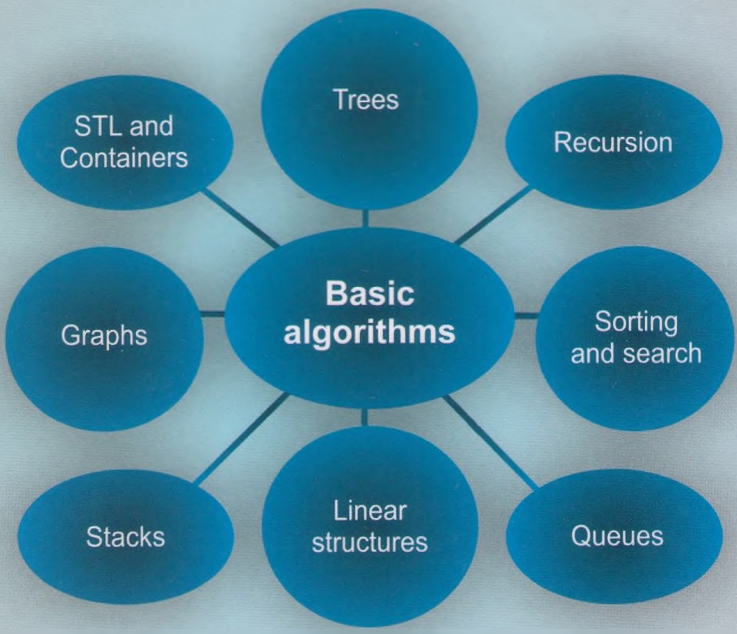
MINISTRY OF EDUCATION AND
SCIENCE OF THE REPUBLIC OF KAZAKHSTAN

A. Seiketov

ALGORITHMS, DATA STRUCTURES AND PROGRAMMING



Almaty, 2016

UDC 004.4
LBC 32.973
S 44

Approved by the Ministry of Education and Science, Republican scientific and practical center «Textbook»

Reviewers:

B.Sh. Kulpeshov – Professor, Doctor of Physics and Mathematics, IITU;
B. Kumalakov – PhD, assistant professor, NU;
B. Buribayev – Professor, Candidate of Technical Sciences, KazNU.

Seiketov A.
S 44 Algorithms, Data Structures and Programming: Textbook. / – Almaty, 2016. – 288 p.

ISBN 978-601-7427-71-9

Discusses the programming from the standpoint of the use of their algorithms in complex data structures including hash function. Describes the methods to search and sort data, analyze the complexity of algorithms. The narration is based on the object-oriented approach using C++ programming language. Shows how the choices of decision task affects the effectiveness and correct implementation of the programs. Contains a large number of code-based programs, illustrating the algorithms.

The content of the textbook is constructed in a sequence of lectures, established by the Ministry of Education and the current model curricula in the Republic of Kazakhstan.

UDC 004.4
RBC 32.973

ISBN 978-601-7427-71-9

© A. Seiketov, 2016
© Association of higher educational institutions of Kazakhstan, 2016

Contents

Introduction **4**
Chapter 1. Principles and basics of programming **6**
1.1 Information systems and programming language 6
1.2 IDE MS Visual Studio 11
1.3 Basic C++ programming 18
1.4 Pointers, Pointer Arithmetic, Functions 38
1.5 Abstract Data Types, Classes 55
1.6 Parameter passing mechanisms. Dynamic memory management 71
1.7 Mechanisms for Code Reuse 83
Chapter 2. Basic algorithms **109**
2.1 Execution Abstraction 109
2.2 Algorithms 113
2.3 Growth of Functions 122
2.4 Asymptotic Notation and Analysis 132
2.5 Guarantees, Predictions, and Limitations 142
Chapter 3. Linear Data Structures **149**
3.1 C-style Strings 149
3.2 The STL and Basic Containers 151
3.3 STL and Container 162
3.4 Queues, Stacks, Using the STL stack Adapter 180
Chapter 4. Recursion **186**
4.1. The basic concept of recursion 186
4.2 Solving problem with recursion 193
Chapter 5. Sorting and Searching **199**
5.1 Linear vs. Binary Search 199
5.2 Basic Sorting Algorithms 203
5.3 Using Hash Tables 221
Chapter 6. Trees and graphs **229**
6.1 Trees 229
6.2 Trees as Search Structures 231
6.3 Using the Tree Based STL Containers 235
6.4 Introduction to Graphs 242
6.5 Fundamental Graph Algorithms 246
Appendixes **262**
Appendix A. 262
Appendix B. 267
Bibliography **287**

Introduction

If you want to speed up a computer program,
You need to do nothing and wait a one year...
(Folklore utterance of programmers)

It is known that the execution speed of computer programs on a straight depends on productivity of the computer. Folk token that is written in an epigraph is quite fair at least because: for saving a leadership in the market vendors of Intel, processors annually increase productivity of processors by 10%. In this year appeared new Intel «Skylake». Results of its first tests testify to fifteen percent increase in productivity. But also it not a limit. Respectively developers promise growth in 20% and 30% of productivity of the processor. Actually, when writing computer programs, is important what the program would work without errors and gave authentic results. Speed of execution of the program today left on the second or third level of a demand.

The amount of information we encounter is sometimes so large that without an effective method of storage and representation, the information is will be useless. Think of all the books contained in your local library. Without the storage system implemented by the librarians, it would be virtually impossible to find a specific book. A data structure, using in the simplest terms, is a structured representation of information. A data structure can be composed of simple pieces of data. Consider as a data structure your address where are you located. There are several items in this data structure, which are simple pieces. Just so, they should be represented in the program.

The series of steps that manipulate and uses a data structure to solve a problem is called an algorithm. Let's think back to the example of the queue at the movie theater. Here, the problem in the management of a group of people waiting to buy tickets. The data structure is a queue of people. The algorithm involves removing a person from the beginning of the queue when the ticket cashier becomes available.

The present textbook, how I conceived it, shall help to decide to you, designated above and many other problems with use of the computers programs. Therefore I used all opportunities which to achievement in target. In my opinion, knowledge and methodology of training can't be hidden from persons interested to seize it. In this regard, I express the gratitude to all, who posted on the Internet and in the open printing of them scientific developments. So some ideas and methodology and implementation of the main part in the present textbook it was taken from course "Algorithms and data structures" of the

Massachusetts university, iCarnegie and also the leading scientists, experts in this area.

I wish success to all who undertook a study of this textbook.

Chapter 1

Principles and basics of programming

1.1 Information systems and programming language

The Program Execution.

On the mission the computer, is a universal device for work with information. By the principles of the device, the computer is a model of the human working with information. The computer system is a hardware and software working in common.

Set of the programs which are stored on the computer forms its software. Set of the programs prepared for work is called the established software. Set of the programs working in this or that time is called a program configuration.

From the point of view of the programmer, computer it is only memory and processor. Any program registers in memory and is processed by the processor, all remaining is insignificant. Memory that place where all information is stored, the programmer shall worry that where and as codes of programs are stored and whether correctly he specified types of variables. The program it is sequence of data and the instructions intended and executed by computing devices.

Mechanical occurs so, data, stored in memory of the computer, transferred into CPU, processed and again return to memory. Therefore, from a line item of the programmer, for the correct understanding of process operation of the computer and the programs enclosed in them, it is enough to have idea how the processor and memory are interconnected among themselves. In parts concerning memory, two of them represent interest, it is RAM and HDD. So we have:

- CPU
- HDD
- RAM

Usually implemented the following sequence. Programs boot into a hard disk drive of the computer where are stored poste restante. In case of program start on execution, the program transferred into a random access memory and from there is controlled, and makes all instructions set by it. The calculating processing device is the central processing unit (CPU).

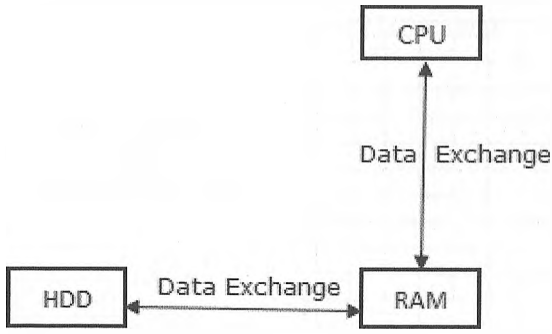


Figure 1.1. Schematic diagram of data exchange

The data interchange diagram between CPU and RAM occurs according to the following diagram.

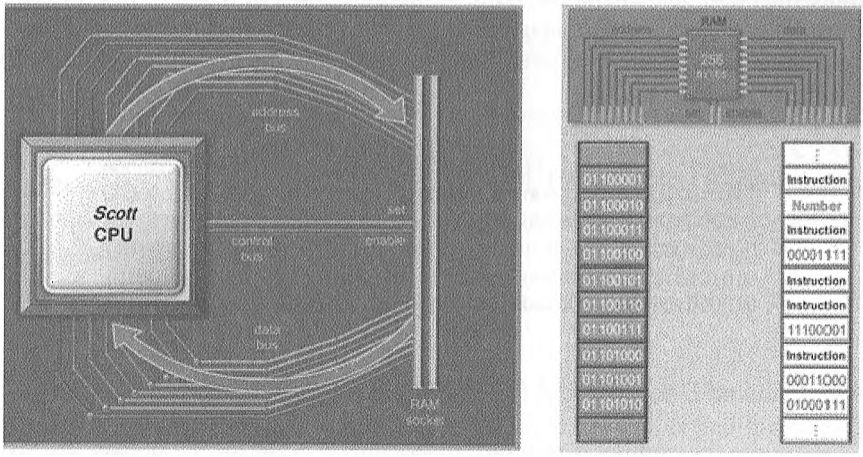


Figure 1.2. The diagram CPU – RAM

As already becomes clear, data processed by the processor can be provided only in the binary code, as the processor is capable to recognize only two line items – 0 and 1. In general, view record of binary numbers in memory of the computer is carried out as follows.

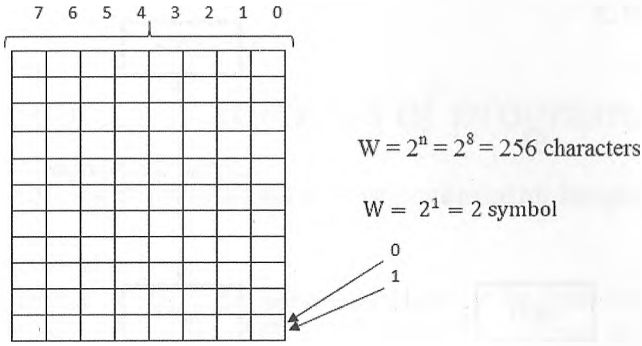


Figure 1.3. Data record system in memory of the computer

The stack of memory of the computer in line contains 8 cells and therefore can place only 8 bits (1 byte). It is clear that in each cell of bit value either 0 or 1 can be written. From here, 1 byte can place in itself a variation from no more than 256 data characters. And though the stack of memory has the strict form, data written in the binary code usually represent record in the form of sequential lines of the program. Here, for example, the normal program cycle in the binary code looks:

```
11000111010001011111101000000000000000000000000000000000
11101011000010011000101101000101111111001000001111000000
00000001100010010100010111111100100000110111110111111100
00000100011111010001001010001011010011011111110010001011
01010101111110000000001100010100100011011101100000100101
01000010000000000100010010101011111111000001111111110111
```

Figure 1.4. Program in the binary code

To write the computer program in the binary code for the person the task from discharge of the impossible. Therefore, programs are written in a clear programming language for the person, or as still speak language of the high level. And only then the compiler transfers ready program to machine code.

The compiler – is the special program which translates a source text of the program from a programming language of the high level in binary codes of instructions of the processor. The code worked out by the compiler registers in a separate executable file. This file is usually launched as the separate program.

The programming language choice, for the scientific studies, in case of development of software products, depends on level of his knowledge and

preferences of the customer. Therefore, for example, the situation in the market with programming languages as of January 2016 looks.

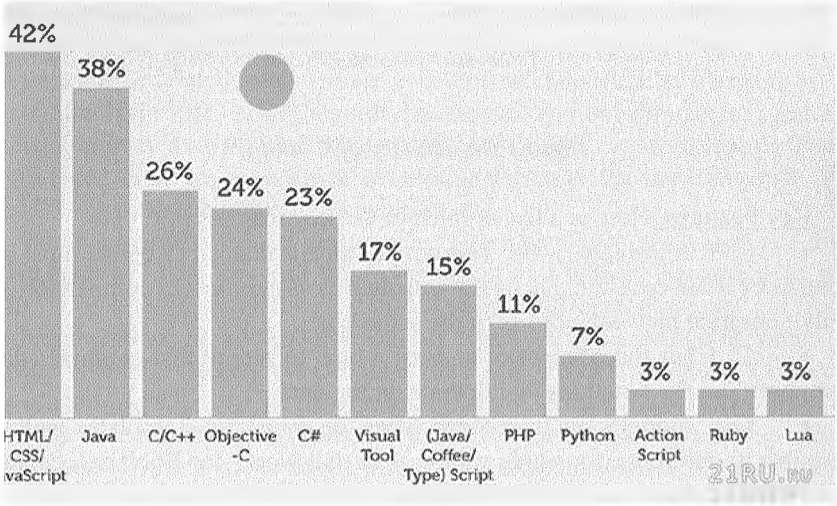


Figure 1.5. Preferences of the market in use of programming languages

In the present textbook the choice of a programming language is defined by the requirement of course of discipline, and it is language C ++. C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation.

History and principal features of language C ++

Development of chronology of programming languages C and C ++ it is provided in fig. 6. For those interested in the timeline, C was first invented in 1970. The language became an ANSI standard in 1980. The first book about ANSI C was published in 1983. C++, originally called "C with Classes," debuted in 1983. The name C++ was coined in 1983. The first C++ book was published in 1985. The Java programming language, in development for several years, debuted in 1995. The C++ language became an ISO standard in 1998.

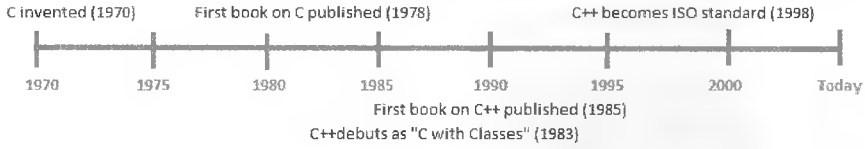


Figure 1.6. History of C and C++

Key Features. Here is a list of some of the key features of C++.

- C++ is *strongly typed*. This simply means that every object must belong to a certain type and that operations such as assignment or comparison are only permitted between objects of the same type.
- C++ has the concept of a *class*, a type of record that combines data members and the functions that operate on the data.
- C++ supports parameterized types, or *templates*. Templates make it possible to define, say, a single vector class that works for Booleans, characters, integers, reals, and so on. The real advantage of C++ templates is that their use does not prohibit the compiler from performing static, or compile-time, type checking.
- C++ supports *inheritance*, a mechanism that makes it possible to build new classes (called *derived classes)* on top of an existing class (called the *base class*) without having to reiterate the base class design for each new class.
- C++ supports *polymorphism*. In C++, polymorphism is achieved through the use of *virtual functions* and pointer variables. Together with inheritance, this turns C++ into a full-fledged object-oriented language.

C++ comes with two libraries known as the *Standard Library* and the *Standard Template Library* (STL) – both of which extend the capabilities of the base language. The Standard Library supplies all the old C libraries as well as new input and output facilities. The STL provides a library of *container types* (types that hold or "contain" collections of objects) as well as a set of *attendant algorithms* (which are general-purpose algorithms for common data structures). That is, the STL supplements the built-in types of C++ with vectors, linked lists, and other useful types.

1.2 IDE MS Visual Studio

Several years ago software companies started to sell "integrated development environments." These are large programs that encompass, within a single package, all the tools needed for developing software. The goal was to reduce the time it took to go through the write-compile-debug cycle by allowing tools to communicate with each other with greater agility. For example, an integrated environment is able to do "incremental compilation," in which the program is compiled in pieces-even as we write other pieces of the same program. Visual C++ is one such integrated environment.

While integrated environments offer clear advantages, they have one disadvantage: we must buy into the environment as a whole. We cannot, as we could with separate tools, mix and match tools of different strengths and from different vendors. For instance, if you use Visual C++ you are almost forced to use the Visual C++ editor, even though you might prefer another one. (You can use other editors for routine programming, but external editors do not provide all of the features and integration of the built-in one.) Even though this is not such a terrible thing, some developers avoid integrated environments for this reason.

The Visual C++ Environment

Task of developing programs involves, to no small extent, the repeated translation of programs from the source level to the machine level, and vice-versa. Compilers translate source code into machine code, and debuggers do something of a reverse translation. Programming, of course, also involves executing machine code as well as managing source code. The management of source code is no simple matter: large programs are divided among many files that are interdependent and that use many pieces provided by third parties and the operating system. To compile the source files of such large programs in the right order is not as easy as it might sound! The process of software development can take advantage of many types of tools, beyond the necessary compilers and debuggers.

Visual C++ is a very large utility and has many components that we will not be using in this course. We invite you to browse its documentation whenever curiosity strikes you, but do not feel uncomfortable if you come across menu items you do not understand. Much of the size of Visual C++ is devoted to what Microsoft calls "application wizards." Application wizards are tools that write programs for you. There are certain types of programs that people need to write very frequently and that are very similar to each other. An application wizard will write most of these for the programmer, who can then customize them with only a little effort. For example, using Visual C++, you

can create an application that is identical to the "Notepad" application without ever writing a single line of code. In this course, we will not use any application wizards, but you should be aware they exist: you might find a use for them later in your career. We don't want you to be confused by the myriad of choices that some of the Visual C++ dialogs will offer you. Despite its complexity, Visual C++ is not hard to use if you restrict yourself to its basic functionality.

Visual C++ has three notions that we will need to understand: solutions, projects, and files. A project is a collection of files, folders and the rules that define their interdependencies. A solution contains one or more projects. While files within a project are intimately linked to each other, projects within a solution are not. In order to compile all the files of a project into an executable program, Visual C++ needs to know how they depend on each other, and in what order to compile them. While projects within a workspace might also be related, they can be compiled separately. For example, one project might be a library (DLL) that is used by another project but compiled independently.

Here so the welcome page of Microsoft Visual Studio approximately can look:

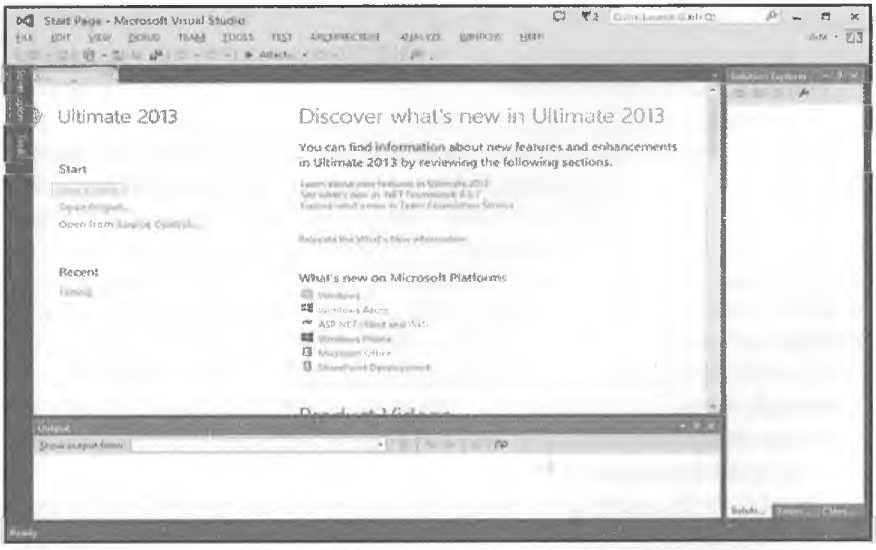


Figure 1.7. Welcome page MS Visual Studio

At this stage it is necessary to select a label of "New Project" and further steps to repeat, as shown in fig. 1.8 and 1.9. Further will open of the Name editor (in the lower part of a window) where it is necessary to set file name. By default, to you will suggest to create the file with extension cpp, it is necessary to agree with a sentence.

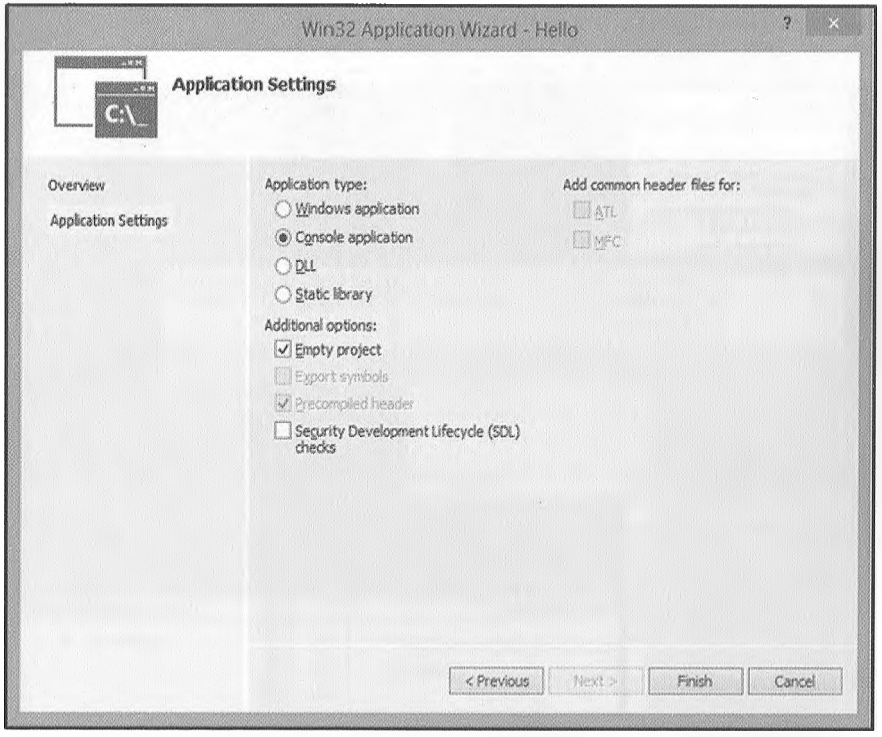


Figure 1.8. Creation of the project

The Application Settings page enables you to choose options that apply to the project. You can see that you are creating a console application and not a Windows application. The Precompiled header option is a facility for compiling header files such as those from the standard library that do not change frequently. When you compile your program after making changes or additions to your code, the precompiled code that has not been changed will be reused as is. This makes compiling your program faster. You can uncheck the Security Development Lifecycle Checks checkbox option; this feature adds functionality for managing large-scale professional projects and we won't be using these. On the right of the dialog there are options for using MFC and ATL, which is outside the scope of this book. For this project you can leave the rest of the options as they are and click Finish.

The application wizard will create the project with default files.

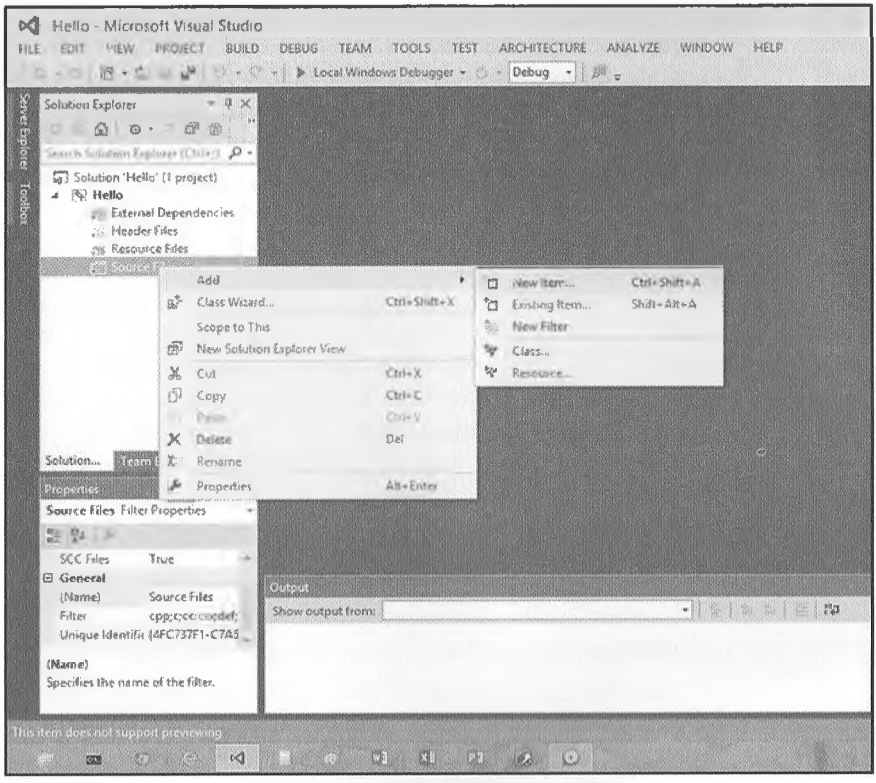


Figure 1.9. Creation elements of the project

The project folder will have the name that you supplied as the project name and will hold the files making up the project definition.

The Solution Explorer tab presents a view of all the projects in the current solution and the files they contain – here, of course, there is just one project. You can display the contents of any file as an additional tab in the Editor pane by double-clicking the name in the Solution Explorer tab. In the Editor pane, you can switch instantly to any of the files that have been displayed by clicking the appropriate tab.

Program structure, variables and data types.

It is traditionally considered to be, the best method to study a programming language it in the course of practical record of the program. Therefore, our first program provided in figure 1.10.

The Hello World program one of the simplest programs which can be written in C ++, but even it already contains fundamental components which are available in all programs in C ++, we will consider them in more detail:

The first line of the program contains a command, called by the directive to a preprocessor. The line begins with a sign a hash (#) and reports to a preprocessor about switching on (start) of standard library of the iostream files in this project. The name is formed from Input/Output Stream. The file includes declarations of library of input-output since, further inevitably these opportunities should be used in the program.

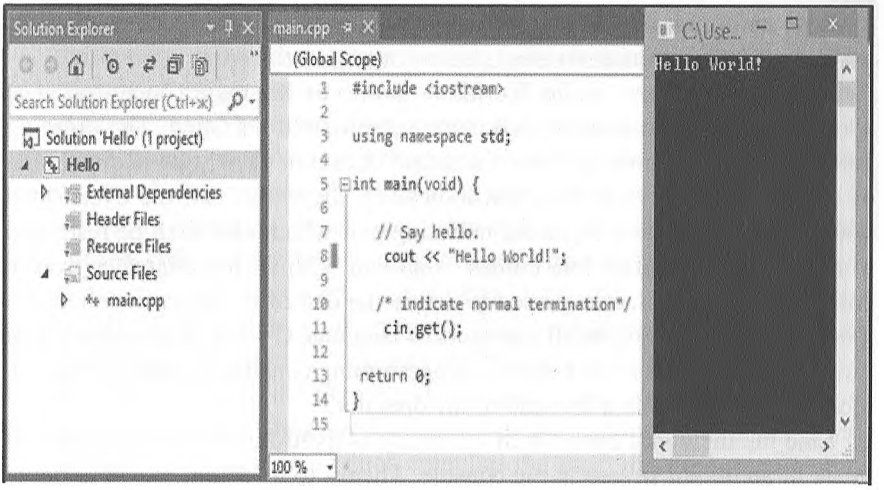


Figure 1.10. "Hello world" program

Second line using namespace std; orders possibility of use of all standard libraries C ++, in language structure they are included in std name space. Without such instruction it should make each function invocation from libraries a personal prefix that undoubtedly would complicate operation.

The line int main () corresponds to the beginning of determination of the principal function. The principal (main) function is that point, from where all programs C ++ begin the execution. For this reason, all programs C ++ shall have a basic function, without it the program won't begin the operation in principle. The word main is accompanied by couple of parentheses (). In C ++ brackets are differentiated with concept of function declaration, in this case they specify that int main expression () also is function. In addition, in brackets can register the list of function arguments, or they can be empty, the void type.

After function declaration, in curly brackets ({}) the function body is de-

clared. In semantic understanding a function body it is space which operators the executing actions implementing this function register. In language C ++ the function body is put into curly brackets.

The line beginning with two signs of oblique line (//), is comments. Any comments in programming don't influence in any way action of the program, their main destination to remind the programmer that it is necessary to execute in the following step. In our case, the comment reminds that in the following step it is necessary to publish the hello message. Comments can be written in one line, or in some lines. In the latter case the multiline comment begins signs separating and increasing, and to come to an end on the contrary, to increase and separate. Writing of comments rather simple, but nevertheless require strict observance of rules.

Next line cout << "Hello World! "; is one of the major operators of C ++ and as though consists of two parts ordering and narrating. The ordering part of cout is a name of a flow of standard output in C ++, two in a row sign of corners (<<) are flow direction indicators, them also call the overloaded operators. Behind these signs the narrating part which else shall be restricted to quotes registers. The line comes to an end with period character from a comma (;). This character is used to define the end of the operator, ruled this mandatory and extends on all operators of language C ++. It is considered that the most widespread syntax error in programming consists in what to forget to include a semicolon in a line end of the operator.

Line the including operator of *cin.get* (); is fixed function of execution of break of an output stream in the console. Thanks to its action it is possible to consider, the text printed in the console.

Return statement of return 0; completes operation of a basic function, the operator to be accompanied by return code (in our example value zero). Return code 0 for main of function is interpreted with understanding of incorrect completion of a program runtime (without errors) during its execution. Actually, it is the most normal method of completion of operation of the console of the program in C ++.

Development phases of the program C ++.

In C ++ for start of the program it is necessary to execute beforehand the following phases of development of the program which consist of five steps: editing, preliminary processing, compiling, binding and execution.

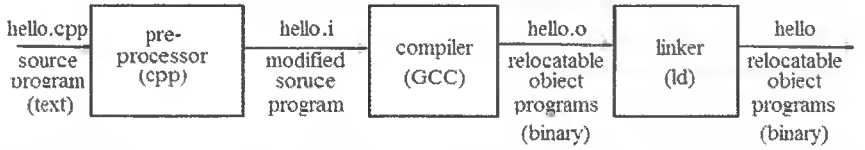


Figure 1.11. Development phases of program

The first step is editing, or creation of the file which contains the source code. The source code in any editor is written. As editors any text editors, or specialized Integrated Development Environments (IDE) can serve.

Preliminary processing (preprocessing), it makes modification of the source code of the program. Preprocessor instructions contain the commands called directives of a preprocessor, report to a preprocessor about execution of some actions. They instruct a preprocessor to include contents of files in a source text of the program. Preprocessing also makes macro definitions.

Compiling is process which transforms beforehand the written source text to an object code. In the beginning check and confirmation is made that syntax of a source text is correct. Quite often, when the program is compiled (especially for the first time), from syntax line items something is wrong. This circumstance is specified in "compile error". Then it is necessary to return to the first step and to edit the source code (to correct an error). Example of C ++ the compiler is GNU Compiler Collection or GCC. Binding is a step which is executed by the same tool which compiles the program. Process of binding involves combining of codes of the object made by the compiler with another beforehand collected by codes of library. Result of operation of binding the performing file which contains compiled and the connected codes of object the programs saved on a hard drive.

After the source code of the program was edited, beforehand is processed, compiled and connected in the file, the program is ready and can be executed. Errors which face at this stage, are known as "runtime errors" which one many more difficult to correct, than the previous errors as they include problems of logic of the program.

Compiling and connection of programs which consist of many filescan be simplified, using makefile. Makefile - the text file which defines the relations and dependences between files in the being created project. The principal advantage of use of makefile consists that it allows the programmer to collect only those files of a source text which changed. It can be very useful working with projects which consist of many files of a source text.

1.3 Basic C++ programming.

Data types and variables.

Programming isn't restricted only to the publication of plain texts on the monitor screen as it was shown in the first sample program. Let's study important concept of programming – concept of a variable.

Let's say it is necessary for you in the memory (in the head) to save two different numbers, let it will be 4 and 2, and then we will add (we don't add!) number 3 to the first, i.e. we save in memory as though record 4+3, but not 7! Now we will write to themselves in memory result, whichwill be calculated as the amount of numbers 4+3 (but not 7) minus the second (3).

That process which you just did in the memory is comparable how the computer with variables operates. How to each value (variable) you in the memory selected a personal place is characteristic. Let's address to a program code in figure 1.12.

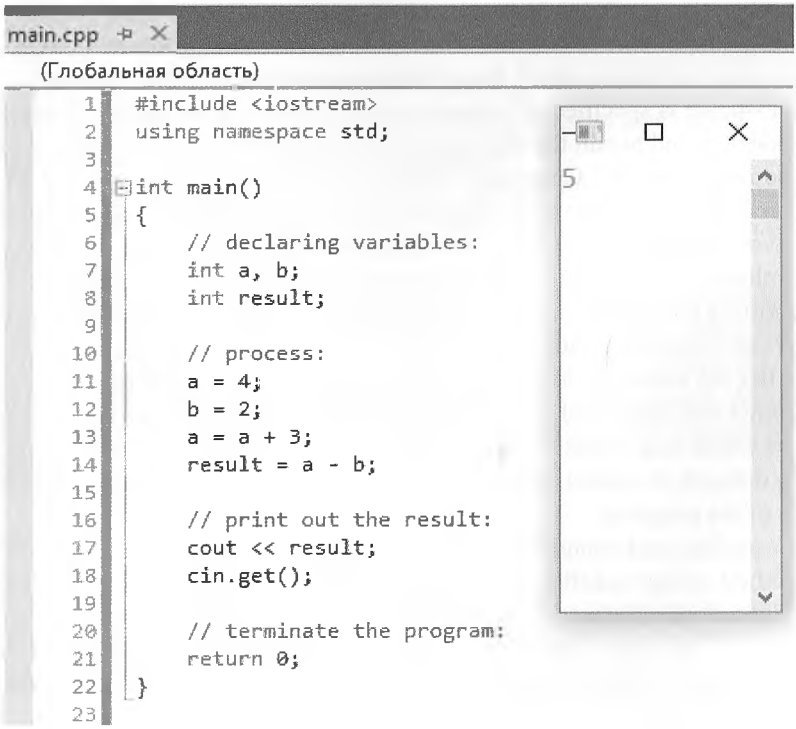


Figure 1.12

In the program two integer variables over which carrying out any actions and a place in memory under result are necessary in the beginning were declared. From this it follows that variable declaration is the requirement of separation of a place in memory of the computer for saving in them some data. In turn each variable needs the identifier which distinguishes it from others. So in the above-stated code variable identifiers were a, b and result. Here we will mark, on identifiers some restrictions with syntax rules are set. Identifiers are admissible if they represent only a combination of letters, digits and underline character and especially ites match reserved keywords. Identifiers are sensitive to the register of the keypad and can begin only with a letter or underline character.

The following standard words are specially reserved keywords, their use as identifiers isn't allowed:

asm,auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, els,e enum, explicit, export, extern, falsefloat, for, friend, goto, if, inline, in,t long, mutable, namespace,e new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, TR, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

Let's in addition specify, alternative representations for some operators can't be used as identifiers as they are reserved keywords in some circumstances, namely:

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq.

Fundamental data types.

When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

Next, you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one.

Fundamental types in C++ are divided into three categories: integral, floating point, and void. Integral types are capable of handling whole numbers. Floating point types are capable of specifying values that may have fractional parts.

The void type describes an empty set of values. No variable of type void can be

specified – it is used primarily to declare functions that return no values or to declare generic pointers to untyped or arbitrarily typed data. Any expression can be explicitly converted or cast to type void. However, such expressions are restricted to the following uses:

An expression statement. (See Expressions, for more information.)

The left operand of comma operator. (See Comma Operator for more information)

The second or third operand of conditional operator (? :). (See Expressions with the Conditional Operator for more information)

The following table explains the restrictions on type sizes. These restrictions are independent of the Microsoft implementation.

Fundamental Types of the C++ Language

Table 1.1

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating-point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating-point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)

* The values of the columns *Size* and *Range* depend on the system the program is compiled for. The values shown above are those found on most 32-bit systems. But for other systems, the general specification is that *int* has the natural size suggested by the system architecture (one *"word"*) and the four integer types char, short, *int* and long must each one be at least as large as the one preceding it, with char being always one byte in size. The same applies to the floating point types float, double and long double, where each one must provide at least as much precision as the preceding one.

Declaration of variables.

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier. For example:

```
6    int a;
7    float mynumber;
```

Listing 1.1

These are two valid declarations of variables. The first one declares a variable of type *int* with the identifier *a*. The second one declares a variable of type *float* with the identifier *mynumber*. Once declared, the variables *a* and *mynumber* can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

```
9    int a, b, c;
```

Listing 1.2

The integer data types *char*, *short*, *long* and *int* can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier *signed* or the specifier *unsigned* before the type name. For example:

```
11   unsigned short int NumberOfSisters;
12   signed int MyAccountBalance;
```

Listing 1.3

By default, if we do not specify either *signed* or *unsigned* most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

```
14   int MyAccountBalance;
```

Listing 1.4

With exactly the same meaning (with or without the keyword signed).

An exception to this general rule is the *char* type, which exists by itself and is considered a different fundamental data type from *signed char* and *unsigned char*, thought to store characters. You should use either `signed` or `unsigned` if you intend to store numerical values in a *char*-sized variable.

`Short` and `long` can be used alone as type specifiers. In this case, they refer to their respective integer fundamental types: `short` is equivalent to `short int` and `long` is equivalent to `long int`. The following two variable declarations are equivalent:

```
16    short Year;
17    short int Year;
```

Listing 1.5

Finally, signed and unsigned may also be used as standalone type specifiers, meaning the same as signed int and unsigned int respectively. The following two declarations are equivalent:

```
19    unsigned NextYear;
20    unsigned int NextYear;
```

Listing 1.6

Scope of variables

All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that *a*, *b*, and *result* were of type *int*. A variable can be either global or local in scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.

Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

The scope of local variables is limited to the block enclosed in braces ({ }) where they are declared.

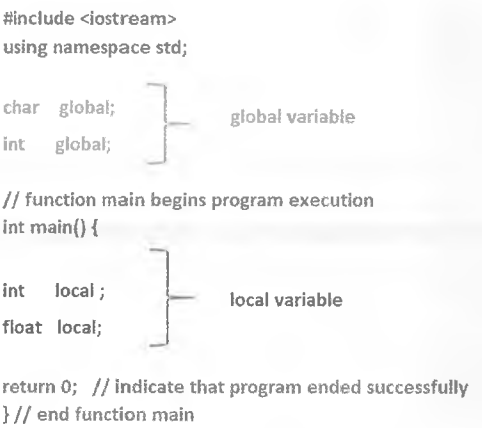


Figure 1.13

Initialization of variables

When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++:

The first one, known as *c-like initialization*, is done by appending an equal sign followed by the value to which the variable will be initialized:

type identifier = initial_value ; (1.1)

For example, if we want to declare an *int* variable called *a* initialized with a value of 0 at the moment in which it is declared, we could write:

```
13   int val 1 = 7:
```

Listing 1.7

The other way to initialize variables, known as *constructor initialization*, is done by enclosing the initial value between parentheses (()):

```
14    int val_2(4);
```

Listing 1.8

Both ways of initializing variables are valid and equivalent in C++.

```
in.cpp
(Глобальная область)
 1  // Printing a line with multiple statements.
 2  #include <iostream>
 3
 4  // function main begins program execution
 5  int main() {
 6
 7      std::cout << "Welcome ";
 8      std::cout << "to programming in C++!\n";
 9      std::cout << "Welcome \"to C++!\n";
10
11      int var_1 = 7;  // initial value = 7
12      int var_2(4);   //  initial value = 4
13      int result;     // initial value undetermined
14
15      var_1 = var_1 + 6;
16      result = var_1 - var_2;
17
18      std::cout << result;
19
20      return 0;   //  indicate that program ended successfully
21
```

Listing 1.9

The size of C++ basic data types

The obvious next question is "how much memory do variables of different data types take?". You may be surprised to find that the size of a given data type is dependent on the compiler and/or the computer architecture!

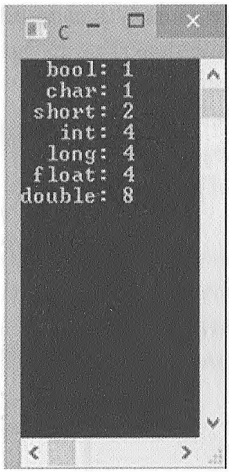
C++ guarantees that the basic data types will have a minimum size. The following example program uses the C++ *size of* operator to display the size, in bytes, of the fundamental data types of the particular C++ implementation used in.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char* argv[]) {

    cout << "  bool: " << sizeof(bool) << endl;
    cout << "  char: " << sizeof(char) << endl;
    cout << " short: " << sizeof(short) << endl;
    cout << "   int: " << sizeof(int) << endl;
    cout << "  long: " << sizeof(long) << endl;
    cout << " float: " << sizeof(float) << endl;
    cout << "double: " << sizeof(double) << endl;
    cin.get();
    return EXIT_SUCCESS;
}
```



Listing 1.10. Data types and the sizeof operator

C++ uses the keyword const to allow programmers to create "read-only" variables. This keyword signals that the variable being declared cannot be modified after it is initialized. Listing 1.11 shows the declaration and initialization of some "read-only" variables, otherwise simply known as "constant variables" or just "constants."

```
const int BOILING_POINT = 100;
const int FREEZING_POINT = 0;
const float PI = 3.14159;
```

Listing 1.11. Constant variables

Introduction to strings

Variables that can store non-numerical values that are longer than one single character are known as strings.

The C++ language library provides support for strings through the standard string class. This is not a fundamental type, but it behaves in a similar way as fundamental types do in its most basic usage.

The first difference with fundamental data types is that in order to declare and use objects (variables) of this type we need to include an additional header file in our source code: <string> and have access to the *std* namespace (which

we already had in all our previous programs thanks to the using namespace statement).

```
.cpp
(глобальная область)
 1  // my first string
 2  #include <iostream>
 3  #include <string>
 4
 5  using namespace std;
 6
 7  int main()
 8  {
 9      string mystring;
10      mystring = "This is the initial string content";
11          cout << mystring << endl;
12      mystring = "This is a different string content";
13          cout << mystring << endl;
14      return 0;
15  }
```

Listing 1.12

As you may see in the previous example, strings can be initialized with any valid string literal just like numerical type variables can be initialized to any valid numerical literal. Both initialization formats are valid with strings:

```
 9      string mystring = "This is a string";
10      string mystring("This is a string");
```

Listing 1.13

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and being assigned values during execution:

```
int addition (int a, int b)

z = addition ( 8 ,  4 );
```

Listing 1.14

These different data types describe not only the fundamental operations that the language performs on the data, but also the range of values that the data types accept. Since different data types have different allowable values, the language can check to ensure that a programmer only assigns appropriate values to a data object. An error occurs if a programmer assigns an inappropriate value, such as a number that is too large or too small, to a data object. This mechanism is known as *type checking*.

In C++, the storage space requirements are left to the discretion of the compiler implementers. Therefore, on one platform a C++ *int* may be two bytes long, while on another it may be four bytes long. Usually, the size in bytes of a variable is unimportant, except that it dictates the range of values that a variable can accept.

Arrays.

C++ provides basic support for a sequence of homogeneous data objects through arrays. Both similarities and differences exist arrays in C++.

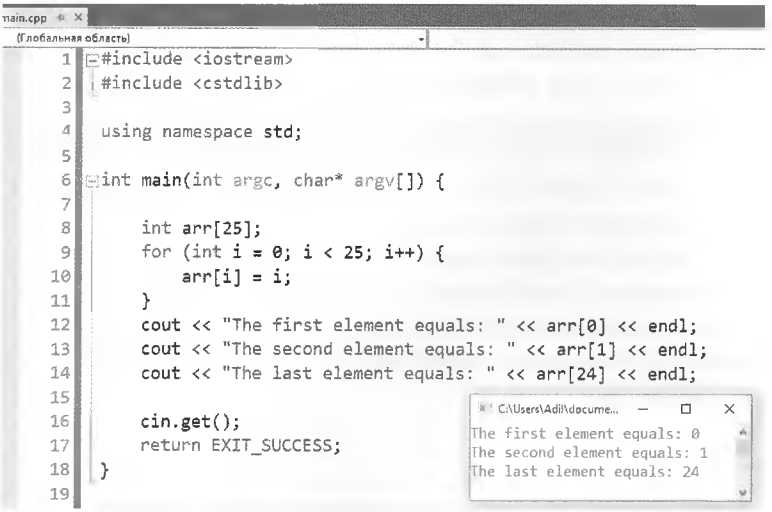
We can declare and initialize an array of ten *ints* using the following.

```
10  // declare and create an array of integers
11  int cpp_array[10];
12
```

Listing 1.15

The double bracket ([]) in the declaration indicates that the line declares an array. Notice the placement of the double bracket ([]) in each of the examples. ([]). A programmer encloses in brackets an index number of the element they wish to access. Indexing of arrays in C++, begins with zero. This means that the first element of an array is accessed at index 0, the second element at index 1, the third element at index 2, and so on. The following Listing 1.16 demonstrates accessing the elements of a C++ array.

One inherent danger that exists in using C++ arrays is the lack of bounds checking. This is not the case in C++, since boundary checking of arrays is not supported. If we have an array of ten elements in C++, we can attempt to access the 12th or 20th or even 100th index of the array.



Listing 1.16. Accessing elements of a C++ array

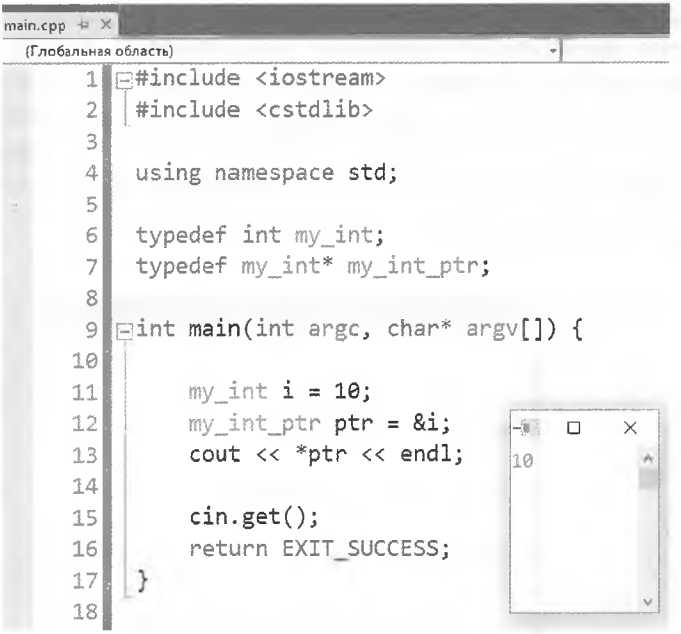
The vector data type provides a much safer alternative to a basic C++ array. In C++ vectors exist as feature-rich array. For example, unlike an array in C++ a vector has a built in function that returns the size of the vector. Vectors also provide bounds checking support, and unlike arrays, they automatically increase in size when the need arises.

Creating New Data Type Names

It is possible in C++ for a programmer to create additional names for existing data types. Creating another name uses the keyword *typedef*. The syntax to create a new name is as follows.

typedeftype-expressionnew-name; (1.3)

It is possible in C++ for a programmer to create additional names for existing data types. Creating another name uses the keyword *typedef*. The syntax to create a new name is as follows.



Listing 1.17. A sample use of typedef

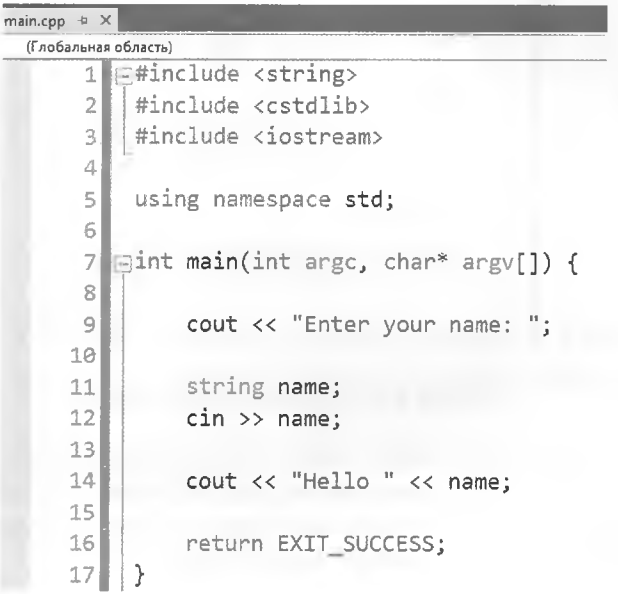
Using data type names created with *typedef* allows programmers to encapsulate their choice of data types. This is good programming practice, since, if the need arises, a programmer can easily change all uses of a particular data type simply by changing the definition of the *typedef*.

Input and Output.

Input and output in C++ is based on the concept of a *stream*. A stream is a sequence of bytes that flow from something to something else. The process of output involves moving bytes, one at a time, from a program to a device. This device could be a monitor, a printer, or even a file on a hard drive. Input is the opposite. Input involves the flow bytes from a device (a keyboard, a file, a network connection) into the program.

The above code first streams data (the characters in the text string "Enter your name:") from the program to a device (the console) in line 9. Stream based output operations use the << operator to indicate the data to write to the stream. The stream used in this line is the output stream referenced by object *cout*. This object is of type *ostream*, which is short for "output stream."

The program in Listing 1.18 then streams data from the keyboard into the program, storing the user entered text in the variable `name`. Stream input operations use the >> operator to specify the variable where the program should place the data it reads from the stream. The stream used in this line is the input stream referenced by object *cin*. The *cin* object is of type *istream*, which is short for "input stream." The Listing then again streams output to the console. In line 14, we see that we can place more than one piece of data into the stream in one statement.



Listing 1.18. Stream based output

We can open and use streams to read and write data to and from many devices besides the console. For example, a program can use a file output stream to write data to the file system. Network communication through sockets is also stream based.

Using the Standard Streams

Three specific streams are always available throughout the lifetime of every C++ program. These are the *standard input*, *standard output*, and *standard error* streams. Each of these standard streams has a specific use. The standard input stream reads data from the console, the standard output stream writes

data to the console, and the standard error stream displays error messages to the console.

Programmers access the standard streams through a set of objects. Objects *cin* and *cout* provide access to the standard input and output streams, respectively. We have seen their use in previous examples. Object *cerr* provides access to the standard error stream. Listing 1.19 demonstrates use of the three standard streams.

```
 9      cout << "Enter your name and age: ";
10
11      string name;
12      int age;
13
14      cin >> name >> age;
15
16      if (age < 0) {
17          cerr << "\nInvalid age entered";
18      }
19      else {
20          cout << "\n" << name << " is " << age;
21      }
```

Listing 1.19. Standard output and the << operator

Programmers do not have to explicitly open and close the three standard streams. These streams are automatically available for use (through their respective objects) when a program begins execution. Programmer's must explicitly open and close all other input and output streams.

File Input and Output

File based input and output is similar to the mechanisms for keyboard and screen I/O. The main difference is that programmers must explicitly open and close files. In pseudocode, a generic program that reads input from a file might look like this.

```
1  #include <fstream>
2  #include <iostream>
3  #include <cstdlib>
4
5  using namespace std;
6
7  int main(int argc, char* argv[]) {
8
9      ifstream inf;
10     inf.open("somefile.txt");
11
12     if (!inf) { // Check to see if file opened
13         cerr << "Could not open file!" << endl;
14         cin.get();
15         return EXIT_FAILURE;
16     }
17
18     int x;
19
20     // While input remains, read an integer.
21     while (inf >> x) {
22         cout << x << endl;
23     }
24
25     inf.close();  // Close the input file
26     cin.get();
27     return EXIT_SUCCESS;
28 }
```



Listing 1.20

The object used in file input is of type *ifstream*. Since this class is not part of the C++ language by default, we must include its library in our program. This is done in line 1. Line 9 declares an object of type *ifstream*, and line 10 calls the member function `open` to open a file. It is good programming practice to check whether an attempt to open a file actually succeeded. Also, note the use of the extraction operator in the conditional of the while-loop in line 20. As long as the extraction attempt succeeds, *true* is returned (and the value read from the file is assigned to `x`). A failure to read another integer is signaled by a value of *false*. This terminates the while-loop.

File output resembles file input. We still need to include a reference to the *fstream* library, but we use an *ofstream* object instead of an *ifstream* object. Here is an example.

main.cpp*

(Global Scope)

```
#include <fstream>
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char* argv[]) {

ofstream onf;
onf.open("output.txt");

if (! onf) { // Check to see if file opened
    cerr << "Could not open file!" << endl;
    return EXIT_FAILURE;
}

for (int i = 1; i <= 10; i++) {
    onf << "This is line " << i << endl;
}

onf.close();  // Close the output file
```

Listing 1.21. File output

Keep in mind that file I/O is stream based, so it is sequential in nature. We read or write one character after another, starting at the beginning of the file and continuing until the end of the file is reached. For technical reasons, it is difficult and very costly to move back and forth, and the only access method that can be implemented reasonably efficiently is a simple beginning to end pass through the file.

The Preprocessor

The *preprocessor* is a tool that C++ programmers use to manipulate the contents of source code files prior to compilation. In the most general sense, the preprocessor performs text substitution and text modification. Higher-level features emerge when we consider the overall effect of these rather basic manipulations. File inclusion, macro substitution, and conditional compilation are three higher-level features the preprocessor provides to a programmer.

Source code files, as authored by programmers, typically need to be modified in various ways before compilation can take place. Because programmers rely on the preprocessor to perform these modifications, knowledge of the

basic use of the preprocessor is essential. Since C++ programs consist entirely of text, a programmer must use the preprocessor to include the declarations of external classes or functions. This is known as file inclusion. The use of other preprocessor features, such as macro substitution, is not necessarily required. Macros exist as a convenience to the programmer. They also provide backward compatibility with C programs.

A programmer interacts with the preprocessor through commands called *preprocessor directives*. Beginning with the number sign (#), preprocessor directives are single-line commands a programmer places into a source code file. Since preprocessor directives are not C++ code, they do not follow the language's scoping rules and therefore can appear on any line in a source code file. The appearance of a preprocessor directive in a source code file instructs the preprocessor to perform some action. The action the preprocessor takes depends on the directive. For some directives, the preprocessor makes exactly one modification in the source code file. An example of this is file inclusion where the preprocessor includes the contents of another file into the file being processed. Preprocessor directives used to define other tasks, such as a macro substitution, can cause the preprocessor to make several modifications in a source code file.

File Inclusion.

File inclusion is a feature of the C++ preprocessor that allows a source code file to use shared code. We consider shared code to be classes or functions declared in other files. In C++, we can only access a shared class or function by including its declaration into our program. This must be done since C++, unlike *Java*, shares code textually. Imagine having to include manually the declaration for every function and class used in a program. That would be quite a cumbersome task. Luckily, the preprocessor automates this through file inclusion.

A programmer issues the #*include* preprocessor directive to instruct the preprocessor to perform file inclusion. This directive takes a file or library name as a parameter. When processing an # *include*directive, the preprocessor replaces the line containing the directive with the contents of the file that the directive specifies. Listing 1 demonstrates file inclusion.

```
(Global Scope)
1  #include <string>
2  #include <cstdlib>
3  #include <iostream>
4  #include <fstream>
5
6  #include "my_functions.h"
7  #include "my_class.h"
8
9  #include "..\another_file1.h"
10 #include "directory\sub\another_file2.h"
11
12 using namespace std;
13
14 int main(int argc, char* argv[]) {
15
16     // Rest of program...
```

Listing 1.22

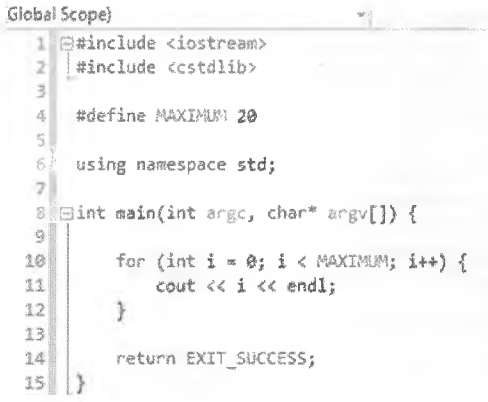
The #*include*directive accepts two different forms of its parameter. The above example demonstrates use of the first form in lines 1 through 4. In this form, angle brackets surround the parameter in the #*include*directive. This signifies that the preprocessor should search for the specified file in an implementation dependent set of places. Typically, this set of places includes system and library directories. Double quotes surround the parameter in the second form of the #*include*directive. In this form, the #*include*directive instructs the preprocessor to look for the specified file in the same directory where the file being preprocessed exists. Lines 6 through 10 of Listing 1 illustrate this form of the directive. C++ programmers typically use the first form to include libraries and the second form to include files they have created.

Macro Substitution

The C++ preprocessor can perform a programmer defined text substitution throughout an entire source code file. This is known as *macro substitution*. Programmers define a macro using the #define preprocessor directive, which can take the following form.

#define *identifier replacement – text* (1.4)

Using the #define directive, a programmer declares an identifier and specifies its replacement text. Macro substitution in a source code file involves the preprocessor replacing every occurrence of the *identifier* with the *replacement-text*. Listing1.23 illustrates macro definition and usage.



Listing 1.23. Macro substitution

We can use macro substitution to implement a constant variable. In the above Listing, #*define* creates an identifier named *MAXIMUM*, and associates with it the replacement text `20`. Anywhere in the program source code that the preprocessor finds `MAXIMUM`, it replaces with `20`. Macro substitution, in this case, allows the identifier `MAXIMUM` to function as a constant variable.

C++ programmers should use the keyword *const* instead of macro substitution to create constant variables. Because the keyword *const*is part of the C++ language (and not a preprocessor feature), constants created with it support type checking better than constants created using macro substitution. Constants created with macro substitution exist in C++ to provide backward compatibility with C programs.

Conditional Compilation

Beyond macro substitution, a more important reason to use `#define` is to support *conditional compilation*. Using #*define*, and some other preprocessor directives, we can instruct the compiler to compile only certain sections of our source code. This is useful in many circumstances, one of which is for inserting debugging code that can be easily enabled and disabled. Below we see an example that uses the #*define*, #*if*, and #*endif* directives.

```
(Global Scope)
1  #include <iostream>
2  #include <cstdlib>
3
4  #define DEBUG
5
6  using namespace std;
7
8  int main(int argc, char* argv[]) {
9
10 #if defined(DEBUG)
11     cerr << "Debugging enabled" << endl;
12 #endif
13
14     int arr[10];
15     for (int i = 0; i < 10; i++) {
16         arr[i] = i;
17
18 #if defined(DEBUG)
19         cerr << "i = " << i << endl;
20         cerr << "arr[i] = " << arr[i] << endl;
21 #endif
22
23     }
24     return EXIT_SUCCESS;
25 }
```

Listing 1.24. Conditional compilation

The *#if* preprocessor directive works similar to a regular if-statement, except that it has to paired with a*#endif* directive. These two directives partition a section of source code that can be conditionally compiled. The preprocessor evaluates the value that follows the *#if*. If this value evaluates to true (non-zero), the preprocessor includes the source code block. If it evaluates to false, the preprocessor omits the source code block. In the above Listing, *defined* (DEBUG) follows the *#if* directives. The preprocessor evaluates this to true only if we have defined an identifier named DEBUG. Since we have defined DEBUG in line 4, the source code blocks partitioned by the *#if* and *#endif* pairs will be compiled. The power of this technique is apparent when we realize all we have to do to disable the debugging code found throughout the program is remove the definition of DEBUG from the program. This causes the preprocessor to omit the debugging code.

Conditional compilation is also often used to prevent multiple definitions of classes and functions that are contained in header files. Including a header file more than once in a program can cause class and function redefinition problems. We can prevent this with a technique that uses conditional compi-

lation. Below we see in line 1 the *#if* directive used to check if the program has defined the identifier _PERSON_H_. If it has not been defined, then the rest of the source code in the example is processed. If it has been defined, the source code is skipped by the preprocessor. The key to this technique is in line 19, where the program defines _PERSON_H_. If we had a program that had several source code files that all included the following header file, the conditional compilation would ensure that the content of the file is included only once. The first time the file was included would result in the definition of _PERSON_H_, which would then prevent the inclusion of the contents of the file a second time.

```
18 #if !defined(_PERSON_H_)
19 #define _PERSON_H_
20
21 class Person {
22
23     // Class declaration...
24 };
25
26 #endif
```

Listing 1.25. Preventing multiple declarations

Shortcut conditional compilation constructs exist that we can use in place of the *defined* operator. The directive *#ifdef identifier* is equivalent to *# if defined (identifier)*. Likewise, the directive *#ifndef identifier* is equivalent to *# if !defined (identifier)*.

1.4 Pointers, Pointer Arithmetic, Functions

Pointers and Indirection

A *pointer* is a variable that stores the memory address of another variable. We have seen already in C++ that data types dictate the range and type of values a variable can store. Variables of data types that we have examined so far store values such as integer numbers, floating-point numbers, and character strings. A pointer variable is unique in that it stores the memory address of another variable. A memory address is the specific location in main memory where a variable exists during program execution.

Programmers use pointers to indirectly access and manipulate other variables. This access and manipulation is considered "indirect" since it is accomplished using a pointer instead of the actual variable being modified. Indirection allows the creation of complex data structures and powerful algorithms. For instance, without pointers and indirection it would not be possible to create a linked list data structure.

Basic Operations.

The declaration of a pointer variable requires the use of some unfamiliar syntax. A pointer declaration must prefix its variable name with an asterisk (*). This signifies to the compiler that the variable declared is a pointer. Listing 1 demonstrates the declaration of a few pointer variables.

```
10  int *int_ptr;
11  float *float_ptr;
12  char *char_ptr;
```

Listing 1.26. Pointer declaration

Even though pointers only store memory addresses, the declaration of a pointer variable must specify a specific data type whose memory address the pointer can store. This is because variables of different data types may not necessarily use the same amount of storage in memory. To allow indirect access and manipulation of a variable using a pointer, the compiler must know the exact size of the variable. Listing 1.26. declares three pointers. They are declared as a pointer to type *int*, a pointer to type *float*, and a pointer to type *char*, respectively.

Pointer initialization also requires some new syntax. We cannot simply initialize a pointer to a non-pointer variable because pointers store memory addresses and not regular values. Instead, we must obtain the memory address of the variable. The address-of (&) operator returns the memory address of where a variable is stored. Listing 1.27 declares an integer variable and sets 100 as its initial value. It then declares a pointer to type *int*, initialized with the address of the variable *i*. In the context of this example, we can say that *ptr* "points" to *i*.

```
9   int i = 100;
10  int *ptr = &i;
```

Listing 1.27. Pointer initialization

Dereference

Indirect access and manipulation of variables using pointers is accomplished using the *dereference* operator. A programmer applies the dereference operator (*) to access or modify the value of the variable pointed to by a pointer. Listing 1.28 demonstrates the indirect access and modification of a variable using a pointer.

```
 8
 9      int i = 100;
10      int *ptr = &i;
11
12      cout << *ptr << endl; // outputs 100
13      *ptr = 200;           // Sets i = 200
14      cout << i << endl;    // outputs 200
```

Listing 1.28. Pointer dereferencing

We can only safely dereference pointers that point to valid memory addresses. Dereferencing pointers that have not been initialized to valid memory addresses causes a run-time error. One technique used to avoid this problem is the initialization of pointer variables to the *null pointer*. The null pointer represents a memory location that cannot be dereferences. Initializing pointers to the null pointer prevents an accidental dereference from causing a run-time error. The null pointer is referenced using the predefined NULL macro.

```
20  int *ptr = NULL;  // Initialize to null pointer
21
22  // ... Some other code ...
23
24  int i = 100;
25  ptr = &i;  // Now set our pointer
```

Listing 1.29. The null pointer

Programmers also use null pointers as sentinel values to indicate the end of some structure or sequence. We examine this more important use of the null pointer later in this course.

Programmers can invoke member functions of an object through a pointer. The arrow operator (->) first dereferences the pointer and then invokes the named member function.

```
#include <iostream>
#include <cstdlib>

using namespace std;

class Thing {
public:
    int number;
    Thing() : number(0) {}
    void method1() {
        cout << "Method 1" << endl;
    }
    void method2() {
        cout << "Method 2" << endl;
    }
};

int main(int argc, char* argv[]) {

    // Direct access
    Thing thing;
    cout << thing.number << endl;
    thing.method1();
    thing.method2();

    // Equivalent indirect access
    Thing *ptr = &thing;
    cout << ptr->number << endl;
    ptr->method1();
    ptr->method2();

    return EXIT_SUCCESS;
}
```

Listing 1.30. The arrow operator

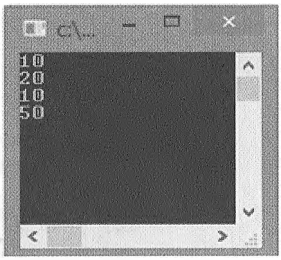
Pointer Arithmetic

Pointer arithmetic is the use of addition and subtraction to change the memory location that a pointer stores. These operations are all relative to the memory location currently stored in a pointer. Subtracting the value 3 from a pointer, for instance, yields the address three locations before the pointer. Incrementing a pointer causes the pointer to "point" to the next memory location.

Pointer arithmetic automatically takes into account the size of the data type

involved. Adding the value 3 to an integer pointer actually adds 3 multiplied by the size of an integer to the address currently stored in the pointer. A typical *int* size is four bytes. In this case, adding the value 3 to an integer pointer adds 12 (3 * 4) to the memory address of the pointer. Knowing the actual number of bytes that pointer arithmetic is adding and subtracting is really not useful to us. It is only important to understand that C++ handles this multiplication automatically. We only need to view pointer arithmetic as a way to access different locations in a data structure (usually an array). Listing1.31 contains a demonstration of pointer arithmetic.

```
7  int array[] = { 10, 20, 30, 40, 50 };
8  int *ptr = &array[0];
9
10 cout << *ptr << endl;        // outputs 10
11
12 ptr++;
13 cout << *ptr << endl;        // outputs 20
14
15 ptr--;
16 cout << *ptr << endl;        // outputs 10
17
18 cout << *(ptr + 4) << endl; // outputs 50
19
```



Listing 1.31. Basic pointer arithmetic

Pointer arithmetic has many important uses in C++ programming. One use is the traversal of arrays.

```
20 int data[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
21 int *ptr = &data[10];  // points to last element in array
22
23 do {
24     cout << *ptr << endl;
25 }
26 while (ptr-- != &data[0]);
```

Listing 1.32. Array traversal

Functions

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++. A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

type name (parameter1, parameter2, ...) { statements } (1.5)

where:

- type is the data type specifier of the data returned by the function.
- name is the identifier by which it will be possible to call the function.
- parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: *int* x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- statements is the function's body. It is a block of statements surrounded by braces { }.
-

```
#include <iostream>

using namespace std;

int addition (int a, int b)
{
  int r;

        r=a+b;

  return (r);
}

int main ()
{
  int z;
  z = addition (8,4);

     cout << "The result is " << z;

  cin.get();
  return 0;
}
```

C:... — □ ×

The result is 12

Listing 1.33

In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution by the main function. So we will begin there.

We can see how the main function begins by declaring the variable z of type int. Right after that, we see a call to a function called addition. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:

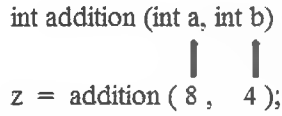


Figure 1.14

The parameters and arguments have a clear correspondence. Within the main function we called to addition passing two values: 5 and 3, that correspond to the int a and int b parameters declared for function addition.At the point at which the function is called from within main, the control is lost by main and passed to function addition. The value of both arguments passed in the call (5 and 3) are copied to the local variables int a and int b within the function. Function addition declares another local variable (int r), and by means of the expression r=a+b, it assigns to r the result of a plus b. Because the actual parameters passed for a and b are 8 and 4 respectively, the result is 12.

Finalizes function addition, and returns the control back to the function that called it in the first place (in this case, main). At this moment the program follows it regular course from the same point at which it was interrupted by the call to addition. But additionally, because the return statement in function addition specified a value: the content of variable r (return (r);), which at that moment had a value of 12. This value becomes the value of evaluating the function call.

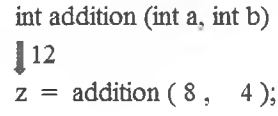


Figure 1.15

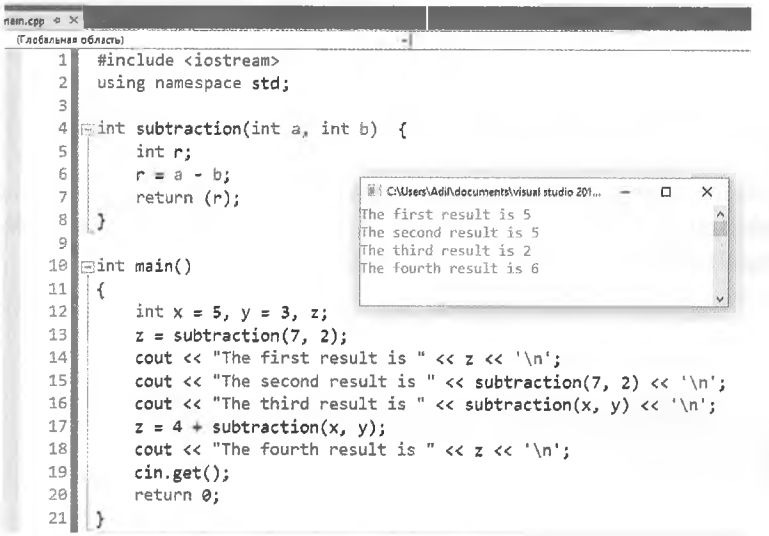
So being the value returned by a function the value given to the function call itself when it is evaluated, the variable z will be set to the value returned by addition (8, 4), that is 12. To explain it another way, you can imagine that the call to a function (addition (8,4)) is literally replaced by the value it returns (12).

The following line of code in main is:

```
23        cout << "The result is " << z;
```

Listing 1.34

That, as you may already expect, produces the printing of the result on the screen.



Listing 1.35

In this case we have created a function called subtraction. The only thing that this function does is to subtract both passed parameters and to return the result.

Nevertheless, if we examine function main we will see that we have made several calls to function subtraction. We have used some different calling methods so that you see other ways or moments when a function can be called.

In order to fully understand these examples you must consider once again that a call to a function could be replaced by the value that the function call itself is going to return. For example, the first case (that you should already know because it is the same pattern that we have used in previous examples):

```
14    z = subtraction (7,2);
15    cout << "The first result is " << z << '\n';
```

If we replace the function call by the value it returns (i.e., 5), we would have:

```
14    z = 5;
15  cout << "The first result is " << z;
```

As well as:

```
18    cout << "The second result is " << subtraction (7,2) << '\n';
```

has the same result as the previous call, but in this case we made the call to subtraction directly as an insertion parameter for cout. Simply consider that the result is the same as if we had written:

```
18    cout << "The second result is " << 5;
```

since 5 is the value returned by subtraction (7,2).
In the case of:

```
20    cout << "The third result is " << subtraction (x,y);
```

The only new thing that we introduced is that the parameters of subtraction are variables instead of constants. That is perfectly valid. In this case the values passed to function subtraction are the values of x and y, that are 5 and 3 respectively, giving 2 as result.

The fourth case is more of the same. Simply note that instead of:

```
22    z= 4 + subtraction (x,y);
```

we could have written:

```
22    z = subtraction (x,y)+ 4;
```

with exactly the same result. I have switched places so you can see that the semicolon sign (;) goes at the end of the whole statement. It does not necessarily have to go right after the function call. The explanation might be once again that you imagine that a function can be replaced by its returned value.

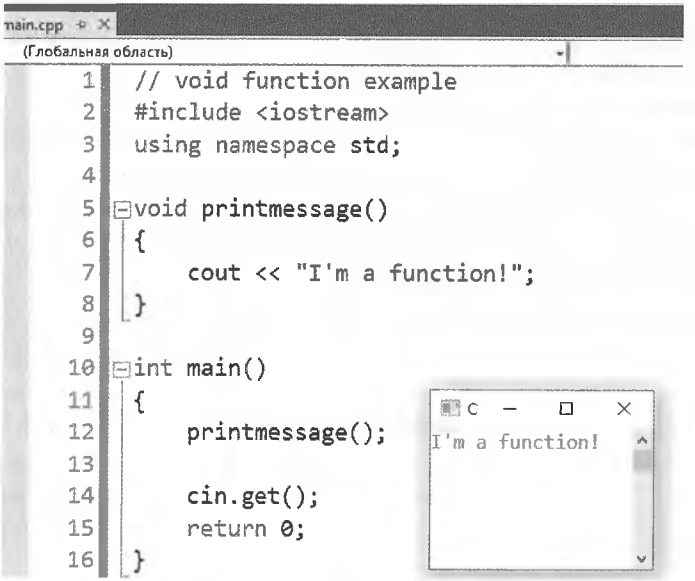
Functions with no type. The use of void.

If you remember the syntax of a function declaration:

type name (argument1, argument2 ...) statement (1.6)

you will see that the declaration begins with a type, that is the type of the function itself (i.e., the type of the datum that will be returned by the function with the return statement). But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the void type specifier for the function. This is a special specifier that indicates absence of type.



Listing 1.36

void can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function printmessage could have been declared as:

```
4  void printmessage ()
5  {
6    cout << "I'm a function!";
7  }
```

Although it is optional to specify void in the parameter list. In C++, a parameter list can simply be left blank if we want a function with no parameters. What you must always remember is that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to *printmessage* is:

```
11   printmessage ();
```

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement.

Arguments passed by value and by reference.

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function addition (Listing 1.37).

What we did in this case was to call to function addition passing the values of x and y, i.e. 8 and 4 respectively, but not the variables x and y themselves.

```
int addition (int a, int b)
                  ↑    ↑
z = addition ( 8 ,   4 );
```

Figure 1.16

This way, when the function addition is called, the value of its local variables a and b become 8 and 4 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use ar-

guments passed by reference, as in the function duplicate of the following example:

```
ain.cpp
(Глобальная область)
1   #include <iostream>
2   using namespace std;
3
4   void duplicate(int& a, int& b, int& c)
5   {
6       a *= 2;
7       b *= 2;
8       c *= 2;
9   }
10  int main()
11  {
12      int x = 1, y = 3, z = 7;
13      duplicate(x, y, z);
14      cout << "x=" << x << ", y=" << y << ", z=" << z;
15      cin.get();
16      return 0;
17  }
18
```

```
C:\Users\Adi...
x=2, y=6, z=14
```

Listing 1.37

The first thing that should call your attention is that in the declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.

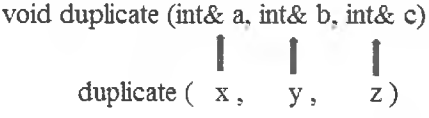
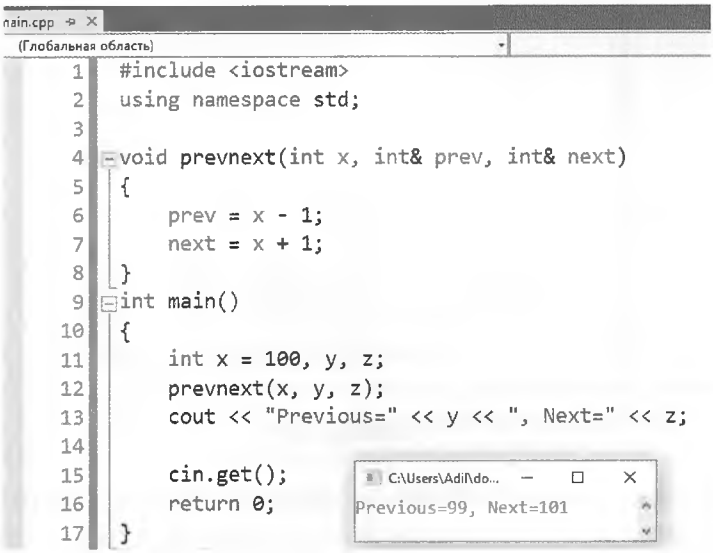


Figure 1.17

To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within

the function will affect the value of x outside it. Any change that we do on b will affect y, and the same with c and z. That is why our program's output, that shows the values stored in x, y and z after the call to duplicate, shows the values of all the three variables of main doubled. If when declaring the following function:

void duplicate (int a, int b, int c)



Listing 1.38

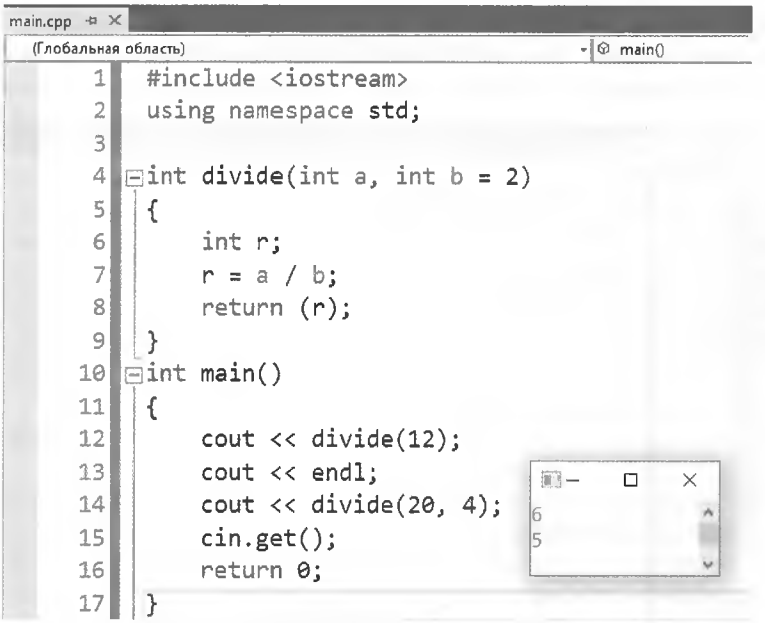
i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and z without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

Default values in parameters.

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default

value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:



Listing 1.39

As we can see in the body of the program there are two calls to function divide. In the first one:

```
13    cout << divide (12);
```

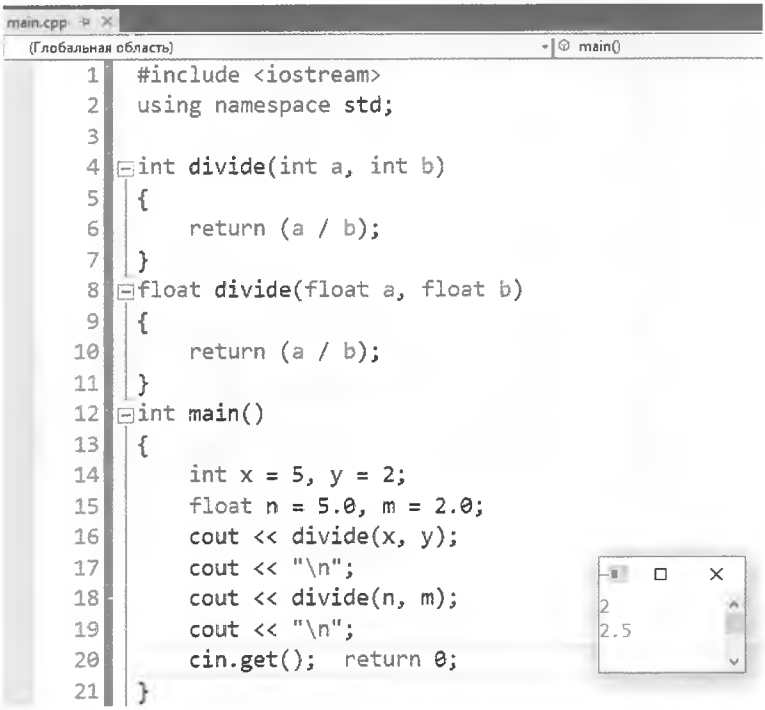
we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with int b=2, not just int b). Therefore the result of this function call is 6 (12/2). In the second call:

```
15    cout << divide (20,4);
```

there are two parameters, so the default value for b (int b=2) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4).

Overloaded functions.

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:



Listing 1.40

In this case we have defined two functions with the same name, operate, but one of them accepts two parameters of type *int* and the other one accepts them of type float. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two *ints* as its arguments it calls to the function that has two *int* parameters in its prototype and if it is called with two floats it will call to the one which has two float parameters in its prototype.

In the first call to operate the two arguments passed are of type *int*, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type float, so the function with the second prototype is called.

This one has a different behavior: it divides one parameter by the other. So the behavior of a call to operate depends on the type of the arguments passed because the function has been *overloaded*.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

Inline functions.

The inline specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

inline type name (arguments ...) { instructions ... } (1.7)

and the call is just like the call to any other function. You do not have to include the inline keyword when calling the function, only in its declaration. Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

Declaring functions.

Until now, we have defined all of the functions before the first appearance of calls to them in the source code.

But there is an alternative way to avoid writing the whole code of a function before it can be used in main or in some other function. This can be achieved by declaring just a prototype of the function before it is used, instead of the entire definition. This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters.

Its form is:

type name (argument_type1, argument_type2, ...); (1.8)

It is identical to a function definition, except that it does not include the body of the function itself (i.e., the function statements that in normal definitions are enclosed in braces { }) and instead of that we end the prototype declaration with a mandatory semicolon (;).

The parameter enumeration does not need to include the identifiers, but

only the type specifiers. The inclusion of a name for each parameter as in the function definition is optional in the prototype declaration. For example, we can declare a function called protofunction with two *int* parameters with any of the following declarations:

```
14 int protofunction (int first, int second);
15 int protofunction (int, int);
```

Listing 1.41

Anyway, including a name for each variable makes the prototype more legible.

main.cpp

(Глобальная область)

```
1  #include <iostream>
2  using namespace std;
3
4  void odd(int a);
5  void even(int a);
6  int main()
7  {
8      int i;
9      do {
10         cout << "Type a number: (0 to exit)";
11         cin >> i;
12         odd(i);
13     } while (i != 0);
14     using namespace std;
15     return 0;
16 }
17 void odd(int a)
18 {
19     if ((a % 2) != 0) cout << "Number is odd.\n";
20     else even(a);
21 }
22 void even(int a)
23 {
24     if ((a % 2) == 0) cout << "Number is even.\n";
25     else odd(a);
26 }
27
```

C:\Users\Adi\docu...

Type a number: (0 to exit)

Listing 1.42

This example is indeed not an example of efficiency. I am sure that at this point you can already make a program with the same result,

but using only half of the code lines that have been used in this example. Anyway this example illustrates how prototyping works. Moreover, in this concrete example the prototyping of at least one of the two functions is necessary in order to compile the code without errors. The first things that we see are the declaration of functions odd and even:

```
4  void odd (int a);
5  void even (int a);
```

This allows these functions to be used before they are defined, for example, in main, which now is located where some people find it to be a more logical place for the start of a program: the beginning of the source code.

Anyway, the reason why this program needs at least one of the functions to be declared before it is defined is because in odd there is a call to even and in even there is a call to odd. If none of the two functions had been previously declared, a compilation error would happen, since either odd would not be visible from even (because it has still not been declared), or even would not be visible from odd (for the same reason).

Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by declaring all functions prototypes at the beginning of a program.

1.5 Abstract Data Types, Classes

Data Structures

We have already learned how groups of sequential data can be used in C++. But this is somewhat restrictive, since in many occasions what we want to store are not mere sequences of elements all of the same data type, but sets of different elements with different data types.

A data structure is a group of data elements grouped together under one name. These data elements, known as *members*, can have different types and different lengths. Data structures are declared in C++ using the following syntax:

```
struct structure_name {
member_type1
member_name1;
member_type2
member_name2;
```

```
member_type3
member_name3;(1.9)
.
.
} object_names;
```

where structure_name is a name for the structure type, object_name can be a set of valid identifiers for objects that have the type of this structure. Within braces { } there is a list with the data members, each one is specified with a type and a valid identifier as its name.

The first thing we have to know is that a data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as structure_name is created and can be used in the rest of the program as if it was any other type. For example:

```
7  struct vegetables {
8      int weight;
9      float price;
10 } ;
11
12 vegetables potato;
13 vegetables tomatoes, carrots;
```

Listing 1.43

We have first declared a structure type called vegetables with two members: weight and price, each of a different fundamental type. We have then used this name of the structure type (vegetables) to declare three objects of that type: potato, tomatoes and carrots, as we would have done with any fundamental data type.

Once declared, vegetables has become a new valid type name like the fundamental ones int, char or short and from that point on we are able to declare objects (variables) of this compound new type, like we have done with potato, tomatoes and carrots.

Right at the end of the struct declaration, and before the ending semicolon, we can use the optional field object_name to directly declare objects of the structure type. For example, we can also declare the structure objects apple, banana and melon at the moment we define the data structure type this way:

```
7  struct vegetables {
8      int weight;
9      float price;
10 } potato, tomatoes, carrots;
```

Listing 1.44

It is important to clearly differentiate between what is the structure type name, and what is an object (variable) that has this structure type. We can instantiate many objects (i.e. variables, like apple, banana and melon) from a single structure type (product).

Once we have declared our three objects of a determined structure type (apple, banana and melon) we can operate directly with their members. To do that we use a dot (.) inserted between the object name and the member name. For example, we could operate with any of these elements as if they were standard variables of their respective types:

```
16 potato.weight;
17 tomatoes.weight;
18 carrots.weight;
19 potato.price;
20 tomatoes.price;
21 carrots.price;
```

Listing 1.45

Each one of these has the data type corresponding to the member they refer to: *potato.weight, tomatoes.weight*and *carrots.weightare* of type *int*, while *potato.price*, *tomatoes.price* and *carrots.price*are of type float.

Let's see a real example where you can see how a structure type can be used in the same way as fundamental types:

The example shows how we can use the members of an object as regular variables. For example, the member *yours.year* is a valid variable of type *int*, and *mine.title* is a valid variable of type string.

```
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

struct movies_t {
  string title;
  int year;
} mine, yours;

void printmovie (movies_t movie);

int main ()
{
  string mystr;

      mine.title = "Fifty Shades of Grey";
      mine.year = 2015;

          cout << "Enter title: ";
      getline (cin,yours.title);
         cout << "Enter year: ";
      getline (cin,mystr);
          stringstream(mystr) >> yours.year;

         cout << "My favorite movie is:\n ";
      printmovie (mine);
          cout << "And yours is:\n ";
      printmovie (yours);
  cin.get();  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

```
c:\users\adil\documents\vis
Enter title: Vikings
Enter year: 2013
My favorite movie is:
 Fifty Shades of Grey (2015)
And yours is:
 Vikings (2013)
```

Listing 1.46

Data structures features that can be used to represent databases, especially if we consider the possibility of building arrays of them:

he objects mine and yours can also be treated as valid variables of type *movies_t*, for example, we have passed them to the function *printmovie* as we would have done with regular variables. Therefore, one of the most important advantages of data structures is that we can either refer to their members individually or to the entire structure as a block with only one identifier.

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

#define N_MOVIES 3

struct movies_t {
      string title;
      int year;
} films [N_MOVIES];

void printmovie (movies_t movie);

int main ()
{
  string mystr;
  int n;

      for (n=0; n<N_MOVIES; n++)
  {
            cout << "Enter title: ";
                  getline (cin,films[n].title);
            cout << "Enter year: ";
                  getline (cin,mystr);
    stringstream(mystr) >> films[n].year;
  }

  cout << "\nYou have entered these movies:\n";
  for (n=0; n<N_MOVIES; n++)
    printmovie (films[n]);
  cin.get(); return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

```
c:\users\adi\documents\visual studio
Enter title: Passengers
Enter year: 2015
Enter title: Doctor Strange
Enter year: 2016
Enter title: Office Christmas Party
Enter year: 2016

You have entered these movies:
Passengers (2015)
Doctor Strange (2016)
Office Christmas Party (2016)
```

Listing 1.47

Pointers to structures

Like any other type, structures can be pointed by its own type of pointers:

```
8  struct movies_t {
9    string title;
10   int year;
11 };
12
13 movies_t amovie;
14 movies_t * pmovie;
15
```

Listing 1.48

Here *amovie* is an object of structure type *movies_t*, and *pmovie* is a pointer to point to objects of structure type *movies_t*. So, the following code would also be valid:

```
45   pmovie = &amovie;
```

Listing 1.49

The value of the pointer *pmovie* would be assigned the reference to the object *amovie* (its memory address).

We will now go with another example that includes pointers, which will serve to introduce a new operator: the arrow operator -> (Listing 1.50).

This is a dereference operator that is used exclusively with pointers to objects with members. This operator serves to access a member of an object to which we have a reference. In the example we used:

```
20       pmovie->title
21
```

Listing 1.50

This is for all purposes equivalent to:

```
22       (*pmovie).title
```

Listing 1.51

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
};

int main ()
{
  string mystr;

  movies_t amovie;
  movies_t * pmovie;
      pmovie = &amovie;

      cout << "Enter title: ";
          getline (cin, pmovie->title);
      cout << "Enter year: ";
          getline (cin, mystr);
  (stringstream) mystr >> pmovie->year;

      cout << "\nYou have entered:\n";
      cout << pmovie->title;
      cout << " (" << pmovie->year << ")\n";

  cin.get(); return 0;
}
```

```
c:\users\adil\documents\visual studio 2012\P
Enter title: Beaut; and the Beast
Enter year: 2017

You have entered:
Beauty and the Beast  (2017)
```

Listing 1.52

And that would access the value pointed by a hypothetical pointer member called title of the structure object *pmovie* (which in this case would not be a pointer). The following panel summarizes possible combinations of pointers and structure members:

Table 1.2

Expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed by a	(*a).b
*a.b	Value pointed by member b of object a	*(a.b)

Classes, basic Syntax.

A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions. An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable. Classes are generally declared using the keyword class, with the following format:

```
class class_name {
access_specifier_1:
member1;
access_specifier_2:                                    (1.10)
member2;
...
} object_names;
```

Where *class_name* is a valid identifier for the class, *object_names* is an optional list of names for objects of this class. The body of the declaration can contain members, that can be both data or function declarations, and optionally access specifiers.

All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called *access specifier*. An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights that the members following them acquire:

- private members of a class are accessible only from within other members of the same class or from their *friends*.
- protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier (Listing 1.53).

Access modifiers in C++ do not repeat for each data member. All data members within that section share the access level of the delimiting modifier. In the above listing, everything defined below line 8 and up until the next access modifier in line 12 has private access.

Another key difference in class specification is that C++ class declarations must end with a semicolon. This semicolon appears in line 13 of the above example.

```
6  class BankAccount {
7
8  private:
9      double sum;
10     string name;
11
12 public:
13     BankAccount(string nm) : name(nm), sum(0) {}
14
15 } ballance ;
16
```

Listing 1.53. A C++ BankAccount class

Constructors.

Constructors are the methods of a class that define what actions to take when creating an object. A C++ class can have multiple constructors. This allows variation in object instantiation since different numbers and types of parameters can exist in each constructor. The following Listing is a modified version of the C++ BankAccount class. This modified version includes an additional constructor.

```
6  class BankAccount {
7
8  private:
9      double sum;
10     string name;
11
12 public:
13     BankAccount(string nm) : name(nm), sum(0) {}
14     BankAccount(string nm, double bal) :
15     name(nm), sum(bal) {}
16
17     double balance() { return sum;}
18     void deposit(double amount) {sum += amount;}
19     void withdraw(double amount) {sum -= amount;}
20     string getName() { return name;}
21 };
```

Listing 1.54. Initializer lists and multiple constructors

The use of *initializer lists* in constructors is the preferred way to specify initial values for class data members. Initializer lists are comma-separated variable initializations that appear prior to the body of a constructor. An exam-

ple of initializer lists appears in Listing1.55. Everything in line 13 following the colon and preceding the empty curly-braces comprises the initializer list. This initializer list sets the initial value of the private data member `name` equal to the value of parameter `nm`. It also sets the initial value of the data member `sum` equal to zero.

```
6  class BankAccount {
7
8  private:
9      double sum;
10     string name;
11
12 public:
13     BankAccount(string nm) : name(nm), sum(0) {}
14     BankAccount(string nm, double bal) :
15     name(nm), sum(bal) {}
16
17     double balance() { return sum;}
18     void deposit(double amount) {sum += amount;}
19     void withdraw(double amount) {sum -= amount;}
20     string getName() { return name;}
21 };
22
23 int main ()
24 {
25     BankAccount account1("checking");
26     BankAccount account2("savings", 200);
27
28     account2.withdraw(100);
29     account1.deposit(100);
```

Listing 1.55. Object instantiation

C++ instantiate an object when the line of code containing the object declaration executes. Object instantiation involves the execution of a class constructor. Listing 1.55 declares two different *BankAccount* objects. Instantiation occurs when the code contained in lines 1 and 2 executes.

Destructor

A *destructor* is a special member function of a C++ class called when an object's lifetime ends. Like a copy constructor, only one destructor can exist for a class. Since they execute when an object's lifetime ends, destructors typically define the actions necessary to release any resources that an object may be using. For example, consider an object that opens a connection to a

database. Until then, we can look at a Listing to see at least what the definition of a destructor looks like in C++.

The difference between the definition of a destructor and a constructor is very subtle.

```
22  ~BankAccount() {
23  if (balance() < 0) {
24      cout << "Warning: negative balance!" << endl;
25  }
26  }
```

Listing 1.56. The destructor

Notice in line 1 of Listing 5 that a tilde (~) exists in front of the name of the class. This signifies that this member function is the destructor for the class.

Declaration vs. Definition

In this discussion on the specification of classes in C++, the term "definition" has been used regarding functions. When we "define" a function, we dictate the function's behavior through the code that exists within the curly braces. The "declaration" of a function, on the other hand, only specifies the interface of the function. This interface includes the function name, the return type, and the list of parameters and their types. The following Listing 1.57 shows both a declaration and definition of the function `average`.

The declaration and definition of a function differ in many aspects. In the above example, notice the declaration of function `average` is followed by a semi-colon and not by curly braces. This specifies that it is the declaration, not the definition. A function declaration does not need to include variable names for the parameters. The definition, on the other hand, looks just the same as any C++ function we have seen so far. It is interesting to note that the definition of the function comes after its actual use in function *main*. This is acceptable since we placed a declaration of the function prior to function *main*. This practice, known as a *forward reference*, is common in C++ programming. It is legal to use a function or a class that is not yet defined, as long as it has been declared. It is illegal, however, to use a function or a class that is not yet defined without including the forward reference.

```
#include <iostream>
#include <cstdlib>

using namespace std;

// function declaration
double average(int, int);

int main(int argc, char* argv[]) {

    cout << average(10, 2) << endl;
    return EXIT_SUCCESS;
}

// function definition
double average(int total, int count) {
    return (total / count);
}
```

Listing 1.57. Declaration vs. Definition

Programmers must fully qualify the names of class member functions that appear outside of a class definition. To qualify a name fully, the function name must be prepended with the class name followed by the *scope resolution operator* (`::`). Listing1.58 defines class member functions outside the class definition.

The rules regarding class specification in C++ offer some flexibility in the placement of member function definitions. A C++ class can include all its member function definitions inside of the class definition. This is the approach taken in the examples we have seen so far. Alternatively, the definitions for one or more class member functions can appear outside the class definition, as long as we still include the declarations inside the class definition.

database. Until then, we can look at a Listing to see at least what the definition of a destructor looks like in C++.

The difference between the definition of a destructor and a constructor is very subtle.

```
22      ~BankAccount() {
23      if (balance() < 0) {
24          cout << "Warning: negative balance!" << endl;
25      }
26  }
```

Listing 1.56. The destructor

Notice in line 1 of Listing 5 that a tilde (~) exists in front of the name of the class. This signifies that this member function is the destructor for the class.

Declaration vs. Definition

In this discussion on the specification of classes in C++, the term "definition" has been used regarding functions. When we "define" a function, we dictate the function's behavior through the code that exists within the curly braces. The "declaration" of a function, on the other hand, only specifies the interface of the function. This interface includes the function name, the return type, and the list of parameters and their types. The following Listing 1.57 shows both a declaration and definition of the function `average`.

The declaration and definition of a function differ in many aspects. In the above example, notice the declaration of function `average` is followed by a semi-colon and not by curly braces. This specifies that it is the declaration, not the definition. A function declaration does not need to include variable names for the parameters. The definition, on the other hand, looks just the same as any C++ function we have seen so far. It is interesting to note that the definition of the function comes after its actual use in function *main*. This is acceptable since we placed a declaration of the function prior to function *main*. This practice, known as a *forward reference*, is common in C++ programming. It is legal to use a function or a class that is not yet defined, as long as it has been declared. It is illegal, however, to use a function or a class that is not yet defined without including the forward reference.

```
#include <iostream>
#include <cstdlib>

using namespace std;

// function declaration
double average(int, int);

int main(int argc, char* argv[]) {

    cout << average(10, 2) << endl;
    return EXIT_SUCCESS;
}

// function definition
double average(int total, int count) {
    return (total / count);
}
```

Listing 1.57. Declaration vs. Definition

Programmers must fully qualify the names of class member functions that appear outside of a class definition. To qualify a name fully, the function name must be prepended with the class name followed by the *scope resolution operator* (`::`). Listing1.58 defines class member functions outside the class definition.

The rules regarding class specification in C++ offer some flexibility in the placement of member function definitions. A C++ class can include all its member function definitions inside of the class definition. This is the approach taken in the examples we have seen so far. Alternatively, the definitions for one or more class member functions can appear outside the class definition, as long as we still include the declarations inside the class definition.

The preprocessor directive *#include* replace the directive with the contents of the specified file. The directives *#define*, *#ifndef* and *#endif* are used to check if a definition file is not included more than once.

Copy Constructors

A *copy constructor* defines the actions that need to be taken to create a copy of an object. Unlike regular constructors, a class can contain only one copy constructor. If a C++ class does not define a copy constructor, the language provides to the class a default copy constructor. This default copy constructor makes a byte-by-byte copy of the object.

Copy constructors are invoked whenever a copy of an object has to be made. There are three situations when copies of objects are made.

1. During declaration that involves initialization
2. When objects are passed by value
3. When objects are returned by value

Using the default copy constructor provided by the runtime system can be dangerous in a program that uses dynamic memory. Situations can arise where two or more objects incorrectly maintain pointers to the same data. Consider the following Listing.

The intent of function main in the above Listing 1.61 is to create two objects of type Array. Each of these objects is to have ten elements. The ten elements in the second object (arr2) should contain copies of the values of the ten elements from the first object (arr1). This copy takes place when arr1 is assigned to arr1.

```
#include <iostream>
#include <cstdlib>

using namespace std;

class Array {

private:
    int *ptr;

public:
    Array(void) : ptr(new int[10]) {}
    ~Array(void) {delete ptr;}
    void display_ptr(void) {cout << ptr << endl;}
};

int main(int argc, char* argv[]) {

    // Create two Array objects. The
    // second should be a copy of the
    // data of the first.
    Array arr1;
    Array arr2 = arr1;

    arr1.display_ptr();
    arr2.display_ptr();

    return EXIT_SUCCESS;
}
```

Listing 1.61. Need for a copy constructor

In reality, only a copy of the pointer variable *ptr* is made during this assignment. Since class Array does not define a copy constructor, a byte-by-byte copy of the object is made. This byte-by-byte copy of the object results in each Array object having the same value for *ptr*. This is not a true copy of the object since the elements in the array, since they exist in the heap, are not copied at all. This type of copy is known as a *shallow copy*. The opposite of a shallow copy is a *deep copy*.

A copy constructor can be used to ensure the creation of a deep copy of an object. The above Listing contains a copy constructor for class `Array`. This copy constructor allocates a new array in memory and copies into it the values from the source array.

```
#include <iostream>
#include <cstdlib>

using namespace std;

class Array {
private:
    int *ptr;

public:
    Array(void) : ptr(new int[10]) {}
    Array(const Array& src) {

        cout << "Copy Constructor Invoked!\n";

        ptr = new int[10];
        for (int i = 0; i < 10; i++) {
            ptr[i] = src.ptr[i];
        }
    }
    ~Array(void) {delete ptr;}
    void display_ptr(void) {cout << ptr << endl;}
};

int main(int argc, char* argv[]) {
    // Create two Array objects. The
    // second should be a copy of the
    // data of the first.
    Array arr1;
    Array arr2 = arr1;
        arr1.display_ptr();
        arr2.display_ptr();

    return EXIT_SUCCESS;
}
```

Listing 1.62. Class Array copy constructor

1.6 Parameter passing mechanisms, Dynamic memory management

Pass by value is the default parameter passing mechanism in C++. Just like other programming languish, when a parameter is passed by value to a function, a copy of the parameter is created and given to the function. This is important, since if we make a change to a parameter that is passed by value, the original variable will remain unchanged. Our change is made to a copy of the original variable. Listing 1.63 illustrates this point.

```
#include <iostream>
#include <cstdlib>

using namespace std;

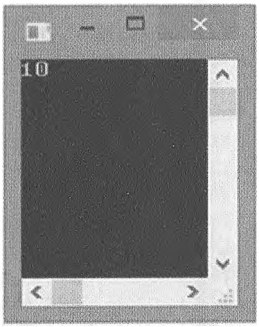
void increment(int x) {
    x++;  // Increment x by 1
}

int main(int argc, char* argv[]) {

    int y = 10;
    increment(y);

    // Variable y remains unchanged.
    cout << y << endl;

    return EXIT_SUCCESS;
}
```



Listing 1.63. Pass by value in C++

Since parameter to function *increment* in Listing1.63 is passed by value, a copy of variable `y` is created and given to the function. This copy is incremented to the value of `11`. Once function *increment*returns, the lifetime of the variable (`x`) that now stores the value of `11` ends. The variable `y` remains unchanged and the program outputs the value `10`.

C++ can also pass objects by value. The following Listing defines a simple class and passes an object of that class by value.

The *main* routine in Listing1.64 passes an instance of class *Person* by value to the function *increment_age*. Inspecting the output of the example yields the result we expect to see. The code in function *increment_ age* does not change the state of the object `person` in function `main`, since the object was passed by value.

Pass by Reference

C++ also supports the *pass by reference* parameter passing mechanism. Unlike pass by value, copies are not made of variables that are passed by reference. Instead, a called function receives a reference, or alias, to the actual parameter supplied by the calling function. For this reason, pass by reference is used to build functions that can modify the variables in the calling function. Even when a function does not need to modify the variables in the calling function, pass by reference is sometimes used to avoid the overhead of pass by value.

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <cassert>

using namespace std;

class Person {
private:
    string name;
    int age;

public:
    Person() : name(""), age(0) {}

    void set_age(int age) { this->age = age; }
    int get_age() { return age; }
    void set_name(string name) { this->name = name; }
    string get_name() { return name; }
};

void increment_age(Person p) {
    p.set_age(p.get_age() + 1);
}

int main(int argc, char* argv[]) {

    Person person;
    person.set_name("Adi Seiko");
    person.set_age(50);

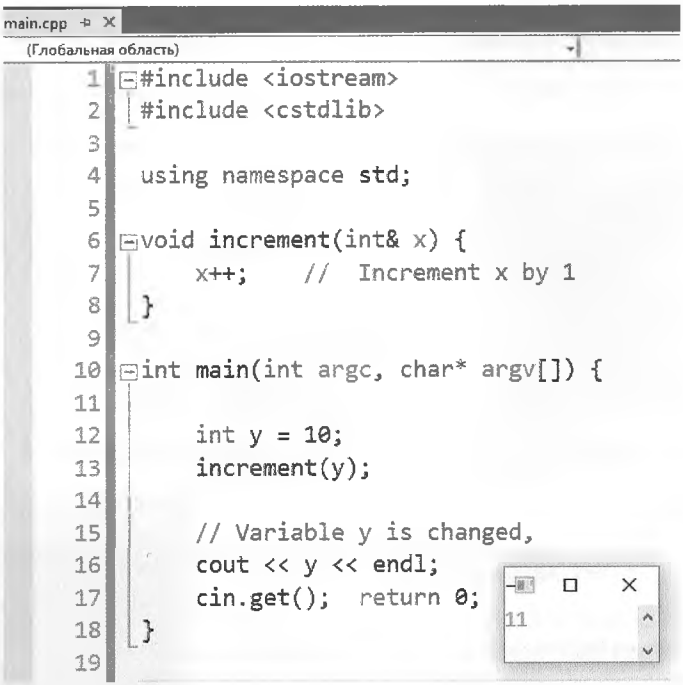
    increment_age(person);
    // age remains unchanged
    cout << person.get_age() << endl;

    return EXIT_SUCCESS;
}
```



Listing 1.64. Passing an object by value

One common use of pass by reference is to create functions that can modify the variables passed to them by a calling function.



Listing 1.65. Pass by reference

Listing 1.65 is the same program presented in Listing 1.63. The only difference is in line 6 where the parameter `x` is declared as a reference parameter using the syntax *int& x*. Do not confuse this use of the ampersand (&) with the address-of operator. Here the ampersand signals to the compiler that this parameter is to be passed by reference. Since it is passed by reference, the increment operation in line 7 affects the original variable `y` found in *main*. Therefore, this program outputs the value `11`.

Passing a parameter by reference is also used as a mechanism to pass large objects to functions. When objects are large, pass by value can result in time-consuming copy operations. Pass by reference is more efficient because it does not involve copying. Even when a function does not intend to modify one of its parameters, pass by reference should be used when the parameter is a large object. We should declare parameters passed by reference that a function should not modify as constants. This is a good practice since it provides protection against accidental modification. Listing 1.66 demonstrates how to pass an object by reference while still preserving the safety of pass by value.

```
#include <iostream>
#include <string>
#include <cstdlib>

using namespace std;

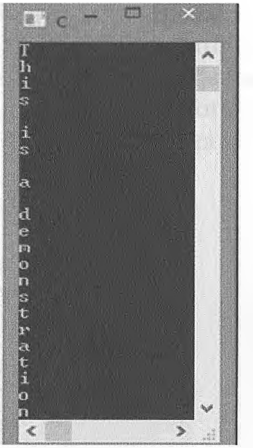
void display_letters(const string& data) {

    for (int i = 0; i < data.length(); i++) {
        cout << data[i] << "\n";
    }
}

int main(int argc, char* argv[]) {

    string s = "This is a demonstration";
    display_letters(s);

    cin.get();
    return EXIT_SUCCESS;
}
```



Listing 1.66. A constant reference

It is sometimes useful to pass a pointer by reference. This is done when a function needs to change the pointer's stored memory address. In other words, a pointer is passed to a function by reference when the function needs to reposition the pointer.

```
#include <iostream>
#include <cstdlib>

using namespace std;

void find_first_greater_than(int*& ptr, int threshold) {

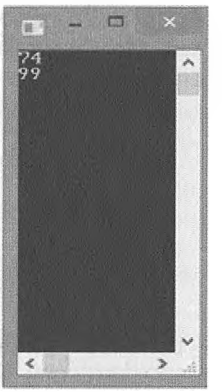
    while (*ptr <= threshold) {
        ptr++;
    }
}

int main(int argc, char* argv[]) {

    int exam_scores[] = { 74, 94, 64, 77, 68,
        99, 58, 89, 74, 88,
        100, 95, 71, 81, 89,
        54, 76, 83, 88, 67 };

    int* score = &exam_scores[0];

    cout << *score << endl;
    find_first_greater_than(score, 98);
    cout << *score << endl;
    cin.get();
    return EXIT_SUCCESS;
}
```



Listing 1.67. Passing a pointer by reference

Pass by value, by default, is also the mechanism used in C++ to return values from functions. It is often useful, however, to return by reference to allow the calling function to modify some stored variable. Listing 1.68 demonstrates how to return by reference. This Listing adds a *replace_letter* function to class Person.

```
class Person {
private:
    string name;
    int age;

public:
    Person() : name(""), age(0) {}

    void set_age(int age) {this->age = age;}
    int get_age() { return age;}
    void set_name(string name) {this->name = name;}
    string get_name() { return name;}

    char &replace_letter(int i) {
        return name[i];
    }
};
```

Listing 1.68. Return by reference

In the above version, notice that the return type of *replace_letter* is a reference to type char. This allows the following use of this function.

```
    Person person;
person.set_name("Adi Sei");
person.set_age(30);

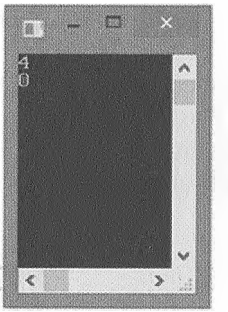
person.replace_letter(1) = 'a';
person.replace_letter(2) = 'n';
person.replace_letter(3) = 'e';

    cout << person.get_name() << endl;
```

Listing 1.69. Using a return by reference function

A common mistake made when returning by reference is returning a reference to a local variable. The lifetime of a local variable ends when the function call that it exists in completes. Returning a reference to something that no longer exists can lead to unexpected results.

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  class my_class {
7  private:
8      int x;
9  public:
10     my_class() : x(0) {}
11     my_class(int p) : x(p) {}
12     int value() { return x; }
13 };
14
15 int main(int argc, char* argv[]) {
16
17     // Allocate a single object
18     my_class* ptr1 = new my_class(4);
19
20     // Allocate an array of objects
21     my_class* ptr2 = new my_class[10];
22
23     cout << ptr1->value() << endl;
24     cout << ptr2->value() << endl;
25
26     cin.get();
27     return EXIT_SUCCESS;
28 }
```



Listing 1.74. Allocating objects

Objects can also be dynamically allocated. The new operator, in addition to allocating the memory for an object, will call a constructor for the object. Listing 1.74 shows objects allocated using new.

Listing 1.74 defines a simple class and illustrates allocation of both a single instance and an array of objects of the class. In line 18, the single instance is allocated. After allocating memory for the object, the `new` operator invokes the single parameter constructor. In line 21, the *new* operator allocates an array of ten elements of type *my_class*. In this case, the default constructor is called for each of the ten objects in the array.

Memory Deallocation

The *delete* operator *deallocates* memory that is allocated using the *new* operator.

To release, or deallocate, memory, the delete operator needs to know what location in memory we want to deallocate. To this end, we supply it with a pointer, which is really just the address that we obtained from the new operator.

```
33  // Dynamically allocate a variable.
34  double* ptr1 = new double;
35
36  // ... use the variable ...
37
38  // The variable is no longer needed,
39  // so we return its memory to the Free Store.
40  delete ptr1;
```

Listing 1.75. The Delete operator

A special syntax exists for the delete operator for use in deallocating arrays. The keyword delete is followed by the double bracket ([]) operator. This signals to the run-time environment that what delete needs to deallocate is actually an array, and not just a variable.

```
33  // Allocate two arrays.
34  int* ptr1 = new int[100];
35
36  int n = 0;
37  cin >> n;
38  int* ptr2 = new int[n];
39
40  // When they are no longer needed,
41  // we deallocate them.
42  delete [] ptr1;
43  delete [] ptr2;
```

Listing 1.76. Deallocating arrays

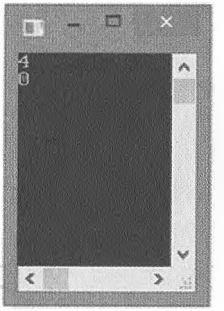
It might be surprising that the size of the array does not have to be specified when using *delete[]*. This is not necessary since the run-time environment automatically maintains the size of allocated arrays.

We have already seen that the `new` operator invokes a constructor when used to allocate objects. Similarly, when deallocating an object the *delete* operator calls the object's destructor.

Some Common Pitfalls

The basic rule for dealing with dynamic allocation and deallocation is straightforward: for each call to `new` (which consumes memory), there must be a corresponding call to *delete* (which releases the memory). Moreover, if at

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  class my_class {
7  private:
8      int x;
9  public:
10     my_class() : x(0) {}
11     my_class(int p) : x(p) {}
12     int value() { return x; }
13 };
14
15 int main(int argc, char* argv[]) {
16
17     // Allocate a single object
18     my_class* ptr1 = new my_class(4);
19
20     // Allocate an array of objects
21     my_class* ptr2 = new my_class[10];
22
23     cout << ptr1->value() << endl;
24     cout << ptr2->value() << endl;
25
26     cin.get();
27     return EXIT_SUCCESS;
28 }
```



Listing 1.74. Allocating objects

Objects can also be dynamically allocated. The new operator, in addition to allocating the memory for an object, will call a constructor for the object. Listing 1.74 shows objects allocated using new.

Listing 1.74 defines a simple class and illustrates allocation of both a single instance and an array of objects of the class. In line 18, the single instance is allocated. After allocating memory for the object, the `new` operator invokes the single parameter constructor. In line 21, the *new* operator allocates an array of ten elements of type *my_class*. In this case, the default constructor is called for each of the ten objects in the array.

Memory Deallocation

The *delete* operator *deallocates* memory that is allocated using the *new* operator.

To release, or deallocate, memory, the delete operator needs to know what location in memory we want to deallocate. To this end, we supply it with a pointer, which is really just the address that we obtained from the new operator.

```
33  // Dynamically allocate a variable.
34  double* ptr1 = new double;
35
36  // ... use the variable ...
37
38  // The variable is no longer needed,
39  // so we return its memory to the Free Store.
40  delete ptr1;
```

Listing 1.75. The Delete operator

A special syntax exists for the delete operator for use in deallocating arrays. The keyword delete is followed by the double bracket ([]) operator. This signals to the run-time environment that what delete needs to deallocate is actually an array, and not just a variable.

```
33  // Allocate two arrays.
34  int* ptr1 = new int[100];
35
36  int n = 0;
37  cin >> n;
38  int* ptr2 = new int[n];
39
40  // When they are no longer needed,
41  // we deallocate them.
42  delete [] ptr1;
43  delete [] ptr2;
```

Listing 1.76. Deallocating arrays

It might be surprising that the size of the array does not have to be specified when using *delete [ ]*. This is not necessary since the run-time environment automatically maintains the size of allocated arrays.

We have already seen that the `new` operator invokes a constructor when used to allocate objects. Similarly, when deallocating an object the *delete* operator calls the object's destructor.

Some Common Pitfalls

The basic rule for dealing with dynamic allocation and deallocation is straightforward: for each call to `new` (which consumes memory), there must be a corresponding call to *delete* (which releases the memory). Moreover, if at

all possible, the matching calls should be close to each other, say, in the same function. Following this principle, allocations are made in the first few lines, and all the corresponding deallocations are performed in the last few lines of the function body. In practice, however, it is often not possible to match up closely allocations and deallocations in code. But, it is always important to think carefully ahead of time about where and when memory should be allocated and deallocated. Do not allocate memory at random throughout your code or deep inside nested function calls, because doing so makes it hard to keep track of those allocations. In general, errors relating to dynamic memory allocation are hard to deal with, and it is best to avoid them from the start.

Memory Leaks

If delete is never used, and one continues to make calls to new, the system will ultimately crash when it runs out of memory in the free store. Bugs of this form are called *memory leaks* and can be very difficult to uncover. One insidious feature of memory leaks is that they lead to seemingly random and irreproducible crashes.

(Global Scope)

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  int main(int argc, char* argv[]) {
7
8      // Allocate until we run out of memory.
9      for (;;) {
10         int* ptr = new int[1000];
11     }
12
13     return EXIT_SUCCESS;
14 }
```

Listing 1.77. Exhausting the free store

The above listing contains a brute force example, but it does demonstrate that a memory leak in a C++ program can exhaust the memory available in the free store. Often though, the effect of a memory leak is more subtle and leads to a "bloated" program that slowly and slowly uses more and more system memory.

Overwrites

When dealing with pointers in C++, it is often easy to overwrite memory. Overwrites are especially troublesome when using dynamic memory. This is because an overwrite in the heap damages the information used to manage dynamic memory. Overwrites also are difficult to detect since they usually do not cause an error immediately. An overwrite might cause the program to crash at a later time, in some other section of code. Even worse, the program may not crash at all, continuing to run with some corrupted data. The Listing below shows a simple, yet very common, memory overwrite. The overwrite is caused by the incorrect use of <= instead of < in line 9. This causes the for-loop to overwrite memory that is not actually part of the array.

```
12  // Input size from user
13
14  int size = 0;
15  cin >> size;
16
17  // Allocate the array
18  int* ptr = new int[size];
19
20  // Initialize the elements
21  for (int i = 0; i <= size; i++) {
22      ptr[i] = 0;
23  }
```

Listing 1.78. Memory overwrite

Using Deallocated Memory

Another common mistake regarding dynamic memory is referencing memory that has already been deallocated. It is never safe to use a pointer whose stored memory address has been deallocated using *delete*. The *delete* operator, for efficiency reasons, does not clear the contents of the memory that it deallocates. This means there is a good chance that memory may still contain valid data long after it has been deallocated. There is no guarantee of this, however, since the memory in question could be reallocated by the run-time system. A good way to prevent accidentally using deallocated memory is to set the pointer equal to the null pointer after the call to *delete*.

Deallocating Memory Twice

Calling delete for a memory location that has already been deallocated is another common mistake regarding dynamic memory management in C++. This may seem like an obvious mistake that is easy to avoid, but when dealing with a program that has allocations and deallocations spread through a large code base, this error is quite common. Careful planning of when allocation and the corresponding deallocation of memory will occur is the best way to avoid deallocating the same memory twice.

1.7 Mechanisms for Code Reuse

Inheritance

Inheritance is a mechanism in C++ (as well as in other programming languages) that facilitates abstraction and code reuse. Inheritance establishes the "is-a" relationships between the classes contained in a program. Using inheritance, new classes can be built based on old classes, allowing child classes to share data members and functions of the parent class.It is through these relationships that the advantages of data abstraction (generalization and specialization) emerge.

Instead of using the keyword extends to denote the parent/child relationship, C++ uses a colon (:).

```
4  class Employee { /* declaration of parent */ };
5  class Manager : Employee { /* declaration of child */ };
```

Listing 1.79. C++ inheritance syntax

The above Listing declares class Employee and class Manager. Programmers and literature often use the terms "parent" and "child" to refer to the classes involved in the "is-a" relationship that inheritance models. Other terms used to describe the classes are "base" and "derived," respectively. As a standard, we use the terms "parent" and "child" throughout this course.

In C++, a child class inherits all non-private data members including constructors. In C++, this includes public and protected data members. In C++, data members declared without an access modifier default to use private access.

```
4  class BankAccount {
5
6  protected:
7      double sum;
8      string name;
9
10 public:
11
12     BankAccount(string nm) : name(nm), sum(0) {}
13
14     double balance() { return sum;}
15     void deposit(double amount) {sum += amount;}
16     void withdraw(double amount) {sum -= amount;}
17     string get_name() { return name;}
18 };
19
20 class SavingsAccount: public BankAccount {
21 protected:
22     double rate;
23
24 public:
25     SavingsAccount(string nm)
26             : BankAccount(nm),  // Call base class constructor
27     rate(0.055) {}
28
29     void add_interest() {sum *= (1 + rate);}
30     double get_rate() { return rate;}
31 };
```

Listing 1.80. Invoking parent class methods

Listing 1.80 contains an example of C++ inheritance that demonstrates how to invoke methods of a parent class. In this Listing, we have defined a class SavingsAccount that inherits from class BankAccount. Our SavingsAccount class provides a way to add interest to the money present in an account. Line 26 is interesting in that it invokes the parent class constructor. In C++, we could accomplish this same task using the keyword super(). In C++, we instead use the name of the parent class followed by the constructor's parameters in parentheses.

C++ contains three types, or levels, of inheritance: public, private, and protected. Public inheritance is the most common type of inheritance used in C++. Private and protected inheritance model a different type of relationship, namely the "uses-a" relationship. For example, to model that a car uses an engine, we could privately inherit from a class Engine when defining a class Car.

The more appropriate way to model this relationship, however, would be to have our class Car contain an instance of class Engine. Modeling the "uses-a" relationship in this manner is known as composition. All uses of inheritance in this course focus on modeling "is-a" relationships. Therefore, we do not use private or protected inheritance.

Polymorphism

A fundamental feature of object-oriented programming languages, polymorphism is the ability of an object to take on several different forms. Put another way, polymorphism allows a programmer to refer to an object of one class as an object of another class. This has two primary uses. First, we can create collections of heterogeneous objects. We can operate on the individual objects in these collections as if they were all of the same type, without the objects losing their real identities. Second, we can code algorithms that make only minimal assumptions about the objects they manipulate. This can allow an algorithm to continue to function correctly even when a programmer introduces new child classes into the system. Both of these uses help create more maintainable solutions.

In C++, polymorphism yields the same benefit as it does in other programming languages: programmers can refer to objects in a way that facilitates elegant solutions. The approach taken to achieve this goal in C++ is the same as in *Java*. Programmers invoke overridden methods of a child class via an ancestor class. In *Java*, this is accomplished by defining a class hierarchy and using parent class references with instances of child classes. In C++, we use *virtual functions* in conjunction with pointers to access objects polymorphically.

To illustrate what virtual functions are and why we need to use pointers in conjunction with them, let's step through an extended example. Suppose we wish to implement a graphics system that can display various geometric shapes on the screen. It is natural to start with a small hierarchy of graphics objects.

```
4  class Shape { /* ... */ };
5  class Circle: public Shape { /* ... */ };
6  class Rectangle: public Shape { /* ... */ };
```

Listing 1.81. A class hierarchy

We can assume that in our application we need to be able to keep track of

a collection of such shape objects. Suppose we want to maintain an array of shapes of either kind. We could try an array of type Shape. This is reasonable, since it is legitimate to make an assignment from a child class to a variable of the base class.

```
Circle C(3);   // radius 3
Shape S[10];
S[0] = C;   // syntactically correct, but ...
```

Listing 1.82. The slicing problem

Unfortunately, the additional parts of the child class are simply stripped off during the assignment. This is known as the *slicing problem*. We lose all the additional data members of the child class. Slicing, however, does not occur when we deal with pointers. Hence, we can salvage our project by using an array of pointers to shapes.

```
Shape *layout[10];
layout[0] = new Circle(3);       // radius 3
layout[1] = new Rectangle(2, 4); // width 2, height 4
```

Listing 1.83. Using pointers

Shapes have several methods associated with them. For example, we want be able to calculate the area a shape takes up on the screen. The *Shape* class, therefore, would have to contain an *area* () method if we wish to invoke *area* () from an array of *Shape**. Since *Shape* is a "generic" class, we could define its *area* () method as follows.

```
class Shape {
private:
    /* ... */

public:
    float area(void) { return 0;}
    /* ... */
};
```

Listing 1.84. More of the Shape class

Next we need class *Circle* and *Rectangle* to override the inherited *area* ()

method of class *Shape* to provide specific details on how to calculate their respective areas. Once this is done, we can try the following.

```
32  cout << layout[0]->area() << endl;   // prints 0
33  cout << layout[1]->area() << endl;   // prints 0
```

Listing 1.85. Calling the wrong area() function

This implementation doesn't work as intended. The problem is that the static type of *layout[0]* is *Shape**, and, therefore, the *area* () method belonging to class *Shape* is invoked, rather than the specific *area* () method for each of those array objects. What we need is a mechanism that checks the dynamic type of *layout[0]*, for example, determine that it is *Circle**, and then call the *area* () method from class *Circle*. This is accomplished in C++ using virtual functions. Below is the proper redefinition of the *Shape* class, particularly its *area* () method. The *area* () method in the child classes need not be redefined (though it is good style to attach the keyword *virtual* to those methods as well).

```
4   class Shape {
5   private:
6       /* ... */
7
8   public:
9       virtual float area(void) { return 0;}
10      /* ... */
11  };
```

Listing 1.86. A virtual function

Assuming that *Circle* and *Rectangle* provide their own *area* () method overriding the one in *Shape*. We obtain the correct output.

```
40  cout << layout[0]->area() << endl;   // prints 28.2743
41  cout << layout[1]->area() << endl;   // prints 8
```

Listing 1.87. Calling the correct area() function

We could also have modified *Shape* by making *area* () not just virtual but also totally undefined. A function of this sort is called a *pure virtual function*.

```
13  class Shape {
14  private:
15      /* ... */
16
17  public:
18      virtual float area(void) = 0; // totally undefined
19      /* ... */
20  };
```

Listing 1.88. A pure virtual function

A class that contains a pure virtual function is known as an *abstract class*. Implementations in C++ only use abstract classes in conjunction with inheritance. Put another way, programmers never create instances of abstract classes. They exist merely to specify the common interface of child classes and to access these child classes polymorphically.

By way of summary, in order to obtain polymorphic behavior, we need to:

- deal with pointers rather than direct objects because of slicing
- declare member functions to be virtual

Templates

Template functions allow a programmer to apply the logic of a function to more than one data type. An ordinary C++ function declaration dictates the data types of its parameters. There are situations, however, when the logical structure of a function makes sense for many different types. For example, to compute the maximum of two values, we perform the same type of calculation regardless of whether the values are integers, floats, or strings. Template functions allow the programmer to create functions independent of the data types of their parameters.

```
40  int max(int x, int y) { return x < y ? y : x;}
41  float max(float x, float y) { return x < y ? y : x;}
42  string max(string& x, string& y) { return x < y ? y : x;}
```

Listing 1.89. Function overloading

In this Listing 1.89, the implementations for each version of function max are identical. The only differences between the three function definitions are the data types of the parameters and the data types of the return values. This approach is undesirable since it is tedious and leads to serious maintenance problems. If we ever need to change the function, we have to edit every single copy by hand. If we would like the function to support additional data types

other than *int*, float, or string, we have to create those versions manually. Using a template function, a C++ programmer creates the logic of a function around a generic data type. The compiler then creates versions of the function for specific data types. Listing 1.90 shows a template version of function max.

```
13  template <typename T>
14 T my_max(T x, T y) {
15     return x < y ? y : x;
16 }
```

Listing 1.90. A template function

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <string>
4
5  using namespace std;
6
7  template <typename T>
8  const T& my_max(const T& x, const T& y) {
9      return x < y ? y : x;
10 }
11
12 int main(int argc, char* argv[]) {
13
14     int i1 = 1;
15     int i2 = 2;
16     cout << "Maximum is: " << my_max(i1, i2) << endl;
17
18     float f1 = 34.4;
19     float f2 = 24.2;
20     cout << "Maximum is: " << my_max(f1, f2) << endl;
21
22     string s1 = "test1";
23     string s2 = "test2";
24     cout << "Maximum is: " << my_max(s1, s2) << endl;
25
26     return EXIT_SUCCESS;
27 }
```

Listing 1.91. Creating and using a template function

Functions template require a new and unique syntax. In line 13 of the above Listing 1.90, the keyword template specifies that the function is a template function. It is followed in the angle brackets by the keyword *typename* and a generic data type name T. The generic data type name T is then used throughout the implementation of the function max. The compiler replaces this generic data type name with the specific data types used by the program.

Instead of the keyword typename, C++ also accepts the keyword class. To

avoid confusion, this course consistently uses the keyword type name when defining templates.

We refer to a template function as a "template" because the compiler actually creates multiple versions of the function based on the generic or "template" definition. To know which versions it needs to create, the compiler looks through the source code of a program, examining the data types of the actual parameters passed to the template function. When compiling the above example program, the compiler creates three versions of function *my_max*, one for type *int*, one for type *float*, and one for type *string*. If we included a call to function *my_max* that used variables of type *double*, the compiler would create a version of function *my_max*that supported type *double*.

When defining and using template functions, a programmer must consider that the actual data types supplied to a template function must support the operators and data members used with the generic data type. Even though the *my_max* function from Listing 1.91 is a template function, it will not support all possible data types. Function *my_max*, because of its implementation in line 8, can only be used with data types that define the < operator. This is an important principle programmers must follow when designing and using template functions. Listing 1.92 illustrates a violation of this principle.

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <string>
4
5  using namespace std;
6
7  template <typename GenericType>
8  void display_length(const GenericType& x) {
9      cout << "Length is: " << x.length();
10 }
11
12 int main(int argc, char* argv[]) {
13
14     string s = "string";
15     int i = 0;
16
17     display_length(s);
18
19     // int does not support function length()
20     // which results in compile error!
21     display_length(i);
22
23     return EXIT_SUCCESS;
24 }
```

Listing 1.92. Incorrectly using a template function

Template functions can utilize more than one generic data type. Listing 5 shows a template function, *some_function*, that has two generic types. This function takes two parameters. The first parameter is of the generic type `X`, and the second is of the generic type `Y`.

```
template <typename X, typename Y>
void output(X x, Y y) {
    cout << x << " " << y << endl;
}

int main(int argc, char* argv[]) {
    output(2, "test");
    output(4.5, true);
    output("test", 3);

    return EXIT_SUCCESS;
}
```

Listing 1.93. Multiple generic types

Template Classes

In addition to template functions, programmers can use C++ to create template classes. A template class in C++ is a class whose definition is independent of a specific data type. For example, we could define a template class Array that works for integers, floats, characters, strings, and user defined classes.

The syntax involved in a template class definition is similar to that of a template function.

```
template <typename ElementType>
class Array {

private:
    ElementType arr[SIZE]; // fixed size array

public:
    // Constructor
    Array(ElementType e) {
        for (int i = 0; i < SIZE; i++) {
            arr[i] = e;
        }
    }

    // Element access
    ElementType& operator[](int i) {
        assert (0 <= i && i < SIZE);
        return arr[i];
    }
};
```

Listing 1.94. A template class

Looking at both Listing 1.93 and Listing 1.94, we can see that the syntax of a template class definition resembles the syntax of a template function definition. In both cases, a line containing the keyword template and a list of the generic data type names precedes the definition. Listing 1.95 demonstrates instantiation of template classes. Notice the programmer specifies the specific type of the generic class inside angle brackets.

```
31  Array<int> i(10);
32  Array<float> f(22.5);
33  Array<string> s("initial value");
```

Listing 1.95. Using a template class

The template keyword must prefix any member function definition that appears outside of the class definition. Listing 1.96 demonstrates this requirement.

```
 6  template <typename ElementType>
 7  class Array {
 8
 9  private:
10      ElementType arr[SIZE]; // fixed size array
11
12  public:
13      Array(ElementType e);
14      ElementType& operator[](int i);
15  };
16
17  template <typename ElementType>
18  Array<ElementType>::Array(ElementType e) {
19      for (int i = 0; i < SIZE; i++) {
20          arr[i] = e;
21      }
22  }
23
24  template <typename ElementType>
25  ElementType& Array<ElementType>::operator[](int i) {
26      assert (0 <= i < SIZE);
27      return arr[i];
28  }
```

Listing 1.96. Member function definitions outside the class definition

The syntax that defines template-class member functions can be unwieldy. This is especially true in template classes that have many parameter types.

One way to alleviate this complexity is to use simple names for the template parameters. To provide readability inside the member-function definitions, define descriptive type definitions for the simple parameter names. Examine the implementation of the template class `Automobile` in the following listing.

```
template <typename ET, typename CT, typename TT, typename IT>
class Automobile {

    protected:
        // more human readable names
        typedef ET EngineType;
        typedef CT ChassisType;
        typedef TT TireType;
        typedef IT InteriorType;

    protected:
         EngineType engine;
         ChassisType chassis;
         TireType tires[4];
         InteriorType interior;

    public:
        Automobile(void);
};
```

Listing 1.97. Template syntax and typedef

Creating the type definitions in Listing 1.98 serves another, more important purpose. Consider deriving a class *formulaOneCar* from a specific type of *Automobile*. The rules regarding inheriting from a template class dictate that *typedef* statements (like the ones found in Listing 1.96) must exist for a derived class to have access to the parameter types of the parent class.

```
class formulaOneCar : public Automobile<void> {

//  ET a;          // error!
    EngineType b; // correct
};
```

Listing 1.98. Providing access to derived classes

Notice from the above listing that an attempt to access the type `ET` fails. The type name *EngineType* succeeds.

Yet another useful application of type definitions appears in Listing 1.99. This listing creates the type name *racecar* to refer to a specific type of the template class *Automobile*. With this type definition in place, the programmer can use the much shorter type name *racecar* instead of the unwieldy *Automobile<TurboEngine, RacingChassis, RacingTire, Fireproof>*.

```
typedef Automobile<TurboEngine, RacingChassis, RacingTire, Fireproof>
racecar;
racecar car1, car2;
```

Listing 1.99. Simplifying declaration

C++ Exception Handling Fundamentals.

C++ exception handling centers around the use of the try keyword and the catch keyword. A programmer encloses code that may trigger an exception within a try block. One or more catch blocks immediately follow the try block. When an exception occurs in a try block, the program execution point is transferred to a catch block corresponding to the type of exception thrown. The catch block performs the necessary tasks to recover from the exceptional condition. In other words, it "handles" the exception. Listing 1.100 demonstrates the use of try and catch blocks in C++.

C++ programmers trigger exceptions using the *throw* statement. In a *throw* statement, a programmer specifies the exception to trigger following the keyword *throw*. Listing 1.101 randomly triggers one of two exceptions using *throw* statements.

```
#include <iostream>
#include <cstdlib>
#include <stdexcept>

using namespace std;

void method_two() {
    throw out_of_range("out of range");
}

void method_one() {

    int *array = new int[100];

    try {
        method_two();
        delete [] array;
    }
    catch (...) {
        delete [] array;
        throw; // rethrow exception
    }
}

int main(int argc, char* argv[]) {

    try {
        method_one();
        return EXIT_SUCCESS;
    }
    catch (exception& e) {
        cerr << e.what() << endl;
    }
    catch (...) {
        cerr << "Unknown exception!" << endl;
    }

    return EXIT_FAILURE;
}
```

Listing 1.100. The try and catch blocks

```
#include <iostream>
#include <cstdlib>
#include <stdexcept>
#include <ctime>

using namespace std;

int main(int argc, char* argv[]) {

    try {

        srand(time(0));
        if (rand() % 2 == 0) {
            throw out_of_range("out of range");
        }
        else {
            throw length_error("length error");
        }

        return EXIT_SUCCESS;
    }
    catch (out_of_range e) {
        cerr << "caught out_of_range exception" << endl;
    }
    catch (length_error e) {
        cerr << "caught length_error exception" << endl;
    }

    return EXIT_FAILURE;
}
```

Listing 1.101. Triggering exceptions

In C++, a variable of any data type can be thrown as an exception using the *throw* statement. C++ can throw objects of user-defined classes, objects of standard classes (such as class *string*), and even variables of the fundamental data types. Typically, it is not useful to throw these basic objects and fundamental data types as exceptions. C++ programmers instead create user-defined exception classes. We examine this practice later in this page.

A function can transfer execution to a *catch* statement that exists in another function. This usually happens when a function (that throws an exception) is called from within a *try* block of another function. Listing 1.102 contains an example of this idea.

```
#include <iostream>
#include <cstdlib>
#include <stdexcept>

using namespace std;

void display_args(int argc, char* argv[]) {

    if (argc == 1) {
        throw runtime_error("Please supply arguments");
    }

    for (int i = 1; i < argc; i++) {
        cout << argv[i] << endl;
    }
}

int main(int argc, char* argv[]) {

    try {
        display_args(argc, argv);
        return EXIT_SUCCESS;

    }
    catch (runtime_error& e) {
        cerr << e.what() << endl;
    }

    return EXIT_FAILURE;
}
```

Listing 1.102. Handling a thrown exception in another function

C++ provides a mechanism to catch exceptions of all possible types. A catch block whose argument contains ellipsis catches all exceptions. This is useful in situations where a programmer wants to ensure that all possible exceptions are caught. An unhandled exception can cause abnormal program termination.

One or more catch blocks immediately follow the try block. When an exception occurs in a try block, the program execution point is transferred to a catch block corresponding to the type of exception thrown. The catch block performs the necessary tasks to recover from the exceptional condition.

```
int main(int argc, char* argv[]) {
    try {
        if (argc == 1) {
            throw runtime_error("runtime error");
        }
        if (argc == 2) {
            throw out_of_range("out of range");
        }
        if (argc == 3) {
            throw length_error("length error");
        }
        return EXIT_SUCCESS;
    }
    catch (...) {
        cerr << "An exception was caught!" << endl;
    }
    return EXIT_FAILURE;
}
```

Listing 1.103. Catching all exceptions

The C++ Standard Exception Hierarchy

The C++ Standard Library provides and uses a standard exception class hierarchy. The classes contained in this hierarchy are thrown by various components of C++ language, the Standard Library, and the Standard Template Library. Figure 1.17 depicts the standard exception hierarchy.

The *bad_alloc* and *out_of_range* exceptions are two standard exceptions that we pay careful attention to in this course. Exception *bad_alloc* is thrown by the `new` operator whenever memory cannot be allocated as requested. Exception *out_of_range* is thrown by the member function *at* of type *vector* when an element whose index is out of range s requested.

Catching standard exception objects by reference allows a programmer to handle exceptions polymorphically. This is useful since it allows a *catch* block to handle an entire group of exceptions from the standard exception hierarchy. For instance, catching class *runtime_error* by reference catches exceptions of type *runtime_error* and all descendents of type *runtime_error*. A *catch* block that catches class *exception* by reference will catch all exceptions in the standard exception hierarchy.

The interfaces for the standard exception hierarchy classes all contain the member function *what*. This function returns a string describing the type of the invoking exception object. This is useful when working polymorphically with exception objects, since it provides a standard mechanism to report on the dynamic type of an exception object. To see the effect of the member function

what, run the example program from Listing 1.90 with a command-line argument of `1000000000` (one billion). A *bad_alloc* exception will be thrown since operator `new` will not be able to allocate enough memory to store a billion integers.

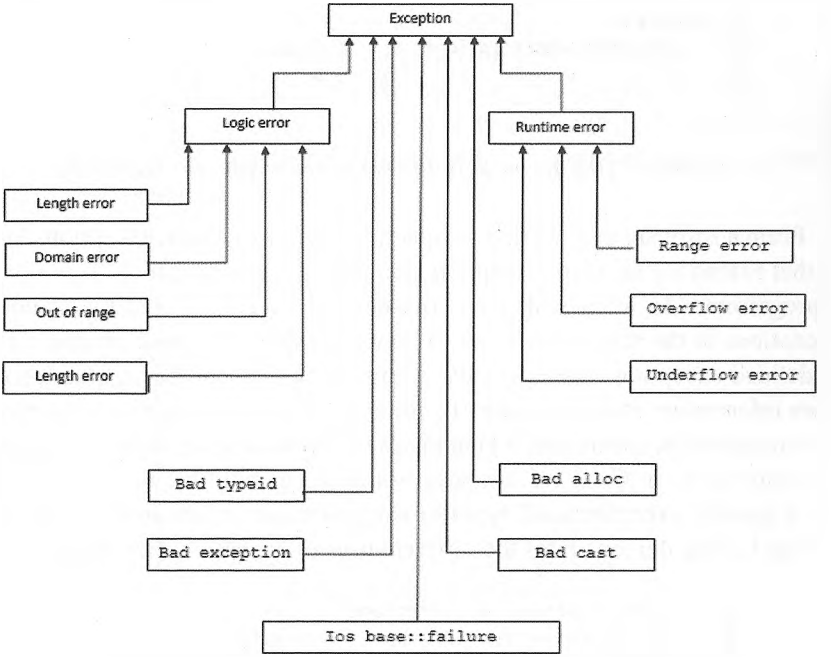


Figure 1.104. The standard exception hierarchy

```
try {

// some code ...

    }
    catch (runtime_error& e) {
// catch all runtime errors
        cerr << e.what() << endl;
    }
    catch (logic_error& e) {
// catch all logic errors
    cerr << e.what() << endl;
}
    catch (exception& e) {
// catch all the rest of the exceptions
    cerr << e.what() << endl;
}
```

Listing 1.105. Catching exceptions polymorphically

Programmers can extend the standard exception hierarchy by defining their own exception classes that inherit from one of the standard exception classes.

```
class DivideByZeroException : public logic_error {

public:
    DivideByZeroException(const string& what_arg ) throw()
            : logic_error ("divide by zero: " + what_arg) {}
 ;
```

Listing 1.106. Extending the standard exception hierarchy

From a software engineering perspective, creating custom exception classes that extend the standard exception hierarchy is a good practice. This allows a programmer to work with their user-defined exceptions and the standard exceptions in the same manner since they both share the same interface. User-defined exception classes are also useful since they can be programmed to store information about the state of a program when the exception occurred.

An exception specification guarantees the types of exceptions that a function can throw. In C++, we can specify that a function throws either one or more specific exceptions, all types of exceptions, or no exceptions. The following Listing demonstrates the different uses of exception specifications.

```
// Some function declarations
void f() throw();   // throws no exceptions
void g() throw(out_of_range);  // throws one
void h() throw(out_of_range, bad_alloc); // throws two
void i();  // can throw any exception (the default)
```

Listing 1.107. Exception specifications

Note for Visual C++ users: Visual C++ departs from the ANSI Standard in its implementation of exception specifications. Exception specifications are accepted but not implemented in Microsoft Visual C++. Setting a non-empty exception specification results in a compile warning. You can avoid this warning by using the warning pragma:

#pragma warning(disable : 4290)

C++ programmers should use exception specifications sparingly since they expose information about the implementation of a function. A child class may wish to override the implementation of inherited functions that have exception specifications. The inherited exception specifications, however, may not apply to the child class implementation. Perhaps the child class implementation

needs to throw an *out_of_range* exception when the parent implementation only throws a *length_error* exception.The main routine of a program that handles exceptions using the standard exception hierarchy typically should follow a standard format. The following Listing contains an example of this format. In this format, a single try block wraps the application source code. Following this try block are multiple catch blocks. These catch blocks start specific and become more general. This allows a programmer to take specific actions for certain exceptions. Catching class exception ensures that all user-defined and standard exception hierarchy classes are caught. Finally, all exceptions are caught to ensure the program exits gracefully if a non-standard exception is thrown anywhere in the application.

```
32 int main(int argc, char* argv[]) {
33
34     try {
35
36         // source ...
37
38         return EXIT_SUCCESS;
39     }
40     catch (out_of_range& e) { // specific
41         cerr << "Out of range" << endl;
42     }
43     catch (exception& e) {  // general
44         cerr << e.what() << endl;
45     }
46     catch (...) { // all
47         cerr << "A non-standard exception" <<
48         " was caught!" << endl;
49     }
50
51     return EXIT_FAILURE;
52 }
```

Listing 1.108. An exception handling main routine

Using Intermediate Handlers

```
#include <cstdlib>
#include <stdexcept>

using namespace std;

void method_two() {
    throw out_of_range("out of range");
}

void method_one() {
    int *array = new int[100];
    method_two();
    delete [] array;
}

int main(int argc, char* argv[]) {

    try {
        method_one();
        return EXIT_SUCCESS;
    }
    catch (exception& e) {
        cerr << e.what() << endl;
    }
    catch (...) {
        cerr << "Unknown exception!" << endl;
    }

    return EXIT_FAILURE;
}
```

Listing 1.109. Exception handling causing a memory leak

Intermediate exception handlers are exception handlers that catch an exception, deallocate some resource, and immediately rethrow the exception. Intermediate exception handlers are used to ensure that all allocated memory is properly deallocated, even when a program transfers control to an exception handler. This gets complicated when we consider that an exception can cause a program to transfer control to a catch block in a different routine.

The above listing illustrates the difficulty that explicit memory management introduces to exception handling and why we need intermediate exception handlers. This Listing includes a function method_one that allocates an integer array in memory. Before method_one deallocates this array, the func-

tion calls method_two. Function method_two immediately throws an exception. Control is then transferred to the nearest appropriate catch block. This catch block, however, exists in function main. The transfer of control back to this catch block skips the execution of the delete [] statement in method_one. This results in a memory leak.

An intermediate exception handler can prevent the memory leak that is present in Listing 1.109. The job of such a handler is to catch any exception thrown by the call to method_two, deallocate the integer array, and rethrow the exception. Listing includes such an exception handler.

Notice from Listing 1.110 that rethrowing a caught exception is done using an empty throw statement.

Summary of Best Practices

The following summarizes C++ exception handling best practices:

- Include all code that might throw exceptions in try blocks;
- Handle exceptions where it is known how to recover from the exception;
- Catch exceptions by reference;
- Use exception specifications only for non-virtual functions that throw specific exceptions;
- Create user-defined exception classes that extend the standard exception hierarchy;
- In main, wrap all code in a try block, catch class exception and catch all exceptions;
- Use intermediate exception handlers to avoid leaking memory.

The modern methods a study of programming, as a rule are guided by the decision of the complex tasks having practical application. Therefore MES of RK recommendations support such methods of training which found broad practice in the Internet on the interactive websites. Below the solution of such task which will be useful not only to fixing of knowledge of the passable material, but also in case of a further study of the following sections is proposed.

```
#include <iostream>
#include <cstdlib>
#include <stdexcept>

using namespace std;

void method_two() {
    throw out_of_range("out of range");
}

void method_one() {

    int *array = new int[100];

    try {
        method_two();
        delete [] array;
    }
    catch (...) {
        delete [] array;
        throw; // rethrow exception
    }
}

int main(int argc, char* argv[]) {

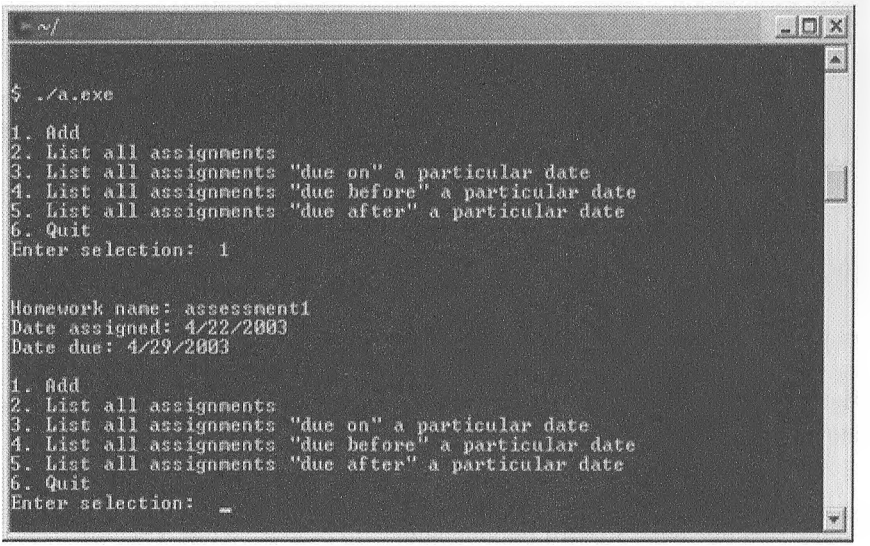
    try {
        method_one();
        return EXIT_SUCCESS;
    }
    catch (exception& e) {
        cerr << e.what() << endl;
    }
    catch (...) {
        cerr << "Unknown exception!" << endl;
    }

    return EXIT_FAILURE;
}
```

Listing 1.110. Intermediate exception handler

Description

The program produced in this exercise maintains a list of homework assignments. Using this program, a user can add homework assignments to the program maintained list. The user can then view all assignments or assignments due on, before, or after a specific date. The program interacts with the user through a menu that displays these options. The following screen shot depicts the program during execution.



Listing 1.111. Homework assignments

A typical use of the application is as follows. The user enters the menu item corresponding to the task they wish to perform. Depending on the task, the user may have to enter additional input. For example, to add a homework assignment, the user first enters the menu item number corresponding to the "add" operation. This is the number 1. Once the user presses the Enter key, the program prompts the user for the name of the assessment. Once this input is received, the program prompts the user for the date assigned and then the date due. This completes the addition of the homework. Each time an operation completes, the program displays another instance of the menu. This loop continues until the user chooses to quit.

This program includes separate classes for modeling a date, an individual assignment, and a collection of assignments.

Class date

Class date models a calendar date. A date, in this context, includes the day, month, and year. Class date contains the following members.

- *date ();*

Default constructor that initializes day, month, and year all to zero

- *date (int day, int month, int year);*

Three parameter constructor that initializes the day, month, and year to more meaningful values

- *int compareTo (date another_date);*

Function that compares two dates

Class date also dictates how its object instances appear when they are sent to an output stream. The insertion operator outputs a date using the "mm/dd/yyyy" format. This method declaration appears below.

friend ostream& operator<< (ostream <stream, date d);

Class homework

Class homework models a homework assignment. For each homework assignment, this program maintains the assignment name, the date assigned, and the date due. Class homework contains the following members.

- *homework ();*

Default constructor that initializes name to the empty string and the assigned and due dates to their default values

- *homework (string name, date assigned_date, date due_date);*

Three parameter constructor that initializes the assignment name, assigned date, and due date

- *int compareTo (homework another_homework);*

Function that compares two homework objects

Similar to class date, class homework also dictates how its object instances are displayed when they are sent to an output stream. The insertion operator outputs a homework object using the following format.

name: *assigned date – due date*

This method declaration appears below.

friend ostream &operator<< (ostream &stream, homework h);

Class homeworklist

Class homeworklist models a list of homework assignments. Class homeworklist maintains this list of assignments internally using an array. This class limits the number of assignments that it can store to 50.

Class homeworklist contains the following members.

- *homeworklist ();*

Default constructor that initializes the size of the list to zero

- *bool add (homework h);*

This method adds an assignment. It returns true if the add operation succeeds and false if the maximum number of assignments is already stored.

- *homeworklist dueafter (date d);*

Returns a new homeworklist containing only those homeworks due after, but not on, the specified date

- *homeworklist duebefore (date d);*

Returns a new homeworklist containing only those homeworks due before, but not on, the specified date

- *homeworklist dueon (date d);*

Returns a new homeworklist containing only those homeworks due exactly on the specified date, but not before or after

Class homeworklist also defines how instances of this class appear when they are sent to an output stream. The following method outputs each homework object stored in the private array on a separate line.

friend ostream &operator<<(ostream &stream, homeworklist hl);

Following is an ordered list of steps that serves as a guide to completing this assessment. Work and test incrementally. Save often.

1.First, complete the constructors for class date. The default constructor should initialize all private data members to zero. The other constructor should initialize the private data members to the corresponding parameter of the constructor. Remember to use initializer lists.

2.Next, complete the class date member function compareTo. This function compares two date objects. It returns zero if the objects are equal, a negative value if the invoking object is less than the date given in the parameter, and a positive value if the invoking object is greater than the date given in the parameter.

3.Then, complete the definition of the insertion operator. This function should output a date object in the form "mm/dd/yyyy". This completes the implementation of class date.

4.For class homework, **begin** by implementing the constructors. These implementations resemble what you did in class date.

5.Next, implement the member function compareTo. To compare two homework objects, you need only compare their due dates.

6.Then, complete the definition of the insertion operator for objects of type homework. This function should output a homework object in the form "<*name*>: <*assigned date*> – <*due date*>". This completes the implementation of class homework.

7.Begin the implementation of class homeworklist by completing the default constructor. This method only needs to initialize the private data member current_size to zero.

8.Next, complete the implementation of the member function add. Do not allow the method to store more than the maximum number of homeworks.

9.Then, complete methods duebefore, dueon, and dueafter. Each of these methods returns a homeworklist object containing a subset of homeworks. These subsets contain all homeworks due before, on, or after a specified date. The single parameter for each of these functions specifies the date.

10. Finally, finish the implementation of the insertion operator defined for class homeworklist. This method should iterate through the array of homework objects, outputting each on a separate line.

Basic algorithms

2.1 Execution Abstraction

Execution as Physical Process

Programming is a difficult task requiring a strict exercise of logic and a capacity to organize substantial amounts of information. Because these challenges require our full attention, we seldom stop to think about all the work that goes on behind the scenes when we execute our finished programs. In this paragraph we will try to give you a feeling for how nearly miraculous it is that the programs that we design end up commanding innumerable devices and mechanisms in complicated sequences of tiny events. These events, which appear insignificant when considered individually, behave according to the specification of our programs when considered together. When writing a program, you might view yourself as the CEO (Chief Executive Officer) of a large company who, though unaware of the individual work of thousands of employees, directs their efforts in one coherent strategy. This requires organization and a lot of middle management, the nature of which we need explore.

- Execution as Physical Process
- Levels of Abstraction

While we are developing programs, we work in an abstract realm: We think of variables and data types rather than of memory chips; of algorithms rather than of moving data among those chips; of program statements rather than of where and how these statements are stored. Even though we are somewhat aware that variables, algorithms, and statements are fleshed out in a running program in the form of values in memory and CPU commands, we don't usually give much thought to how this happens. We don't give much thought to this because we don't have to: The computer hardware and operating system are very good at translating our abstract programs into electrical currents that change in the appropriate ways. We are so oblivious to that translation that we often forget that it takes place at all, and we confuse the program with its execution, even though the program is a static reflection of thought and the execution is a dynamic process that takes place in the physical world.

Sometimes, the physical medium in which programs execute does not

cause programs to behave "incorrectly," but it causes them to be unexpectedly slow. If you have learned to analyze the performance of a program using asymptotic analysis, this technique is very useful when designing algorithms, and gives a good approximation to how quickly an algorithm will execute. But like all useful techniques that analyze complex systems, asymptotic analysis takes advantage of simplifying assumptions that don't always hold when the program executes. The limitations of computer technology, for instance, entail, among other things, that memory does not have uniform access times. That is, reading the value of one variable in a program may take much longer than reading the value of another variable. This problem is particularly insidious because we cannot know beforehand which variables will be slower than others, and, worse, the variables that are slow at one point in the program's execution may become fast at a later point. In this chapter, we will explore this difficulty, and we will learn how to speed up programs that get bogged down by this peculiarity of real, physical computers.

Levels of Abstraction

An "abstraction" is a notion that describes multiple instances of objects or events but that is more succinct than the instances. A good abstraction captures relevant features of the instances while discarding those that are accidental. For example, when we talk about a "car" we may think of an expensive object that moves under its own power and is big enough to carry people, even though there are many types of cars that differ substantially from each other. An electric car is rather different from one powered by a combustion engine, but often we don't care about such differences and prefer to use the more general concept of "car." This concept hides the accidental difference in the propulsion system but is nevertheless sufficiently specific to distinguish a car from, say, an airplane.

Abstraction plays a central role in information technology. You are already familiar with it. For example, method inheritance in object-oriented languages such as C++ or C++ provides a framework for abstraction. An "electric car" class can inherit methods from the "car" class, which in turn, may inherit methods from a "transportation device" class. Each class captures characteristics that are useful, without specifying characteristics that are not relevant at its level of abstraction. Within each of these classes, it is the implementation of the public methods that reveals further details. The code written for each public method of the "electric car" class may reveal, for instance, that the simple "car" may actually subsume complex objects such as electric generators, batteries, CV joints, and so on. The detail is necessary to design the car, but the abstraction is extremely useful because, once the car is designed, we can ef-

fortlessly instantiate thousands of cars to build, say, a model of a city's streets. Imagine what Herculean effort would be required if, in building the city model, we had to separately write the complete code for each part of each car.

Back when computers were first invented this is pretty much what people did: For each computation they needed to carry out, a team of engineers would design a custom computer down to each single electronic tube. Thousands of such tubes, each very limited, would be interconnected with myriad wires to form a system that would, if not a single wire was out of place, carry out the desired computation. You can imagine how difficult and error-prone this was. You can also see that each of these engineers had to have comprehensive knowledge of all the technology involved, from algorithms to electronics.

Things have improved. Today, when you write a small program, you manage to put to work hundreds of millions of transistors, all without needing to know (though you may know) how a transistor works and without needing to actually manipulate any wires. You do this by relying on abstractions.

C++ and C++ give you the abstraction of an object space. You describe object classes, and then instantiate objects of those types that behave according to the stipulated rules. The compilers for these languages allow you to think in these terms, and free you from having to think at a lower-level, by automatically translating your programs into other programs that use lower-level abstractions. For instance, some C++ compilers use the simpler notions that are available in C, perhaps implementing a class instance as a C pointer that points to a struck which contains pointers for each of the object's variables and methods, and so on. The resulting C program will behave exactly like the original C++ program, but it will probably be much longer and much harder to read. Much like cars can be driven by people who don't know how cars work, the C++ compiler allows us to program using classes without knowing how the classes are implemented in a real computer.

But the C programming model is itself an abstraction. The C program that implements the C++ program is itself very simple compared to the actual program that gets executed directly by the CPU. For example, in C, you can name variables with mnemonic names, instead of having to specify cryptically the locations within memory where the variables are stored, and you can write array[i], instead of having to compute the location of the it element of array yourself. You can also write c = a + b instead of having to give the CPU specific instructions on how to carry out the addition. This is because the C compiler takes care of translating C programs into machine code. "Machine code" is the language that embodies the lower-level abstractions provided by the CPU chip. This is the same for all languages that execute on any one computer, and includes basic instructions like "move byte" and "is a byte zero?"

This is as far as software goes and as far as programmers go. But the engi-

neers at Intel and other hardware companies do not stop there. Machine code instructions are themselves translated by "peripheral" parts of the CPU into even simpler instructions that have meaning deep inside of the CPU. Deep inside the CPU, these instructions are, in turn, used to turn switches on and off in a complicated dance achieved by logic "gates." These "gates" are in turn implemented with silicon transistors, and transistors are themselves abstractions of physical electronic processes.

You may have noticed that abstractions seem to come in hierarchies, each level referring to the same underlying facts in more or less detail. A C++ class and a fragment of machine code may both refer to the same program. Whether we choose to manipulate one or the other depends on what we are trying to do. The higher levels of abstraction (the C++ class in this case) lack detail, whereas the lower levels lack clarity. Think of a sequence of maps of decreasing scale: the first one is perhaps a plan of your home, and the last one is a map of the world. Now suppose that we made maps of the whole world at all these different scales so that, for instance, your living room would appear in the highest-scale map along with all the other living rooms in the planet. That would be quite a few living rooms, don't you think? Now, suppose that we want to use these maps to find Andorra, a small country that happens to be nestled between Spain and France. The lowest-scale map, the world map that schools hang on their walls, is too coarse to show tiny Andorra. The highest-scale map, however, is far too detailed: it is impossible for us to figure out which living rooms are inside Andorra, and which are in France or Spain-because in such a detailed map, the borders among these countries are no more noticeable than the walls of your house. But suppose that, instead of Andorra, we are looking to find out whether we can fit an extra bookshelf in our cousin's living room. We would surely be unable to figure this out by looking at a map of Western Europe!

As we have seen, when we program in C++ we work at a pretty high level of abstraction that ignores many details of how computers work. But there are many higher levels as well. A person answering a mobile phone in the street is actually using a computer that is probably partially programmed with C++. Yet for that person, the device is not a computer, and answering the call does not involve thoughts of classes or method invocations.

Why go over this business of abstraction now? Abstraction is ubiquitous, and we know that you hardly need to have it explained. However, while most people readily accept the advantages of abstraction, not everyone keeps its limitations in mind. For example, the borders drawn between states on a map might gloss over certain details-such as where a house straddles the border between two states-and in that sense, the map might be considered to be inaccurate because it implies that entities are either in one state or another. Such

inaccuracies do not usually have any practical importance, but, occasionally, they do. It is for such occasions that, while working at a high level of abstraction, we must remain vigilant for cases that show imperfections in the abstractions we use.

2.2 Algorithms

When we write a computer program, we are generally implementing a method that has been devised previously to solve some problem. This method is often independent of the particular computer to be used-it is likely to be equally appropriate for many computers and many computer languages. It is the method, rather than the computer program itself, that we must study to learn how the problem is being attacked. The term algorithm is used in computer science to describe a problem-solving method suitable for implementation as a computer program. Algorithms are the stuff of computer science: They are central objects of study in many, if not most, areas of the field.

Most algorithms of interest involve methods of organizing the data involved in the computation. Objects created in this way are called data structures, and they also are central objects of study in computer science. Thus, algorithms and data structures go hand in hand. In this book we take the view that data structures exist as the byproducts or end products of algorithms and that we must therefore study them in order to understand the algorithms. Simple algorithms can give rise to complicated data structures and, conversely, complicated algorithms can use simple data structures. We shall study the properties of many data structures in this book; indeed, the book might well have been called Algorithms and Data Structures in C++.

When we use a computer to help us solve a problem, we typically are faced with a number of possible different approaches. For small problems, it hardly matters which approach we use, as long as we have one that solves the problem correctly. For huge problems (or applications where we need to solve huge numbers of small problems), however, we quickly become motivated to devise methods that use time or space as efficiently as possible.

The primary reason to learn about algorithm design is that this discipline gives us the potential to reap huge savings, even to the point of making it possible to do tasks that would otherwise be impossible. In an application where we are processing millions of objects, it is not unusual to be able to make a program millions of times faster by using a well-designed algorithm. We shall see such an example on numerous other occasions throughout the book. By contrast, investing additional money or time to buy and install a new computer holds the potential for speeding up a program by perhaps a factor of only 10 or

100. Careful algorithm design is an extremely effective part of the process of solving a huge problem, whatever the applications area.

When a huge or complex computer program is to be developed, a great deal of effort must go into understanding and defining the problem to be solved, managing its complexity, and decomposing it into smaller subtasks that can be implemented easily. Often, many of the algorithms required after the decomposition are trivial to implement. In most cases, however, there are a few algorithms whose choice is critical because most of the system resources will be spent running those algorithms. Those are the types of algorithms on which we concentrate in this book. We shall study a variety of fundamental algorithms that are useful for solving huge problems in a broad variety of applications areas.

The sharing of programs in computer systems is becoming more widespread, so although we might expect to be using a large fraction of the algorithms in this book, we also might expect to have to implement only a small fraction of them. For example, the C++ libraries contain implementations of a host of fundamental algorithms. However, implementing simple versions of basic algorithms helps us to understand them better and thus to more effectively use and tune advanced versions from a library. More important, the opportunity to reimplement basic algorithms arises frequently. The primary reason to do so is that we are faced, all too often, with completely new computing environments (hardware and software) with new features that old implementations may not use to best advantage. In other words, we often implement basic algorithms tailored to our problem, rather than depending on a system routine, to make our solutions more portable and longer lasting. Another common reason to reimplement basic algorithms is that, despite the advances embodied in C++, the mechanisms that we use for sharing software are not always sufficiently powerful to allow us to conveniently tailor library programs to perform effectively on specific tasks.

Computer programs are often overoptimized. It may not be worthwhile to take pains to ensure that an implementation of a particular algorithm is the most efficient possible unless the algorithm is to be used for an enormous task or is to be used many times. Otherwise, a careful, relatively simple implementation will suffice: We can have some confidence that it will work, and it is likely to run perhaps 5 or 10 times slower at worst than the best possible version, which means that it may run for an extra few seconds. By contrast, the proper choice of algorithm in the first place can make a difference of a factor of 100 or 1000 or more, which might translate to minutes, hours, or even more in running time. In this book, we concentrate on the simplest reasonable implementations of the best algorithms. We do pay careful attention to carefully coding the critical parts of the algorithms, and take pains to note where low-level optimization effort could be most beneficial.

The choice of the best algorithm for a particular task can be a complicated process, perhaps involving sophisticated mathematical analysis. The branch of computer science that comprises the study of such questions is called analysis of algorithms. Many of the algorithms that we study have been shown through analysis to have excellent performance; others are simply known to work well through experience. Our primary goal is to learn reasonable algorithms for important tasks, yet we shall also pay careful attention to comparative performance of the methods. We should not use an algorithm without having an idea of what resources it might consume, and we strive to be aware of how our algorithms might be expected to perform.

Implementation and Empirical Analysis

We design and develop algorithms by layering abstract operations that help us to understand the essential nature of the computational problems that we want to solve. In theoretical studies, this process, although valuable, can take us far afield from the real-world problems that we need to consider. Thus, in this book, we keep our feet on the ground by expressing all the algorithms that we consider in an actual programming language: C++. This approach sometimes leaves us with a blurred distinction between an algorithm and its implementation, but that is small price to pay for the ability to work with and to learn from a concrete implementation.

Indeed, carefully constructed programs in an actual programming language provide an effective means of expressing our algorithms. In this book, we consider a large number of important and efficient algorithms that we describe in implementations that are both concise and precise in C++. English-language descriptions or abstract high-level representations of algorithms are all too often vague or incomplete; actual implementations force us to discover economical representations to avoid being inundated in detail.

We express our algorithms in C++, and this book is about algorithms, rather than not only about C++ programming. Certainly, we consider C++ implementations for many important tasks, and when there is a particularly convenient or efficient way to do a task in C++, we will take advantage of it. But the vast majority of the implementation decisions that we make are worth considering in any modern programming environment. Translating the programs in this book, to another modern programming language is a straightforward task. On occasion, we also note when some other language provides a particularly effective mechanism suited to the task at hand. Our goal is to use C++ as a vehicle for expressing the algorithms that we consider, rather than to dwell on implementation issues specific to C++.

If an algorithm is to be implemented as part of a large system, we use

abstract data types or a similar mechanism to make it possible to change algorithms or implementations after we determine what part of the system deserves the most attention. From the start, however, we need to have an understanding of each algorithm's performance characteristics, because design requirements of the system may have a major influence on algorithm performance. Such initial design decisions must be made with care, because it often does turn out, in the end, that the performance of the whole system depends on the performance of some basic algorithm, such as those discussed in this book.

Implementations of the algorithms in this book have been put to effective use in a wide variety of large programs, operating systems, and applications systems. Our intention is to describe the algorithms and to encourage a focus on their dynamic properties through experimentation with the implementations given. For some applications, the implementations may be quite useful exactly as given; for other applications, however, more work may be required. For example, using a more defensive programming style than the one that we use in this book is justified when we are building real systems. Error conditions must be checked and reported, and programs must be implemented such that they can be changed easily, read and understood quickly by other programmers, interface well with other parts of the system, and be amenable to being moved to other environments.

Notwithstanding all these comments, we take the position when analyzing each algorithm that performance is of critical importance so that we focus our attention on the algorithm's essential performance characteristics. We assume that we are always interested in knowing about algorithms with substantially better performance, particularly if they are simpler.

To use an algorithm effectively, whether our goal is to solve a huge problem that could not otherwise be solved, or whether our goal is to provide an efficient implementation of a critical part of a system, we need to have an understanding of its performance characteristics. Developing such an understanding is the goal of algorithmic analysis.

One of the first steps that we take to understand the performance of algorithms is to do empirical analysis. Given two algorithms to solve the same problem, there is no mystery in the method: We run them both to see which one takes longer! This concept might seem too obvious to mention, but it is an all-too-common omission in the comparative study of algorithms. The fact that one algorithm is 10 times faster than another is unlikely to escape the notice of someone who waits 3 seconds for one to finish and 30 seconds for the other to finish, but it is easy to overlook as a small constant overhead factor in a mathematical analysis. When we monitor the performance of careful implementations on typical input, we get performance results that not only give us a direct indicator of efficiency but also provide us with the information that

we need to compare algorithms and to validate any mathematical analyses that may apply. When empirical studies start to consume a significant amount of time, mathematical analysis is called for. Waiting an hour or a day for a program to finish is hardly a productive way to find out that it is slow, particularly when a straightforward analysis can give us the same information.

The first challenge that we face in empirical analysis is to develop a correct and complete implementation. For some complex algorithms, this challenge may present a significant obstacle. Accordingly, we typically want to have, through analysis or through experience with similar programs, some indication of how efficient a program might be before we invest too much effort in getting it to work.

The second challenge that we face in empirical analysis is to determine the nature of the input data and other factors that have direct influence on the experiments to be performed. Typically, we have three basic choices: use actual data, random data, or perverse data. Actual data enable us truly to measure the cost of the program in use; random data assure us that our experiments test the algorithm, not the data; and perverse data assure us that our programs can handle any input presented them. For example, when we test sorting algorithms, we run them on data such as the words in Moby Dick, on randomly generated integers, and on files of numbers that are all the same value. This problem of determining which input data to use to compare algorithms also arises when we analyze the algorithms.

It is easy to make mistakes when we compare implementations, particularly if differing machines, compilers, or systems are involved, or if huge programs with ill-specified inputs are being compared. The principal danger in comparing programs empirically is that one implementation may be coded more carefully than the other. The inventor of a proposed new algorithm is likely to pay careful attention to every aspect of its implementation and not to expend so much effort on the details of implementing a classical competing algorithm. To be confident of the accuracy of an empirical study comparing algorithms, we must be sure to give the same attention to each implementation.

One approach that we often use in this book, is to derive algorithms by making relatively minor modifications to other algorithms for the same problem so that comparative studies really are valid. More generally, we strive to identify essential abstract operations and start by comparing algorithms on the basis of their use of such operations. For example, the comparative empirical results are likely to be robust across programming languages and environments, as they involve programs that are similar and that make use of the same set of basic operations. For a particular programming environment, we can easily relate these numbers to actual running times. Most often, we simply

want to know which of two programs is likely to be faster, or to what extent a certain change will improve the time or space requirements of a certain program.

Perhaps the most common mistake made in selecting an algorithm is to ignore performance characteristics. Faster algorithms are often more complicated than brute-force solutions, and implementers are often willing to accept a slower algorithm to avoid having to deal with added complexity. As we saw with union-find algorithms, however, we can sometimes reap huge savings with just a few lines of code. Users of a surprising number of computer systems lose substantial time waiting for simple quadratic algorithms to finish solving a problem, even though N log N or linear algorithms are available that are only slightly more complicated and could therefore solve the problem in a fraction of the time. When we are dealing with huge problem sizes, we have no choice but to seek a better algorithm, as we shall see.

Perhaps the second most common mistake made in selecting an algorithm is to pay too much attention to performance characteristics. Improving the running time of a program by a factor of 10 is inconsequential if the program takes only a few microseconds. Even if a program takes a few minutes, it may not be worth the time and effort required to make it run 10 times faster, particularly if we expect to use the program only a few times. The total time required to implement and debug an improved algorithm might be substantially more than the time required simply to run a slightly slower one-we may as well let the computer do the work. Worse, we may spend a considerable amount of time and effort implementing ideas that should improve a program but actually do not do so.

We cannot run empirical tests for a program that is not yet written, but we can analyze properties of the program and estimate the potential effectiveness of a proposed improvement. Not all putative improvements actually result in performance gains, and we need to understand the extent of the savings realized at each step. Moreover, we can include parameters in our implementations and use analysis to help us set the parameters. Most important, by understanding the fundamental properties of our programs and the basic nature of the programs' resource usage, we have the potential to evaluate their effectiveness on computers not yet built and to compare them against new algorithms not yet designed. In this we outline our methodology for developing a basic understanding of algorithm performance.

Algorithm Analysis

In this section, we outline the framework within which mathematical analysis can play a role in the process of comparing the performance of algorithms

and thus lay a foundation for us to be able to consider basic analytic results as they apply to the fundamental algorithms that we consider throughout the book. We shall consider the basic mathematical tools that are used in the analysis of algorithms, both to allow us to study classical analyses of fundamental algorithms and to make use of results from the research literature that help us understand the performance characteristics of our algorithms.

What is an algorithm and why do we want to analyze one? An algorithm is "a...step-by-step procedure for accomplishing some end". We are interested in algorithms which have been precisely specified using an appropriate mathematical formalism--such as a programming language.

Given such an expression of an algorithm, what can we do with it? Well, obviously we can run the program and observe its behavior. This is not likely to be very useful or informative in the general case. If we run a particular program on a particular computer with a particular set of inputs, then all know is the behavior of the program in a single instance. Such knowledge is anecdotal and we must be careful when drawing conclusions based upon anecdotal evidence.

In order to learn more about an algorithm, we can "analyze" it. By this we mean to study the specification of the algorithm and to draw conclusions about how the implementation of that algorithm--the program--will perform in general. But what can we analyze? We can

- determine the running time of a program as a function of its inputs;
- determine the total or maximum memory space needed for program data;
- determine the total size of the program code;
- determine whether the program correctly computes the desired result;
- determine the complexity of the program – e.g., how easy is it to read, understand, and modify; and,
- determine the robustness of the program – e.g., how well does it deal with unexpected or erroneous inputs?

We shall see many examples of each of these reasons throughout the book. Empirical analysis might suffice for some of these tasks, but mathematical analysis can be more informative (and less expensive!), as we shall see.

The analysis of algorithms can be challenging indeed. Some of the algorithms in this book are well understood, to the point that accurate mathematical formulas are known that can be used to predict running time in practical situations. People develop such formulas by carefully studying the program to find the running time in terms of fundamental mathematical quantities and then doing a mathematical analysis of the quantities involved. On the other hand, the performance properties of other algorithms in this book are not fully understood-perhaps their analysis leads to unsolved mathematical questions, or perhaps known implementations are too complex for a detailed analysis to

be reasonable, or (most likely) perhaps the types of input that they encounter cannot be characterized accurately.

Several important factors in a precise analysis are usually outside a given programmer's domain of influence. First, C++ programs are translated into bytecode, and the bytecode is interpreted or translated into runtime code on a virtual machine (VM). The compiler, translator, and VM implementations all have an effect on which instructions on an actual machine are executed, so it can be a challenging task to figure out exactly how long even one C++ statement might take to execute. In an environment where resources are being shared, even the same program can have varying performance characteristics at two different times. Second, many programs are extremely sensitive to their input data, and performance might fluctuate wildly depending on the input. Third, many programs of interest are not well understood, and specific mathematical results may not be available. Finally, two programs might not be comparable at all: one may run much more efficiently on one particular kind of input and the second may run efficiently under other circumstances.

All these factors notwithstanding, it is often possible to predict precisely how long a particular program will take, or to know that one program will do better than another in particular situations. Moreover, we can often acquire such knowledge by using one of a relatively small set of mathematical tools. It is the task of the algorithm analyst to discover as much information as possible about the performance of algorithms; it is the task of the programmer to apply such information in selecting algorithms for particular applications. In this and the next several sections, we concentrate on the idealized world of the analyst. To make effective use of our best algorithms, we need to be able to step into this world, on occasion.

The first step in the analysis of an algorithm is to identify the abstract operations on which the algorithm is based in order to separate the analysis from the implementation. Thus, for example, we separate the study of how many times one of our union-find implementations executes the code fragment i = a[i] from the analysis of how many nanoseconds might be required to execute that particular code fragment on our computer. We need both these elements to determine the actual running time of the program on a particular computer. The former is determined by properties of the algorithm; the latter by properties of the computer. This separation often allows us to compare algorithms in a way that is independent of particular implementations or of particular computers.

Although the number of abstract operations involved can be large, in principle, the performance of an algorithm typically depends on only a few quantities, and typically the most important quantities to analyze are easy to identify. One way to identify them is to use a profiling mechanism (a mechanism avail-

able in many C++ implementations that gives instruction-frequency counts) to determine the most frequently executed parts of the program for some sample runs. Or, like the union-find algorithms of our implementation might be built on a few abstract operations. In either case, the analysis amounts to determining the frequency of execution of a few fundamental operations. Our modus operandi will be to look for rough estimates of these quantities, secure in the knowledge that we can undertake a fuller analysis for important programs when necessary. Moreover, as we shall see, we can often use approximate analytic results in conjunction with empirical studies to predict performance accurately.

We also have to study the data and to model the input that might be presented to the algorithm. Most often, we consider one of two approaches to the analysis: we either assume that the input is random and study the average-case performance of the program, or we look for perverse input and study the worst-case performance of the program. The process of characterizing random inputs is difficult for many algorithms, but for many other algorithms it is straightforward and leads to analytic results that provide useful information. The average case might be a mathematical fiction that is not representative of the data on which the program is being used, and the worst case might be a bizarre construction that would never occur in practice, but these analyses give useful information on performance in most cases. For example, we can test analytic results against empirical results. If they match, we have increased confidence in both; if they do not match, we can learn about the algorithm and the model by studying the discrepancies.

In the next three sections, we briefly survey the mathematical tools that we shall be using throughout the book. This material is outside our primary narrative thrust, and readers with a strong background in mathematics or readers who are not planning to check our mathematical statements on the performance of algorithms in detail may wish to skip to and to refer back to this material when warranted later in the book. The mathematical underpinnings that we consider, however, are generally not difficult to comprehend, and they are too close to core issues of algorithm design to be ignored by anyone wishing to use a computer effectively.

First, in, we consider the mathematical functions that we commonly need to describe the performance characteristics of algorithms. Next, in, we consider the O-notation, and the notion of is proportional to, which allow us to suppress detail in our mathematical analyses. Then, in, we consider recurrence relations, the basic analytic tool that we use to capture the performance characteristics of an algorithm in a mathematical equation. Following this survey, we consider examples where we use the basic tools to analyze specific algorithms, in.

2.3 Growth of Functions

Most algorithms have a primary parameter N that affects the running time most significantly. The parameter N might be the degree of a polynomial, the size of a file to be sorted or searched, the number of characters in a text string, or some other abstract measure of the size of the problem being considered: it is most often directly proportional to the size of the data set being processed. When there is more than one such parameter we often reduce the analysis to just one parameter by expressing one of the parameters as a function of the other or by considering one parameter at a time (holding the other constant) so that we can restrict ourselves to considering a single parameter N without loss of generality. Our goal is to express the resource requirements of our programs (most often running time) in terms of N, using mathematical formulas that are as simple as possible and that are accurate for large values of the parameters. The algorithms in this book typically have running times proportional to one of the following functions.

1

Most instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that the program's running time is constant.

log N

When the running time of a program is logarithmic, the program gets slightly slower as N grows. This running time commonly occurs in programs that solve a big problem by transformation into a series of smaller problems, cutting the problem size by some constant fraction at each step. For our range of interest, we can consider the running time to be less than a large constant. The base of the logarithm changes the constant, but not by much: When N is 1 thousand, log N is 3 if the base is 10, or is about 10 if the base is 2; when N is 1 million, log N is only double these values. Whenever N doubles, log N increases by a constant, but log N does not double until N increases to N^2.

N

When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. When N is 1 million, then so is the running time. Whenever N doubles, then so does the running time. This situation is optimal for an algorithm that must process N inputs (or produce N outputs).

N log N

The N log N running time arises when algorithms solve a problem by breaking it up into smaller subproblems, solving them independently, and then combining the solutions. For lack of a better adjective (linearithmic?), we simply say that the running time of such an algorithm is N log N. When N is 1

million, N log N is perhaps 20 million. When N doubles, the running time more (but not much more) than doubles.

N^2

When the running time of an algorithm is quadratic, that algorithm is practical for use on only relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop). When N is 1 thousand, the running time is 1 million. Whenever N doubles, the running time increases fourfold.

N^3

Similarly, an algorithm that processes triples of data items (perhaps in a triple-nested loop) has a cubic running time and is practical for use on only small problems. When N is 100, the running time is 1 million. Whenever N doubles, the running time increases eightfold.

2^N

Few algorithms with exponential running time are likely to be appropriate for practical use, even though such algorithms arise naturally as brute-force solutions to problems. When N is 20, the running time is 1 million. Whenever N doubles, the running time squares!

The running time of a particular program is likely to be some constant multiplied by one of these terms (the leading term) plus some smaller terms. The values of the constant coefficient and the terms included depend on the results of the analysis and on implementation details. Roughly, the coefficient of the leading term has to do with the number of instructions in the inner loop: At any level of algorithm design, it is prudent to limit the number of such instructions. For large N, the effect of the leading term dominates; for small N or for carefully engineered algorithms, more terms may contribute and comparisons of algorithms are more difficult. In most cases, we will refer to the running time of programs simply as "linear," "N log N," "cubic," and so forth. We consider the justification for doing so in detail in.

Eventually, to reduce the total running time of a program, we focus on minimizing the number of instructions in the inner loop. Each instruction comes under scrutiny: Is it really necessary? Is there a more efficient way to accomplish the same task? Some programmers believe that the automatic tools provided by modern C++ compilers can produce the best machine code or that modern VMs will optimize program performance; others believe that the best route is to implement critical methods in native C or machine code. We normally stop short of considering optimization at this level, although we do occasionally take note of how many machine instructions are required for certain operations in order to help us understand why one algorithm might be faster than another in practice.

This table indicates the relative size of some of the functions that we en-

counter in the analysis of algorithms. The quadratic function clearly dominates, particularly for large N, and differences among smaller functions may not be as we might expect for small N. For example, $N^{3/2}$ should be greater than $N \lg^2 N$ for huge values of N, but $N \lg^2 N$ is greater for the smaller values of N that might occur in practice. A precise characterization of the running time of an algorithm might involve linear combinations of these functions. We can easily separate fast algorithms from slow ones because of vast differences between, for example, lg N and N or N and N^2, but distinguishing among fast algorithms involves careful study.

Table 2.1. Values of commonly encountered functions

Lg N	√N	N	NLg N	$N(\text{Lg } N)^2$	$N^{3/2}$	N^2
3	3	10	33	110	32	100
7	10	100	664	4414	1000	10000
10	32	1000	9966	99317	31623	1000000
13	100	10000	132877	1765633	1000000	100000000
17	316	100000	1660964	27588016	31622777	10000000000
20	1000	1000000	19931569	397267426	1000000000	1000000000000

For small problems, it makes scant difference which method we use-a fast modern computer will complete the job in an instant. But as problem size increases, the numbers we deal with can become huge. As the number of instructions to be executed by a slow algorithm becomes truly huge, the time required to execute those instructions becomes infeasible, even for the fastest computers.

Table 2.2. Seconds conversions

Number level	Time
10^2	1.7 minutes
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries
10^{11}	never

Table 2.2 gives conversion factors from large numbers of seconds to days, months, years, and so forth; Table 2.3 gives examples showing how fast algorithms are more likely than fast computers to be able to help us solve problems without facing outrageous running times.

The vast difference between numbers such as 10^4 and 10^8 is more obvious when we consider them to measure time in seconds and convert to familiar units of time. We might let a program run for 2.8 hours, but we would be unlikely to contemplate running a program that would take at least 3.1 years to complete. Because 2^{10} is approximately 10^3, this table is useful for powers of 2 as well. For example, 2^{32} seconds is about 124 years.

Table 2.3. Time to solve huge problems

operations per second	problem size 1 million			problem size 1 billion		
	N	N lg N	N^2	N	N lg N	N^2
10^6	seconds	seconds	weeks	hours	hours	never
10^9	instant	instant	hours	seconds	seconds	decades
10^{12}	instant	instant	seconds	instant	instant	weeks

For many applications, our only chance of being able to solve huge problem instances is to use an efficient algorithm. This table indicates the minimum amount of time required to solve problems of size 1 million and 1 billion, using linear, N log N, and quadratic algorithms, when we can execute 1 million, 1 billion, and 1 trillion instructions per second. A fast algorithm enables us to solve a problem on a slow machine, but a fast machine is no help when we are using a slow algorithm.

A few other functions do arise. For example, an algorithm with N^2 inputs that has a running time proportional to N^3 is best thought of as an $N^{3/2}$ algorithm. Also, some algorithms have two stages of sub problem decomposition, which lead to running times proportional to $N \log^2 N$. It is evident from Table 2.1 that both of these functions are much closer to N log N than to N^2.

The logarithm function plays a special role in the design and analysis of algorithms, so it is worthwhile for us to consider it in detail. Because we often deal with analytic results only to within a constant factor, we use the notation "log N" without specifying the base. Changing the base from one constant to another changes the value of the logarithm by only a constant factor, but specific bases normally suggest themselves in particular contexts. In mathematics, the natural logarithm (base e = 2.71828 ...) is so important that a special abbreviation is commonly used: $\log_e N = \ln N$. In computer science, the binary logarithm (base 2) is so important that the abbreviation $\log_2 N = \lg N$ is commonly used.

The smallest integer larger than lg N is the number of bits required to represent N in binary, in the same way that the smallest integer larger than $\log_{10} N$ is the number of digits required to represent N in decimal. The C++ statement

$$\text{for (lg N = 0; N > 0; lg N++, N /= 2) ;} \tag{2.1}$$

is a simple way to compute the smallest integer larger than lg N. A similar method for computing this function is

$$\text{for (lgN = 0, t = 1; t < N; lgN++, t += t) ;} \tag{2.2}$$

This version emphasizes that $2^n \leq N < 2^{n+1}$ when n is the smallest integer larger than lg N.

Occasionally, we iterate the logarithm: We apply it successively to a huge number. For example, lg lg 2^{256} = lg 256 = 8. As illustrated by this example, we generally regard log log N as a constant, for practical purposes, because it is so small, even when N is huge.

We also frequently encounter a number of special functions and mathematical notations from classical analysis that are useful in providing concise descriptions of properties of programs. Table 2.4 summarizes the most familiar of these functions; we briefly discuss them and some of their most important properties in the following paragraphs.

Our algorithms and analyses most often deal with discrete units, so we often have need for the following special functions to convert real numbers to integers:

[x]: largest integer less than or equal to x

[x]: smallest integer greater than or equal to x.

For example, [π]: and [e] are both equal to 3, and [lg(N +1)] is the number of bits in the binary representation of N. Another important use of these functions arises when we want to divide a set of N objects in half. We cannot do so exactly if N is odd, so, to be precise, we divide into one subset with [N/2 objects and another subset with [N/2] objects. If N is even, the two subsets are equal in size ([N/2] = N/2); if N is odd, they differ in size by 1 ([N/2] + 1 = [N/2]). In C++, we can compute these functions directly when we are operating on integers (for example, if N $\geq$0, then [N/2] is [N/2] and N – (N/2) is [N/2]), and we can use floor and ceil from the C++. lang. Math package to compute them when we are operating on floating point numbers.

A discretized version of the natural logarithm function called the harmonic numbers often arises in the analysis of algorithms. The Nth harmonic number is defined by the equation

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N} \tag{2.3}$$

The natural logarithm ln N is the area under the curve 1/x between 1 and N; the harmonic number H_N is the area under the step function that we define by evaluating 1/x at the integers between 1 and N. This relationship is illustrated in Figure 2.1. The formula

$$H_N \approx lnN + \gamma + 1/(12N) \tag{2.4}$$

where γ = 0.57721 ... (this constant is known as Euler's constant) gives an excellent approximation to H_N. By contrast with [lg N] and [lg N], it is better to use the log method of C++ to compute H_N than to do so directly from the definition.

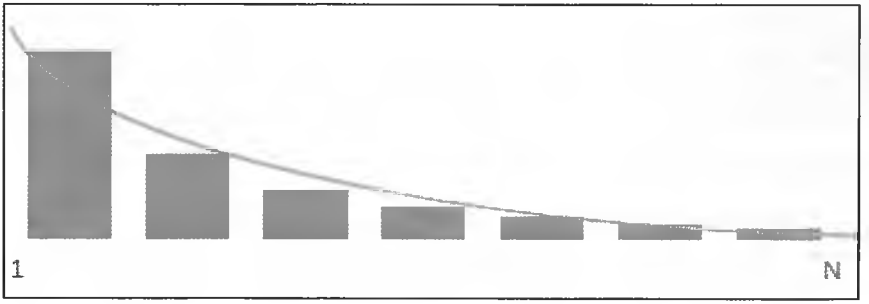


Figure 2.1. Harmonic numbers

The harmonic numbers are an approximation to the area under the curve y = 1/x. The constant γ accounts for the difference between H_N and $\ln N = \int_1^N \frac{dx}{x}$

The sequence of numbers: 0 1 1 2 3 5 8 13 21 34 ..., that are defined by the formula

$$F_N = F_{N-1} + F_{N-2}, \text{ for } N \geq 2 \text{ with } F_0 = 0 \text{ and } F_1 = 1 \tag{2.5}$$

We also have occasion to manipulate the familiar factorial function N!. Like the exponential function, the factorial arises in the brute-force solution to problems and grows much too fast for such solutions to be of practical interest. It also arises in the analysis of algorithms because it represents all the ways to arrange N objects.

This table summarizes the mathematical notation that we use for functions

and constants that arise in formulas describing the performance of algorithms. The formulas for the approximate values extend to provide much more accuracy, if desired (see reference section).

In this text, we are concerned primarily with the running time. We also consider the memory space needed to execute the program. There are many factors that affect the running time of a program. Among these are the algorithm itself, the input data, and the computer system used to run the program. The performance of a computer is determined by the hardware:

- processor used (type and speed),
- memory available (cache and RAM), and disk available;
- the programming language in which the algorithm is specified;
- the language compiler/interpreter used; and
- the computer operating system software.

A detailed analysis of the performance of a program which takes all of these factors into account is a very difficult and time-consuming undertaking. Furthermore, such an analysis is not likely to have lasting significance. The rapid pace of change in the underlying technologies means that results of such analyses are not likely to be applicable to the next generation of hardware and software.

In order to overcome this shortcoming, we devise a "model" of the behavior of a computer with the goals of simplifying the analysis while still producing meaningful results. The next section introduces the first in a series of such models.

A Detailed Model of the Computer

In this section we develop a detailed model of the running time performance of C++ programs. The model developed is independent of the underlying hardware and system software. Rather than analyze the performance of a particular, arbitrarily chosen physical machine, we consider the implementation of the C++ programming language as a kind a "virtual C++ machine".

A direct consequence of this approach is that we lose some fidelity--the resulting model will not be able to accurately predict the performance of all possible hardware/software systems. On the other hand, we shall see that the resulting model is still rather complex and rich in detail.

The Basic Axioms

The running time performance of the C++ virtual machine is given by a set

of axioms which we shall now postulate. For now we consider only operations on integers. The first axiom addresses the running time of simple variable references:

Axiom-1. The time required to fetch an integer operand from memory is a constant, T_{fetch} , and the time required to store an integer result in memory is a constant, T_{store}.

According to Axiom T_{fetch}, the assignment statement

$$y = x; \tag{2.6}$$

Has running time $T_{fetch}+ T_{store}$. I.e., the time taken to fetch the value of variable x is T_{fetch} and the time taken to store the value in variable y is T_{store}.

We shall apply Axiom to manifest constants too: The assignment

$$Y = 1; \tag{2.7}$$

Also has running time $T_{fetch}+ T_{store}$. To see why this should be the case, consider that the constant typically needs to be stored in the memory of the computer, and we can expect the cost of fetching it to be the same as that of fetching any other operand.

The next axiom addresses the running time of simple arithmetic operations on integers:

Axiom-2. The times required to perform elementary operations on integers, such as addition, subtraction, multiplication, division, and comparison, are all constants. These times are denoted by

$$T+T\text{-}TxT\div, \text{ and } T < \text{ respectively.} \tag{2.8}$$

According to Axiom (1, 2), all the simple operations on integers can be accomplished in a fixed amount of time. In order for this to be feasible, the number of bits used to represent an integer must be fixed. Typically, between 16 and 64 bits are used to represent an integer. It is precisely because the number of bits used is fixed that we can say that the running times are also fixed. If arbitrarily large integers are allowed, then the basic arithmetic operations can take an arbitrarily long amount of time.

By applying Axioms, we can determine that the running time of a statement like

$$Y = y + 1; \tag{2.9}$$

Is $2T_{fetch} + T++ T_{store}$. This is because we need to fetch two operands, y and 1; add them; and, store the result back in y.

C++ syntax provides several alternative ways to express the same computation:

```
y += 1;
++y;
y++;
```
(2.10)

We shall assume that these alternatives require exactly the same running time as the original statement.

The third basic axiom addresses the function call/return overhead:

Axiom-3. The time required to call a function is a constant, T_{call}, and the time required to return from a function is a constant, T_{return}.

When a function is called, certain housekeeping operations need to be performed. Typically this includes saving the return address so that program execution can resume at the correct place after the call, saving the state of any partially completed computations so that they may be resumed after the call, and the allocation of a new execution context (stack frame or activation record) in which the called function can be evaluated. Conversely, on the return from a function, all of this work is undone. While the function call/return overhead may be rather large, nevertheless it entails a constant amount of work.

In addition to the function call/return overhead, additional overhead is incurred when parameters are passed to the function:

Axiom-4. The time required to pass an integer argument to a function or procedure is the same as the time required to store an integer in memory, Tstore.

The rationale for making the overhead associated with parameter passing the same as the time to store a value in memory is that the passing of an argument is conceptually the same as assignment of the actual parameter value to the formal parameter of the function.

According to Axiom-1, the running time of the statement

```
y = f (x);
```
(2.11)

would be $T_{fetch} + 2\ T_{store} + T_{call} + T_f(x)$, where $T_f(x)$ is the running time of function f for input x. The first of the two stores is due to the passing of the parameter x to the function f; the second arises from the assignment to the variable y.

Average Running Times

Suppose we are interested in a function $T_{average}(n)$ which gives the running time on average for n inputs, regardless of the values of those inputs. In other

words, if we run the program, a large number of times on a selection of random inputs of length n, what will the average running time be?

We can write the sum of the running times given in the following form

$$T_{average}(n) = t1+t2n+ \quad (2.12)$$

where pi is the probability that all line of the program is executed. The probability *pi* is given by

$$pi = P[a_j > (\max a_j)] \quad (2.13)$$

I.e., pi is the probability that the (i) array entry, , is larger than the maximum of all the preceding array entries, a0,a1,...ai-1.

In order to determine pi, we need to know (or to assume) something about the distribution of input values. For example, if we know a priori that the array passed to the function FindMaximum is ordered from smallest to largest, then we know that pi=1. Conversely, if we know that the array is ordered from largest to smallest, then we know that pi=0.

In the general case, we have no a priori knowledge of the distribution of the values in the input array. In this case, consider the (i) iteration of the loop. In this iteration is compared with the maximum of the I values, a0,a1,...ai-1 preceding it in the array. Line of Program is only executed if is the largest of the i+1 values a0,a1,...ai-1. All things being equal, we can say that this will happen with probability 1/(i+1). Thus

$$Pi = P[a_i > (\max a_i)] = \frac{1}{i+1}. \quad (2.14)$$

Substituting this expression for pi in Equation and simplifying the result we get

$$T_{average}(n) = t1+t2n+\sum_{i=1}^{n-1} p_i t_3 = t1+t2n+t_3(Hn-1) \quad (2.15)$$

where $Hn = \sum_{i=1}^{n} \frac{1}{i}$, is the harmonic number.

Best-Case and Worst-Case Running Times

In reasoning's above, we derived the average running time of Program which finds the largest element of an array. In order to do this we had to determine the probability that a certain program statement is executed. To do this, we made an assumption about the average input to the program.

The analysis can be significantly simplified if we simply wish to deter-

mine the worst case running time. For Program, the worst-case scenario occurs when line worst is executed in every iteration of the loop. We saw that this corresponds to the case in which the input array is ordered from smallest to largest. In terms of Equation, this occurs when pi=1. Thus, the worst-case running time is given by

$$T_{worstcase}(n) = (t1-t3)+ (t2+t3) \text{ x } n. \quad (2.16)$$

Similarly, the best-case running time occurs when line is never executed. This corresponds to the case in which the input array is ordered from largest to smallest. This occurs when pi=0 and best-case running time is

$$T_{bestcase}(n) = t1+ t2\ n. \quad (2.17)$$

The Last Axiom

In this section we state the last axiom we need for our detailed model of the C++ virtual machine. This axiom addresses the time required to allocate and to free dynamically allocated storage:

Axiom-4. The time required to allocate a fixed amount of storage from the heap using operator new is a constant, Tnew. This time does not include any time required for initialization of the storage. Similarly, the time required to return a fixed amount of storage to the heap using operator delete is a constant, Tdelete. This time does not include any time spent cleaning up the storage before it is returned to the heap.

By applying Axioms-3 and Axioms-4, we can determine that the running time of the statement

```
int* ptr = new int;
```
(2.18)

is $T_{new}+ T_{delete}$. Similarly, the running time of delete ptr;

2.4 Asymptotic Notation and Analysis

An Asymptotic Upper Bound-Big Oh

Suppose we are considering two algorithms, A and B, for solving a given problem. Furthermore, let us say that we have done a careful analysis of the

running times of each of the algorithms and determined them to be TA(n) and TB(n), respectively, where n is a measure of the problem size. Then it should be a fairly simple matter to compare the two functions TA(n) and TB(n) to determine which algorithm is the best!

But is it really that simple? What exactly does it mean for one function, say TA(n), to be better than another function, TB(n)? One possibility arises if we know the problem size a priori. E.g., suppose the problem size is n0 and TA(n0)<TB(n0). Then clearly algorithm A is better than algorithm B for problem size n0.

In the general case, we have no a priori knowledge of the problem size. However, if it can be shown, say, that $\Theta n \geq 0$: TA(n)≤TB(n), then algorithm A is better than algorithm B regardless of the problem size.

Unfortunately, we usually don't know the problem size beforehand, nor is it true that one of the functions is less than or equal the other over the entire range of problem sizes. In this case, we consider the asymptotic behavior of the two functions for very large problem sizes.

In 1892, P. Bachmann invented a notation for characterizing the asymptotic behavior of functions. His invention has come to be known as big oh notation:

Definition (Big Oh). Consider a function f(n) which is non-negative for all integers n≥0. We say that "f(n) is big oh g(n)," which we write f(n)=O(g(n)), if there exists an integer n0 and a constant c>0 such that for all integers n≥n0, f(n)≤cg(n).

A Simple Example

Consider the function f(n)=8n+128 shown in Figure 2.2. Clearly, f(n) is non-negative for all integers n≥0. We wish to show that f(n)=O(n2). According to Definition (Big Oh), in order to show this we need to find an integer n0 and a constant c>0 such that for all integers n≥n0, f(n)≤cn2.

It does not matter what the particular constants are – as long as they exist! E.g., suppose we choose c=1. Then

$$F(n) \leq cn2 \Rightarrow 0 \leq (n-16)(n+8). \qquad (2.19)$$

Since (n+8)>0 for all values of n≥0, we conclude that (n0 -16)≥0. I.e., n0 =16.

So, we have that for c=1 and n0 =16, f(n)≤cn2 for all integers n≥n0. Hence, $f(n)=O(n^2)$.

Figure 2.2 clearly shows that the function $f(n) = n^2$ is greater than the function f(n)=8n+128 to the right of n=16.

Of course, there are many other values of c and n_0 that will do. For example, c=2 and $n_0 = 2+4\sqrt{17} \approx 10.2$ will do, as will c=4 and $n_0 = 1+\sqrt{33} \approx 6.7$ (See Figure 2.2).

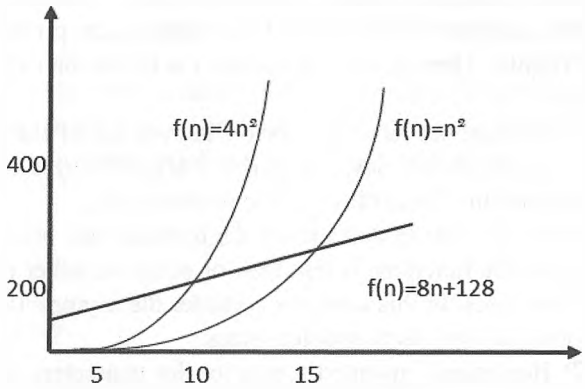


Figure 2.2. Showing that f(n)=8n+128=O(n2)

Physical vs. Logical Running Time

Asymptotic analysis is the determination of the amount of resources used by an algorithm. Usually, this resource is the running time of the algorithm. The running time of a program is above all a physical quantity. In principle, we can use a stopwatch and determine how long the program takes to execute. In fact, all computer systems support an internal clock, and one can measure the running time even without resorting to an external timing device.

The first thing to note is that running time is not a fixed quantity; rather, the running time in almost all programs depends on the particular input given to the program. For example, a sorting algorithm will obviously take longer on a list of one million numbers than on a list of one hundred numbers. Hence, we really need to express running time as a function of input. This is much too tedious, and in practice, it is quite impossible to describe the running time of a program for all conceivable inputs. As the example of sorting algorithms shows, one can lump together a whole class of inputs, such as a list of a given length. Note also, though, that a sorting algorithm may behave quite differently when presented with an already sorted list, a reverse sorted list, or a random list. The running times may differ quite drastically. Nonetheless, it is custom-

ary to refer to the running time of a program on input of size n, meaning the worst possible running time for all such inputs. This measure is called worst case complexity. Alternatively, one can analyze the average amount of time it takes to process input of a certain size, which is called the average case complexity. Unfortunately, the latter is much harder to deal with than the former, so we will focus on worst case complexity.

Another problem with measuring running time is that it obviously depends very much on the machine used to execute the program. Additionally, it depends on the compiler used, the level of optimization, the load of the computer, and so on. In order to avoid these numerous complications, one uses a different measure: logical running time, which is the number of steps in the execution of the algorithm. By "one step," we mean essentially one CPU cycle. Assignments of built-in types, arithmetic operations, logical tests, and so on can all be performed in "one step." Filling an array with zeros, on the other hand, will take as many steps as the size of the array.

Counting the number of steps produces a good and helpful measure of running time. Nonetheless, if an algorithm is to be executed on a specific computer, it is still necessary to perform measurements of the physical running time in a few cases. In conjunction with the theoretical analysis, one can then predict the real running time on arbitrary inputs fairly accurately.

Determining the Logical Running Time

As one might imagine, counting the number of steps is a daunting task in all but the most simple-minded programs. It is important to ignore small details and look at the large-scale behavior. As an example, consider the selection sort algorithm.

Setting up the local variables takes a small, constant number of steps. The body of the outer loop is executed v.size() - 1 times, where v.size() is the number of vector elements. The number of steps performed by the inner loop decreases with each iteration of the outer loop. For the sake of this discussion, we will let n equal the number of elements in the vector. During its first execution, the inner loop iterates (n - 1) times.

During its second execution (when i equals 1), the inner loop iterates (n - 2) times. The body of this inner loop contains a single if-statement. This statement can be executed in a constant amount of time. Hence, we have the following sum that represents the number of operations performed by selection sort. Each term in this sum represents the number of steps performed by each iteration of the outer loop.

Example 1.
The number of steps performed by selection sort

$$(n - 1) + (n - 2) + (n - 3) + ... + 3 + 2 + 1 \qquad (2.20)$$

The sum of this expression equals (n2 - n) / 2. We summarize this result by saying that selection sort is a quadratic algorithm. Note that quadratic means that a doubling in input size will result in a four-fold increase in running time. A ten times larger input will take a hundred times longer to process. In other words, quadratic algorithms do not scale up very well to large inputs.

The opposite of quadratic algorithms are logarithmic algorithms. In logarithmic algorithms, the number of steps required to solve a problem does not increase significantly even when the number of inputs increases dramatically. One example of a logarithmic algorithm is a binary search. In a binary search, the items to search (the input) are halved during each iteration. This allows a binary search to work very well even on very large data sets.

Big-Oh Notation

The mathematical artifact that allows us to suppress detail when we are analyzing algorithms is called the O-notation, or "big-Oh notation," which is defined as follows.

Definition. A function g(N) is said to be O(f(N)) if there exist constants c_0 and N_0 such that $g(N) < c_0 f(N)$ for all $N > N_0$.

We use the O-notation for three distinct purposes:

- To bound the error that we make when we ignore small terms in mathematical formulas
- To bound the error that we make when we ignore parts of a program that contribute a small amount to the total being analyzed
- To allow us to classify algorithms according to upper bounds on their total running times

We consider the third use and discuss briefly the other two here.

The constants c_0 and N_0 implicit in the O-notation often hide implementation details that are important in practice. Obviously, saying that an algorithm has running time O(f(N)) says nothing about the running time if N happens to be less than N_0, and c_0 might be hiding a large amount of overhead designed to avoid a bad worst case. We would prefer an algorithm using N^2 nanoseconds

over one using log N centuries, but we could not make this choice on the basis of the O-notation.

Often, the results of a mathematical analysis are not exact but rather are approximate in a precise technical sense: The result might be an expression consisting of a sequence of decreasing terms. Just as we are most concerned with the inner loop of a program, we are most concerned with the leading terms (the largest terms) of a mathematical expression. The O-notation allows us to keep track of the leading terms while ignoring smaller terms when manipulating approximate mathematical expressions and ultimately allows us to make concise statements that give accurate approximations to the quantities that we analyze.

That is, N^2 is a good approximation to this expression when N is large. These manipulations are intuitive, but the O-notation allows us to express them mathematically with rigor and precision. We refer to a formula with one O-term as an asymptotic expression.

For a more relevant example, suppose that (after some mathematical analysis) we determine that a particular algorithm has an inner loop that is iterated $2NH_N$ times on the average, an outer section that is iterated N times, and some initialization code that is executed once. Suppose further that we determine (after careful scrutiny of the implementation) that each iteration of the inner loop requires a_0 nanoseconds, the outer section requires a_1 nanoseconds, and the initialization part a_2 nanoseconds.

This simpler form is significant because it says that, for large N, we may not need to find the values of a_1 or a_2 to approximate the running time. In general, there could well be many other terms in the mathematical expression for the exact running time, some of which may be difficult to analyze. The O-notation provides us with a way to get an approximate answer for large N without bothering with such terms.

Continuing this example, we also can use the O-notation to express running time in terms of a familiar function, ln N. In terms of the O-notation, the approximation in Table 2.4 is expressed as $H_N = \ln N + O(1)$. Thus, $2a_0N \ln N + O(N)$ is an asymptotic expression for the total running time of our algorithm. We expect the running time to be close to the easily computed value $2a_0N \ln N$ for large N. The constant factor a_0 depends on the time taken by the instructions in the inner loop.

Furthermore, we do not need to know the value of a_0 to predict that the running time for input of size 2N will be about twice the running time for input of size N for huge N because.

That is, the asymptotic formula allows us to make accurate predictions without concerning ourselves with details of either the implementation or the analysis. Note that such a prediction would not be possible if we were to have only an O-approximation for the leading term.

The kind of reasoning just outlined allows us to focus on the leading term when comparing or trying to predict the running times of algorithms. We are so often in the position of counting the number of times that fixed-cost operations are performed and wanting to use the leading term to estimate the result that we normally keep track of only the leading term, assuming implicitly that a precise analysis like the one just given could be performed, if necessary.

When a function f(N) is asymptotically large compared to another function g(N) (that is, g(N)/f(N) 0 as N), we sometimes use in this book the (decidedly nontechnical) terminology about f(N) to mean f(N) + O(g(N)). What we seem to lose in mathematical precision we gain in clarity, for we are more interested in the performance of algorithms than in mathematical details. In such cases, we can rest assured that, for large N (if not for all N), the quantity in question will be close to f(N). For example, even if we know that a quantity is N(N - 1)/2, we may refer to it as being about $N^2/2$. This way of expressing the result is more quickly understood than the more detailed exact result and, for example, deviates from the truth only by 0.1 percent for N = 1000. The precision lost in such cases pales by comparison with the precision lost in the more common usage O(f(N)). Our goal is to be both precise and concise when describing the performance of algorithms.

In a similar vein, we sometimes say that the running time of an algorithm is proportional to f(N) when we can prove that it is equal to cf(N) + g(N) with g(N) asymptotically smaller than f(N). When this kind of bound holds, we can project the running time for, say, 2N from our observed running time for N, as in the example just discussed. Figure 2.3 gives the factors that we can use for such projection for functions that commonly arise in the analysis of algorithms. Coupled with empirical studies this approach frees us from the task of determining implementation-dependent constants in detail. Or, working backward, we often can easily develop an hypothesis about the functional growth of the running time of a program by determining the effect of doubling N on running time.

Big-Oh notation provides a good way to compare the running time of algorithms. In asymptotic analysis, we use Big-Oh notation to refer to the highest order, or most dominant term from a function's runtime analysis.

The most important property of Big-Oh notation is that it allows one to ignore lower order terms and constants, as in the following example.

Example 2
Using Big-Oh notation

$$500n^3 + 10n^2 + 17n + 121345 = O(n^3) \quad (2.21)$$

In the above example, the highest order term in the runtime equation is $500n^3$. As we did the lower order terms, we can also ignore the constant in this term and state that the running time is $O(n^3)$.

Most algorithms one encounters fall into one of the following classes.

O(1) – constant time
O(log(n)) – logarithmic time
O(n) – linear time
O(n log(n)) – "n-log-n" time
$O(n^2)$ – quadratic time
$O(n^3)$ – cubic time
$O(2^n)$ – exponential time

Algorithms that have running time up to O(n log(n)) scale very well to large problem instances. Quadratic algorithms and cubic algorithms even more so, show severely degraded performance when the input becomes large. Lastly, exponential algorithms can be used only on very small inputs.

It is important to express logical running time in terms of physical time, using familiar units such as seconds, minutes, hours, and so forth to get a feel for what these characterizations really mean. First, let's consider an O(n log(n)) algorithm that takes 1 second on an input of size 1000.

Table 2.4. An O(n log(n)) algorithm that takes 1 second on an input of size 1000

input size	**running time**
1000	1 second
2000	2.2 seconds
5000	6.2 seconds
10000	13.3 seconds
100000	2.77 minutes
106	33.3 minutes
107	6.48 hours

By comparison, here is a quadratic algorithm that also takes 1 second on an input of size 1000.

Table 2.5. An $O(n^2)$ algorithm that takes 1 second on an input of size 1000

input size	running time
1000	1 second
2000	4 seconds
5000	25 seconds
10000	1.66 minutes
100000	2.77 hours
106	11.5 days
107	3.25 years

For very large inputs of size ten million, the quadratic algorithm becomes entirely useless. Even for one million, it may be useless: suppose it is part of some bank accounting system that needs to be updated on a daily basis. On the other hand, quadratic performance is superb when compared with exponential running time. To see how really catastrophic exponential running time is, suppose that an exponential algorithm takes 1 second on input of size 10. We then can expect approximately the following running times for larger instances.

Table 2.6. An $O(2^n)$ algorithm that takes 1 second on an input of size 1000

input size	running time
10	1 seconds
15	32 seconds
20	17.1 minutes
25	9.1 hours
30	12.1 days
35	1.09 years
40	34.9 years
50	35700 years
100	4.02 1019 years

The last figure greatly exceeds the age of the universe (about 15 x 109

years), and the algorithm is already useless at inputs of size 30. We only have acceptable performance up to size 25 or so. Thus, in some sense, an exponential algorithm is almost as bad as no algorithm at all.

Unfortunately, there are some famous computational problems that seem to require exponential running times. Finding out whether a Boolean circuit with n inputs can ever yield true as output is one of them: there are 2n possible input combinations, and no shortcuts are known to simplify the computation. In practice, one often has to deal with circuits of several hundred inputs.

Basic Rules

How does one determine the asymptotic running time of an algorithm? Since we ignore constants, any sequence of basic statements, such as 1) assignments of built-in types or 2) operations on built-in types (addition, multiplication, comparison, increment, decrement, Boolean operations, etc.), takes constant time (O(1)).

Note, though, that function calls have to be accounted for separately. First, there is the cost of organizing the function call. If only reference parameters are used, that cost is O(1). But, if call-by-value is used, we have to count the cost of copying the input. The same comment applies to returning the result of the computation. And, of course, we have to count the evaluation of the function body.

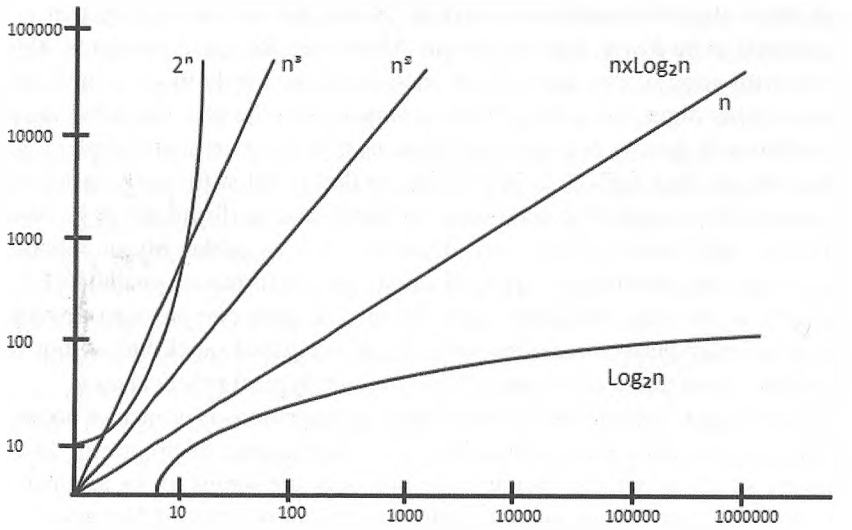


Figure 2.3. Some properties of growth-rate functions

2.5 Guarantees, Predictions, and Limitations

The running time of most algorithms depends on their input data. Typically, our goal in the analysis of algorithms is somehow to eliminate that dependence: We want to be able to say something about the performance of our programs that depends on the input data to as little an extent as possible, because we generally do not know what the input data will be each time the program is invoked.

Studying the worst-case performance of algorithms is attractive because it allows us to make guarantees about the running time of programs. We say that the number of times certain abstract operations are executed is less than a certain function of the number of inputs, no matter what the input values are. If the guarantees are low, as is the case with binary search, then we are in a favorable situation, because we have eliminated cases for which our program might run slowly. Programs with good worst-case performance characteristics are a basic goal in algorithm design.

There are several difficulties with worst-case analysis, however. For a given algorithm, there might be a significant gap between the time required for it to solve a worst-case instance of the input and the time required for it to solve the data that it might encounter in practice. For example, quick union requires time proportional to N in the worst case, but only log N for typical data. More important, we cannot always prove that there is an input for which the running time of an algorithm achieves a certain bound; we can prove only that it is guaranteed to be lower than the bound. Moreover, for some problems, algorithms with good worst-case performance are significantly more complicated than are other algorithms. We often find ourselves in the position of having an algorithm with good worst-case performance that is slower than simpler algorithms for the data that occur in practice, or that is not sufficiently faster that the extra effort required to achieve good worst-case performance is justified. For many applications, other considerations-such as portability or reliability-are more important than improved worst-case performance guarantees. For example, as we saw, weighted quick union with path compression provides provably better performance guarantees than weighted quick union, but the algorithms have about the same running time for typical practical data.

Studying the average-case performance of algorithms is attractive because it allows us to make predictions about the running time of programs. In the simplest situation, we can characterize precisely the inputs to the algorithm; for example, a sorting algorithm might operate on an array of N random integers, or a geometric algorithm might process a set of N random points in the plane with coordinates between 0 and 1. Then, we calculate the average number of times that each instruction is executed and calculate the average

running time of the program by multiplying each instruction frequency by the time required for the instruction and adding them all together.

There are also several difficulties with average-case analysis, however. First, the input model may not accurately characterize the inputs encountered in practice, or there may be no natural input model at all. Few people would argue against the use of input models such as "randomly ordered file" for a sorting algorithm, or "random point set" for a geometric algorithm, and for such models it is possible to derive mathematical results that can predict accurately the performance of programs running on actual applications. But how should one characterize the input to a program that processes English-language text? Even for sorting algorithms, models other than randomly ordered inputs are of interest in certain applications. Second, the analysis might require deep mathematical reasoning. For example, the average-case analysis of union-find algorithms is difficult. Although the derivation of such results is normally beyond the scope of this book, we will illustrate their nature with a number of classical examples, and we will cite relevant results when appropriate (fortunately, many of our best algorithms have been analyzed in the research literature). Third, knowing the average value of the running time might not be sufficient: we may need to know the standard deviation or other facts about the distribution of the running time, which may be even more difficult to derive. In particular, we are often interested in knowing the chance that the algorithm could be dramatically slower than expected.

In many cases, we can answer the first objection listed in the previous paragraph by turning randomness to our advantage. For example, if we randomly scramble an array before attempting to sort it, then the assumption that the elements in the array are in random order is accurate. For such algorithms, which are called randomized algorithms, the average-case analysis leads to predictions of the expected running time in a strict probabilistic sense. Moreover, we are often able to prove that the probability that such an algorithm will be slow is negligibly small.

The field of computational complexity is the branch of analysis of algorithms that helps us to understand the fundamental limitations that we can expect to encounter when designing algorithms. The overall goal is to determine the worst-case running time of the best algorithm to solve a given problem, to within a constant factor. This function is called the complexity of the problem.

Worst-case analysis using the Onotation frees the analyst from considering the details of particular machine characteristics. The statement that the running time of an algorithm is O(f(N)) is independent of the input and is a useful way to categorize algorithms in a way that is independent of both inputs and implementation details, separating the analysis of an algorithm from any particular

implementation. We ignore constant factors in the analysis; in most cases, if we want to know whether the running time of an algorithm is proportional to N or proportional to log N, it does not matter whether the algorithm is to be run on a nanocomputer or on a supercomputer, and it does not matter whether the inner loop has been implemented carefully with only a few instructions or badly implemented with many instructions.

When we can prove that the worst-case running time of an algorithm to solve a certain problem is O(f(N)), we say that f(N) is an upper bound on the complexity of the problem. In other words, the running time of the best algorithm to solve a problem is no higher than the running time of any particular algorithm to solve the problem.

We constantly strive to improve our algorithms, but we eventually reach a point where no change seems to improve the running time. For every given problem, we are interested in knowing when to stop trying to find improved algorithms, so we seek lower bounds on the complexity. For many problems, we can prove that any algorithm to solve the problem must use a certain number of fundamental operations. Proving lower bounds is a difficult matter of carefully constructing a machine model and then developing intricate theoretical constructions of inputs that are difficult for any algorithm to solve. We rarely touch on the subject of proving lower bounds, but they represent computational barriers that guide us in the design of algorithms, so we maintain awareness of them when they are relevant.

When complexity studies show that the upper bound of an algorithm matches the lower bound, then we have some confidence that it is fruitless to try to design an algorithm that is fundamentally faster than the best known, and we can start to concentrate on the implementation. For example, binary search is optimal, in the sense that no algorithm that uses comparisons exclusively can use fewer comparisons in the worst case than binary search.

We also have matching upper and lower bounds for pointer-based union-find algorithms. Tarjan showed in 1975 that weighted quick union with path compression requires following less than $O(\lg^* V)$ pointers in the worst case, and that any pointer-based algorithm must follow more than a constant number of pointers in the worst case for some input. In other words, there is no point looking for some new improvement that will guarantee to solve the problem with a linear number of i = a[i] operations. In practical terms, this difference is hardly significant, because $\lg^* V$ is so small; still, finding a simple linear algorithm for this problem was a research goal for many years, and Tarjan's lower bound has allowed researchers to move on to other problems. Moreover, the story shows that there is no avoiding functions like the rather complicated $\log^*$ function, because such functions are intrinsic to this problem.

Many of the algorithms in this book have been subjected to detailed ma-

thematical analyses and performance studies far too complex to be discussed here. Indeed, it is on the basis of such studies that we are able to recommend many of the algorithms that we discuss.

Not all algorithms are worthy of such intense scrutiny; indeed, during the design process, it is preferable to work with approximate performance indicators to guide the design process without extraneous detail. As the design becomes more refined, so must the analysis, and more sophisticated mathematical tools need to be applied. Often, the design process leads to detailed complexity studies that lead to theoretical algorithms that are rather far from any particular appli-cation. It is a common mistake to assume that rough analyses from complexity studies will translate immediately into efficient practical algorithms; such assumptions can lead to unpleasant surprises. On the other hand, computational complexity is a powerful tool that tells us when we have reached performance limits in our design work and that can suggest departures in design in pursuit of closing the gap between upper and lower bounds.

In this book, we take the view that algorithm design, careful implementation, mathematical analysis, theoretical studies, and empirical analysis all contribute in important ways to the development of elegant and efficient programs. We want to gain information about the properties of our programs using any tools at our disposal, then modify or develop new programs on the basis of that information. We will not be able to do exhaustive testing and analysis of every algorithm that we run in every programming environment on every machine, but we can use careful implementations of algorithms that we know to be efficient, then refine and compare them when peak performance is necessary. Throughout the book, when appropriate, we shall consider the most important methods in sufficient detail to appreciate why they perform well.

Data Structures and Algorithms in C++

The amount of information we encounter is sometimes so large that without an effective method of storage and representation, the information is rendered useless. Think of all the books contained in your local library. Without the storage system implemented by the librarians, it would be virtually impossible to find a specific book.

A data structure, in the simplest terms, is a structured representation of information. Consider your address as a data structure. There are several items in this data structure, all of which are simple pieces of data. First, there is the street address, which is usually a number followed by a street name. Add to that the city name, the state or province abbreviation, and the postal code, and you have a very useful data structure composed simply of numbers and words.

Data structures can also be complex in that they can contain other data structures. For example, a library contains many bookcases, which in turn contain many books.

Sitting on the shelf of one of these bookcases could be a cardboard box that also contains a few books. In this example, the bookcase is a data structure composed of books and boxes of books. The boxes are also data structures, since they store books as well. Here we have a data structure (a bookcase) that is composed of another data structure (a box). Both data structures provide a way to store information (the books) to help solve some problem (finding a specific book). A filing cabinet full of folders of papers is another good example of how data structures can be composed of other data structures. In this case, the cabinet as a whole is a data structure. The cabinet is composed of cabinet drawers, which are composed of folders, which are composed of files.

In addition to having a data structure that represents information, some sequence of actions, or series of steps, needs to be taken to solve a problem. The series of steps that manipulate and uses a data structure to solve a problem is called an algorithm. Let's think back to the example of the line at the movie theater. Here, the problem at hand is the management of a group of people waiting to buy tickets. The data structure is a line of people.

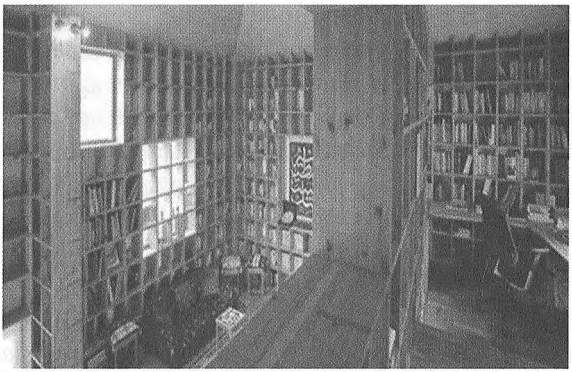


Figure 2.4. A filing cabinet as a data structure Algorithms

The algorithm involves removing a person from the beginning of the line (until the line is empty) when the ticket cashier becomes available.

- Wait for the cashier to become available.
- Remove the customer that is at the beginning of the line.
- Let the cashier help this removed customer.
- Go to step 1.

The steps contained in an algorithm are performed in a mechanical fashion. From the example above, first we do step 1, then we do step 2, then we do step 3, then we repeat with step 1. Algorithms can involve this type of repetition, and they can also involve simple branching or decision making (if "this," then do "that"). Since computers can perform these tasks very rapidly, it should be no surprise that algorithms play such a prominent role in computers and computer science.

Multiple Solutions and Multiple Representations

One of the prevailing questions we address in this course is how do we use data structures and algorithms to solve problems? With so many different types of problems, and so many different possible representations and algorithms, finding a solution for a problem may seem like a daunting task. One fact that is on our side is that typically there is not just one solution for any given problem. Often there are many solutions to a problem, each with advantages and disadvantages. Sometimes enough solutions exist that the real problem is determining which approach to take based on these trade-offs.

A library contains an example of multiple approaches to solving a problem, and the advantages and disadvantages of each approach. Here we are faced with the problem of storing a large number of books in a way that allows users of the library to find specific books easily.

One solution to this problem involves using a hierarchical system to store together books on related topics. For example, this approach stores all books on "Politics" together. It further separates and groups books into sub-topics of "Politics." One shelf would contain the books on "History of Politics" while another shelf would contain the books on "Modern Politics." The representation continues to group the books into more detailed categories. Perhaps it divides the books in "History of Politics" into "History of Spanish Politics," "History of Brazilian Politics," "History of Italian Politics," and so on. The algorithm to find a specific book involves a person walking first to the "Politics" bookshelf, then to the "History of Politics" shelf, and so on until they find the book they seek.

Another representation involves storing books alphabetically by author. In this system, perhaps only top-level categories exist. Within these categories, the representation arranges books alphabetically by the name of the book's author. When we find the book we are looking for in this system, we would also find all the books written by the same author next to our book on the same shelf. After we found a book using the first representation, we would find the

other books on the same specific topic, and not the books written by the same author. This difference could be an advantage or disadvantage, depending on the reason for locating the book in the first place. For instance, when researching a specific topic it would be helpful to see all the other books on the topic.

Table 2.7

Problem	**Solution**
Remembering which groceries to buy	A grocery list
Tracking inventory by category and sub-category	A category list of sub-categories
Grading students over an entire semester	A list of students each with a list of grades
Managing your work tasks	A list of tasks and priorities
Performing the pre-flight procedure for an airplane	A check-list

Each of the problems in the table above essentially is solved using some type of list to represent the data. The algorithms involved in solving the problems may vary slightly, but essentially the algorithm traverses each list and performs some action (updating a grade, marking an item as complete).

Linear Data Structures

The idea of ordering structure of the program so it would be rather simple to describe life events use in programs methods, as it was marked already earlier, started in the beginning of the 70th years. At the same time, the prevailing events in human life, especially those that required repeated application and processing had the usually linear structures. Therefore, not surprisingly, in this direction we will also continue the way to knowledge. The first linear structures were probably presented in the form of the linear arrays. Taking this circumstance into account, we will consider the linear array for C programming language.

3.1 C-style Strings

This mechanism is *C-style strings*. C-style strings are neither as safe nor as easy to use as the string class. We discuss C-style strings only because they are occasionally encountered in C++ programming.

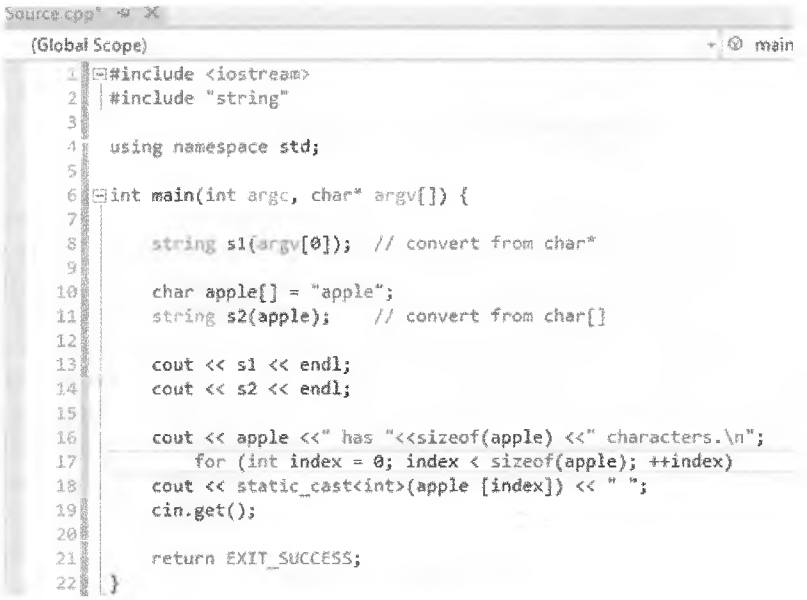
In the C programming language, arrays of type char provide support for character strings. With the introduction of C++, this mechanism became known as C-style strings. The following Listing illustrates how a C-style string stores the individual characters of the "apple".

'a'	'p'	'p'	'l'	'e'	'\0'

Listing 3.1. C-style string is an array of type char

The *null character*, or *null terminator*, is a special character that terminates the array to indicate the end of a C-style string. C-style strings therefore require one extra byte of storage in addition to the number of characters in the string. For example, to store the five-letter word "apple", a C-style string uses a six-element array. The array uses five elements to store the letters of "apple" and one element to store the null character.

During the remainder of this course, we manipulate character strings using the standard string class. This class provides ease of use, convenience, and safety that C-style strings lack. A constructor of the string class converts a C-style string into a string object. Listing demonstrates the use of this constructor.



Listing 3.2 Converting from a C-style string and the length of the string, and then the ASCII values of all of the characters.

Advanced String Operations

Class string also provides advanced character string support. Beyond the basic support of initialization, concatenation, length, input and output, class string provides a set of higher-level string handling functions. While a list of some of these follows, please refer to chapter 36 of the *Schildt* reference for a complete list.

- *erase*

Removes a sequence of characters from a string

- *find*

Searches a string for the occurrence of another string

- *substr*

Returns, as a string, part of another string

- *replace*

Replaces a substring of characters with another string

- *insert*

Inserts a string into another string

The following Listing 3.3 demonstrates the above member functions.

```
#include <iostream>
#include  "string"

using namespace std;

int main (int argc, char* argv[]);

string s1("Demonstrating all the advanced");
string s2("string functions!!!");

// erase the exclamation marks
s2.erase(16, 3);

// replace 'all the' with 'some'
s1.replace(14, 7, "some");

// insert a space at the beginning of s2
s2.insert(0, " ");

// extract everything after "some" from s1
int index = s1.find("some");
string s3 = s1.substr(index);

cout << s1 << s2 << endl << s3 << endl;

cin.get();

return EXIT_SUCCCESS;
}
```

Listing 3.3. Some advanced string functions

3.2 The STL and Basic Containers

STL Overview

Standard C++ includes a fairly extensive set of libraries and, in particular, the *Standard Template Library (STL)*. The STL provides general-purpose components for common programming tasks. These components are characterized by their flexibility, efficiency, and theoretical soundness. The library is organized into three main abstractions.

- Containers
- Iterators
- Algorithms

The containers include strings, vectors, lists, sets, stacks, and the like. The containers are organized as a collection of independent C++ classes. All the

STL containers classes are template classes, and therefore can accommodate items of arbitrary types. The user interfaces of the individual container classes are surprisingly small. Instead, there is a large and well-organized collection of STL algorithms that perform many of the tasks one might expect to see handled by a member function. For example, there is a universal remove() function that works on all kinds of containers. Other examples of STL algorithms are methods for searching, sorting, replacement, and various functional operations. Another interesting conceptual feature in the STL is that iterators form the link between containers and algorithms. Access to the items held in a container is always mediated by iterators, a sort of generalized array index or pointer.

Unlike similar libraries, the STL focuses strongly on algorithmic abstractions. Its implementation relies heavily on templates, but it uses little inheritance and virtual functions. As a consequence, the efficiency of STL components often equals that of components developed using traditional C++ code.

It is best to think of the STL as more of a language extension rather than a traditional library. More complicated data structures such as graphs and hash tables, which are not part of the library, can easily be built on top of existing STL components. As we will see, using library components allows designers of a class to focus on developing the conceptual design and functionality of the new class, rather than on spending an inordinate amount of time programming and debugging standard structures.

There is another reason why a C++ programmer should be keenly aware of the STL. As mentioned earlier, the STL has a sound theoretical foundation and provides an excellent conceptual framework that can be used to guide the implementation of user-defined data structures and algorithms. The ideas on which the STL is built can often be exploited for other purposes. For example, the use of iterators to gain access to the items stored in some class is often a superior solution to more traditional access methods. Designing algorithms so that they receive data via iterators often enhances their versatility considerably and may actually help slightly in writing the code: the iterator approach often brings out the salient features of an algorithm.

Containers

The standard containers are available via the following header files, whose names indicate the corresponding container type.

```
1: #include <string>        // strings
2: #include <vector>        // arrays
3: #include <list>          // cyclic doubly linked lists
4: #include <deque>         // hybrid list/array
5: #include <queue>         // queue
6: #include <stack>         // stack
7: #include <bitset>        // bit-vectors
8: #include <set>           // general sets
9: #include <map>           // associative arrays
```

Listing 3.4. The standard containers

Numerical calculations are supported in the following.

```
1: #include <complex>       // complex numbers
2: #include <valarray>      // numerical arrays
3: #include <numeric>      // numerical algorithms
4: #include <cmath>         // math functions
```

Listing 3.5. Numerical calculation headers

Unfortunately, not all C++ installations fully support the STL at this point; for example, bitset may be missing. On the other hand, some implementations of the STL have added extensions such as the following.

```
1: #include <hash_set>      // hash set
2: #include <hash_map>      // hash map
3: #include <slist>         // singly linked list
```

Listing 3.6. Some common STL extensions

These are not part of the standardized part of the library, and one should be careful using them when portability is an issue.

Iterators

Iterators provide a uniform interface between containers and algorithms in the STL. Iterators are modeled after plain C++ pointers. In particular, operators operator*, operator++ and so forth are properly overloaded, so that the use of iterators is very similar to the use of pointers. For example, listed below is the standard idiom to traverse a container and print out its elements.

```
1: string A = "This is a string";
2:
3: string::iterator it;    // create iterator
4: for (it = A.begin(); it != A.end(); ++it) {
5:     cout << *it << endl;
6: }
```

Listing 3.7. Traversing a container using iterators

Here it is a suitable iterator for container A. Note that the member functions A.begin() and A.end() return iterators, one pointing at the first element in A, and the other pointing at "one-past-the-end," meaning at a hypothetical position that would be occupied by the last element if another element were added to the container.

Iterators can be used to get access to a single item, but in sequential containers, they also provide access to whole blocks of items: we can always step an iterator from some initial value to an end value. Such a block of items is called a *range* in STL terminology. If we wanted to print out only the first ten characters in the string above, we could write the following.

```
1: int main (){                    ;
2: string A = "This is a string"
3: string::iterator it;    // create iterator
4: for (it = A.begin(); it != A.begin();+10;++it){
5:     cout << *it << endl;
6: }
```

Listing 3.8. Traversing a range

A great many operations make sense for individual items, as well as for ranges thereof, and it is a good idea to use iterators to implement such operations.

Algorithms

The core of the STL is its extensive collection of algorithms. Since iterators form the interface between containers and algorithms in the STL, the algorithms are (for the most part) implemented as free functions. Using free functions rather than member functions has the most desirable effect of reducing their number considerably. In some special cases, though, one may be forced to use member functions, either because an iterator would not provide enough information, or for efficiency reasons (as we will see, the erase operation and

the STL list class are examples). As already noted, another major advantage is that iterators naturally allow the user to perform operations on part of a container, rather than just the whole container, essentially without any added effort.

```
 1: #include <string>
 2: #include <cstdlib>
 3: #include <iostream>
 4: #include <algorithm>
 5:
 6: using namespace std;
 7:
 8: int main(int argc, char* argv[]) {
 9:
10:     string s = "This is a test";
11:     cout << s << endl;
12:
13:     // replace spaces with hyphens
14:     replace(s.begin(), s.end(), ' ', '-');
15:     cout << s << endl;
16:
17:     // reverse the string
18:     reverse(s.begin(), s.end());
19:     cout << s << endl;
20:
21:     return EXIT_SUCCESS;
22: }
```

Listing 3.9. The replace and reverse function

Throughout the remainder of this course, we will examine many of the STL algorithms. For the string container, it is worthwhile to examine the replace and reverse functions. The replace function substitutes all occurrences of an element with a particular value with another value. In the Listing below, all the characters in the string that are spaces are replaced with hyphens. The reverse function reverses a portion of a container. The Listing 3.9 reverses an entire string.

Class vector provides a safe and feature-rich alternative to an array. Similar to an array, a vector sequentially stores a series of objects of identical data types. Since it is an STL container, class vector supports generic programming. We can create a vector that stores integers, or one that stores strings, or one that stores any other primitive or user defined type.

Arrays can be dangerous to use because they provide no functionality to the programmer. When using arrays, the programmer is responsible for creating any higher-level functionality, such as resizing or reversing the array. These implementations can be tricky and, if not coded correctly, can introduce subtle bugs into an application. Vectors provide a safer alternative to arrays since they provide member functions that implement these (and other) higher-level tasks.

```
 1: #include <string>
 2: #include <cstdlib>
 3: #include <iostream>
 4: #include <vector>
 5:
 6: using namespace std;
 7:
 8: int main(int argc, char* argv[]) {
 9:
10:     vector<int> v1;
11:     vector<double> v2;
12:     vector<bool> v3;
13:     vector<string> v4;
14:
15:     return EXIT_SUCCESS;
16: }
```

Listing 3.10. Declaring vector objects

Listing 3.10 demonstrates the declaration of objects of class vector. Notice that in order to use objects of type vector, we have to include the library that defines the vector class. This is done inline 4 in the above Listing.

It is interesting to note that we can also declare a vector of vector objects. This provides an implementation for a two-dimensional data structure such as a matrix. Listing 3.11 illustrates the declaration of a vector of vectors. To avoid confusion with the >> operator, most compilers require a space following the first greater-than symbol. This applies to any nested template declaration, not just vectors of vectors.

```
#include <string>
#include <vector >
                  >
using namespace  std;

int main(int argc, char*  argv[] ) {

   vector <vector<int> > matrix;

    return  EXIT_SUCCESS;
}
```

Listing 3.11. A vector of vector objects

Class vector contains a set of constructors programmers can use to set the initial size of the vector and the initial values of the elements. Listing 3.12 demonstrates these constructors.

```
#include <string>
#include <cstdlib>
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char* argv[]) {

    vector<int> v1;          // initially empty
    vector<int> v2(5);       // 5 elements, initialized to 0
    vector<int> v3(10, 1);   // 10 elements, initialized to 1
    vector<int> v4(v3);      // v4 is a copy of v3

    return EXIT_SUCCESS;
}
```

Listing 3.12. Vector constructors

We can access the elements stored in a vector in a few different ways. First, class vector overloads operator[]. This allows us to access elements using syntax similar to array subscripting. Just like array subscript access, the overloaded bracket operator of class vector does not provide out-of-bounds access checking. When bounds checking is required, we must use the at() method. There are also special access methods for the first and last element in the array, as demonstrated in Listing 3.13.

```
1: #incluse <isotream>
2: #include <vector>
3:
4: using namespace std;
5: int main(int argc, char* argv[]) {
6: vector<int> v(10);
7:
8: v[1] = 2;
9: v.at(2) = 45;
10:v.front() = v.back();
```

Listing 3.13. Element access

In addition to providing, the standard member functions discussed above, class vector also supplies some other functions that you may not expect as part of an array class. For example, method push_back appends a data item to the end of the vector. This method automatically resizes the vector to accommodate the new element. Method pop_back performs the opposite task. This member function removes the last element from the vector, decreasing the size of the vector by one. The empty function returns true if the vector contains zero elements, and false otherwise.

```
 2: vector<int> v;
 3:
 4: // push elements in
 5: for (int i = 0; i < 5; i++) {
 6:     v.push_back(i);
 7:     cout << "Size: " << v.size() << endl;
 8: }
 9:
10: // pop elements off
11: for (int j = 0; j < 5; j++) {
12:     v.pop_back();
13:     cout << "Size: " << v.size() << endl;
14: }
15: cout << endl << v.empty() << endl;
16: return EXIT_SUCCESS;
17: }
```

Listing 3.14. Others member functions

In programming, one often needs collect, or aggregate, objects. Thus, class vector is one of the most frequently used STL containers. As an STL container, class vector provides access to its elements via iterators. Iterators allow interoperability of vector objects and the STL algorithms.

```
 1: vector<int> v;
 2:
 3: for (int i = 0; i < 10; i++) {
 4:     v.push_back(i);
 5: }
 6:
 7: // Iterate though elements
 8: vector<int>::iterator it;
 9: for (it = v.begin(); it != v.end(); it++) {
10:     cout << *it << endl;
11: }
12: cin.get();
13: return EXIT_SUCCESS;
14: }
```

Listing 3.15. Class vector iterators

Listing demonstrates the use of an iterator to traverse the elements stored in a vector. The second for-loop, which outputs the contents of the vector to the standard output stream, utilizes class vector member functions begin and end. These functions return iterators to the invoking vector. Function begin returns an iterator to the first element in the vector, and function end returns an iterator positioned beyond the last element in the vector. In the second for-loop, Listing initializes the iterator it to the iterator returned by function begin. During each iteration, the for-loop increments the iterator it. This causes the iterator to traverse through the vector. The traversal ends when the iterator equals the iterator returned from end.

```
1: int main(int argc, char* argv[]) {
2:vector<int> v;
3:ostream_iterator<int> out(cout, "\n");
4:copy(v.rbegin(),v.rend(),out);
5: return EXIT_SUCCESS;
6:
7:}
```

Listing 3.16. Vector and the STL

The STL function copy copies a sequence of elements. The first and second parameters are iterators that delimit the sequence. This third parameter specifies where to copy the sequence. In the Listing above, the third parameter is an output stream iterator initialized to the standard output stream. This causes function copy to output the elements stored in the vector to the console.

```
 1: #Include <isotream>
 2:  #include <vector>
 3: #include <string>
 4: #include <cstdlib>
 5:
 6: using namespace std;
 7:
 8: int main (int arge, char* argv[])
 9: vector<int> v;
10:
11: for (int i = 0; i < 10; i++) {
12:     v.push_back(i);
13: }
14:
15: ostream_iterator<int> out(cout, "\n");
16: copy(v.rbegin(), v.rend(), out);
```

Listing 3.17. Reverse iterators

If we would like to output the elements in the vector in reverse (from back to front), we need only switch the iterators used. The following Listing uses functions rbegin and rend instead of using functions begin and end. Functions begin and rend return iterators to the last element and one position before the first element, respectively.

Class vector also provides support for element insertion and removal from anywhere in the vector. Functions insert and erase perform these tasks. These functions take vector iterators as parameters. These iterators indicate the location to insert, or the elements to erase.

```
 1: // A vector with 10 elements
 2: vector<int> v(10);
 3:
 4: // Insert an element at the beginning
 5: v.insert(v.begin(), 50);
 6: cout << v[0] << endl;
 7: cout << v.size() << endl;
 8:
 9: // Erase the last five elements
10: vector<int>::iterator it = v.end() - 5;
11: v.erase(it, v.end());
12: cout << v.size() << endl;
```

Listing 3.18. Function insert and erase

To appreciate the power of the insert and erase functions we need to understand exactly what these functions do. Let's consider the vector in Figure 3.1 and the tasks that function insert performs to insert an element (with a value of 5) at the front of that vector.

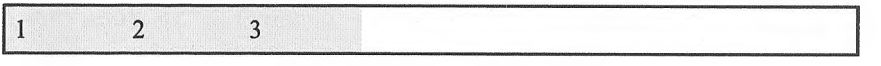


Figure 3.1. A vector with three elements

Size and *capacity* are two distinct properties of a vector. The size of the vector in Listing 1 is three, since it stores three elements. This vector also has reserved four memory locations to allow for the addition of new elements. The number of reserved memory locations plus the size is known as the capacity. The capacity of the vector in Listing 1 is seven. Having a capacity larger than its size allows operations such as insert or push_back to complete without having to allocate additional memory. Memory allocation only takes place when we exhaust the reserved memory locations.

Before it can prepend the new element, function insert must copy the existing elements one location to the right to create an empty space for the new element. Listing illustrates this task.

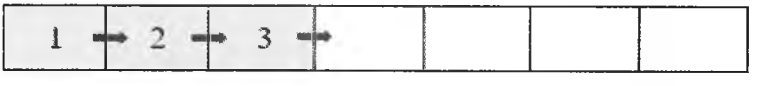


Figure 3.2. Copying elements

Listing represents the state of the vector after the function copies the elements one space to the right.

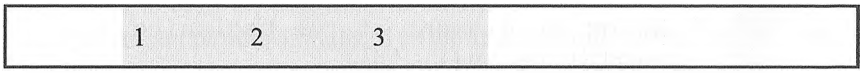


Figure 3.3. Copied elements

The new element with the value 5 is then copied into the empty location.



Figure 3.4. After insertion is complete

To insert just one item at the front of the vector, function insert had to first move all the other elements. Function erase performs similarly. To erase the first element in a vector, function erase must copy all remaining elements one location to the left. All this copying can become expensive if we are dealing with a vector that contains a large number of elements.

3.3 STL and Container

Using the STL deque Container

An object of type deque (pronounced "deck") can store and provide access to a linear sequence of elements. In this respect, a deque is similar to a vector. In fact, the two classes share nearly identical class interfaces.

```
 9:     count <<  "Enter your name and age:" ;
10:
11:     string name;
12:     int age;
13:
14:     cin >> name >> age;
15:
16:     if (age < 0) {
17:     cerr ,, '\Invalid age entered';
18:
19:     else {
20: cout << "\n" << name << is << age;
21:   }
```

Listing 3.19. Deques and Vectors

In the Listing above, we can see that both classes provide similar constructors, element access methods, insertion and removal methods, and iterator support. Deques and vectors differ, however, in their ability to handle element insertions and removals from the front of the respective structures. In class vector, element insertion and removal from the end of the vector is accomplished using functions push_back and pop_back. For implementation reasons we examine shortly, vectors contain no corresponding push_front and pop_front methods. Class deque, on the other hand, does provide methods push_front and pop_front. These methods do not require the element copying overhead.

```
 8: int main(int argc, char* argv[]) {
 9:
10: deque<int> d(10);  // 10 elements, initialized to 0
11:
12: d.push_front(2);
13: cout << d.front() << endl; // Outputs "2"
14:
15: d.pop_front();
16: cout << d.front() << endl; // Outputs "0"
17:
18:     return EXIT_SUCCESS;
19: }
```

Listing 3.20. Push front and pop front

Counting the number of items that possess certain properties can be done using the count and count_if functions. The count function accepts a range of elements in a container (through a beginning and ending iterator) and a value to count. The function returns number of occurrences of the value within the specified range. The count_if function uses a programmer supplied function to determine if an element in the specified range is counted. This programmer supplied function must take a single argument and return a value of type bool. The following Listing demonstrates both the count and count_if functions used with a deque.

```
bool is_odd(int i) {        // a predicate
    return ((i % 2) == 1);
}

int main(int argc, char* argv[]) {

    deque<int> numbers;
    for (int i = 0; i < 20; i++) {
        numbers.push_back(i);
    }

    cout << count(numbers.begin(), numbers.end(), 10) << endl;
    cout << count_if(numbers.begin(), numbers.end(), is_odd)
    << endl;

    return EXIT_SUCCESS;
}
```

Listing 3.21. The count and count_if functions

Implementation

Class deque and vector differ in their implementation. As we have seen in, class vector essentially wraps a C++ array, providing a safe and feature-rich interface. To accommodate efficient insertions and deletions of elements from both the front and back, a deque implementation is typically based on a series of arrays. While this implementation is interesting, we will ignore the details. Comparing element storage strategies between the two containers is enough to illustrate the main difference between deques and vectors.

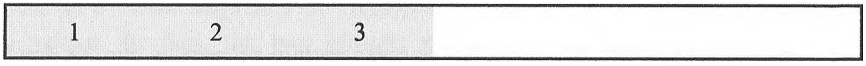


Figure 3.5. Element storage in a vector

Listing represents how elements are stored in a vector. In this Listing, the shaded squares represent the elements of the vector (the size) and the non-shaded squares represent the reserved memory (the capacity). This vector contains three elements, and has four reserved memory locations. In this arrangement, inserting an element at the beginning of the vector is not a trivial task since all the elements need to be moved to make room for the newly added element.



Figure 3.6. Element storage in a deque

Deques reserve memory locations at both the front and back of their stored elements. Figure illustrates this fundamental difference between vectors and deques. Reserving memory at both the front and back of the structure allows for efficient insertion and removal of elements at both ends, at the cost of a more complex implementation.

Non-Contiguous List

A *linked list* stores a linear sequence of elements. Using a linked list and its associated operations, we can build, manipulate, and maintain a list of elements. Recall that vectors also store a sequence of elements. Linked lists share many of the same operations as vectors. For example, both data structures support element insertion and removal, element traversal, and other basic operations.

The efficiency of certain operations differs between linked lists and vectors. For example, the operation associated with removing the first element is more efficient in a linked list than it is in a vector. On the other hand, accessing a random element in a sequence stored in a vector is more efficient than it is in a linked list. These differences in efficiency are rooted in how each data structure stores elements in memory. Vectors use contiguous storage whereas a linked list stores elements arbitrarily in memory. All the elements in a vector, for instance, appear one after another in memory.

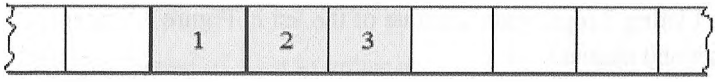


Figure 3.7. A vector in memory

Linked lists store elements in non-contiguous memory locations. This means that adjacent elements in the sequence the list maintains are not stored in adjacent memory locations. Figure 3.8 shows a linked list in memory.

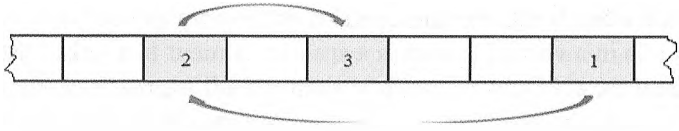


Figure 3.8. A linked list in memory

Storing elements in non-contiguous memory locations is only possible if each element somehow references the next element in the list. The reference from one element to the next appears in Figure as an arrow. The linked list in this Listing begins with the element whose value is 1. This element references the next element (value 2) which references the third element (value 3). The reference is typically known as a "link," which explains the name "linked list." It is important to notice that extra memory is required to store the link for each item in the list. This additional memory requirement per list element is a fundamental difference between linked lists and vectors.

Storing elements in contiguous or non-contiguous memory affects the efficiency of certain operations. When elements are stored in contiguous memory locations, random element access is efficient because the implementation can compute directly the location of any element. Storing elements in contiguous memory locations has an adverse impact on other operations. We have already seen the overhead involved in inserting an element in the middle of a vector. This overhead exists since the implementation must move values in memory to create "space" for the inserted item.

Non-contiguous storage also has its advantages and disadvantages. One advantage of a linked list is that we can remove or add an element by simply rearranging the links. For example, we can easily remove the second element from the list in Figure 3.9. To do this we update the link that points from element one to point it to element three. If the memory where element two exists was allocated dynamically (using new), then we also must deallocation this memory. This element removal process is the same regardless of the size of the list. Listing 3 represents the state of the list in Figure 3.9 after the removal of the second element.

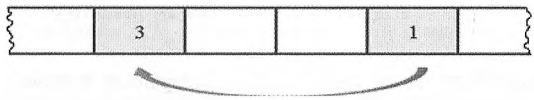


Figure 3.9. After removing element two

On the other hand, random element access in a linked list is not as efficient as it is in a vector. To access a specific element in a linked list, we must traverse all the elements from the beginning until we reach the item we seek. This is because the use of non-contiguous storage prevents the implementation from easily calculating where a specific element is stored.

Linking Elements Together

As we noted above, storing in memory only the element data is not sufficient to maintain the structure of a linked list. Each element must also indicate where in memory the next element is located. Since we are dealing with memory locations, linking together the elements contained in a linked list is a perfect application for pointers. With this in mind, we can build a mental model of a linked list that resembles Figure 3.10.

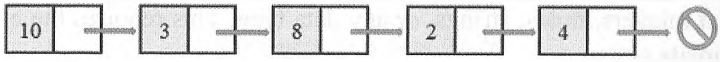


Figure 3.10. Pointers as links

In Listing we see a linked list containing five elements. The data from each list item is paired with a pointer (represented by the arrows) that indicates the location in memory of the next element. Notice that the pointer associated with the last item is set to the null pointer. This is a standard technique used to denote the last item in a list.

A *node* in a linked list contains the data for one element and any bookkeeping information necessary to maintain the list representation. In the example in Figure 2.11, the node contains an integer (the data), and a pointer to the next node in the list. The structure represented in Figure 2.11 is known as a *singly-linked list* since only one link exists for each node. Because only one link exists for each node, we can traverse a singly-linked list only from front to back.

To facilitate traversal of a list in either direction, we can add an additional pointer to each node that points to the previous node in the list.

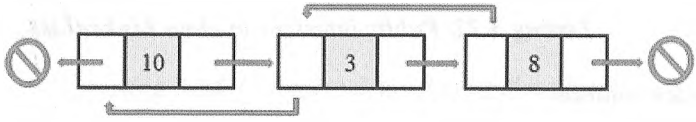


Figure 3.11. Doubly-linked lists

In Figure 3.11, one additional pointer has been added to each node. This new pointer references the previous node in the list. This type of linked list is a *doubly-linked list*. A doubly-linked list allows one to move forward and backward through a list.

A Template List Class

We start by examining some high-level requirements that we want our class to support. Let's assume that this list class, which we name *LinkedList*, needs to support only forward traversal. For this reason, we can implement the class as a singly-linked list. We can also assume that class *LinkedList* needs to support insertion and removal of elements at both the front and back of the list. To support this requirement, our class will maintain pointers to both the beginning and end of the list. Finally, we would like to use this class to store lists of integers, floats, strings, or any data type. This requires the class to be a template class.

The following C++ declaration contains the public members of our class.

```
template <class T>
class LinkedList {

public:
    LinkedList();   // Default constructor
    LinkedList(const LinkedList<T>& src);  // Copy constructor
    ~LinkedList();  // Destructor

    T& front();  // Element access for front of list
    T& back();   // Element access for back of list
    int size();  // count of elements in list
    bool empty(); // Size > 0

    void push_front(const T&);  // Insert element at beginning
    void push_back(const T&);   // Insert element at end
    void pop_front();  // Remove element from beginning
    void pop_back();  // Remove element from end

    void dump();  // Output contents of list
};
```

Listing 3.22. Public interface of class LinkedList

Implementation

Now that we have an idea of the member functions that the class implements, let's shift our focus and think about how we can represent a node of the linked list. We know a node must store the element data and any bookkeeping information necessary to maintain the structure of the linked list. Since this is a singly-linked list implementation, a pointer to the next node in the list is the only bookkeeping information required.

```
 1: template <class T>
 2: class LinkedList {
 3:
 4: private:
 5:     class Node {
 6:         T data;
 7:         Node* next;
 8:     };
 9:
10: public:
11:     // rest of public members ...
```

Listing 3.23. Class Node

The above class is an adequate representation for a node in a singly-linked list. It contains a member to store the data for the node and a pointer to the next node. Within the context of our linked list class, however, class *Node* needs to be augmented with a few more features. First, we add a constructor that allows initialization of the data members of the class. We also add a friend statement that allows the member functions of class *LinkedList* to access the private data members of class *Node*.

```
 6: template <class T>
 7: class LinkedList {
 8:
 9: private:
10:     class Node {
11:         friend class LinkedList<T>;
12:
13:     private:
14:         T data;
15:         Node* next;
16:
17:     public:
18:         Node(T d, Node* n = NULL) : data(d), next(n) {}
19:     ;
```

Listing 3.24. Complete definition of class Node

Listing fully defines class *Node* within class *LinkedList*. This is done mainly as a matter of convenience so that we do not have to specify the template type each time we declare an object of type *Node*. The definition of class *Node* in this Listing adds a constructor and includes explicit statements of the private and public data members. Also, the *friend* statement allows objects of class *LinkedList* to access the private data members of class *Node*.

```
 6: template <class T>
 7: class LinkedList {
 8:
 9: private:
10:     class Node {
11:         friend class LinkedList<T>;
12:
13:     private:
14:         T data;
15:         Node* next;
16:
17:     public:
18:         Node(T d, Node* n = NULL) : data(d), next(n) {}
19:     ;
20:
21:     Node* head;  // Beginning of list
22:     Node* tail;  // End of list
23:     int count;    // Number of nodes in list
```

Listing 3.25. All the private data members

Other private data members need to be added to class *LinkedList*. First, we add pointers to the beginning and end of the list. Our member function implementations use these pointers to manipulate the list. We will also add a data member to store the size of the list.

Simple Member Functions

We can now implement some of the simpler member functions of class LinkedList. Listing 2.26 contains implementations for the default constructor and member functions front, back, size and empty.

```
30: // Default constructor
31: LinkedList(void) : head(NULL), tail(NULL), count(0) {}
32:
33: // Returns a reference to first element
34: T& front(void) {
35:     assert (head != NULL);
36:     return head->data;
37: }
38:
39: // Returns a reference to last element
40: T& back(void) {
41:     assert (tail != NULL);
42:     return tail->data;
43: }
44:
45: // Returns count of elements of list
46: int size(void) {
47:     return count;
48: }
49:
50: // Returns whether or not list contains any elements
51: bool empty(void) {
52:     return count == 0;
53: }
```

Listing 3.26. Some of the simpler member function

The default constructor only has to initialize the three private data members, *head, tail,* and *count*. The two pointers, *head* and *tail*, are initialized to the null pointer to indicate that they do not initially point to valid nodes. The data member *count* is also initialized to `0` to indicate the list contains zero elements. Member functions *front* and *back* return a reference to the first and last elements in the list. Finally, member function *size* returns the number of elements in the list, and member function `empty` returns whether or not the list contains any elements.

Inserting and Removing from the Beginning

Inserting an element at the beginning of our list is a relatively straightforward process. The tasks involved are:

1. Create a new Node object.
2. Link the new node to the beginning of the list.
3. Reposition the head pointer to reflect the new beginning of the list.
4. Increase the count of elements by one.

Since we also need to maintain a pointer to the end of the list, a special case exists when we are adding an element to an empty list. In this case, the item we add does not only become the item at the beginning of the list, it also becomes the item at the end of list. This special case is known as a *boundary*

condition. A boundary condition is a point, or a case, in an algorithm that must be handled differently. Figure 3.12 and 3.13 illustrate this idea.

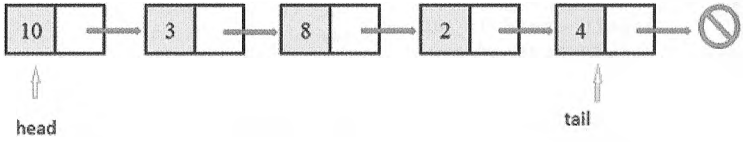


Figure 3.12. Five items in our linked list

Figure 3.13 depicts a five element linked list along with pointers *head* and *tail*.

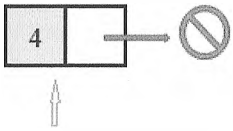


Figure 3.13. One item in our linked list

This list in Figure 3.13, on the other hand, contains only one element. In this situation, the lone element is the element at the beginning and the end of the list. This is reflected in Figure 3.13 with both the head and tail pointers pointing to that node. Listing 3.27 contains an implementation of method push_front that considers this special case.

```
85: // Insert an element at the beginning
86: template <class T>
87: void LinkedList<T>::push_front(T d) {
88:
89:     Node* new_head = new Node(d, head);
90:
91:     if (this->empty()) {
92:         head = new_head;
93:         tail = new_head;
94:     }
95:     else {
96:         head = new_head;
97:     }
98:     count++;
99: }
```

Listing 3.27. Method push_front

The implementation of method *pop_front*, for reasons similar to method *push_front*, must also account for a boundary condition. In this method, the special case occurs when the size of the list equals one.

```
8           // Remove an element from the beginning
9   template <typename T>
10  void LinkedList<T>::pop_front(void) {
11
12      assert(head != NULL);
13
14      Node* old_head = head;
15
16      if (this->size() == 1) {
17          head = NULL;
18          tail = NULL;
19      }
20      else {
21          head = head->next;
22      }
23
24      delete old_head;
25      count--;
26  }
```

Listing 3.28. Method pop_front

Method *pop_front* must perform essentially the opposite tasks that method *push_front* had to perform. Instead of allocating memory, method *pop_front* deallocates the memory of the node object that it unlinks from the front of the list. This method also decreases the count of the number of elements by one.

Inserting and Removing from the End

The implementation of appending (push_back) an element to the list is similar to that of prepending (push_front) an element. The pointer used in the operation is the only major difference. In push_back, we append an element by linking a new node to the tail pointer.

```
101: // Insert an element at the end
102: template <class T>
103: void LinkedList<T>::push_back(T d) {
104:
105:     Node* new_tail = new Node(d, NULL);
106:
107:     if (this->empty()) {
108:         head = new_tail;
109:     }
110:     else {
111:         tail->next = new_tail;
112:     }
113:
114:     tail = new_tail;
115:     count++;
116: }
```

Listing 3.29. Method push_back

Method *pop_back*, on the other hand, is not similar to any of the methods we have implemented so far. The steps required to complete this operation appear in the following list.

1. Reposition the tail pointer to the node prior to the last element.
2. Deallocate the memory used for the last node in the list.

The first step listed above, repositioning the tail pointer, introduces some complexity into the implementation. Repositioning requires us to move the tail pointer back one node. Since each node only has a pointer to the next element and not the previous element, we cannot perform this operation as efficiently as we removing the head element in method *pop_front*.

The only way to get a pointer to the node just before the last node is to traverse, or walk, the list. This traversal takes place in lines 29 through 31 in the following example.

```
138: // Remove an element from the end
139: template <class T>
140: void LinkedList<T>::pop_back(void) {
141:
142:     assert(tail != NULL);
143:
144:     Node* old_tail = tail;
145:
146:     if (this->size() == 1) {
147:         head = NULL;
148:         tail = NULL;
149:     }
150:     else {
151:
152:         // Traverse the list to node just before tail
153:         Node* current = head;
154:         while (current->next != tail) {
155:             current = current->next;
156:         }
157:
158:         // Unlink and reposition
159:         current->next = NULL;
160:         tail = current;
161:     }
162:
163:     delete old_tail;
164:     count--;
165: }
```

Listing 3.30. Method pop_back

The Destructor

The destructor for our linked list class must deallocate any memory we may have allocated (in methods push_front and push_back). We can leverage our implementation of method pop_front to provide a simple, effective destructor implementation.

```
76: // Destructor
77: template <class T>
78: LinkedList<T>::~LinkedList(void) {
79:
80:     while (! this->empty()) {
81:         this->pop_front();
82:     }
83: }
```

Listing 3.31. The destructor

Method *pop_front*, instead of method *pop_back*, was not chosen arbitrarily. Think back to the implementation of method *pop_back*and try to reason why using method *pop_back* would not be as efficient as method *pop_front*.

The Copy Constructor

Since our linked list class allocates memory dynamically, we need to implement a copy constructor. This is to ensure that when a copy of a list is made, a copy of the list's data is also made. Otherwise, we could have two list objects' head and tail pointers point to the same nodes.

```
63: // Copy constructor
64: template <class T>
65: LinkedList<T>::LinkedList(const LinkedList<T>& src) :
66: count(0), head(NULL), tail(NULL) {
67:
68:     Node* current = src.head;
69:     while (current != NULL) {
70:         this->push_back(current->data);
71:         current = current->next;
72:     }
73:
74: }
```

Listing 3.32. The copy constructor

Displaying the List Contents

The final member function we implement for class LinkedList is method print_list. This method displays the contents of the list to the standard output stream.

```
167: // Display the contents of the list
168: template <class T>
169: void LinkedList<T>::dump(void) {
170:
171:     cout << "(";
172:
173:     Node* current = head;
174:
175:     if (current != NULL) {
176:
177:         while (current->next != NULL) {
178:             cout << current->data << ", ";
179:             current = current->next;
180:         }
181:         cout << current->data;
182:     }
183:
184:     cout << ")" << endl;
185: }
```

Listing 3.33. Method print_list

Using the STL list Container

The list container is the Standard Template Library's linked list class. Page 819 of the Schildt reference book contains a complete Listing of the member functions that comprise the interface of class list.

The container class list supports element insertion and removal from both the front and back. In class list, member functions push_front, pop_front, push_back, and pop_back accomplish these tasks. Class list also provides access to the data in the first and last elements via methods front and back. Furthermore, similar to our class LinkedList, method size returns the number of elements in a list and method empty reports whether or not a list is empty.

```
 8: int main(int argc, char* argv[]) {
 9:
10:     list<int> l1;
11:     l1.push_front(10);
12:     cout << l1.size() << endl; // count of 1
13:
14:     list<string> l2(10);
15:     cout << l2.size() << endl; // count of 10
16:
17:     list<double> l3(1, 2.3);   // initial value of 2.3
18:     cout << l3.back() << endl;
19:     l3.pop_back();
20:     cout << l3.empty() << endl;
```

Listing 3.34. The STL list container

In addition to the default constructor, the STL list class provides a constructor that specifies an initial number of elements in a list. Line 14 of the above Listing uses this constructor to declare a list that initially contains ten elements. This constructor also has an optional second parameter. A programmer can use this second parameter to specify the default value for the initial elements.

An example of this usage is demonstrated in line 17 of Listing 3.23. This line declares a list that initially contains one element with the value 2.3. Omitting the second argument invokes the default constructors of the initial elements.

Traversing a List

Traversing through the elements contained in a list is a common application of iterators.

Using the iterator returned by function begin and the iterator returned by function end, a loop construct can visit all the elements in a list.

```
1: // Populate
2: list<int> l;
3: for (int i = 1; i <= 10; i++) {
4:     l.push_back(i);
5: }
6:
7: // Traverse
8: list<int>::iterator it;
9: for (it = l.begin(); it != l.end(); it++) {
10:    cout << *it << endl;
11: }
```

Listing 3.35. Traversing a list

The STL list is a doubly-linked list. This means the list should support traversal from back to front. At first thought, one might attempt the following for a reverse traversal.

```
 1: // Populate
 2: list<int> l;
 3: for (int i = 1; i <= 10; i++) {
 4:     l.push_back(i);
 5: }
 6:
 7: // Reverse Traverse
 8: list<int>::iterator it;
 9: it = l.end();
10: do {
11:     --it;
12:     cout << *it << endl;
13: }
14: while (it != l.begin());
```

Listing 3.36. Wrong way to traverse in reverse

The code in Listing does traverse the entire list in reverse, but it is clearly not as readable as the forward traverse Listing. To implement a more readable version of a reverse traverse, we need iterators that point to the last element in the list and one position before the first element. These iterators exist, and are returned by functions begin and rend, respectively.

```
 1: // Populate
 2: list<int> l;
 3: for (int i = 1; i <= 10; i++) {
 4:     l.push_back(i);
 5: }
 6:
 7: // Reverse Traverse
 8: list<int>::reverse_iterator it;
 9: for (it = l.rbegin(); it != l.rend(); it++) {
10:     cout << *it << endl;
11: }
```

Listing 3.37. Reverse iterators

The iterators returned by functions rbegin and rend are *reverse iterators* and are used for reverse traversals. Reverse iterators operate much the same way as regular iterators, except they handle increment operations in reverse. Notice the use of the increment operator (++) in the for-loop in Listing 2.37. Incrementing the reverse iterator moves it backwards through the list. Reverse iterators, found also in the other STL containers (string, vector, and deque)

that we have examined, provide an effective and intuitive way to traverse a structure in reverse order.

Inserting and Removing Elements

Linked lists perform efficient element insertions and removals, regardless of the positions of elements with the list. This is due to the linked nature of the structure. Member functions insert and erase perform this functionality in STL list class. To indicate the range of elements to insert or remove, both of these functions use iterators. Function erase either takes one argument indicating the element to remove, or two arguments that indicate the range of elements to remove. Listing 2.38 demonstrates the single argument use of function erase.

```
1: // Erase the second element from a list
2: list<int> l(10);
3: list<int>::iterator second = l.begin();
4: second++;
5: l.erase(second);
```

Listing 3.38. Using function erase

Method insert is used similar to method erase. Listing inserts an element with the value 5 after the first list element.

```
1: // Insert an element after the first element
2: list<int> l(10);
3: list<int>::iterator it = l.begin();
4: it++;
5: l.insert(it, 5);
```

Listing 3.39. Function insert

Using function insert, we can insert an entire sequence of elements into a list. Listing 7 demonstrates this use of the function.

```
1: // Insert a range of elements
2: list<double> l1(10, 2.14);
3: list<double> l2(5, 0.0);
4:
5: list<double>::iterator it = l1.begin();
6: it++;
7:
8: l1.insert(it, l2.begin(), l2.end());
9: cout << l1.size() << endl;
```

Listing 3.40. Inserting a range of elements

3.4 Queues, Stacks, Using the STL stack Adapter

First-In, First-Out

A queue is a linear data structure that utilizes a first-in, first-out element insertion and removal policy. First-in, first-out means that the first element added to a queue is the first element removed from a queue. Put another way, things are only removed from the beginning and only added to the end of queues.

Queues are used in the real world in many places. Remember from we discussed a line at a movie theater as a data structure. This line is really just a queue. People enter the line at the back and leave the line only when they reach the front and buy their tickets. Airplanes waiting to land and take off from an airport also wait in queues. Anywhere a "First Come, First Served" sign is seen, a queue is at work.

Queues have many different applications in computer science and software development. A queue can be used to provide a *buffer*. A buffer provides temporary storage for information that needs to be processed later. Network routers buffer incoming data packets in a queue to give the routers time to process outgoing packets. Operating systems can use a queue to implement a scheduling policy that decides which program to run next. Applications that stream audio and video across the Internet buffer a certain number of packets in a queue to account for temporary slow downs in connection speeds.

queue Adapter

The Standard Template Library contains *adapters* that provide a new, but similar interface to an existing container. The queue adapter provides the interface suitable for a queue data structure. It is based, by default, on the deque container. When an object of type queue is used, an object of type deque is actually instantiated, just with an interface more suitable for a queue. Listing 1 illustrates the entire, small interface of the queue adapter.

```
1: #include <iostream>
2: #include <string>
3: #include <cstdlib>
4: #include <queue>
5:
6: using namespace std;
7:
8: int main(int argc, char* argv[]) {
9:
10:     queue<int> q;
11:
12:     // push and pop
13:     q.push(1);
14:     q.pop();
15:
16:     // front and back
17:     q.push(1);
18:     q.push(2);
19:     cout << q.front() << endl;
20:     cout << q.back() << endl;
21:
22:     // size and empty
23:     cout << q.size() << endl;
24:     cout << q.empty() << endl;
25:
26:     return EXIT_SUCCESS;
27: }
```

Listing 3.41. The queue interface

Only six member functions exist in the queue adapter interface. Methods push and pop insert and remove elements from a queue, and methods front and back provide access to the data stored in the beginning and end of a queue. Similar to the other STL containers we have examined, method size returns the number of elements stored in a queue and method empty returns true if a queue is empty and false if a queue is storing at least one element.

Notice from line 4 in the above Listing that inclusion of the <queue> library is necessary to use the queue adapter.

One thing that the queue adapter lacks is support for iterators. Without iterators, a method to traverse the elements stored in a queue does not exist. Essentially, a queue hides all the elements it stores except for the first and last. If an application requires a queue, but also requires access to the elements stored in the queue, a deque should be used instead. Table 3.1 shows the member functions of class queue and their deque equivalents.

queue member		deque equivalent
1	push	push_back
2	pop	pop_front
3	front	front
4	back	back
5	size	size
6	empty	empty

Last-In, First-Out

A stack is a linear data structure that provides access to only the most recently added element. In a stack, items can only be added at and removed from one end of the sequence. This amounts to a "Last-in, First-Out" element insertion and removal policy. Put another way, the next item that can be removed from a stack is always the element that was most recently added to the stack. We can always add more elements to a stack, but each time we do, the newly added element becomes the element that can be removed first.

A stack of plates or dishes in a kitchen is a real world example of a stack data structure. Consider how one uses a stack of plates. When a plate is needed for dining, one is taken from the top of the stack of plates. It is not a good idea to try to take a plate from the bottom, since this probably would topple the entire stack. When plates are put away, they are placed on top of the stack. Because we only take plates from the top, a plate placed on top of the stack becomes the next plate that we use.

Real world examples like the stack of plates make it is easy to think of a stack as a vertical structure. With this mental model in mind, let's step through adding a few elements to an initially empty stack. First we add (typically referred to as "pushing" onto the stack) an element with the value 1. Figure 2.14 represents our stack with this lone element.



Figure 3.14. After adding element 1

Things become more interesting after we push a second element onto the stack. Figure 2.15 shows the state of the stack after we add an element with the value 2.



Figure 3.15. After adding element 2

The element shaded in gray is now at the top of the stack. Adding this second element effectively hides the first element since the second element is now the only element that can be removed. Also, it is the only element in the stack whose value we can access. This is an important property of the stack data structure that differs from vectors and deques. In a stack, we have access to the number of elements stored in the data structure, but we do not have access to all the values of the elements. We can "peek" into the stack to return the value of the element that sits at the top. A peek operation performed on the stack in Figure 3.15 would return the value 2.



Figure 3.16. After adding element 3

Figure 3.16 shows the stack after we push a third element onto the stack. The element added contains the value 3. If we were to remove an element from the stack (often referred to as "popping" an element), our stack would then look like Figure 3.15. We have examined how stacks operate and have seen the primary operations associated with stacks. Summarizing, "pushing" adds an element to the top of a stack, "popping" removes the element at the top of a stack, and "peeking" returns the value of the element at the top of a stack.

Applications of Stacks

An important data structure in Computer Science, stacks have many different applications. These applications range from simple tasks such as reversing the characters in a string to seemingly more complex tasks such as the evaluation of arithmetic expressions or the navigation of mazes. It is important to gain a solid understanding of the concept of a stack since later in this course we encounter many uses of stacks in the context of other data structures and algorithms.

Using the STL stack Adapter

The STL stack adapter provides an interface suitable for use as a stack. Listing demonstrates all the member functions of the stack adapter.

```
 1: #include <iostream>
 2: #include <string>
 3: #include <cstdlib>
 4: #include <stack>
 5:
 6: using namespace std;
 7:
 8: int main(int argc, char* argv[]) {
 9:
10:     stack<int> s;
11:
12:     // push and pop
13:     s.push(1);
14:     s.pop();
15:
16:     // top
17:     s.push(10);
18:     s.push(11);
19:     cout << s.top() << endl;
20:
21:     // size and empty
22:     cout << s.size() << endl;
23:     cout << s.empty() << endl;
24:
25:     return EXIT_SUCCESS;
26: }
```

Listing 3.42. The stack adapter

Only six member functions exist in the stack adapter interface. Methods push and pop insert and remove elements from a stack respectively, and method top returns a reference to the data stored at the top of the stack. As with the queue adapter, method size returns the number of elements stored in a stack and method empty returns true if a stack is empty and false if a stack is not empty.

To use the STL stack adapter in a program, a programmer must include the <stack> library. This is done in line 4 of Listing 3.42.

Listing 3.43 shows a more practical example of the stack adapter. This

Listing 3.43 displays the lines of a text file in reverse order. A stack provides a natural data structure to solve this problem.

```
// open file specified by command-line argument
ifstream inf(argv[1]);

if ( !inf ) {
    cerr << "cannot open " << filename << " for input" << endl;
    return EXIT_FAILURE;
}

stack<string> s;
string line;

// read file line by line
while (getline(inf, line)) {
    s.push(line);
}

inf.close();

// print lines in reverse
while (!s.empty()) {
    cout << s.top() << endl;
    s.pop();
}
```

Listing 3.43. Displaying lines of a file in reverse

Recursion

4.1. The basic concept of recursion

Recursive Functions

A *recursive function* is a function that calls itself. The use of recursive functions, called *recursion*, can yield elegant solutions to otherwise complex problems. C++, like many other programming languages, supports recursion.

A programmer must define recursive functions carefully in order to avoid creating a function that repeatedly calls itself forever. A pseudocode version of a typical recursive function looks like the following example.

```
if (simplest case) then
   solve directly
else
   make recursive call to a simpler case
```

Example 4.1. A typical recursive function

A key to creating and using effective recursive functions is learning to think of a problem in terms of a similar, but smaller problem. Eventually, a problem becomes small enough that a function can solve it without using recursion. This is called the *base case*.

Calculating factorials is one example of a problem that we can solve using recursion. The factorial of a number is the product of all the integers from that number to one. For example, the factorial of five (often called "five factorial") equals 5 * 4 * 3 * 2 * 1. This evaluates to 120.

```
5! = 5 * 4 * 3 * 2 * 1 = 120
4! = 4 * 3 * 2 * 1 = 24
3! = 3 * 2 * 1 = 6
2! = 2 * 1 = 2
1! = 1
0! = 1
```

Example 4.2. Some factorials

Three factorial (3 * 2 * 1) equals the value 6. An exclamation point denotes the factorial of a number. Thus, "five factorial" can be expressed as 5!. Example 2 lists some of the first several positive integers and the calculation of their factorials. The factorial for zero is a special case and is defined to equal 1.

We can express factorials recursively, that is, in terms of other factorials. Consider the factorial calculation for the value 5. From Example 2, the calculation is 5! = 5 * 4 * 3 * 2 * 1. But, from examining the factorial calculation for 4, we know that 4! = 4 * 3 * 2 * 1. Recursively then, 5! = 5 * 4!. Example lists the recursive definitions of the same numbers from Example 3.2. Since we cannot express zero factorial recursively, it is the base case.

5! = 5 * 4!
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!
0! = 1

Example 4.3. Recursive factorials

Listing 4.1 contains C++ code that recursively calculates factorials. Notice that function factorial follows the recursive function pattern outlined in Example 4.3.

```
 1: #include <iostream>
 2: #include <cstdlib>
 3: #include <string>
 4:
 5: using namespace std;
 6:
 7: int factorial(int n) {
 8:
 9:     if (n == 0) {
10:         // base case
11:         return 1;
12:     }
13:     else {
14:         // recursive call
15:         int value = factorial(n - 1);
16:         return n * value;
17:     }
18: }
19:
20: int main(int argc, char* argv[]) {
21:
22:     cout << factorial(5) << endl;
23:     return EXIT_SUCCESS;
24: }
```

Listing 4.1. Calculating a factorial recursively

Execution of the program in Listing 4.1 outputs the expected value of 120. We know this is correct, but how did the function achieve this result? Adding output statements to function factorial gives us a better idea how this example works. Listing 4.1 contains an updated function factorial that outputs a line indicating when an instance of the function begins and when an instance of the function is about to end. This modified version also outputs the return value of function factorial.

```
 1: int factorial(int n) {
 2:
 3:     cerr << "factorial(" << n << ") begin" << endl;
 4:
 5:     if (n == 0) {
 6:         cerr << "factorial(" << n << ") returns 1" << endl;
 7:         return 1;  // base case
 8:     }
 9:     else {
10:         int ret = n * factorial(n - 1); // recursive call
11:         cerr << "factorial(" << n << ") returns " << ret << endl;
12:         return ret;
```

Listing 4.2. A verbose function factorial

Example 4.4 contains the output of the factorial program after the addition of the output statements to function factorial.

The output in Example 4.4 shows that the program first calls function factorial with the argument 5. Function main performs this initial call. During the execution of factorial(5) it makes a call to function factorial with an argument value of 4. The instance of factorial(4) then begins execution and makes a call to factorial(3), which in turn makes a call to factorial(2). This behavior continues until factorial(0) begins and returns the value 1. After this, factorial(1) returns to factorial(2) the value 1, factorial(2) returns to factorial(3) the value 2, and so on until factorial(5) returns to main the value 120.

```
factorial(5) begin
factorial(4) begin
factorial(3) begin
factorial(2) begin
factorial(1) begin
factorial(0) begin
factorial(0) returns 1
factorial(1) returns 1
factorial(2) returns 2
factorial(3) returns 6
factorial(4) returns 24
factorial(5) returns 120
120
```

Example 4.4. Output

The Call Stack

The *call stack* is an area of a program's memory used to manage the currently executing function and all pending function calls. Listing lends some insight into what is meant by a "pending function call." Looking back at Listing , we can see that even though the instance of function factorial with the parameter 5 was the first to begin execution, it is the last to finish execution. The program pauses the execution of factorial(5) "pending" the completion of the function call to factorial(4). Likewise, the program pauses the execution of factorial(4) pending the completion of the function call to factorial(3). This series of nested function calls is managed using the call stack. Consider the following example C++ program.

```
 1: #include <iostream>
 2: #include <cstdlib>
 3:
 4: using namespace std;
 5:
 6: void method3(void) {
 7:     cout << "Method 3" << endl;
 8: }
 9:
10: void method2(void) {
11:     method3();
12: }
13:
14: void method1(void) {
15:     method2();
16: }
17:
18: int main(int argc, char* argv[]) {
19:     method1();
20:     return EXIT_SUCCESS;
21: }
```

Listing 4.3. A program with nested function calls

Stepping through the function calls in Listing demonstrates how a program uses the call stack to manage the function calls in a program. As in all C++ programs, the first function that executes in the program in Listing is function main. Listing 1(a) represents the state of the call stack after function main begins execution, but before it calls method1. Since function main is the routine currently in execution, the information needed to run this function sits on the

top of the call stack. This information, which includes among other things the local variables of the function, is known as an *activation record.*

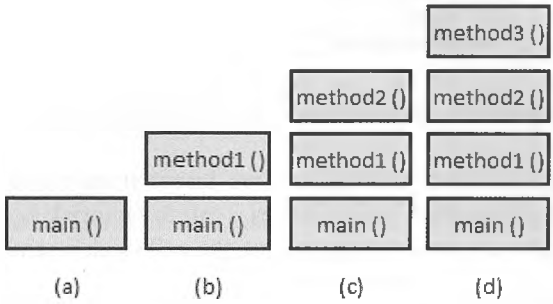


Figure 4.1. The call stack during various points

When main calls function method1, the run-time system pushes an activation record for method1 onto the top of the call stack. The run-time system then halts execution of function main, pending the completion of method1. At this point in the program, Figure 4.1 (b) represents the state of the call stack. Function method 1 then immediately calls method 2. The run-time system pauses execution of method1 and pushes an activation record for method 2 onto the top of the stack. This state of the call stack at this point in the program corresponds to Figure 4.1 (c). After function method 2 calls method 3, the call stack resembles Figure 4.1 (d).

At this point during the execution of the program, function calls are nested four levels deep. The program currently is executing function method 3, with functions method 2, method 1, and main all suspended. After the program finishes execution of function method 3, the run-time system pops off the activation record for method 3 from the top of the stack. Execution of method 2 then resumes. The call stack would then again resemble Figure 4.1 (c). When execution of method 2 completes, the run-time system pops it off the stack also, putting the call stack back to the state represented in Figure 4.1 (b). As the nested functions in the program end, the call stack grows shorter and shorter. Eventually, function main finishes and the run-time system pops its activation record off the stack, ending the execution of the program.

The call stack operates in the same manner when dealing with recursive functions as it does with regular functions. Revisiting the factorial example, let's trace the call stack as it manages the recursive calls to function factorial. The program execution begins with function main. Function main then calls function factorial with the parameter 5. The run-time system pushes the information needed to execute factorial with the parameter 5 on top of the stack. The call stack, at this point resembles Figure 4.2.



Figure 4.2. The call stack while factorial (5) executes

The function call factorial (5), as we know from earlier examination, makes a call to factorial(4), which makes a call to factorial(3), and so on until factorial(0) begins execution. At this point, the call stack resembles Figure 4.3.

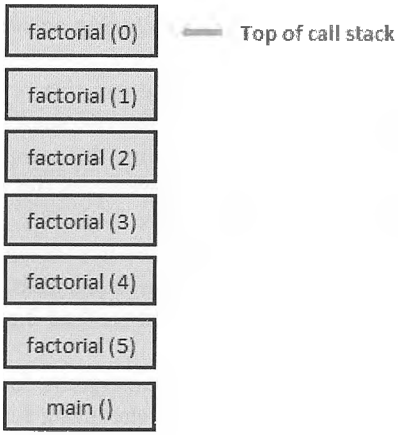


Figure 4.3. The call stack while factorial(0) executes

Function factorial(0) is the base case of the recursion. When this instance of the function finishes, it returns the value 1 to the function whose activation record sits below it. Then the run-time system pops the activation record for factorial(0) off the stack and resumes execution with function factorial(1). The nested functions continue to "unwind" in this manner, passing values back to their calling functions, until function main completes and the program terminates.

Debuggers often provide a feature that allows programmers to inspect the call stack of a program. This feature can prove invaluable when trying to debug a recursive function. Figure 4.4 shows an example call-stack window from Microsoft Visual C++.

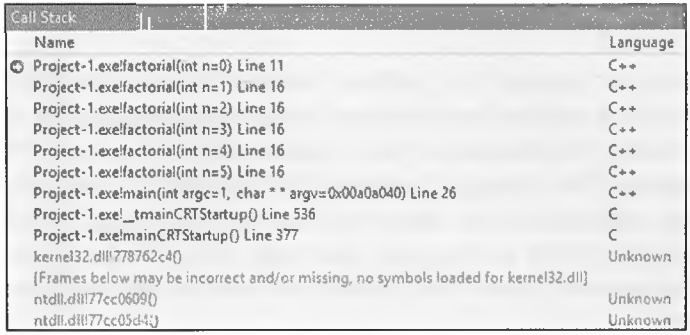


Figure 4.4. Microsoft Visual C++ call stack window

Removing Recursion

Recursion comes with a price: the run time system has to maintain a complicated call stack, on top of having to perform the usual evaluations contained in the function. Often, we can eliminate recursion in favor of iteration, which does not make similar demands on the run-time system.

```
1: #include <iostream>
2: #include <cstdlib>
3: #include <string>
4:
5: using namespace std;
6:
7: int factorial(int n) {
8:
9:     int x = 1;
10:    for (int i = 2; i <= n; ++i) {
11:        x *= i;
12:    }
13:    return x;
14:}
```

Listing 4.4. Non-recursive factorial calculation

It is always possible to eliminate recursion, and it is worthwhile to think about replacing recursion by loops. In some cases, however, recursion may well be the superior method. Non-recursive versions of certain algorithms may be so much more complicated to program that the gain in efficiency is not worth the added effort.

4.2 Solving problem with recursion

Divide and conquer is a problem solving technique that utilizes recursion to solve a problem by "dividing" the problem into smaller and smaller sub-problems. The base case of the recursion solves the group of the smallest sub-problems. The "conquer" portion of this problem solving technique then combines these solutions to create the solution to the original problem.

Generally, divide and conquer algorithms utilize two recursive function calls. Having two recursive calls continually divides the problem space into two parts.

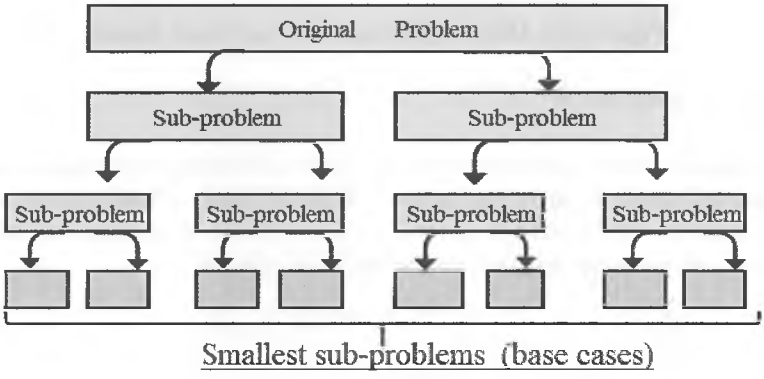


Figure 4.5. Dividing a problem

Figure 4.5 illustrates a typical "divide" step that uses two recursive calls. When the recursion reaches the base case, the sub-problems are solved directly. The solutions to these sub-problems are then combined together (as the recursion unwinds) and eventually form the solution to the original problem.



Figure 4.6. The combined solution

Figure 4.6 shows the solved sub-problems combined into a solution for the original problem.

Consider the problem of calculating the sum of the squares of a range of integers. We can apply the divide and conquer approach to reduce the range continually until we reach a sub-problem size that is easily calculated. Listing 4.5 contains the source code for this recursive, divide-and-conquer based algorithm.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int ssq(int m, int n) {

    if (m == n) {
        return m * m; // base case
    }
    else {
        int middle = (m + n) / 2;
        // recursive divide
        return ssq(m, middle) + ssq(middle + 1, n);
    }
}

int main(int argc, char* argv[]) {

    cout << "ssq(1,10) = " << ssq(1, 10) << endl;

    return EXIT_SUCCESS;
}
```

Listing 4.5. Finding the sum of the squares of a range of integers

```
#include <iostream>
#include <cstdlib>

using namespace std;

int find_smallest(int a[], int size) {

    if (size == 1) {
        return a[0]; // base case
    }
    else {

        // Search the first half of the array for the smallest element.
        int s1 = find_smallest(a, size / 2);

        // Search the second half of the array for the smallest element.
        int s2 = find_smallest(a + size / 2, size - size / 2);

        return (s1 < s2) ? s1 : s2;
    }
}

int main(int argc, char* argv[]) {

    int arr[] = {13, 19, 12, 11,
                 15, 19, 23, 12,
                 13, 22, 18, 19,
                 14, 17, 23, 21};

    cout << "smallest: " << find_smallest(arr, 16) << endl;

    return EXIT_SUCCESS;
}
```

Listing 4.6. Finding the smallest element in an array

Another example of a simple and effective divide and conquer based algorithm appears in Listing 4.6.

Function find_smallest determines the smallest element stored in an array by continually dividing the array into two smaller pieces. When these pieces are small enough that they only contain one element, the algorithm then compares the elements stored in two pieces and returns the smaller of the two elements.

Backtracking.

Backtracking is a problem solving technique that involves examining all possibilities in the search for a solution. An example of backtracking can be seen in the process of finding the solution to a maze. During the exploration of a maze, we have to make decisions involving which path to explore. When faced with a choice of paths to explore, we decide whether to go north, south, east, or west. A backtracking approach to find our way through a maze involves considering all possible paths until we find a solution.

Backtracking involves pursuing one possible solution until the algorithm determines that it is or is not a solution. If a backtracking algorithm discovers a chosen possibility is not a solution, the algorithm "backs up" and chooses to pursue another possibility from the set of possible solutions. For example, the mouse needs to find the solution to the maze in order to get to the cheese in Listing 1. Let's place ourselves in the mouse's situation and see how we can use backtracking to find the solution. Immediately, at position A, we have a choice between moving north or south. For the sake of the example, we will choose north, but we will remember that there is another path leading south that we have not explored. Going north, we find out quickly that the path leads to a dead end. In this situation, we must backtrack to position A and visit the path we did not choose. Moving south from position A, another choice appears at position B where we can continue to move south or change direction and explore the path to the east.

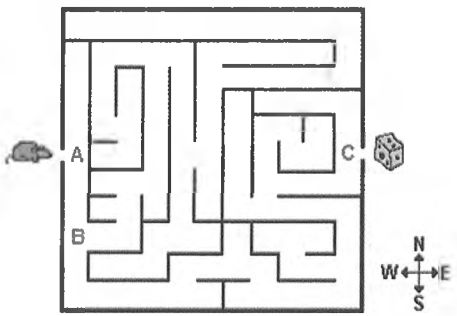


Figure 4.7. A maze

If we choose to go east, we will surely have to make many other decisions on which paths to take as we continue to explore. If one of these paths leads to the solution, our search is complete. If all paths branching off to the east lead to dead ends, we must backtrack to position B and explore the path that leads south. This process is guaranteed to find a solution to the maze since it considers all possible paths.

Backtracking algorithms that use recursion operate basically the same way as other recursive algorithms. Similar to any other recursive algorithm, programmers design backtracking algorithms around base cases that are solved without recursion. In the maze example, the base case exists when the mouse is adjacent to the exit of the maze (at position C). In this situation, the choice to go east is obvious and a recursive search is not needed. Recursive backtracking algorithms also reduce a problem to a smaller sub-problem. The recursion, applied in the maze example, effectively makes the maze smaller and smaller until it reaches the base case.

An Example: Eight Queens

A classic problem that we can solve using backtracking is the *Eight Queens* problem. This difficult problem deals with placing eight queens on a chessboard such that no two queens are attacking each other.

The game of chess is played on a board containing 64 squares of alternating color. Two players take turns moving a set of pieces on these squares. The object of the game is to capture the opponent's most important piece, the "king." While the king is the most important piece, the "queen" is the most powerful piece. In the game of chess, queens can "attack" or "capture" other pieces in two different ways. First, a queen can attack pieces on those squares that are in the same row or the same column as the queen. Second, a queen can attack pieces occupying the squares that run diagonally through the square that the queen occupies. Figure 4.8 shows the two different ways that queens attack. Note that the squares highlighted in red indicate the squares that the queen in the Figure can attack.

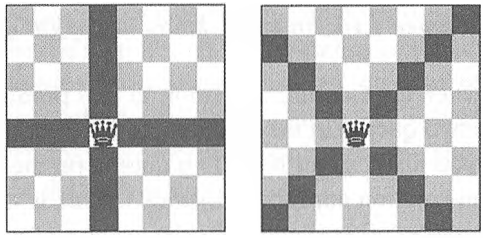


Figure 4.8. Queens attack in two different ways

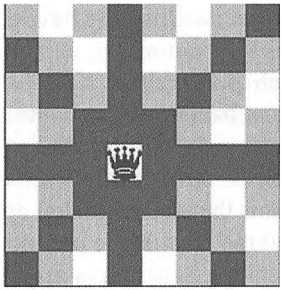


Figure 4.9. All the squares a queen can attack

Combining these two methods together, we see (again in red) all the squares on a chessboard that a queen can attack in Figure 4.8. We consider the queen in Figure 4.8 to be "attacking" any piece that occupies a red square.

A queen is considered the most powerful chess piece since it attacks the largest number of squares out of any other piece. Are queens so powerful that eight of them cannot be placed on a board without any two of them attacking each other? Spend some time and see if you can create a solution. If you do not have access to a chessboard, printing a copy of this page will give you, from the following Figure, an empty board and eight queens.

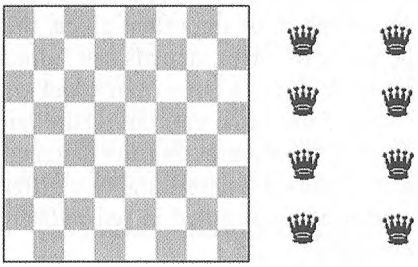


Figure 4.10. An empty board and eight queens

Remember, when considering a solution to this problem you must make sure not to place two queens in the same row or in the same column. To complicate matters, you also cannot place two queens on the same diagonal. An example of a non-solution appears in Listing 5. This is a non-solution since the queen in the upper left corner attacks the queen in the lower right corner, and vice-versa.

The Solution

To solve the *Eight Queens* problem, we use a backtracking algorithm. The algorithm involves placing queens, column by column, onto the chessboard. The algorithm terminates when it places all eight queens on the board with no two queens attacking each other. The algorithm backtracks when it reaches a situation where a new queen cannot be placed on the board without attacking a queen already on the board. When the algorithm reaches this situation, it moves the piece that it most recently added to the board to another location. The idea here is that moving this piece may create a combination that allows the algorithm to add more pieces. For example, consider the chessboard in Listing 6. Here, we have placed seven queens successfully in the first seven columns such that no two queens are attacking each other. We must backtrack, however, since no legal space in column eight exists where we can place the eighth queen.

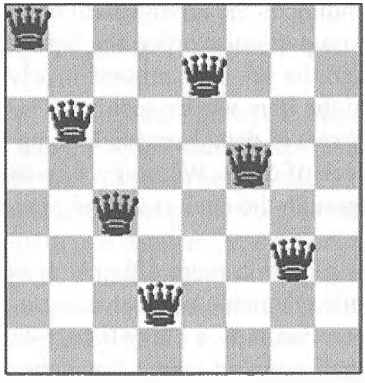


Figure 4.11. Seven queens placed, but we must backtrack

The backtracking portion of the algorithm would then attempt to move the seventh queen to another spot in the seventh column to open a spot in the eighth column for the eighth queen. If the seventh queen cannot be moved because no other legal spaces exist in column seven, the algorithm must remove that queen and back up and consider moving the queen in column six. Eventually, the algorithm repeats this process of placing queens and backing up until it finds a combination that solves the problem.

Sorting and Searching

5.1 Linear vs. Binary Search

Searching: A Common Task

Searching for data is a common task found not just in computer science but also in the real world. In the most basic sense, finding an element stored in a vector is similar to searching for a person's name in a telephone book. In both cases, we want to find one piece of data that exists somewhere in a larger set of similar data.

In both computer science and in the real world, the process involved in searching differs depending on the arrangement of the data set being searched. If a set is arranged in no particular order, the best search process we can use is a simple pass through the set. This process involves examining every item in the set until we find the item we are seeking. This is probably an adequate solution for only small sets of data. Imagine looking for a specific card from a completely shuffled deck of cards. With only fifty-two total cards in the deck, a front-to-back pass through the deck is a valid approach to finding a specific card.

Larger sets of data require a more efficient searching approach. Imagine trying to find a person's telephone number using a telephone book that lists people in random order. Examining every listing during a front-to-back pass through the book probably would take a few days. This is definitely not an efficient approach, and is the reason why telephone book listings are alphabetized. Leveraging our knowledge of how the telephone book arranges its listings, we can perform a more efficient search. This search involves opening the telephone book and comparing the listings on the page to the listing we are seeking. If we are searching for "Doe, John", and the page we opened the book to contains only listings for "Smith", we know we must flip to another page closer to listings for "Doe". We continue this approach until we reach the page containing our listing. This search probably only takes a minute to complete. This is fast compared to the time that a front-to-back pass through the book would take.

Both of these real world approaches to searching have computer-science algorithm analogues. A front-to-back pass through a data set is known as a *linear search*. A more formal variant of searching through the telephone book is known as a *binary search*.

Linear Search

A linear search is a simple search algorithm that finds an item by examining a data set in order. A linear search is sometimes known as a sequential search. In C++, we can implement a linear search as a for-loop using only a few lines of code.

```
// Finding an element in a vector using linear search
template <class T>
int linear_search(const vector<T>& v, const T& item) {

    for (int i = 0; i < v.size(); i++) {
        if (v[i] == item) {
            return i; // item found
        }
    }
    return -1; // item not found
}
```

Listing 5.1. Finding an element in a vector using a linear search

Besides the simplicity of its implementation, the main advantage of a linear search is that it does not require sorted data. Even if the data contained in a data set is stored in random order, a linear search still works as expected. This is because a linear search makes no assumption on the arrangement of the data. It simply examines every item in a data set until it finds the desired item.

The main disadvantage of a linear search is its suitability for only small data sets. Remember, a linear search examines all items in a set until it finds the item. In the best case, this happens when the first item examined by a linear search is the item. In the worst case, the search terminates when the last item is examined. In this situation, the search examines every item in the data set. For a vector or array that contains several million elements, this could take a considerable amount of time, especially if the search is repeated many times. On average, a linear search examines half the items in a data set.

The STL find function performs a linear search on a container. This function accepts three arguments. The first two arguments are iterators that specify a range to search. The third argument is the value that the function attempts to find. This function returns an iterator to the first position that contains the value sought. If the search is unsuccessful, the find function returns an iterator equal to the second iterator. An example usage of the find function appears in find.cpp. This program populates a vector with the first twenty-five Fibonacci numbers. It then allows the user to enter a number and reports whether or not the number is one of those Fibonacci numbers.

A binary search is a fast search algorithm suitable for data sets of any reasonable size encountered. Unlike a linear search, it is suitable even for very large data sets because it eliminates large numbers of comparisons. A binary search differs from a linear search in that it requires sorted data to operate successfully.

We can explore how a binary search operates by considering how it would find an element in the following vector.



Figure 5.1. A vector containing nine elements

Given the vector pictured in Figure 5.1, we can use a binary search to find the element that contains the value 9. All binary searches begin by comparing the item being sought against the item in the middle of the data set. In this example, the middle element contains the value 21.



Figure 5.2. Selecting the middle element

The value we are seeking is 9. Since 9 is less than 21, we know that 9 must be stored in an element somewhere to the left of the middle element. With this in mind, we can safely ignore the right half of the vector and continue our search, only considering the left half of the vector. Figure 5.3 demonstrates this partitioning of the vector.



Figure 5.3. Partitioning the vector

After we partition the vector, the middle element changes. We then compare the value stored here to the value we seek. Since 5 is less than 9, we know that 9 must be stored in the right half of this partition. We can then ignore the left half of this partition, and compare against the midpoint of the right half.



Figure 5.4. Partitioning the vector, again

Figure 5.4 highlights the midpoint of the current vector partition in green. The value stored here equals the value we seek so our search is complete. For this vector, which contains nine elements, only three comparisons were needed to find the element that stored the value 9. Starting from the left side of the vector, a linear search needs only three comparisons also to find the element that stores the value 9.

Even for a vector that contains a billion elements, a binary search algorithm needs to make at most thirty comparisons to find any element. An implementation of a binary search algorithm appears in Listing 5.2.

```
// Finding an element in a vector using binary search
template <class T>
int binary_search(const vector<T>& v, const T& item) {

    int low = 0;
    int high = v.size() - 1;
    int mid;

    while (low <= high) {

        // find the midpoint
        mid = (low + high) / 2;

        if (v[mid] < item) {
            low = mid + 1; // look in upper portion
        }
        else if (v[mid] > item) {
            high = mid - 1; // look in lower portion
        }
        else {
            return mid; // found it!
        }
    }

    return -1; // item not found
}
```

Listing 5.2. Finding an element in a vector using a binary search

The real advantage of binary search appears when we consider larger data

sets. The following table lists the maximum number of comparisons a binary search algorithm has to make to find any element in vectors of various sizes.

Table 5.1 Maximum comparisons of binary search

Size of vector	Max comparisons
10	4
100	7
1,000	10
10,000	13
100,000	17
1,000,000	20
10,000,000	23
100,000,000	27
1,000,000,000	30

The only disadvantage of a binary search is that it requires sorted data. Without sorted data, the midpoint comparisons are meaningless and the search would incorrectly report that an element is not present in the data set. Given a vector of unsorted items, how can we sort the items so that we can use a binary search? There are many approaches that we can take.

Instead of relying on their own binary search implementations, C++ programmers can use the STL binary_search function. The STL binary_search interface resembles the find function interface. The only difference is that function binary_search returns a bool value indicating the success of the search. An example use of the STL binary_search appears in wordsearch.cpp. This program stores the words contained in a text file into a vector. The user then enters a word via the console. The program then performs a binary search to see if the text file contains the word. A sample text file containing the full text of Nathaniel Hawthorne's *The House of the Seven Gables* can be found in internet.

5.2 Basic Sorting Algorithms

Sorting Overview

Many different basic sorting algorithms exist. Each of these algorithms has unique characteristics, behaviors, and requirements. For example, for the same set of data, one algorithm may perform far fewer comparisons than another

algorithm. Similarly, during execution another algorithm may swap the positions of data items far less often than another algorithm. The behavior of some algorithms differ when presented with data that is almost sorted as opposed to data that is completely shuffled. It is the differences between the set of these properties that make each sorting algorithm unique. These characteristics also make an algorithm more or less applicable in certain situations.

Because professional programmers rarely use basic sorting algorithms, we examine only one algorithm in this page. In page Fast Sorting Algorithms, we examine a sorting algorithm that is typically faster than all basic sorting algorithms.

Selection Sort

Selection sort is a basic sorting algorithm that works by making iterations, or passes, through the data being sorted. Each pass results in the placement of one element into its correct location. As a result, each subsequent pass has to inspect one fewer item than the previous pass.



Figure 5.5. An unsorted array

Figure 5.5 contains an unsorted array. Since this array contains eight elements, a selection sort must make seven passes to sort the data. Each pass places one element into the position that will result in a sorted array. The next two figures represent the first two passes, respectively. In each figure, the filled arrow indicates the position that the pass is seeking to fill. During each pass, the algorithm looks for the smallest remaining element. The pass ends with the algorithm swapping the smallest element into the position under consideration.

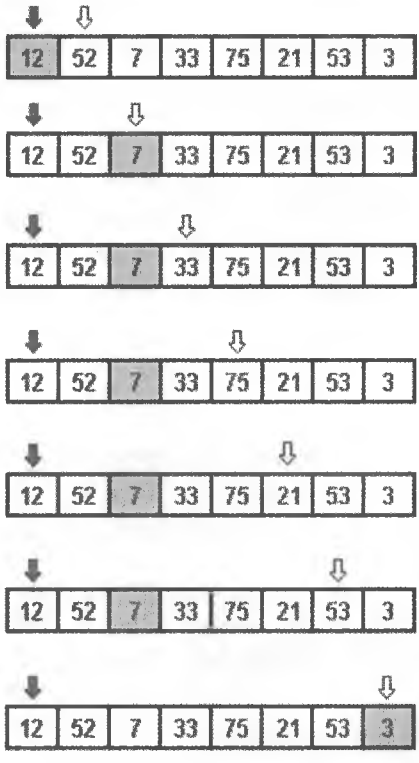


Figure 5.6. The first pass

After the first pass, the lowest element is placed into the first position of the array. The next pass, illustrated by Figure 5.7 places the second lowest element into the second position of the array.

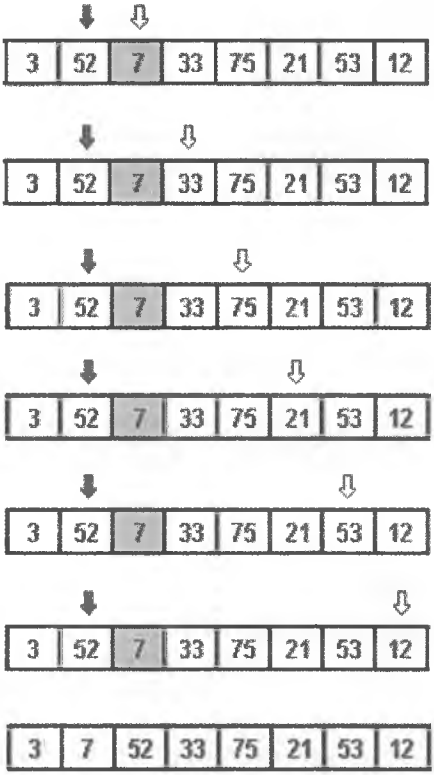


Figure 5.7. The second pass

The selection sort algorithm always makes one fewer passes than the number of elements in the array. This is true even if the original array is already sorted. The algorithm has no mechanism to detect when the array is sorted, so it must make all required passes. A C++ implementation of the selection sort appears in Listing 5.3.

```
// Sorting using selection sort
template <class T>
void selection_sort(vector<T>& v) {

    for (int i = 0; i < v.size() - 1; i++) {
        int best = i;
        for (int j = i + 1; j < v.size(); j++) {
            if (v[j] < v[best]) {
                best = j;
            }
        }

        if (best != i) {
            T temp = v[i];
            v[i] = v[best];
            v[best] = temp;
        }
    }
}
```

Listing 5.3. Selection sorting algorithm

Mergesort

We now turn our attention to mergesort. Mergesort runs in worst-case runningtime, and the number of comparisons used is nearly optimal. It is a fine example of arecursive algorithm.

The fundamental operation in this algorithm is merging two sorted lists. Because thelists are sorted, this can be done in one pass through the input, if the output is put in athird list. The basic merging algorithm takes two input arrays *A* and *B*, an output array *C*,and three counters, *Actr*, *Bctr*, and *Cctr*, which are initially set to the beginning of theirrespective arrays. The smaller of *A*[*Actr*] and *B*[*Bctr*] is copied to the next entry in *C*, andthe appropriate counters are advanced. When either input list is exhausted, the remainderof the other list is copied to *C*. An example of how the merge routine works is provided forthe following input.

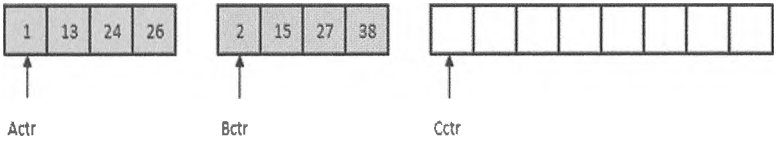


Figure 5.8

If the array *A* contains 1, 13, 24, 26, and *B* contains 2, 15, 27, 38, then the algorithmproceeds as follows: First, a comparison is done between 1 and 2. 1 is added to *C*, andthen 13 and 2 are compared.

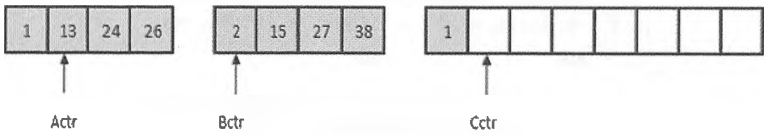


Figure 5.9.

2 is added to *C*, and then 13 and 15 are compared.

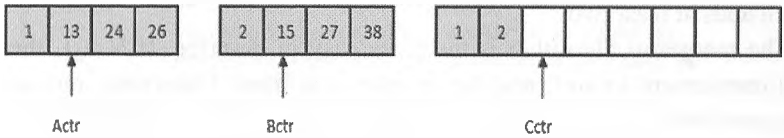


Figure 5.10.

13 is added to *C*, and then 24 and 15 are compared. This proceeds until 26 and 27 arecompared.

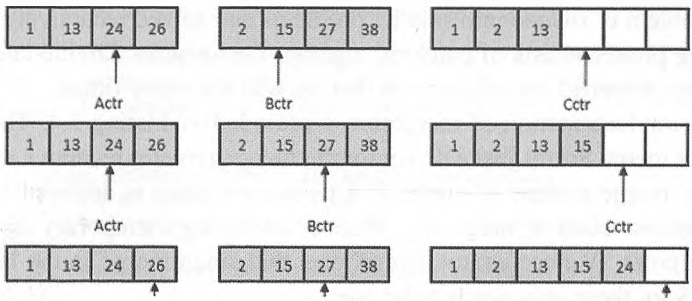


Figure 5.11.

26 is added to *C*, and the *A* array is exhausted.

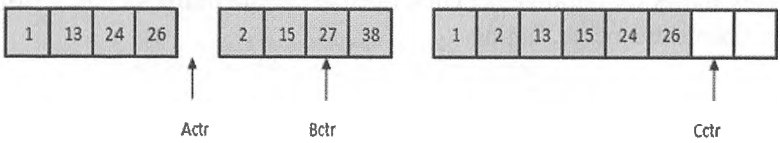


Figure 5.12.

The remainder of the *B* array is then copied to *C*.

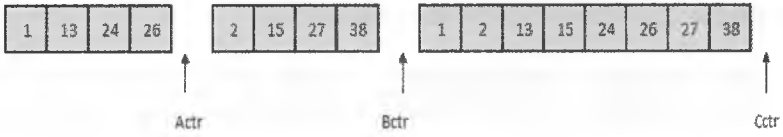


Figure 5.13.

The time to merge two sorted lists is clearly linear, because at most $N-1$ comparisonsare made, where N is the total number of elements. To see this, note that every comparisonadds an element to *C*, except the last comparison, which adds at least two.

The mergesort algorithm is therefore easy to describe. If $N = 1$, there is only oneelement to sort, and the answer is at hand. Otherwise, recursively mergesort the first halfand the second half. This gives two sorted halves, which can then be merged togetherusing the merging algorithm described above. For instance, to sort the eight-element array 24, 13, 26, 1, 2, 27, 38, 15, we recursively sort the first four and last four elements, obtaining1, 13, 24, 26, 2, 15, 27, 38. Then we merge the two halves as above, obtaining the finallist 1, 2, 13, 15, 24, 26, 27, 38. This algorithm is a classic divide-and-conquer strategy. Theproblem is *divided* into smaller problems and solved recursively. The *conquering* phaseconsists of patching together the answers. Divide-and-conquer is a very powerful use ofrecursion that we will see many times.

An implementation of mergesort is provided in Listing 5.4. The one-parameter merge Sort is just a driver for the four-parameter recursive mergeSort.

The merge routine is subtle. If a temporary array is declared locally for each recursivecall of merge, then there could be log$N$ temporary arrays active at any point. A close examinationshows that since merge is the last line of mergeSort, there only needs to be one.

Merge sort routinestemporary array active at any point, and that the temporary array can be created in thepublic mergeSort driver. Further, we can use any part of the temporary array; we will usethe same portion as the input array a. This allows the improvement described at the end ofthis section. Listing 5.5 implements the merge routine.

```
6                 // Mergesort algorithm (driver).
7     template <typename Comparable>
8         void mergeSort( vector<Comparable> & a )
9     {
10        vector<Comparable> tmpArray( a.size( ) );
11        mergeSort( a, tmpArray, 0, a.size( ) - 1 );
12    }
13
14    template <typename Comparable>
15        void mergeSort( vector<Comparable> & a,
16        vector<Comparable> & tmpArray, int left, int right )
17    {
18            if( left < right )
19    {
20            int center = ( left + right ) / 2;
21        mergeSort( a, tmpArray, left, center );
22        mergeSort( a, tmpArray, center + 1, right );
23        merge( a, tmpArray, left, center + 1, right );
24    }
25    }
```

Listing 5.4. Mergesort algorithm

```
7   template <typename Comparable>
8
9   void merge( vector<Comparable> & a, vector<Comparable> & tmpArray,
10      int leftPos, int rightPos, int rightEnd )
11  {
12      int leftEnd = rightPos - 1;
13      int tmpPos = leftPos;
14      int numElements = rightEnd - leftPos + 1;
15
16                                          // Main loop
17  while( leftPos <= leftEnd && rightPos <= rightEnd )
18      if( a[ leftPos ] <= a[ rightPos ] )
19      tmpArray[ tmpPos++ ] = std::move( a[ leftPos++ ] );
20      else
21      tmpArray[ tmpPos++ ] = std::move( a[ rightPos++ ] );
22
23  while( leftPos <= leftEnd )             // Copy rest of first half
24      tmpArray[ tmpPos++ ] = std::move( a[ leftPos++ ] );
25
26  while( rightPos <= rightEnd )           // Copy rest of right half
27      tmpArray[ tmpPos++ ] = std::move( a[ rightPos++ ] );
28
29                                          // Copy tmpArray back
30  for( int i = 0; i < numElements; ++i, --rightEnd )
31  a[ rightEnd ] = std::move( tmpArray[ rightEnd ] );
32  }
```

Listing 5.5

Analysis of Merge sort

Merge sort is a classic example of the techniques used to analyze recursive routines: We have to write a recurrence relation for the running time. We will assume that N is a powerof 2 so that we always split into even halves. For N = 1, the time to merge sort is constant,which we will denote by 1. Other wise, the time to merge sort N numbers is equal to the as a Listing 5.5.

Merge routine time to do two recursive merge sorts of size $N/2$, plus the time to merge, which is linear. The following equations say this exactly:

$$T(1) = 1$$
$$T(N) = 2T(N/2) + N \tag{5.1}$$

This is a standard recurrence relation, which can be solved several ways. We will show two methods. The first idea is to divide the recurrence relation through by N. The reason fordoing this will become apparent soon. This yields.

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1 \tag{5.2}$$

This equation is valid for any N that is a power of 2, so we may also write

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1 \tag{5.3}$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{n/8} + 1$$

and

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1 \tag{5.4}$$

Now add up all the equations. This means that we add all of the terms on the left-hand sideand set the result equal to the sum of all of the terms on the right-hand side. Observe that the term $T(N/2)/(N/2)$ appears on both sides and thus cancels. In fact, virtually all the terms appear on both sides and cancel. This is called telescoping a sum. After everything is added, the final result is

$$\frac{T(N)}{N} = \frac{T(1)}{1} + \log N \tag{5.5}$$

because all of the other terms cancel and there are log N equations, and so all the 1s at the end of these equations add up to log N. Multiplying through by N gives the final answer.

$$T(N) = N \log N + N = O(N \log N) \tag{5.6}$$

Notice that if we did not divide through by N at the start of the solutions, the sum would not telescope. This is why, it was necessary to divide through by N.

An alternative method is to substitute the recurrence relation continually on the right-handside. We have

$$T(N) = 2T(N/2) + N \tag{5.7}$$

Since we can substitute $N/2$ into the main equation,

$$2T(N/2) = 2(2(T(N/4)) + N/2) = 4T(N/4) + N \tag{5.8}$$

we have

$$T(N) = 4T(N/4) + 2N \tag{5.9}$$

Again, by substituting $N/4$ into the main equation, we see that

$$4T(N/4) = 4(2T(N/8) + N/4) = 8T(N/8) + N \tag{5.10}$$

So we have

$$T(N) = 8T(N/8) + 3N \tag{5.11}$$

Continuing in this manner, we obtain

$$T(N) = 2kT(N/2k) + k \cdot N \tag{5.12}$$

Using $k = \log N$, we obtain

$$T(N) = NT(1) + N \log N = N \log N + N \tag{5.13}$$

The choice of which method to use is a matter of taste. The first method

tends toproduce scrap work that fits better on a standard 81/2 × 11 sheet of paper leading to fewer mathematical errors, but it requires a certain amount of experience to apply. The second method is more of a brute-force approach.

Recall that we have assumed $N = 2k$. The analysis can be refined to handle cases when N is not a power of 2. The answer turns out to be almost identical (this is usually the case).

Although mergesort's running time is $O(N \log N)$, it has the significant problem that merging two sorted lists uses linear extra memory. The additional work involved in copying to the temporary array and back, through out the algorithm, slows the sort considerably.

This copying can be avoided by judiciously switching the roles of a and tmp Array at alternate levels of the recursion. A variant of mergesort can also be implemented nonrecursively.

The running time of mergesort, when compared with other $O(N \log N)$ alternatives, depends heavily on the relative costs of comparing elements and moving elements in the array (and the temporary array). These costs are language dependent.

For instance, in Java, when performing a generic sort (using a Comparator), an element comparison can be expensive (because comparisons might not be easily inlined, and thusthe overhead of dynamic dispatch could slow things down), but moving elements is cheap (because they are reference assignments, rather than copies of large objects). Merge sortuses the lowest number of comparisons of all the popular sorting algorithms, and thus is agood candidate for general-purpose sorting in Java. In fact, it is the algorithm used in thestandard Java library for generic sorting.

On the other hand, in classic C++, in a generic sort, copying objects can be expensive ifthe objects are large, while comparing objects often is relatively cheap because of the ability of the compiler to aggressively perform inline optimization. In this scenario, it might be reasonable to have an algorithm use a few more comparisons, if we can also use significantly fewer data movements. *Quicksort,* which we discuss in the next section, achieves this trade off and is the sorting routine that has been commonly used in C++ libraries. NewC++11 move semantics possibly change this dynamic, and so it remains to be seen whet her quick sort will continue to be the sorting algorithm used in C++ libraries.

Quicksort

Quicksort is a fast sorting algorithm that uses a divide-and-conquer prob-

lem solving approach. Unlike the basic sorting algorithms we have already examined, quick sort uses recursion. Given an array of elements to sort, the algorithm recursively divides the array into smaller and smaller arrays. Quick sort then sorts these very small arrays and combines the results to create a sorted version of the original array. Because of its recursive nature, the quick sort implementation can be hard to understand. Before examining an implementation, we consider the idea behind the algorithm.

The quick sort algorithm can be summarized in four basic steps. Presented in the context of sorting an array, these steps are as follows.

1. If the size of the array is zero or one, then return the array. This is the base case.
2. Select an element from the array to be used as the pivot element. This is the pivot selection step.
3. Create two new arrays. Place all the elements from the original array that are less than the pivot element into one of these sub-arrays and all the elements that are greater than the pivot element into the other sub-array. This is the partitioning step.
4. Return the array that contains the result of the quick sorted sub-array that contains the elements less than the pivot, followed by the pivot, followed by the result of the quicksorted sub-array that contains the elements greater than the pivot. This is the recursive divide step.

Stepping through an example illustrates how these four steps sort a set of data. Consider the array presented in Figure 5.8.

12	52	7	33	75	21	53	3

Figure 5.14. An unsorted array

Since the array in Figure 5.14 contains more than one element, quick sort enters step two and selects a pivot element. There are many different strategies we can use to pick this pivot element. One that is simple involves choosing the middle element of the array as the pivot element. This element contains the value 33. After selecting the pivot element, quicksort partitions the remaining elements into two sub-arrays. One sub-array contains the elements of the partitioned array whose values are less than 33, and the other sub-array contains the elements whose values are greater than 33. Figure 5.15 illustrates this first pivot and partition step. In this figure, a circle denotes the pivot element.

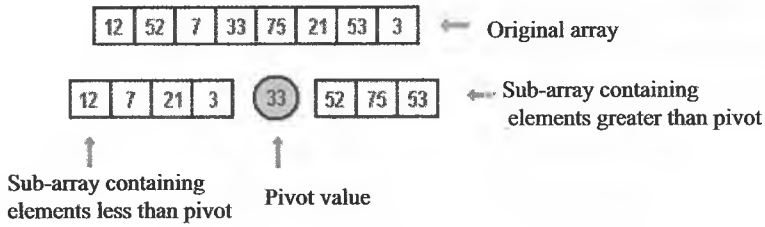


Figure 5.15. Pivot selection and partition

The first partition creates two smaller arrays. The quicksort algorithm then recursively calls itself on these arrays. Notice that both of these arrays contain two or more elements. Therefore, the algorithm selects pivot elements for each array and partitions their remaining elements into smaller arrays. The result of this appears in Figure 5.16.

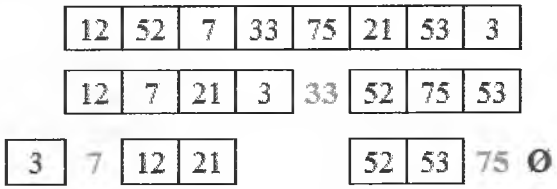


Figure 5.16. Second level partitioning

The quick sort algorithm only needs to select pivot elements and partition sub-arrays that contain more than one element. From Figure 3, we see after two partitions we are left with four sub-arrays. These arrays appear in the bottom row of the figure. Starting from the left side of the figure, the first sub-array contains only one element (the value 3). The second sub-array contains elements 12 and 21, and the third sub-array contains elements 52 and 53. The fourth sub-array contains zero elements. The empty set sign (a circle with a diagonal line through it) denotes that this sub-array contains zero elements. The quick sort algorithm does not need to pivot and partition the first and fourth arrays since they each contain less than two elements. At this point, these two sub-arrays are sorted. Since the second and third arrays each contain more than one element, there is the possibility they are not in sorted order. Quick sort must recursively pivot and partition these arrays. This is demonstrated in Figure 5.17.

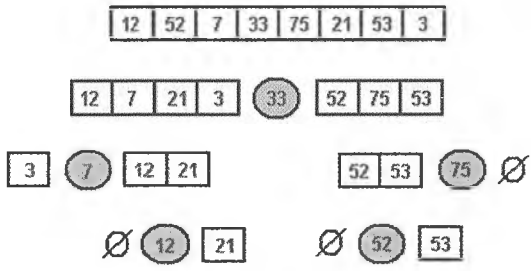


Figure 5.17. After last necessary partition

The recursive nature of quicksort breaks down the original array into smaller and smaller sub-arrays. The pivoting and partitioning of these sub-arrays eventually results in the sorting of the entire array. Figure 5.18 depicts the sorted version of the original array from our ongoing example.

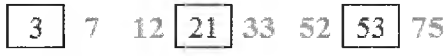


Figure 5.18. The sorted array

An Implementation

A quicksort implementation appears in Listing 5.5. This quicksort implementation uses a basic pivot selection strategy of selecting the middle element. This is a simple approach to implement, but one that can lead to efficiency problems when the algorithm selects either the smallest or largest value as the pivot. In these cases, the performance of the algorithm degrades since the algorithm can only recursively quicksort one sub-array instead of two.

A more advanced quicksort implementation appears on page 347 of the *Weiss* textbook. The Weiss implementation uses a different pivot and partition strategy known as *median-of-three partitioning*. This optimized strategy improves quicksort's performance by preventing the algorithm from selecting the minimum or maximum value as the pivot. This implementation also specifies a minimum array size that the algorithm will recursively operate on. When presented with a portion of an array that is smaller than this cut-off size, the Weiss quicksort implementation performs an insertion sort.

```
template <class T>
void quick_sort(vector<T>& v, int low, int high) {

    // Do not solve recursively when faced with
    // only 1 or 2 elements
    if (low == high) {
        return ;
    }
    else if (low + 1 == high) {
        if (v[low] > v[high]) {
            swap(v[low], v[high]);
            return ;
        }
    }

    // select pivot
    int middle = (low + high) / 2;
    T pivot = v[middle];
    swap(v[middle], v[high]);

    // partition
    int i;
    int j;
    for (i = low, j = high - 1; ;) {

        while (v[i] < pivot && i < j) i++;
        while (pivot < v[j] && i < j) j--;

        if (i < j) {
            swap(v[i], v[j]);
        }
        else {
            break;
        }
    }

    // place pivot in correct location
    if (i != high - 1 && j != high - 1) {
        swap( v[i], v[high]);
    }

    // quicksort sub-vectors
    if (i == low && j == low) {
        quick_sort(v, low + 1, high);
    }
    else if (i == high - 1 && j == high - 1) {
        quick_sort(v, low, high - 1);
    }
    else {
        quick_sort(v, low, i - 1);
        quick_sort(v, i + 1, high);
    }

}
```

Listing 5.6. A quicksort implementation

Analysis of Quicksort

Like mergesort, quicksort is recursive; therefore, its analysis requires solving a recurrenceformula. We will do the analysis for a quicksort, assuming a random pivot (no median-of-three partitioning) and no cutoff for small arrays. We will take $T(0) = T(1) = 1$, as inmergesort. The running time of quicksort is equal to the running time of the two recursivecalls plus the linear time spent in the partition (the pivot selection takes only constanttime). This gives the basic quicksort relation.

$$T(N) = T(i) + T(N - i - 1) + cN \tag{5.15}$$

where $i = |S_1|$ is the number of elements in S_1. We will look at three cases.

Worst-Case Analysis

The pivot is the smallest element, all the time. Then $i = 0$, and if we ignore $T(0) = 1$,
which is insignificant, the recurrence is

$$T(N) = T(N - 1) + cN, N > 1 \tag{5.16}$$

We telescope, using Equation (7.2) repeatedly. Thus,

$$T(N - 1) = T(N - 2) + c(N - 1) \tag{5.17}$$
$$T(N - 2) = T(N - 3) + c(N - 2) \tag{5.18}$$

$$\vdots$$

$$T(2) = T(1) + c(2) \tag{5.19}$$

Adding up all these equations yields

$$T(N) = T(1) + c \sum_{i=2}^{N} i = \Theta(N^2) \tag{5.20}$$

as claimed earlier. To see that this is the worst possible case, note that the total cost of allthe partitions in recursive calls at depth d must be at most N. Since the recursion depth isat most N, this gives an $O(N2)$ worst-case bound for quicksort.

Best-Case Analysis

In the best case, the pivot is in the middle. To simplify the math, we assume that the two subarrays are each exactly half the size of the original, and although this gives a slight over estimate, this is acceptable, because we are only interested in a Big-Oh answer.

$$T(N) = cN \log N + N = \Theta(N \log N) \tag{5.21}$$

Using the STL Sorting Functions

The Standard Template Library includes functions that programmers can use to sort a variety of containers. One of these functions is the sort function. The sort function performs a fast sort (typically a quicksort) on a container.

```
 9: int random_less_than_50(void) {
10:
11:     return (rand() % 50);
12: }
13:
14: int main(int argc, char* argv[]) {
15:
16:     // seed the random number generator
17:     srand( time(0) );
18:
19:     // create a vector with ten random elements (all less than 50)
20:     vector<int> v;
21:     back_insert_iterator< vector<int> > back_itr(v);
22:     generate_n(back_itr, 10, random_less_than_50);
23:
24:     // display its contents
25:     ostream_iterator<int> out(cout, " ");
26:     copy(v.begin(), v.end(), out);
27:
28:     // sort the vector
29:     sort(v.begin(), v.end());
30:
31:     // display its contents, sorted
32:     cout << endl;
33:     copy(v.begin(), v.end(), out);
34:
35:     return EXIT_SUCCESS;
36: }
```

Listing 5.7. Using the sort algorithm

Listing 5.7 uses a few other functions that warrant explanation. First, the STL function generate_n is used in Listing 2 to populate a vector with ten random numbers. This function takes as parameters an output iterator, a number of items to generate, and a function to use to generate each item. The generated items are placed in the output iterator specified in the first parameter. In this

case, the output iterator is a back_insert_iterator for the vector v. Thus, all items generated by the function in the third parameter are placed at the end of vector v. The function specified in the third parameter is a wrapper for the C-library function rand. The function rand returns a random number from the runtime's random number generator. This random number can be quite large, so a wrapper function is used to reduce the number returned to a smaller range of values.

The sort function can operate only on containers that provide random access iterators. Containers such as vector, deque, and string all provide random access iterators. Since the STL list container does not provide random access iterators the general-purpose sort function on a list object results in a compile-time error. Since linked lists often need to be sorted, the list container has a special sort member function. Listing 5.8 demonstrates its use.

In addition to sorting, the STL provides the ability to *shuffle* the contents of a container. Shuffling, the opposite of sorting, is the random arrangement of data in a set. The following listing demonstrates the STL function random_shuffle.

```
14: int main(int argc, char* argv[]) {
15:
16:     // seed the random number generator
17:     srand( time(0) );
18:
19:     // create a list with ten random elements
20:     // (all less than 50)
21:     list<int> ls;
22:     back_insert_iterator< list<int> > back_itr(ls);
23:     generate_n(back_itr, 10, random_less_than_50);
24:
25:     // display its contents
26:     ostream_iterator<int> out(cout, " ");
27:     copy(ls.begin(), ls.end(), out);
28:
29:     // sort the list
30:
31:     // The following can NOT be used since lists
32:     // do not provide random access iterators
33:
34:     // sort(ls.begin(), ls.end());
35:
36:     ls.sort();
37:
38:     // display its contents again
39:    cout << endl;
40:    copy(ls.begin(), ls.end(), out);
41:
42:    return EXIT_SUCCESS;
43: }
```

Listing 5.8. Sorting a list

```
 1: #include <iostream>
 2: #include <cstdlib>
 3: #include <ctime>
 4: #include <list>
 5:#include  <vector>
 6:#include <deque>
 7:#include <algorithm>
 8:
 9:using namespace  std;
10:
11:int main (int argc, char * argv []) {
12: // create a deque with 10 elements
13: deque<int> d;
14: for (int i = 1; i <= 10; i++) {
15:     d.push_back(i);
16: }
17:
18: ostream_iterator<int> out(cout, " ");
19:
20: for (int j = 1; j <= 5; j++) {
21:
22:     // display its contents
23:     copy(d.begin(), d.end(), out);
24:     cout << endl;
25:
26:     random_shuffle(d.begin(), d.end());
27:     copy(d.begin(), d.end(), out);
28:     cout << endl << endl;
29:
30:     sort(d.begin(), d.end());
31: }
```

Listing 5.9. Shuffling a container

5.3 Using Hash Tables

Overview of Hash Tables

A *hash table* is a data structure that supports very fast element insertion, removal, and retrieval. A hash table that is properly configured for the elements it contains can support these operations in a fixed amount of time. This is unlike any other data structures and algorithms that we have examined.

A hash table is a type of map. Maps are *associative* structures. Conceptually, associative data structures store data in key-value pairs. This means that for every value stored there is a corresponding key used to access the value. A real world example of an associative data structure is a dictionary. In a dictionary, data is stored in key-value pairs. The keys are the words, and the values are the definitions. To access a definition, one must use the corresponding key.

Hash maps and hash sets are two different types of hash tables. Hash maps are associative structures that store key-value pairs, whereas hash sets are structures that keep track of the membership of items within a collection. Hash sets do not map keys to values. They merely store a collection of keys. A list of commonly misspelled words, for instance, could be stored using a hash set. In this example, there is no mapping of a key to a value like there was in the dictionary example. Each word serves as only a key. A hash set is then used to report whether or not a word exists in this list of words.

Hash Functions

The hash function depends on the key type. Strictly speaking, we need a different hash function for each kind of key that might be used. For efficiency, we generally want to avoid explicit type conversion, striving instead for a throwback to the idea of considering the binary representation of keys in a machine word as an integer that we can use for arithmetic computations. Hashing predates high-level languages – on early computers, it was common practice to view a value as a string key at one moment and an integer the next. Some high-level languages make it difficult to write programs that depend on how keys are represented on a particular computer, because such programs, by their very nature, are machine dependent and therefore are difficult to transfer to a new or different computer. Hash functions generally are dependent on the process of transforming keys to integers, so machine independence and efficiency are sometimes difficult to achieve simultaneously in hashing implementations. We can typically hash simple integer or floating-point keys with just a single machine operation, but string keys and other types of compound keys require more care and more attention to efficiency.

Operations using hash tables are efficient because the position of a stored value can be calculated using the key. Hash tables are implemented typically as an array of values. A *hash function* is used to map a key to an index within this array. This index is where the key's corresponding value is stored. Other search algorithms, such as a linear or binary search, cannot map a key to its value's position as quickly as hash tables. These algorithms must perform more comparisons to find the index of the stored value. The time this search takes to complete increases as the number of stored items increases.

Table 5.2 A sample hash function

Data	Letter	Ascii	Index
Hanson, Bob	a	97	7
Adderly, Maria	d	100	0
Cruz, Jose	r	114	4
Song, Albert	o	111	1
Appleton, Mary	p	112	2

A mapping of keys to indexes is demonstrated in Table 5.2. In this figure, the hash function generates an index based on the rightmost digit of the ASCII value of the second letter of the person's last name. For example, in the name "Hanson, Bob", the second letter is an "a". The ASCII value of "a" is 97. The rightmost digit of 97 is 7. Therefore, the record for "Hanson, Bob" is stored at index 7 of the hash table.

Table 5.3 A sample hash table

0	Adderly, Maria
1	Song, Albert
2	Appleton, Mary
3	
4	Cruz, Jose
5	
6	
7	Hanson, Bob
8	
9	

The hash table in Table 5.3 shows each name in its mapped position. The advantage of a hash table is that to find an entry, one only has to apply the hash function to the key.

A popular method used in implementing a hash function is the *division method*. Like all hash functions, the division method maps a key to an index within the hash table. The division method implementation involves the conversion of a key to a variable of type unsigned int. Then, this value is divided by the size of the hash table. The remainder of this division is used as the index.

```
1: class hash_function {
2:
3: public:
4:
5:     unsigned int mm;
6:
7:     hash_function(unsigned int m = 6151)
8:             : mm(m) {}
9:
10:    unsigned int operator()(const string& s) const {
11:        unsigned int res = 0;
12:        for (int i = 0; i < s.size(); i++) {
13:            res = res * mm + s[i];
14:        }
15:        return res;
16:    }
17: };
```

Listing 5.10. A hash function based on the division method

A problem arises when we consider that there are typically many more keys than there are locations within a hash table. This means that a hash function can potentially map two or more distinct keys to the same index. Consider if we tried to add the name "Anderson, Jeff" to the hash table pictured in Figure 2. Applying the hash function yields an index of 0. The name "Adderly, Maria", however, already exists at index 0. When a hash function maps more than one key to the same index a *collision* occurs.

Hash table implementations are complicated by the fact that they must handle potential collisions. These mechanisms to handle collisions are discussed in detail in chapter 20 of the *Weiss* textbook. Basically, collisions decrease the performance of hash table operations. The best way to reduce the number of collisions, and thus increase the efficiency of a hash table, is to use a good hash function. A good hash function evenly distributes the mapping of keys throughout the positions in the hash table.

It is good software engineering practice to separate the hash function from the implementation of the hash table. This allows a programmer to customize a hash function to the data stored by a particular instance of a hash table. Listing 1 defines a hash table as a class that overloads operator ().

Memoizing: An Application of Hash Tables

When it is computationally expensive to calculate a function value y = f(x), it may be a good idea to store the value for future use. In other words, we calculate f(x) only the first time the function value on this particular input x is required, and then store (x, f(x)) in a hash table. Any future requests for f(x)

will then result in a table lookup and be much faster than a recalculation. This technique is called *memoizing*.

Note that we have to modify the function a bit to take advantage of the stored values. Here is the structure of a memoized function.

```
int f_memo(int x) {

  if ( there is an entry (x,y) in hash table ) {
    return y;
    }
    else {
    compute y = f(x) directly
    store (x,y) in table
    return y;
  }
}
```

Example 5.1. A typical memoized function

Since the hash table has to persist between function calls, it would be natural to organize this as a class that overloads operator ().

A Hash Table Implementation

The following header file and implementation file declares and defines a template hash map class. The class takes four template parameters. The first is the key type and the second is the value type. The third parameter specifies the hash function and the fourth parameter specifies how to compare two objects of the key's data type.

- *hashmap.h*- declaration of class HashMap

The return type of the search member function of class HashMap is an STL class that we have not yet encountered. Function search returns an object of type pair. Class pair is an STL class that contains two data members. These members are named first and second. This class is used often by functions that need to return more than one piece of data. The search member function needs to return whether or not the item was found and what position it was found in. Thus, function search returns an object of type class pair<bool, int>.

The file *hash-driver.cpp* demonstrates use of class HashMap and its member functions.

The choice of the hashing method that is best suited for a particular application depends on many different factors, as we have discussed when examining the methods. All the methods can reduce the symbol-table search and insert operations to constant-time operations, and all are useful for a broad variety of applications. Roughly, we can characterize the three major methods (linear probing, double hashing, and separate chaining) as follows: Linear probing is the fastest of the three (if sufficient memory is available to ensure that the table is sparse), double hashing makes the most efficient use of memory (but requires extra time, to compute the second hash function), and separate chaining is the easiest to implement and deploy (provided that a good storage allocator is available). Table 5.1 gives empirical data and commentary on the performance of the algorithms.

The choice between linear probing and double hashing depends primarily on the cost of computing the hash function and on the load factor of the table. For sparse tables (small a), both methods use only a few probes, but double hashing could take more time because it has to compute two hash functions for long keys. As a approaches 1, double hashing far outperforms linear probing.

Comparing linear probing and double hashing against separate chaining is more complicated, because we have to account precisely for memory usage. Separate chaining uses extra memory for links; the open-addressing methods use extra memory implicitly within the table to terminate probe sequences. The following concrete example illustrates the situation: Suppose that we have a table of M lists built with separate chaining, that the average length of the lists is 4, and that items and links each occupy a single machine word. The assumption that items and links take the same amount of space is justified in many situations because we would replace huge items with links to the items. With these assumptions, the table uses 9M words of memory (4M for items and 5M for links), and delivers an average search time of 2 probes. But linear probing for 4M items in a table of size 9M requires just $(1 + 1/(1 - 4/9))/2 = 1.4$ probes for a search hit, a value that is 30 percent faster than separate chaining for the same amount of space; and linear probing for 4M items in a table of size 6M requires 2 probes for a search hit (on the average), and thus uses 33 percent less space than separate chaining for the same amount of time. Furthermore, we can use a dynamic method such as Listing 5.1 to ensure that the table can grow while staying sparsely populated.

The argument in the previous paragraph indicates that it is not normally justifiable to choose separate chaining over open addressing on the basis of performance. However, separate chaining with a fixed M is often chosen in practice for a host of other reasons: it is easy to implement (particularly re-

move); it requires little extra memory for items that have preallocated link fields for use by symbol-table and other ADTs that may need them; and, although its performance degrades as the number of items in the table grows, the degradation is graceful and takes place in a manner that is unlikely to harm the application because it still is a factor of M faster than sequential search.

Many other hashing methods have been developed that have application in special situations. Although we cannot go into details, we consider three examples briefly to illustrate the nature of specially adapted hashing methods.

One class of methods moves items around during insertion in double hashing to make successful search more efficient. In fact, Brent developed a method for which the average time for a successful search can be bounded by a constant, even in a full table (see reference section). Such a method might be useful in applications where search hits are the predominant operation.

Another method, called ordered hashing, exploits ordering to reduce the cost for unsuccessful search in linear probing to be close to the cost for successful search. In standard linear probing, we stop the search when we find an empty table position or an item with a key equal to the search key; in ordered hashing, we stop the search when we find an item with a key greater than or equal to the search key (the table must be constructed cleverly if this procedure is to work) (see reference section). This improvement by introducing ordering in the table is on the same order as what we achieved by ordering the lists in separate chaining. This method is designed for applications where search misses predominate.

A symbol table that has a fast search miss and somewhat slower search hit can be used to implement an exception dictionary. For example, a text-processing system might have an algorithm for hyphenating words that works well for most words, but does not work for bizarre cases (such as "bizarre"). Only a few words in a huge document are likely to be in the exception dictionary, so nearly all the searches are likely to be misses.

These examples are only a few of a large number of algorithmic improvements that have been suggested for hashing. Many of these improvements are interesting and have important applications. Our usual cautions must be raised against premature use of advanced methods except when the requirements are serious and the performance/complexity tradeoffs are carefully considered, because separate chaining, linear probing and double hashing are simple, efficient, and acceptable for most applications.

The problem of implementing an exception dictionary is an example of an application where we can recast our algorithm slightly to optimize performance for the most frequently performed operation – in this case, search miss. For example, suppose that we have a 1000-item exception dictionary, have 1 million items to look up in the dictionary, and expect virtually all the searches to end as misses. This situation might arise if the items were bizarre

English-language words or random 32-bit integers. One way to proceed is to hash all the words to, say, 15-bit hash values (table size about 2^{16}). The 1000 exceptions occupy 1/64 of the table, and most of the 1 million searches end immediately with search misses, finding the empty table position on the first probe. But if the table contains 32-bit words, we can do much better by converting it into a bit-exception table and using 20-bit hash values. If we have a search miss (as we do most of the time), we finish the search with one bit test; a search hit requires a secondary test in a smaller table. The exceptions occupy 1/1000 of the table; search misses are by far the most likely operation; and we accomplish the task with 1 million directly indexed bit tests. This solution exploits the basic idea that a hash function produces a short certificate that represents a key – an essential concept that is useful in applications other than symbol-table implementations.

Hashing is preferred to the binary-tree structures of Chapters 6 as the symbol-table implementation for many applications, because it is somewhat simpler and can provide optimal (constant) search times, if the keys are of a standard type or are sufficiently simple that we can be confident of developing a good hash function for them. The advantages of binary-tree structures over hashing are that the trees are based on a simpler abstract interface (no hash function need be designed); the trees are dynamic (no advance information on the number of insertions is needed); the trees can provide guaranteed worst-case performance (everything could hash to the same place even in the best hashing method); and the trees support a wider range of operations (most important, sort and select). When these factors are not important, hashing is certainly the search method of choice, with one more important proviso: When keys are long strings, we can build them into data structures that can provide for search methods that are even faster than hashing.

Trees and graphs

6.1 Trees

A Tree as a Data Structure

A *tree* is a data structure that generally supports element insertion, access, and removal in less than linear time. Trees consist of an arrangement of *nodes* such that the following conditions hold.

- Only one node has no predecessor: the *root*.
- Every node other than the root has a unique predecessor.
- Starting at any node, one can reach the root by repeatedly stepping from a node to its predecessor.

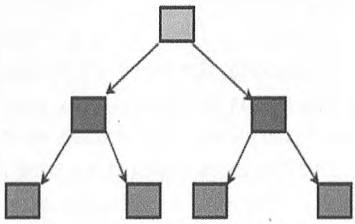


Figure 6.1. A basic tree

All nodes other than the root node have a predecessor, or *parent* node. These nodes are said to be *child* nodes of the parent. Nodes that have no children are *leaf* nodes. The *height* of a tree is the number of levels of nodes, including the root node, which exist in the tree. The tree in Figure 6.1 has a height of three and contains four leaf nodes.

Trees have many applications in Computer Science. The representation of categories and sub-categories for an online bookstore, for instance, is a common application of trees. Programmers use trees to represent file systems. Trees are also used in the implementation of compilers and in file compression algorithms.

Types of Trees

Tree implementations can be categorized into different types. This cate-

gorization process takes into account several properties and characteristics of trees. A property that categorizes a tree implementation is the maximum number of children that can exist in any given node. If the maximum number of children in any given node is not limited in an implementation, we consider the tree to be a *general hierarchy*.

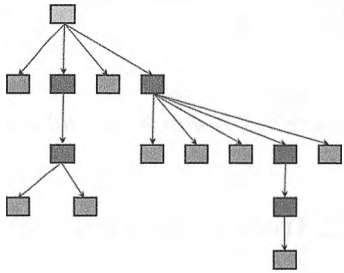


Figure 6.2. A general tree structure

Tree implementations that limit the number of child nodes for any given node form a separate class of trees. The most common type of these implementations limits the number of nodes to two. These types of trees are known as *binary trees*. Less commonly used implementations that limit the number of child nodes to three is known as a *ternary trees*.

Binary trees can be further categorized based on the ordering of elements within the tree. A *heap* is a binary tree that maintains elements in either increasing or decreasing order. At each level in a heap, the value at a node is either less than or greater than the values at all nodes below it. Two types of heaps exist: *min heaps* and *max heaps*. A min heap contains elements that always have increasingly larger values at each level in the tree. This means the root node of the tree contains the smallest element in the tree. Max heaps have their largest element stored at the root and store their remaining elements in decreasing order.

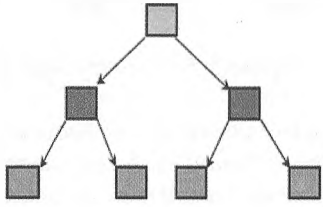


Figure 6.3. A binary tree

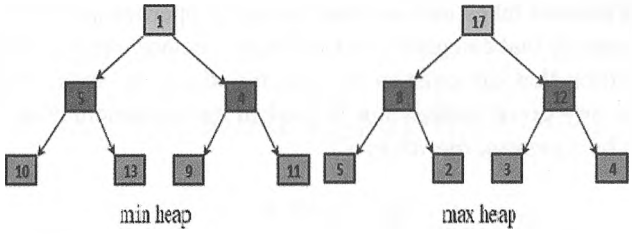


Figure 6.4. A min heap and a max heap

6.2 Trees as Search Structures

Binary search trees are another type of binary tree characterized by the ordering of the elements with the tree. Binary search trees maintain elements in a sorted order. Because of this, binary search trees effectively support searching for individual elements.

Binary search trees maintain items in a sorted order. All elements less than the element at the root of the tree are stored in the tree rooted at the root's left child. All elements greater than the root are stored in the tree rooted at the root's right child. This principle is applied recursively to all nodes in the tree. For any node in a binary search tree, all elements in the sub-tree in the left child are less than the element in the node. Also, all elements in the sub-tree in the right child are greater than the element in the node.

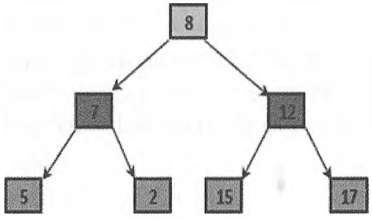


Figure 6.5. A binary search tree

Not all binary search trees have all leaf nodes at the same level in the tree. Binary search trees can be *unbalanced*. An unbalanced tree is a tree that has left and right sub-trees whose heights differ by more than one level. Binary search trees become unbalanced when elements of either increasing or decreasing size are continually added to the structure. The performance of a binary search tree approaches linear time as the structure becomes too unbalanced.

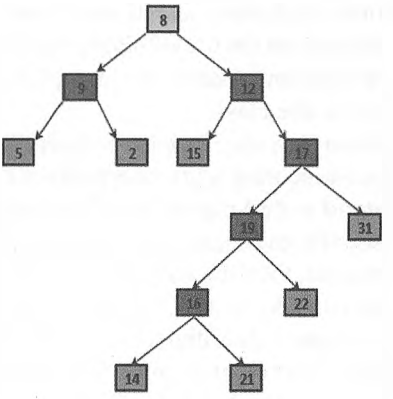


Figure 6.6. An unbalanced binary search tree

The nodes in a binary search tree do not have to contain two child nodes. A binary search tree limits, but does not require the number of child nodes to be two. A node in a binary search tree can contain only one child. A binary search tree that contains one or more of these types of nodes is an *incomplete* tree.

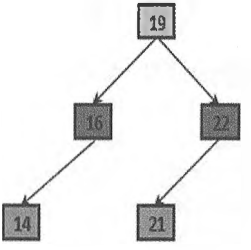


Figure 6.7. An incomplete binary search tree

The BSTree Class Interface

A binary search tree is a tree based data structure that maintains an ordered collection of items. Because of how they maintain the ordering of their stored items, binary search trees can perform certain tasks more efficiently than other data structures. For example, traversing items in sorted order (or in reverse sorted order) is an efficient operation in a binary search tree. This is not the case for a hash table. The ability to search for a range of items is another task easily accomplished using a binary search tree, but not using a hash table.

The implementation of a binary search tree class is covered in chapter 19 of *Weiss*. This page focuses on the use of binary search trees to solve problems relating to storing a collection of items. As such, we examine the interface of an existing binary search tree class.

There are many tasks that may have to be perfomed on a tree structure; a common one is that of executing a given operation P on each element of the tree. P is then understood to be a parameter of the more general task of visting all nodes or, as it is usually called, of tree traversal. If we consider the task as a single sequential process, then the individual nodes are visited in some specific order and may be considered as being laid out in a linear arrangement. In fact, the description of many algorithms is considerably facilitated if we can talk about processing the next element in the tree based in an underlying order.

Listing 6.1 contains the declaration of class `BSTree`. This is a binary search tree class that supports operations for element insertion, access, and removal. It also contains a method that returns the number of elements currently stored in the binary search tree.

```
 1: template <class T>
 2: class BSTree {
 3:
 4: protected:
 5:     BSTNode<T> *root;     // root of tree
 6:     int count;            // size of tree
 7:
 8: public:
 9:     BSTree() : root(NULL), count(0) {}
10:     BSTree(const BSTree&);
11:     virtual ~BSTree() {if (root) delete root;}
12:
13:     virtual int size() const { return count; }
14:     virtual bool insert( const T& x );
15:     virtual const T* const search( const T& x );
16:     virtual bool remove( const T& x );
17:
18: protected:
19:     virtual BSTNode<T>* copy_tree(BSTNode<T>* nodep);
20:     virtual bool insert_helper(BSTNode<T> *&nodep, const T &x);
21:     virtual const T* const search_helper(BSTNode<T> * nodep, const T &x);
22:     virtual bool remove_helper(BSTNode<T> *&nodep, const T &x);
23:     virtual BSTNode<T>* remove_leftmost_child(BSTNode<T> *&nodep);
24:
25: };
```

Listing 6.1. The BSTree interface

Class *BSTree* is a template class. This allows us to create instances of the class that can store different data types. There is an important consideration we must make before using this class. Internally, some of the methods of the class use operators ==, < and > to perform the element insertion, access, and removal tasks. Thus, any object we store in this binary search tree class must provide these operators.

The public data members of class *BSTree* that we use the most have rather straightforward usage. We use the *insert* and *remove* operations to add and delete elements to and from the binary search tree. Both of these functions return a `bool` value.

The *insert* function returns *true* when the element is successfully added and *false* when the element already exists. The *remove* function returns `true` if the element is successfully removed and *false* if the element is not found in the binary search tree. Function *size* returns the count of the number of elements in the binary search tree.

```
10:template <typename T>
11:const T* const BSTree<T>::search_helper(BSTNode<T> *nodep, const T &x) {
12:     if (nodep == 0) {
13:         return NULL;
14:     }
15:
16:     if (x == nodep->data) {
17:         return &(nodep->data);
18:     }
19:
20:     if (x < nodep->data) {
21:         return search_helper(nodep->left, x);
22:     }
23:     else {
24:         return search_helper(nodep->right, x);
25:     }
26:}
```

Listing 6.2. Internal use of operators of type `T`

The `search` function provides a way to access elements that exist in the binary search tree. If the requested element is not found in the binary search tree, the function returns the null pointer. Otherwise, the function returns a constant pointer to a constant version of the element. The pointer must be constant to prevent a user from attempting to gain access to another portion of memory through pointer arithmetic. The element the pointer points to must be constant to ensure that the integrity of the tree is maintained. Changing the value of a stored element could possibly change where the element should be stored in the binary search tree. The correct process of updating a stored element involves first removing the element, then updating a copy of the element, then inserting the updated copy.

An Application: Customer Database

The following files comprise a program that manages a list of customer contacts. A binary search tree is a good choice to use in this implementation since it allows for efficient element insertion and access.

- main.cpp – Main application source code
- Customer.h – Declaration of a customer class
- Customer.cpp – Implementation of class Customer
- CustomerDatabase.h – Declaration of a customer database class
- CustomerDatabase.cpp – Implementation of class Customer Database
- bst.h – A binary search tree implementation
- contacts.dat – A sample data file.

The complete code of the program is provided in appendix B.

Using Heaps

Heaps are often used to provide efficient access to either the minimum or maximum of a set of values. One use of this functionality is the implementation of a priority queue. A priority queue is a data structure that behaves similar to regular queue in that it provides `push`, `pop`, `size`, and `empty` methods. Priority queues differ from regular queues since the element that a priority queue `pop` removes depends on an assigned priority, and not the "First-In First-Out" principle. Priority queues are used often enough that STL provides a `priority_queue` adapter. The `priority_queue` adapter is covered in detail is given below.

6.3 Using the Tree Based STL Containers

Sets

The STL `set` container stores unique items in an ordered collection. The tasks of element addition, removal, and access are guaranteed by the STL standard to take logarithmic time in a `set`. Inserting items in sorted order will not degrade these operations to linear time, as is the case with binary search trees.

A set could be the implementation structure for the vertices and/or edges in a trees. One property of a set is that putting an item, which is already there, into the set doesnot change the set. These properties can prove useful indetermining whether items occur in a passage of text, for example. All the characters inthe text could be put into a set. When the process is finished, the set contains exactlyone copy of every item that appeared in the text.There are two basic ways to implement sets. The first explicidy records the presence or absence of each item in the base type (itemType) in the representation of the set variable. The second records only those items that are in a set variable at a particular time.If an item is not listed as being in the set, it is not in the set. That is, the presence of each item in the set is explicitly recorded; the absence of an item is implicit.

Declaring objects of class `set` is similar to declaring other STL containers. A programmer specifies the data type that the set can contain as a template parameter. To access the `set` container, a programmer must include the `<set>` library.

```
1:  set<int> s1;
2:  set<int> s2;
```

Listing 6.3. Declaring set objects

Adding items to a `set` is done using the `insert` member function.

```
1: set<int> s1;
2: set<int> s2;
3:
4: for (int i = 0; i < 20; i++) {
5:     s1.insert(i);
6:     s2.insert(30 - i);
7: }
```

Listing 6.4. Adding items to a set

The member function `size` returns the number of items stored in a set.

```
 1: set<int> s1;
 2: set<int> s2;
 3:
 4: for (int i = 0; i < 20; i++) {
 5:     s1.insert(i);
 6:     s2.insert(30 - i);
 7: }
 8:
 9: cout << "size of s1: " << s1.size() << endl;
10: cout << "size of s2: " << s2.size() << endl;
```

Listing 6.5. Reporting the number of items stored in a set

The `find` function searches for an item in a `set`. This function returns an iterator to the found item. If the item is not found, function `find` returns an iterator equal to the iterator returned by function `end`.

```
1: if (s1.find(10) != s1.end()) {
2:     cout << "s1 contains 10\n";
3: }
```

Listing 6.6. Searching for an item in a set

The STL `set` container can be used in conjunction with the STL generic algorithms. Four of the STL generic algorithms are geared specifically for use with sets. These are functions `set_intersection`, `set_union`, `set_difference`, and `set_symmetric_difference`. The following listing demonstrates the use and effect of these functions. It is important to understand that these four functions can also be used on other STL containers (such as `vectors` and `deques`). They are more efficient, however, when used with sets. This is because the STL `set` container maintains items in sorted order.

```
ostream_iterator<int> out(cout, " ");

// Set intersection
cout << "\nset intersection: ";
set_intersection(s1.begin(), s1.end(),
                 s2.begin(), s2.end(),
                 out);

// Set union
cout << "\nset union: ";
set_union(s1.begin(), s1.end(),
          s2.begin(), s2.end(),
          out);

// Set difference
cout << "\nset difference: ";
set_difference(s1.begin(), s1.end(),
               s2.begin(), s2.end(),
               out);

// Set symmetric difference
cout << "\nset symmetric difference: ";
set_symmetric_difference(s1.begin(), s1.end(),
                         s2.begin(), s2.end(),
                         out);
```

Listing 6.7. The set STL algorithms

Maps

The STL `map` container is an associative structure that stores data in key-value pairs. The `map` container is considered associative since it maps, or associates, one piece of data (a key) with another piece of data (a value).

Declaring an object of type `map` involves specifying both the key and value data types. The key is specified with the first template parameter, and the value

is specified with a second template parameter. For instance, the following listing declares two `map` objects. The first is a map of type `string` to `string`. The second is a map of type `string` to `int`.

```
1: // A map of strings to strings
2: map<string, string> m1;
3:
4: // A map of strings to ints
5: map<string, int> m2;
```

Listing 6.8. Declaring STL map objects

Inserting elements into a map requires both the key and value. The following listing inserts two key-value pairs into a map. In this listing, the keys are "apple" and "orange". They map to the values "a small red fruit" and "a small orange fruit", respectively. Note that the version of the `insert` method used here takes only one parameter. To use this method correctly, we have to place the key and value into an object of type `pair<string, string>`. If we were inserting data into a map of type `int` to `double`, for instance, we would have to place our keys and values in objects of type `pair<int, double>` to use the `insert` method.

```
1: // A map of strings to strings
2: map<string, string> m1;
3:
4: // A map of strings to ints
5: map<string, int> m2;
6:
7: m1.insert(pair<string, string>("apple", "a small red fruit"));
8: m1.insert(pair<string, string>("orange", "a small orange fruit"));
```

Listing 6.9. Using the insert function

Using the `insert` method of the STL `map` container is an effective but cumbersome method to add data to a map. An alternative mechanism for adding items to a map is the overloaded double bracket operator. The following listing uses this operator to insert an item into a map.

```
1: M1["banana"] = "a long yellow fruit";
```

Listing 6.10. A second way to add items to a map

The overloaded double bracket operator also provides a way to access values based on their keys.

```
1: cout << m1["apple"] << endl;
2: cout << m1["orange"] << endl;
3: cout << m1["banana"] << endl;
```

Listing 6.11. Accessing items in a map

Similar to the `set` container, the `map` container provides `size`, `find`, and `count` member functions. A programmer can also iterate through the items stored in a map. The implementation of a map iteration warrants examination since it is slightly more complex than a standard container iteration.

```
1: map<string, string>::iterator it = m1.begin();
2: for ( ; it != m1.end(); it++) {
3:
4:     cout << it->first << ": " << it->second << endl;
5: }
```

Listing 6.12. Traversing the items in a map

The above listing iterates through the elements of the map object `m1`. The important aspect to notice is how the key and value of each object is accessed. Internally, a map stores the key and value for each data item in an object of type `pair<key, value>`. Therefore, an iterator pointing to an item in a map points to an object of the same `pair<key, value>` type. To access an item's key through an iterator, one must dereference the iterator and then use the `pair` object's `first` data member. The data member `second` provides access to the value.

Keys stored in a `map` must be unique. Put another way, a key can only map to one value. There is another STL container, a `multimap`, which allows keys to map to more than one value.

Priority Queues

The STL `priority_queue` adapter provides to programmers an interface suitable for as a priority queue. As their name suggests, priority queues behave similar to regular queues. Priority queues, however, allow access to only the item with the highest priority.

```
 1: #include <iostream>
 2: #include <cstdlib>
 3: #include <queue>
 4:
 5: using namespace std;
 6:
 7: int main(int argc, char* argv[]) {
 8:
 9:     priority_queue<int> pq;
10:
11:     pq.push(1);
12:     pq.push(4);
13:     pq.push(2);
14:
15:     cout << pq.top() << endl; // outputs '4'
16:     pq.pop();
17:     cout << pq.top() << endl; // outputs '2'
18:     pq.pop();
19:     cout << pq.top() << endl; // outputs '1'
20:
21:     cout << pq.size() << endl; // outputs '1'
22:
23:     return EXIT_SUCCESS;
24: }
```

Listing 6.13. Basic use of the priority queue

The `priority_queue` adapter requires some mechanism to determine the relative priority of elements. By default, the `priority_queue` adapter uses the < operator of the elements to determine priority. Many data types and classes define this operator, but many classes, especially user-defined classes, do not. To use a `priority_queue` with objects that define the < operator is straightforward. The above listing does exactly this.

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <queue>

using namespace std;

class StringCompare {
public:
    bool operator()(const string& s1, const string& s2) {

        if (s1.length() < s2.length()) {
            return true;
        }
        else {
            return false;
        }
    }
};

int main(int argc, char* argv[]) {

    // Create a priority queue that assigns longer
    // strings a higher priority.
    priority_queue<string, vector<string>, StringCompare> pq;

    pq.push("small string");
    pq.push("a slightly longer string");
    pq.push("another small string");

    cout << pq.top() << endl;
    pq.pop();

    cout << pq.top() << endl;
    pq.pop();

    cout << pq.top() << endl;

    return EXIT_SUCCESS;
}
```

Listing 6.14. A user defined compare function

As long as the semantics of the supplied < make sense for the application at hand, the implementation is relatively painless.There is an alternative mechanism that a programmer can implement to use a `priority_queue` with objects that do not provide a < operator. This mechanism involves providing the `priority_queue` adapter with a compare function. This compare function dictates how the `priority_queue` adapter should determine the priority of the stored items.

Supplying a compare function to the `priority_queue` is also advanta-

geous when an existing < operator is not suitable for use in determining priority. The above listing actually demonstrates this concept since class `string` does provide a < operator. If the listing did not supply the user-defined compare function, the priority queue would use the < operator. This would cause the program to assign the highest priority to the string that alphabetically precedes the other strings.

6.4 Introduction to Graphs

Graphs in the Real World

Many rich examples of graphs exist in the real world. You may be using one such example, the Internet, right now. The interconnected networks that make up the Internet are a common example of a graph. The computers form the nodes of the graph. Since not every machine is connected to every other machine, the links between machines also are part of the graph. Each such link between a pair of computers may be directed or undirected, depending on whether messages can be sent in only one direction or in both.

A great many problems are naturally formulated in terms of objects and connections between them. Electric circuits are another obvious example where interconnections between objects play a central role. Circuit elements like transistors, resistors, and capacitors are intricately wired together. Such circuits can be represented and processed within a computer in order to answer simple questions like "Is everything connected together?" as well as complicated questions like "If this circuit is built, will it work?" In this case, the answer to the first question depends only on the properties of the interconnections (wires), while the answer to the second question requires detailed information about both the wires and the objects that they connect.

Actually, we've already encountered graphs in several instances in previous chapters. Linked data structures are actually representations of graphs, and some of the algorithms that we'll see for processing graphs are similar to algorithms that we've already seen for processing trees and other structures.

A similar example is traffic networks, such as railroad links between stations, or airplane flights connecting cities. The nodes here are railroad stations and cities, and the links are given by the rails and the flights.

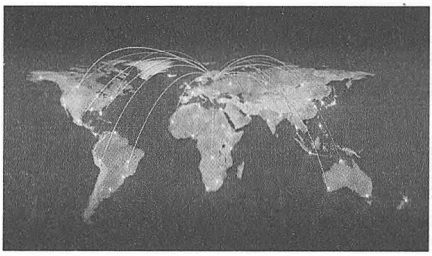


Figure 6.8. Flights connecting cities as a graph

A third example is "job scheduling," where the objects are tasks to be performed, say in a manufacturing process, and interconnections indicate which jobs should be done before others. Here we might be interested in answering questions like "When should each task be performed?" A graph is a mathematical object which accurately models such situations. In this chapter, we'll examine some basic properties of graphs, and in the next several chapters we'll study a variety of algorithms for answering questions of the type posed above.

Graph theory is a major branch of combinatorial mathematics and has been intensively studied for hundreds of years. Many important and useful properties of graphs have been proved, but many difficult problems have yet to be resolved. We'll be able here only to scratch the surface of what is known about graphs, covering enough to be able to understand the fundamental algorithms.

As with so many of the problem domains that we've studied, graphs have only recently begun to be examined from an algorithmic point of view.

Although some of the fundamental algorithms are quite old, many of the interesting ones have been discovered within the last ten years. Even trivial graph algorithms lead to interesting computer programs, and the nontrivial algorithms that we'll examine are among the most elegant and interesting (though difficult to understand) algorithms known.

Lastly, consider a company's organizational chart. In this case, the nodes are the employees, and links indicate the supervisor-subordinate relationship. You may be tempted to think that this type of hierarchy is actually a tree, but we have to consider the fact that an employee may have more than one supervisor. For example, in the following organizational chart the "QA Manager" reports to both the "V.P. of Sales" and the "V.P. of Technology." This prohibits a tree-based representation.

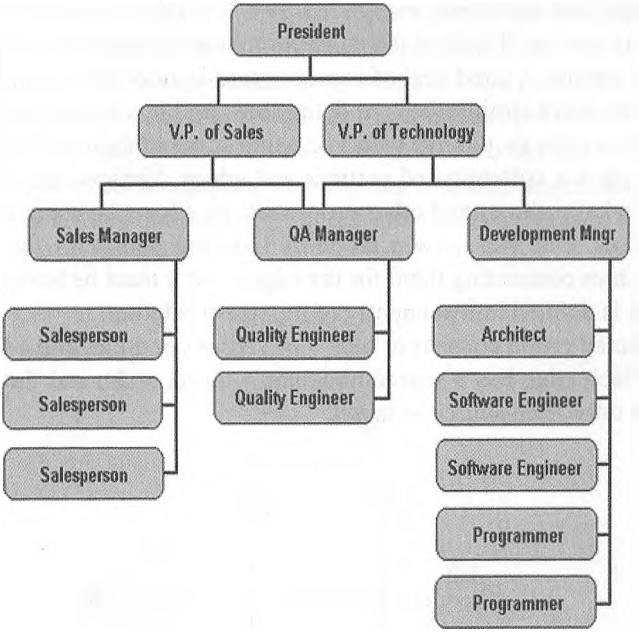


Figure 6.9. An organizational chart as a graph

A third example is "job scheduling," where the objects are tasks to be performed, say in a manufacturing process, and interconnections indicate which jobs should be done before others. Here we might be interested in answering questions like "When should each task be performed?" A graph is a mathematical object which accurately models such situations. In this chapter, we'll examine some basic properties of graphs, and in the next several chapters we'll study a variety of algorithms for answering questions of the type posed above.

Graphs Defined

Graphs are one of the most versatile structures used in computer programming. Thesorts of problems that graphs can help solve are generally quite different from those we've dealt with thus far in this book. If you're dealing with general kinds of data storage problems, you probably won't need a graph, but for some problems – and they tend to be interesting ones – a graph is indispensable.

Our discussion of graphs is divided into two part. In this part we'll cover the algorithms associated with unweighted graphs, show some algorithms that

these graphs can represent, and present two Workshop applets to model them. In the next part we'll look at the more complicated algorithms associated with weighted graphs. A good deal of nomenclature is associated with graphs. Most of the terms have straightforward definitions, and it is convenient to put them in one place even though we won't be using some of them until later.

A graph is a collection of vertices and edges. Vertices are simple objects which can have names and other properties; an edge is a, connection between two vertices. One can draw a graph by marking points for the vertices and drawing lines connecting them for the edges, but it must be borne in mind that the graph is defined independently of the representation.

A *directed graph* consists of a set `V` of vertices or nodes and a set `E` of edges or arcs. Each edge has a source node and a target node, and the edge can be traversed only from source to target.

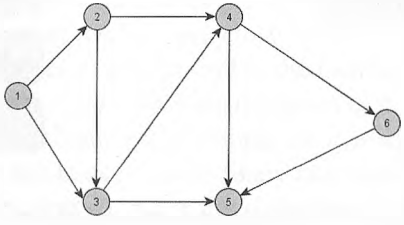


Figure 6.10. Adirected graph

Many of the basic questions one can ask about a graph revolve around path existence. A *path* is any sequence of vertices connected by edges. The number of edges in a path is the *length* of the path.

Given two nodes `s` and `t` (the source and the target), we would like to determine whether there is a path from `s` to `t`. This is the *single pair reachability problem*. Closely related is the *all pairs reachability problem*, which asks whether there is a path between any two vertices in the graph. In terms of a computer network, this translates into the ability to send a message from any machine to any other machine (assuming some suitable forwarding mechanism).

Several data structures can be used to represent a directed graph. The appropriate choice of data structure depends on the operations that will be applied to the vertices and arcs of the digraph. One common representation for a digraph $G = (V,E)$ is the *adjacency matrix*. Suppose $V = \{1, 2, \ldots, n\}$. The adjacency matrix for G is an n x n matrix A of booleans, where $A[i, j]$ is true if and only if there is an arc from vertex i to j. Often, we shall exhibit adjacency matrices with 1 for true and 0 for false; adjacency matrices may even be imple-

mented that way. In the adjacency matrix representation the time required to access an element of an adjacency matrix is independent of the size of V and E. Thus the adjacency matrix representation is useful in those graph algorithms in which we frequently need to know whether a given arc is present.

For the sake of completeness, we mention that one often deals with *undirected* graphs where each edge can be traversed in either direction. An undirected edge can always be thought of as a pair of directed edges, so we will focus on the directed case.

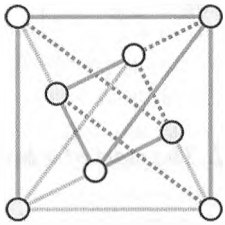


Figure 6.11. Anundirected graph

An undirected graph $G = (V, E)$ consists of a finite set of vertices V and a set of edges E. It differs from a directed graph in that each edge in E is an unordered pair of vertices.† If (v, w) is an undirected edge, then $(v, w) = (w, v)$. Hereafter we refer to an undirected graph as simply a graph.

Graphs are used in many different disciplines to model a symmetric relationship between objects. The objects are represented by the vertices of the graph and two objects are connected by an edge if the objects are related. In this chapter we present several data structures that can be used to represent graphs.

Perhaps the most appealing feature of graphs is that they can be rendered visually, and pictures of graphs are often helpful in motivating and describing important properties. Graph algorithms usually are best understood in conjunction with a little drawing that shows what the algorithm does.

6.5 Fundamental Graph Algorithms

Breadth-First Search.

The Concept. A *breadth-first search* is an algorithm that explores the nodes in a graph in order of increasing distance from the starting node. A breadth-first

search can be used to answer questions about reachability. In other words, we can perform a breadth-first search to determine which nodes can be reached from a starting node. This is an important application since graphs can contain nodes that are disconnected from other nodes.

Figures 6.12 through 6.18 demonstrate a breadth-first search of a simple graph.

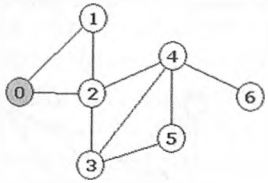


Figure 6.12. The start of a breadth-first search

The starting node in the breadth-first exploration is highlighted in Figure 6.12. The algorithm begins by examining each of the outgoing edges of this start node. New nodes are discovered when the algorithm examines these outgoing edges. These "discovered" nodes are placed into a queue. The breadth-first search algorithm then examines each discovered node in turn to determine if new nodes can be reached from the discovered node.

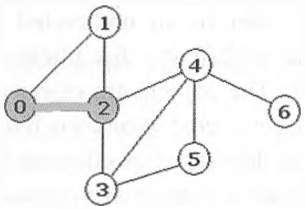


Figure 6.13. Node 2 discovered

In the figure above, we see that the algorithm has discovered node `2` by examining one of the edges connected to node `0`. The figure shades node `2` to denote that the node has been discovered. Once a node has been discovered, it is placed into a queue of nodes that have been discovered, but not explored. The algorithm then finishes exploring the starting node by examining its remaining edge. This edge leads to node `1`.

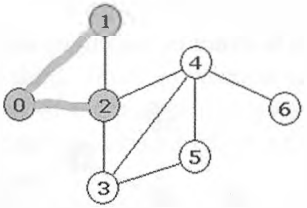


Figure 6.14. Node 1discovered

A node is completely explored when all of its edges have been examined. When the exploration of a node is complete, the algorithm selects another node to explore. This selection is not arbitrary. The chosen node is the node at the front of the queue of discovered, but not explored nodes. At this point in this algorithm, this means that the algorithm considers node 2. In Figures 4 and 5, we see the algorithm explore the edges this node.

After examining all the edges from node 2, the algorithm considers the edges of node 1. The edges connected to node 1, however, all lead to nodes that have already been discovered. The algorithm moves on and considers the edges connected to node 4.

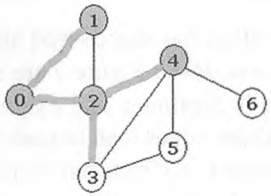


Figure 6.15. Node 4 discovered

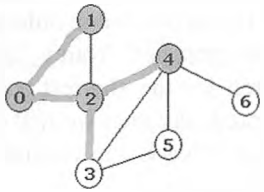


Figure 6.16. Node 3 discovered

Node 4 is connected to four edges. Two of these edges lead to nodes that have already been discovered. The other two edges lead to nodes 5 and 6.

Both of these nodes are marked as "discovered" and placed into the queue. The algorithm then moves on to examine the edges connected to node 3.

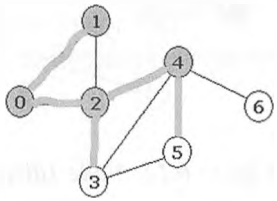


Figure 6.17. Node 5 discovered

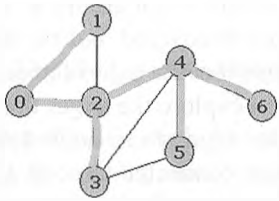


Figure 6.18. Node 6 discovered

At this point, the algorithm has discovered all the nodes in the graph. We can see this because all the nodes in Figure 7 are shaded. The algorithm, however, does not know this. It continues and examines the edges connected to nodes 3, 5, and 6. All of these edges lead to nodes previously discovered. The algorithm finally terminates when it runs out of nodes that need to be explored.

In this method we consider the single-source shortest path problem, given an edge-weighted graph and a vertex v, find the shortest weighted path to every other vertex V.

Why do we find the shortest path to every other vertex if we are interested only in the shortest path? It turns out that in order to find the shortest path from it is necessary to find the shortest path from v_0 to every other vertex.

Clearly, when we search for the shortest path, we must consider all the vertices. If a vertex is ignored, say then we will not consider any of the paths from that pass through. But if we fail to consider all the paths we cannot be assured of finding the shortest one.

Furthermore, suppose the shortest path passes through some intermediate node. I.e., the shortest path is of the form that the portion of P between v is also the shortest path from to suppose it is not. Then there exists another shorter path. But then, P would not be the shortest path, because we could obtain a shorter one by replacing the portion of P between v_0 and by the shorter path.

An Analogy

An analogy you might think about in relation to depth-first search is a maze. The maze perhaps one of the people-size ones made of hedges, popular in England consists of narrow passages (think of edges) and intersections where passages meet (vertices).

Suppose that someone is lost in the maze. She knows there's an exit and plans to traverse the maze systematically to find it. Fortunately, she has a ball of string and a marker pen. She starts at some intersection and goes down a randomly chosen passage, unreeling the string. At the next intersection, she goes down another randomly chosen passage, and so on, until finally she reaches a dead end.

```
1: void bfs(vector< list<int> >& adj_lists, int start_node) {
2:
3:     queue<int> not_yet_explored;
4:     set<int> discovered;
5:
6:     // Mark the start node as being discovered,
7:     // and place it in the queue of nodes to explore
8:     not_yet_explored.push(start_node);
9:     discovered.insert(start_node);
10:
11:    while (! not_yet_explored.empty()) {
12:
13:        // Get a node to explore.
14:        int node_to_explore = not_yet_explored.front();
15:        not_yet_explored.pop();
16:
17:        // Examine all the edges of the node
18:        list<int>::iterator edges = adj_lists[node_to_explore].begin();
19:        for ( ; edges != adj_lists[node_to_explore].end(); edges++) {
20:
21:            // See if the edge leads to a node that we
22:            // have not yet discovered
23:            if (discovered.count(*edges) == 0) {
24:
25:                // We have discovered a new node!
26:                // Add this node to the queue of nodes
27:                // to explore.
28:                discovered.insert(*edges);
29:                not_yet_explored.push(*edges);
30:
31:                cout << "Found " << *edges <<
32:                " from " << node_to_explore << endl;
33:
34:            }
35:        }
36:    }
37:
38: }
```

Listing 6.15. Breadth-first search implementation

At the dead end she retraces her path, reeling in the string, until she reaches the previous intersection. Here she marks the path she's been down so she won't take it again, and tries another path. When she's marked all the paths leading from that intersection, she returns to the previous intersection and repeats the process.

An Implementation

An implementation of a breadth-first search appears in Listing 6.13. This program explores the graph in Figures 6.12 through 6.18. Because the nodes in this graph are referenced via integers, we can use a vector of lists of type int to maintain adjacency lists.

Rule 1: Visit the next unvisited vertex (if there is one) that's adjacent to the current vertex, mark it, and insert it into the queue.

Rule 2: If you can't carry out Rule 1 because there are no more unvisited vertices, remove a vertex from the queue (if possible) and make it the current vertex.

Rule 3: If you can't carry out Rule 2 because the queue is empty, you're finished.

Depth-First Search

As we saw in the depth-first search, the algorithm acts as though it wants to get as far away from the starting point as quickly as possible. In the breadth-first search, on the other hand, the algorithm likes to stay as close as possible to the starting point. It visits all the vertices adjacent to the starting vertex, and only then goes further a field. This kind of search is implemented using a queue instead of a stack.

The Concept. A *depth-first search* is an algorithm that explores the nodes in a graph in reverse order of increasing distance from the starting node. A depth-first search explores nodes deeper into a graph and then works its way back to the nodes that are close to the starting node.

To search deeper into a maze first, a depth-first search algorithm keeps track of the nodes to be explored in a stack rather than a queue.

This causes newly discovered nodes to be explored before previously discovered nodes. In a breadth-first search, we explored nodes in the order that we discovered them. In a depth-first search, we explore nodes as we discover them.

From a given vertex *v*, the depth-first search strategy of graph traversal proceeds along a pathfrom *v* as deeply into the graph as possible before backing up. That is, after visiting a vertex, a depth-first search visits, if possible, an unvisited adjacent vertex.

The depth-first search strategy has a simple recursive form:

```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Recursive version.
depth-first search(v: Vertex)
Mark v as visited
for (each unvisited vertex u adjacent to v)
dfs(u)
```

Figures 6.19 through 6.25 illustrate a depth-first search of the same graph from the breadth-first search example.

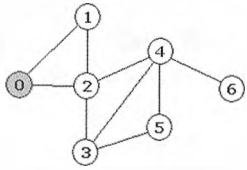


Figure 6.19. First step of a depth-first search

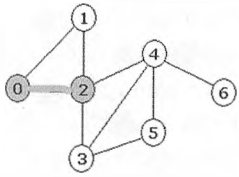


Figure 6.20. Second step of a depth-first search

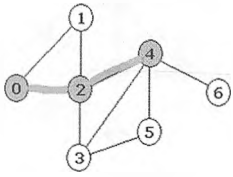


Figure 6.21. Third step of a depth-first search

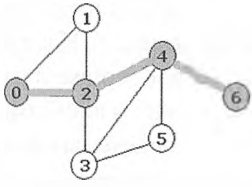


Figure 6.22. Fourth step of a depth-first search

Depth-first search yields valuable information about the structure of a graph. Perhaps the most basic property of depth-first search is that the predecessor subgraph $G\pi$ does indeed form a forest of trees, since the structure of the depth-first trees exactly mirrors the structure of recursive calls of DFS-VISIT. That is, $u = \pi[v]$ if and only if DFS-VISIT(v) was called during a search of u's adjacency list. Additionally, vertex v is a descendant of vertex u in the depth-firstforest if and only if v is discovered during the time in which u is gray.

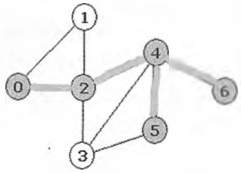


Figure 6.23. Fifth step of a depth-first search

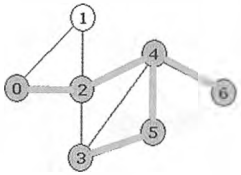


Figure 6.24. Sixth step of a depth-first search

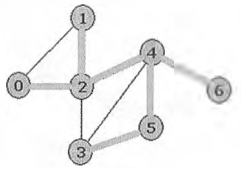


Figure 6.25. Final step of a depth-first search

Another important property of depth-first search is that discovery and finishing times have parenthesis structure. If we represent the discovery of vertex u with a left parenthesis "(u"and represent its finishing by a right parenthesis "u)", then the history of discoveries andfinishings makes a well-formed expression in the sense that the parentheses are properlynested. For example, the depth-first search of Figure 5.23 corresponds to the parenthesization shown in Figure 5.25. Another way of stating the condition of parenthesis structure is given in theorem.

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

• the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and neither u nor v is adescendant of the other in the depth-first forest,

• the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is adescendant of v in a depth-first tree, or

• the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is adescendant of u in a depth-first tree.

Proof We begin with the case in which $d[u] < d[v]$. There are two subcases to consider,according to whether $d[v] < f[u]$ or not. The first subcase occurs when $d[v] < f[u]$, so v wasdiscovered while u was still gray. This implies that v is a descendant of u. Moreover, since v was discovered more recently than u, all of its outgoing edges are explored, and v is finished,before the search returns to and finishes u. In this case, therefore, the interval $[d[v], f[v]]$ is entirely contained within the interval $[d[u], f[u]]$. In the other subcase, $f[u] < d[v]$, and in equality implies that the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.

The case in which $d[v] < d[u]$ is similar, with the roles of u and v reversed in the above argument. Another interesting property of depth-first search is that the search can be used to classify theedges of the input graph $G = (V, E)$. This edge classification can be used to glean important information about a graph.

We can define four edge types in terms of the depth-first forest $G\pi$ produced by a depth-first search on G.

- *Tree edges* are edges in the depth-first forest $G\pi$. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
- *Back edges* are those edges (u, v) connecting a vertex u to an ancestor v in a depthfirst tree. Self-loops, which may occur in directed graphs, are considered to be back edges.
- *Forward edges* are those non tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
- *Cross edges* are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

An Implementation

A depth-first search implementation differs from a breadth-first search implementation only in the container used to store unexplored nodes. In a breadth-first search, we explore nodes in the order that they are discovered. Thus, a

queue is used to maintain the discovered but not explored nodes. A depth-first search algorithm always explores the most recently discovered node. To facilitate this, a stack is used instead of a queue to store the unexplored nodes. In all actuality, though, we can leverage the fact that the run-time system maintains function calls in a stack, and code the routine recursively. This altogether eliminates the need to use a container to store unexplored nodes.

```
1:
2: void dfs_helper(vector< list<int> >& adj_lists, set<int>& discovered, int
3: node) {
4:
5:     // Examine all the edges of the node
6:     list<int>::iterator edges = adj_lists[node].begin();
7:     for ( ; edges != adj_lists[node].end(); edges++) {
8:
9:         // See if the edge leads to a node that we
10:        // have not yet discovered
11:        if (discovered.count(*edges) == 0) {
12:
13:            // We have discovered a new node!
14:            // Add this node to the queue of nodes
15:            // to explore.
16:            discovered.insert(*edges);
17:            cout << "Found " << *edges <<
18:            " from " << node << endl;
19:            dfs_helper(adj_lists, discovered, *edges);
20:        }
21:    }
22: }
23:
24: void dfs(vector< list<int> >& adj_lists, int start_node) {
25:
26:    // Mark the start node as being discovered
27:    set<int> discovered;
28:    discovered.insert(start_node);
29:    dfs_helper(adj_lists, discovered, start_node);
}
```

Listing 6.16. Depth-first search implementation

Shortest Path Calculation

Shortest path calculations are a family of algorithms that determine the shortest distance between nodes in a graph. Different shortest path algorithms exist because different types of graphs require slightly different approaches. In an unweighted graph, for instance, a simple breadth-first search can be used to calculate the shortest path between nodes. For weighted graphs, different and slightly more complex algorithms must be used. We can use *Dijkstra's algorithm* if all the weights in a weighted graph are positive. An algorithm known as the *Bellman-Ford algorithm* solves the shortest path problem for graphs that contain edges with negative weights.

If the graph is weighted, the problem (apparently) becomes harder, but we can still use theideas from the unweighted case.

```
void Graph::unweighted( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;
        s.dist = 0;
        q.enqueue( s );
while( !q.isEmpty( ) )
    {
    Vertex v = q.dequeue( );

    for each Vertex w adjacent to v
        if( w.dist == INFINITY )
        {
            w.dist = v.dist + 1;
            w.path = v;
            q.enqueue( w );
        }
    }
}
```

Listing 6.17. Pseudocode for unweighted shortest-path algorithm

We keep all of the same information as before. Thus, each vertex is marked as either*known* or *unknown.* A tentative distance *dv* is kept for each vertex, as before. This distanceturns out to be the shortest path length from *s* to *v* using only *known* vertices asintermediates. As before, we record *pv,* which is the last vertex to cause a change to *dv*.

The general method to solve the single-source shortest-path problem is known as Dijkstra's algorithm.This thirty-year-old solution is a prime example of a greedy algorithm.

Greedy algorithms generally solve a problem in stages by doing what appears tobe the best thing at each stage. For example, to make change in U.S. currency, most people count out the quarters first, then the dimes, nickels, and pennies. This greedy algorithm gives change using the minimum number of coins. The main problem with greedy algorithms is that they do not always work. The addition of a 12-cent piece breaks the coin-changing algorithm for returning 15 cents, because the answer it gives (one 12-cent piece and three pennies) is not optimal (one dime and one nickel).

Dijkstra's algorithm proceeds in stages, just like the unweighted shortest-path algorithm.At each stage, Dijkstra's algorithm selects a vertex, *v*, which has the smallest *dv* among all the *unknown* vertices and declares that the shortest path from *s* to *v* is *known.* The remainder of a stage consists of updating the values of *dw*. In the unweighted case, we set $dw = dv + 1$ if $dw = \infty$. Thus, we essentially lowered the value of *dw* if vertex *v* offered a shorter path. If we apply the same logic to the weighted case, then we should set dw = dv +cv,w if this new value for dw would be an improvement. Put simply, the algorithm decides whether or not it is a good idea to use v on the path to w.

To be necessary for program implementation of an algorithm two arrays: logical visited – for information storage about the visited peaks and numerical distance in which the found shortest ways will be skidded. So, there is a graph G= (V, E). Each of the peaks entering a great number of V is initially marked as not visited, i.e. false value is appropriated to array cells of visited. As the most favorable ways only should be found, in each element of a vector distance such number which certainly is more than any potential way registers (this number is normal call infinity, but in the program use, for example the maximum value of specific data type). As starting point the peak of s is selected and the zero way is attributed to it: distance[s]=0 since there is no edge from s in s (the method doesn't provide loops). Further, there are all adjacent peaks (in which there is an edge from s) [let t and u] will be those and in turn are researched, namely route cost from s in turn in each of them is calculated:

- distance[t]=distance[s] + weight of incident s and t of an edge;
- distance[u]=distance[s] + weight of incident s and u of an edge.

But it is quite probable that in this or that peak from s there are several ways therefore the price of a way to such peak in an array of distance should be reconsidered, then the greatest (nonoptimal) value is ignored, and the smallest to be put in compliance to peak.

After processing of peaks, adjacent to s, it is marked as visited: visited[s]=true, and the active becomes that peak, the way from s to which is minimum. Let's say the way from s to u is shorter, than from s in t, therefore, the peak of u to become the active above in a described way her neighbors, except for s peak are researched. Further, u is marked as passed: visited[u]=true, the active becomes t peak, and all procedure repeats for it.

Dijkstra's algorithm proceeds until all peaks, available from s, aren't probed.

Now on the specific graph (fig. 6.26) we will find all shortest ways be-

tween source and all remaining peaks, using Dijkstra's algorithm. The size (the number of edges) of the graph figured below is equal to 7 (|E |=7), and an order (quantity of peaks) – 6 (|V |=6). It is the weighed graph, some numerical value therefore route value optionally is defined by number of the edges lying between pair of peaks is delivered each of his edges in compliance.Dijkstra's algorithm proceeds until all peaks, available from s, aren't probed.

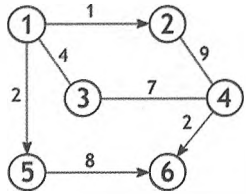


Figure 6.26. The weighed graph

From all peaks entering a great number of V we will select one, that from which it is necessary to find the shortest ways to remaining available peaks. Let that will be peak of 1 (fig. 6.27). Length of a way to all peaks, except the first, is initially equal to infinity, and to it – 0 since a graph has no loops.

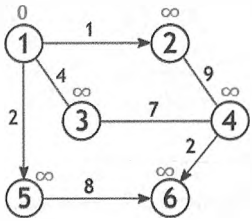


Figure 6.27. Choice of start peak

At peak of 1 exactly 3 neighbor (peak 2, 3, 5) and to calculate length of a way to them, it is necessary to put weight of the arcs lying between peaks: 1 and 2, 1 and 3, 1 and 5 with value of the first peak (with zero):

- 21+0
- 34+0
- 52+0

As it was already marked, the turned-out values are appropriated to peaks, only in that case if they it is "better" (less) than those which appear currently. And as each of three numbers is less than infinity, they become the new values determining length of a way from peak 1 to peaks of 2, 3 and 5 (fig. 6.28).

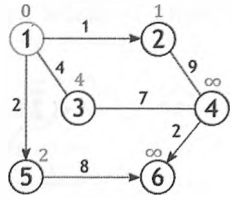


Figure 6.28. The next distances to peaks 2, 3 and 5

Further, the active peak is marked as visited, the status of "active" (a red circle) passes to one of her "neighbors", namely to peak of 2 (figure 6.29) as it the next to the earlier active peak.

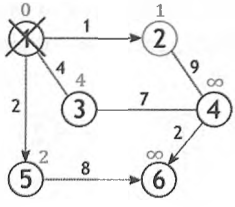


Figure 6.29. New active peak

At peak 2 only one not considered neighbor to whom distance from it is equal 9, but we need to calculate length of a way from source peak for what it is necessary to put the value attributed to peak 2 with an arc weight from it in peak 4: 41+9The condition of "brevity" (10 <∞) is satisfied, therefore, the peak 4 receives new value of length of a way (fig. 6.30).

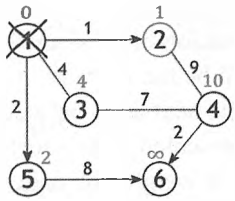


Figure 6.30. Distance to the 4th peak through peak 2

The peak 2 stops being the active and is deleted from the list of not visited peaks. Now the same method researches neighbors of peak 5, and the distance (fig. 6.31) to them is calculated.

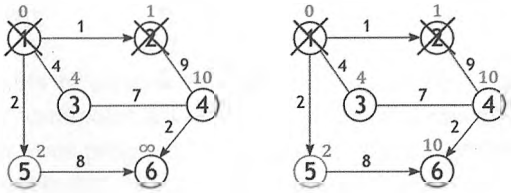


Figure 6.31. Distances to peaks 5 and 6

When business reaches survey of neighbors of peak 3 then it is important not to be mistaken since the peak 4 was already probed also the distance of one of possible ways from a source to it is calculated. If to start on on it through peak 3, then will make a way 4+7=11, and 11> 10 therefore new value is ignored, old remains (fig. 6.31).

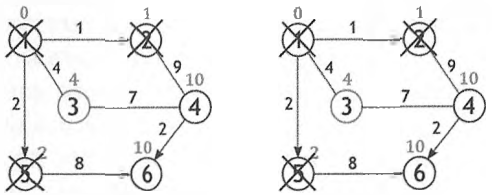


Figure 5.32. Optimum distances to all peaks

When all peaks of a graph, or all those that are available from a source are marked as visited, then operation of an algorithm of Dijkstra will come to the end, and all found ways will be the shortest.

In the program finding the next ways between peaks by means of Dijkstra's method, a graph will be presented in the form not of a binary connectivity matrix. Instead of units in it weights of edges will be exposed, function of zero will remain former: to show between what peaks there are no edges or they are, but are negatively directed.Program code on C ++it is given in the appendix B.

So, for example, the list of the most optimum distances lying between peak 1 and all remaining peaks, the considered graph will look:

- 1→1=0
- 1→2=1
- 1→3=4
- 1→4=10
- 1→5=2
- 1→6=10

Appendixes

Appendix A.

C++ Header Files and Standart Function.

It is a list of commonly used C++ header files. Other header files are introduced in the chapters when needed.

cassert

This library contains only the function assert . You use

assert(*assertion*);

to test the validity of an assertion. If *assertion* is false , assert writes an error message and terminates program execution. You can disable all occurrences of assert in your program by placing the directive#define NDEBUG before the include directive.

cctype

Most functions in this library classify a given ASCII character as a letter, a digit, and so on. Two other functions convert letters between uppercase and lowercase.The classify cation functions return a true value if ch belongs to the specified group; otherwise they return false.

isalnum(ch) Returns true if ch is either a letter or a decimal digit
isalpha(ch) Returns true if ch is a letter
iscntrl(ch) Returns true if ch is a control character (ASCII 127 or 0 to 31)
isdigit(ch) Returns true if ch is a decimal digit
isgraph(ch) Returns true if ch is printable and nonblank
islower(ch) Returns true if ch is a lowercase letter
isprint(ch) Returns true if ch is printable (including blank)
ispunct(ch) Returns true if ch is a punctuation character
isspace(ch) Returns true if ch is a whitespace character: space, tab, carriage return, new line,or form feed
isupper(ch) Returns true if ch is an uppercase letter
isdigit(ch) Returns true if ch is a hexadecimal digit

toascii(ch) Returns the ASCII code for ch

tolower(ch) Returns the lowercase version of ch if ch is an uppercase letter; otherwisereturns ch

toupper(ch) Returns the uppercase version of ch if ch is a lowercase letter; otherwisereturns ch

cmath

The C++ functions in this library compute certain standard mathematical functions. These functions are overloaded to accommodate float, double, and long double. Unless otherwise indicated, each function has one argument, with the return type being the same as the argument type (either float, double, or long double).

acos(x) Returns the arc cosine

asin(x) Returns the arc sine

atan(x) Returns the arc tangent

atan2(x, y) Returns the arc tangent of x /y

ceil(x) Rounds up

cos(x) Returns the cosine

cosh(x) Returns the hyperbolic cosine

exp(x) Returns *e* raised to the power x

fabs(x) Returns the absolute value

floor(x) Rounds down

fmod(x, y) Returns x modulo y

log(x) Returns the natural log

log10(x) Returns the log base 10

modf(x, iptr) For arguments x and iptr , returns the fractional part of x and sets iptr to pointto the integer part of x

pow(x, y) Returns x raised to the power y

sin(x) Returns the sine

sinh(x) Returns the hyperbolic sine

sqrt(x) Returns the square root

tan(x) Returns the tangent

tanh(x) Returns the hyperbolic tangent

cstdlib

abort() Terminates program execution abnormally

abs(x) Returns the absolute value of an integer

atof(s) Converts a string argument to floating point

atoi(s) Converts a string argument to an integer
exit(x) Terminates program execution, returns x to the operating system
rand() Returns a pseudorandom integer
srand(x) Initializes the pseudorandom number generator to x
srand() Initializes the pseudorandom number generator to 1

cstring

This library enables you to manipulate C strings that end in \0 . Unless noted otherwise, these functions return a pointer to the resulting string in addition to modifying an appropriate argument. The argument ch is a character, n is an integer, and the other arguments are strings.

strncat(toS, fromS, n) Copies at most n characters of from S to the end of toS and appends '\0'
strcmp(str1, str2) Returns an integer that is negative if str1 < str2 , zero if str1 ==str2 , and positive if str1 > str2
stricmp(str1, str2) Behaves like strcmp , but ignores case
strncmp(str1, str2, n) Behaves like strcmp , but compares the first n characters of each string
strncpy(toS, fromS, n) Copies n characters of fromS to toS , truncating or padding with'\0' as necessary
strspn(str1, str2) Returns the number of initial consecutive characters of str1 that arenot in str2
strcspn(str1, str2) Returns the number of initial consecutive characters of str1 that arein str2
strlen(str) Returns the length of str , excluding '\0'
strlwr(str) Converts any uppercase letters in str to lowercase without alteringother characters
strupr(str) Converts any lowercase letters in str to uppercase without alteringother characters
strchr(str, ch) Returns a pointer to the fi rst occurrence of ch in str ; otherwisereturns nullptr
strrchr(str, ch) Returns a pointer to the last occurrence of ch in str ; otherwise returnsnullptr
strpbrk(str1, str2) Returns a pointer to the fi rst character in str1 that also appears instr2 ; othewise returns nullptr
strstr(str1, str2) Returns a pointer to the fi rst occurrence of str2 in str1 ; otherwise returns nullptr
strtok(str1, str2) Finds the next token in str1 that is followed by str2 , returns a pointer to the token, and writes nullptr immediately after the token in str1

fstream

Defi nes the C++ classes that support fi le I/O.

iomanip

The manipulators in this library affect the format of stream operations. Note that iostream contains additional manipulators.

setbase(b) Sets number base to b = 8, 10, or 16
setfill(f) Sets fi ll character to f
setprecision(n) Sets fl oating-point precision to integer n
setw(n) Sets fi eld width to integer n
dec Tells the subsequent operation to use decimal representation
endl Inserts the new-line character \n and fl ushes the output stream
ends Inserts the null character \0 into an output stream
flush Flushes an output stream
hex Tells the subsequent I/O operation to use hexadecimal representation
oct Tells the subsequent I/O operation to use octal representation
ws Extracts whitespace characters from the input stream

iostream

The manipulators in this library affect the format of stream operations. Note that iomanip contains additional manipulators.

stdexcept

Defines several exception classes that can be thrown or extended for specific exceptional conditions.

string

This library enables you to manipulate C++ strings. Described here is a selection of the methods that this library provides. In addition, you can use the following operators with C++ strings: = , + , == , != , < ,<= , > , >= , << , and >> . Note that positions within a string begin at 0.

erase() Makes the string empty
erase(pos, len) Removes the substring that begins at position pos and contains lencharacters

find(subString) Returns the position of the substring subString within the string

length() Returns the number of characters in the string (same as size)

replace(pos, len, str) Replaces the substring that begins at position pos and contains lencharacters with the string str

size() Returns the number of characters in the string (same as length)

substr(pos, len) Returns the substring that begins at position pos and contains lencharacters

Appendix B.

Library of the used programs

Bank Account program

main.cpp | bankaccount.cpp | bankaccount.h | makefile

(Global Scope) - main(i

```
#include <iostream>
#include <string>
#include <cstdlib>

using namespace std;

#include "bankaccount.h"

string get_account_name(void) {

    string name;
    cout << "Enter account name: ";
    getline(cin, name);
    return name;
}

void display_menu(void) {

    cout << endl;
    cout << "1) Make a deposit" << endl;
    cout << "2) Make a withdrawl" << endl;
    cout << "3) Check balance" << endl;
    cout << "4) Quit" << endl;
}

int prompt_user(void) {

    cout << endl << "Enter a menu item number 1-4: ";
    string action;
    cin >> action;

    if (action.length() == 1) {
        return action[0] - '0';
    }
    else {
        return -1;
    }
}
```

main.cpp | bankaccount.cpp | bankaccount.h | makefile

(Global Scope)

```
39
40  double get_amount(void) {
41
42      cout << endl << "Enter a dollar amount: ";
43      double amount = 0;
44      cin >> amount;
45      return amount;
46  }
47
48  int main(int argc, char* argv[]) {
49
50      BankAccount account(get_account_name());
51
52      const int DEPOSIT = 1;
53      const int WITHDRAW = 2;
54      const int BALANCE = 3;
55      const int QUIT = 4;
56
57      int action = 0;
58
59      while (action != QUIT) {
60
61          display_menu();
62          action = prompt_user();
63
64          double amount;
65          if (action == DEPOSIT) {
66              amount = get_amount();
67              account.deposit(amount);
68
69          }
70          else if (action == WITHDRAW) {
71              amount = get_amount();
72              account.withdraw(amount);
73
74          }
75          else if (action == BALANCE) {
76              cout << endl << "Balance for " << account.getName() << endl;
77              cout << "$" << account.balance() << endl;
78          }
79      }
80      cin.get();
81      return EXIT_SUCCESS;
82  }
```

C:\Users\Adil\documents\visual studio 2012\Project

Enter account name:

main.cpp | bankaccount.cpp | bankaccount.h | makefile

(Global Scope)

```
#include "bankaccount.h"

double BankAccount::balance() {
    return sum;
}
void BankAccount::deposit(double amount) {
    sum += amount;
}
void BankAccount::withdraw(double amount) {
    sum -= amount;
}
string BankAccount::getName() {
    return name;
}
```

main.cpp | bankaccount.cpp | bankaccount.h | makefile

(Global Scope)

```
#include <string>
#include <cstdlib>
#include <iostream>

#ifndef _BANKACCOUNT_H
#define _BANKACCOUNT_H

using namespace std;

class BankAccount {

private:
    double sum;
    string name;

public:
    BankAccount(string nm) : name(nm), sum(0) {}
    BankAccount(string nm, double bal) :
    name(nm), sum(bal) {}

    double balance();
    void deposit(double);
    void withdraw(double);
    string getName();
};

#endif
```

Homework program

date.h | date.cpp | homework.h | homework.cpp | homeworklist.h | homeworklist.cpp | main.cpp

(Глобальная область)

```
1  #ifndef _DATE_H
2  #define _DATE_H
3
4  #include <iostream>
5
6  using namespace std;
7
8  class date;
9
10 ostream &operator<< (ostream &stream, date d);
11
12 class date {
13
14 private:
15     int day;
16     int month;
17     int year;
18
19 public:
20     // The default constructor
21     date ();
22
23     // Initializes the date with meaningful data
24     date (int day, int month, int year);
25
26     // This follows the Java semantics -- we'll look at overloading operators
27     // a little bit later.
28     int compareTo (date another_date);
29
30     // This is the standard insertion operator. It should produce "mm/dd/yyyy"
31     friend ostream &operator<< (ostream &stream, date d);
32 };
33
34 #endif // Nothing belongs below here
```

date.h | date.cpp | homework.h | homework.cpp | homeworklist.h | homeworklist.cpp | main.cpp

(Глобальная область)

```cpp
#include <iostream>
#include "date.h"

date::date() : day(0), month(0), year(0) {}

date::date(int day, int month, int year) :
day(day), month(month), year(year) {}

int date::compareTo(date another_date) {

    if (year != another_date.year)
        return year - another_date.year;

    if (month != another_date.month)
        return month - another_date.month;

    if (day != another_date.day)
        return day - another_date.day;

    return 0;
}

ostream &operator<< (ostream &stream, date d) {

    stream << d.month << '/' << d.day << '/' << d.year;
    return stream;
}
```

date.h | date.cpp | homework.h | homework.cpp | homeworklist.h | homeworklist.cpp | main.cpp

(Глобальная область)

```cpp
#ifndef _HOMEWORKLIST_H
#define _HOMEWORKLIST_H

#include "homework.h"

const int LIST_MAX = 50; // Maximum of 50 assignments; change if desired

class homeworklist;

ostream &operator<< (ostream &stream, homeworklist hl);

class homeworklist {

private:
    homework list[LIST_MAX]; // Array of individual assignments
    int current_size; // total currently within array

public:
    homeworklist(); // default constructor
    bool add (homework h); // adds a new assignment to this collection

    homeworklist dueafter (date d);

    homeworklist duebefore (date d);

    homeworklist dueon (date d);

    friend ostream &operator<< (ostream &stream, homeworklist hl);
};

#endif // Nothing belongs below here
```

date.h | date.cpp | homework.h | homework.cpp | homeworklist.h | homeworklist.cpp | main.cpp

(Глобальная область)

```
15 #include <iostream>
16
17 #include "date.h"
18
19 class homework;
20
21 ostream &operator<< (ostream &stream, homework h);
22
23 class homework {
24
25 private:
26     string name; // Important note: Spaces within a name are not supported
27     date assigned_date;
28     date due_date;
29
30 public:
31     homework ();
32     homework (string name, date assigned_date, date due_date);
33
34     // This should follow the Java syntax -- we'll discuss overloading
35     // oeprators soon enough. The comparison should be based ont he due
36     // date.
37     int compareTo (homework another_homework);
38
39     // The standard insertion operator.
40     // It should output as follows:
41
42     friend ostream &operator<< (ostream &stream, homework h);
43
44 };
45
46 #endif // Nothing belongs below here
```

date.h | date.cpp | homework.h | homework.cpp | homeworklist.h | homeworklist.cpp | main.cpp

(Глобальная область)

```
1  #include <iostream>
2  #include <string>
3  #include "homework.h"
4
5  homework::homework() : name(""), assigned_date(), due_date() {}
6
7  homework::homework(string name, date assigned_date, date due_date) :
8  name(name), assigned_date(assigned_date), due_date(due_date) {}
9
10 int homework::compareTo(homework another_homework) {
11     return (this->due_date.compareTo(another_homework.due_date));
12 }
13
14 ostream &operator<< (ostream &stream, homework h) {
15     stream << h.name << ": " << h.assigned_date << '-' << h.due_date;
16     return stream;
17 }
18
```

date.h | date.cpp | homework.h | homework.cpp | homeworklist.h | homeworklist.cpp | main.cpp

(Глобальная область)

```
#include "homeworklist.h"

homeworklist::homeworklist() : current_size(0) {}

bool homeworklist::add(homework h) {
    if (current_size == LIST_MAX)
        return false;
    list[current_size++] = h;
    return true; }

homeworklist homeworklist::dueafter(date d) {
    homeworklist new_list;
    homework reference_date("reference assignment", d, d);
    for (int index = 0; index < current_size; index++) {
        if (list[index].compareTo(reference_date) > 0) {
            new_list.add(list[index]);
        }
    }
    return new_list; }

homeworklist homeworklist::duebefore(date d) {
    homeworklist new_list;
    homework reference_date("reference assignment", d, d);
    for (int index = 0; index < current_size; index++) {
        if (list[index].compareTo(reference_date) < 0) {
            new_list.add(list[index]);
        }
    }
    return new_list; }

homeworklist homeworklist::dueon(date d) {
    homeworklist new_list;
    homework reference_date("reference assignment", d, d);
    for (int index = 0; index < current_size; index++) {
        if (list[index].compareTo(reference_date) == 0) {
            new_list.add(list[index]);
        }
    }
    return new_list; }

ostream &operator<< (ostream &stream, homeworklist hl) {
    for (int index = 0; index < hl.current_size; index++) {
        stream << hl.list[index] << endl;
    }
    return stream; }
```

e.h | date.cpp | homework.h | homework.cpp | homeworklist.h | homeworklist.cpp | main.cpp* | ×

(Глобальная область) | string_to_date(string date_string)

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <cstdio>

#include "date.h"
#include "homework.h"
#include "homeworklist.h"

using namespace std;

// Constants
const int EMPTY = 0;
const int ADD = 1;
const int LIST = 2;
const int DUE_ON = 3;
const int DUE_BEFORE = 4;
const int DUE_AFTER = 5;
const int QUIT = 6;

// Function declarations
date string_to_date (string);
void display_menu (void);
int read_menu_selection (void);
void display_due(homeworklist, int);
date read_date(string);

// Function definitions
int main (int argc, char *argv[]) {
    string hw_name;
    date hw_asg_date;
    date hw_due_date;
    homework new_asg;

    homeworklist hl;

    int choice = EMPTY;
    while (choice != QUIT) {
        display_menu();
        choice = read_menu_selection();
        switch (choice) {
        case ADD:
            // read in the name, date assigned, and due date
            cout << endl << "Homework name: ";
            cin >> hw_name;

            hw_asg_date = read_date("Date assigned: ");
            hw_due_date = read_date("Date due: ");
```


date.h
date.cpp
homework.h
homework.cpp
homeworklist.h
homeworklist.cpp
main.cpp*
(Глобальная область)
string_to_date(string date_string)
// create and store the homework object
new_asg = homework(hw_name, hw_asg_date, hw_due_date);
hl.add(new_asg);
break;
case LIST:
cout << hl << endl;
break;
case DUE_AFTER: // falls through
case DUE_BEFORE: // falls through
case DUE_ON:
// Due after, before, and on are all
// handled by this case. Since there
display_due(hl, choice);
break;
case QUIT:
break;
default:
choice = EMPTY;
break;
} // end switch (choice)
} // end main work loop (while)
return EXIT_SUCCESS;
}
void display_menu(void) {
cout << "\n1. Add\n";
cout << "2. List all assignments\n";
cout << "3. List all assignments \"due on\" a particular date\n";
cout << "4. List all assignments \"due before\" a particular date\n";
cout << "5. List all assignments \"due after\" a particular date\n";
cout << "6. Quit\n";
}
int read_menu_selection(void) {
int choice = 0;
cout << "Enter selection: ";
cin >> choice;
cout << endl;
return choice;
}

date.h | date.cpp | homework.h | homework.cpp | homeworklist.h | homeworklist.cpp | main.cpp*

(Глобальная область) | string_to_date(string date_string)

```
96
97  void display_due(homeworklist hl, int choice) {
98
99      date d = read_date("Enter a date: ");
100
101     if (DUE_AFTER == choice) {
102         cout << endl << hl.dueafter (d);
103     } else if (DUE_ON == choice) {
104         cout << endl << hl.dueon (d);
105     } else if (DUE_BEFORE == choice) {
106         cout << endl << hl.duebefore (d);
107     }
108
109 }
110
111 date read_date(string prompt) {
112
113     // display prompt, read input, convert to date
114     cout << prompt;
115
116     string date_whole;
117     cin >> date_whole;
118
119     return string_to_date(date_whole);
120 }
121
122 date string_to_date (string date_string) {
123
124     // convert from a string to a date
125     int day, month, year;
126     int posn;
127
128     posn = date_string.find_first_of ("/", 0);
129     month = atoi(date_string.substr(0, posn).c_str());
130
131     day = atoi(date_string.substr(posn + 1,
132                                   date_string.find_last_of("/", 0)).c_str());
133
134     posn = date_string.find_first_of("/", posn + 1) + 1;
135     year = atoi(date_string.substr(posn, date_string.length()).c_str());
136
137     return date (day, month, year);
138 }
139
140 // eof
```

Declaration of class HashMap

hashmap.h | hashmap.cpp | hash-driver.cpp

(Global Scope)

```
1   // template hashmap class
2
3   #ifndef  _HASHPMAP_H_
4   #define  _HASHPMAP_H_
5
6   #include <iostream>
7   #include <vector>
8   #include <algorithm>
9   #include <stdexcept>
10
11  using namespace std;
12
13  // we do not compute prime numbers but use a table instead
14  static const int num_primes = 25;
15  static const unsigned long prime_list[] = {
16              53, 97, 193, 389, 769, 1543, 3079, 6151, 12289, 24593, 49157, 98317,
17              196613, 393241, 786433, 1572869, 3145739, 6291469, 12582917, 25165843,
18              50331653, 100663319, 201326611, 402653189, 805306457
19          };
20
21  template <typename key_type, typename value_type, typename hash_func, typename key_equal>
22  class HashMap {
23
24  protected:
25      // hashtable entries
26      class Entry {
27      public:
28          key_type key;
29          value_type value;
30          bool used;
31
32          Entry() : used(false) {}
33      };
34
35      int entries;      // number of entries
36      int prime;        // index to size table
37
38      vector<Entry> *ht;
39      hash_func hf;         // hash function on key_type
40      key_equal eq;         // equality predicate on key_type
41
42      int table_size() const { return prime_list[prime];}
43      float load_factor() const { return float(size()) / table_size();}
44      int resize();
```

```
hashmap.h  hashmap.cpp  hash-driver.cpp
HashMap<key_type, value_type, hash_func, key_equal>
39      hash_func hf;        // hash function on key_type
40      key_equal eq;        // equality predicate on key_type
41
42      int table_size() const { return prime_list[prime];}
43      float load_factor() const { return float(size()) / table_size();}
44      int resize();
45
46  public:
47
48      HashMap(const hash_func& h, const key_equal& e)
49              : entries(0), prime(0),
50              ht(new vector<entry>(prime_list[0])),
51      hf(h), eq(e) {}
52
53      virtual ~HashMap() { delete ht;}
54
55      virtual int size() const { return entries;}
56
57      virtual value_type operator[](int i) throw(out_of_range) {
58
59          if (i < 0 || i > table_size()) {
60              throw out_of_range("Index out of range");
61          }
62          return (*ht)[i].value;
63      }
64
65      virtual pair<bool, int> search(const key_type& k);
66      virtual int insert(const key_type& k, const value_type& v);
67      virtual void remove(const key_type& k);
68  };
69
70  #endif
```

Hasmap.cpp definition of class HashMap

```
hashmap.h  hashmap.cpp  hash-driver.cpp
(Глобальная область)
1   #include "hashmap.h"
2
3   using namespace std;
4
5   template <typename key_type, typename value_type, typename hash_func, typename key_equal>
6   pair<bool, int> HashMap<key_type, value_type, hash_func, key_equal>::search(const key_type& k) {
7
8       int p = hf(k) % table_size();
9
10      while ((*ht)[p].used) {
11          if (eq((*ht)[p].key, k)) {     // equality predicate for key_type
12              return pair<bool, int>(true, p);
13          }
```

Hasmap.cpp

hashmap.h | hashmap.cpp | hash_driver.cpp

HashMap<key_type, value_type, hash_func, key_equal>

```
63      }
64
65      template <typename key_type, typename value_type, typename hash_func, typename key_equal>
66          int HashMap<key_type, value_type, hash_func, key_equal>::resize() {
67
68              if (prime == num_primes - 1) {
69                  cerr << "maximal table size reached, aborting ... "<< endl;
70                  exit(2);
71              }
72
73              int mm = prime_list[prime];
74              prime++;
75              int m = prime_list[prime];
76              vector<Entry>*ptr = new vector<Entry>(m);
77
78              for (int i = 0; i < mm; ++i) {
79
80                  if ((*ht)[i].used) {
81                      key_type kk = (*ht)[i].key;
82
83                      int p = hf(kk) % m;
84
85                      while (p < m && (*ptr)[p].used) {
86                          p++;
87                      }
88                      if (p == m) {
89                          p = 0;
90                      }
91                      while ((*ptr)[p].used) {
92                          p++;
93                      }
94                      (*ptr)[p].key = kk;
95                      (*ptr)[p].value = (*ht)[i].value;
96                      (*ptr)[p].used = true;
97                  }
98              }
99              delete ht;
100             ht = ptr;
101             return m;
102         }
```

hashmap.h hashmap.cpp hash-driver.cpp*

(Глобальная область)

```
#include<iostream>
#include<string>
#include<algorithm>
#include"hashmap.h"
#include"hashmap.cpp"

using namespace std;

// Here is our hash function. It is implemented
// as a class that overloads the () operator.
/*!Begin Snippet*/
class hash_function {

public:

    unsigned int mm;

    hash_function(unsigned int m = 6151)
        : mm(m) {}

    unsigned int operator()(const string& s) const {
        unsigned int res = 0;
        for (unsigned int i = 0; i < s.size(); i++ ) {
            res = res * mm + s[i];
        }
        return res;
    }
};
/*!End Snippet*/

// We supply this class to the HashMap so that
// the HashMap knows how to compare two objects
// of the key type.
class equal_predicate {

public:

    equal_predicate() {}

    bool operator()(const string& A, const string& B) const {
        return A == B;
    }
};
```

Use of class HashMap and its member functions

hashmap.h hashmap.cpp hash-driver.cpp

(Global Scope)

```
49      equal_predicate eq;
50
51      // HashMap<key, value, hash, equality>
52      HashMap< string, int, hash_function, equal_predicate> table(hf, eq);
53
54      // inserts
55      table.insert("first", 1);
56      table.insert("second", 2);
57      table.insert("third", 3);
58      table.insert("fourth", 4);
59      table.insert("fifth", 5);
60
61      cout << "table contains " << table.size() << " elements\n";
62
63      // search and remove
64      pair<bool, int> result = table.search("third");
65      if (result.first) {
66          cout << "third found at position " << result.second << "\n";
67      }
68      else {
69          cout << "third not found\n";
70      }
71
72      table.remove("third");
73
74      result = table.search("third");
75      if (result.first) {
76          cout << "third found at position " << result.second << "\n";
77      }
78      else {
79          cout << "third not found\n";
80      }
81
82      cout << "table contains " << table.size() << " elements\n";
83
84      return EXIT_SUCCESS;
85  }
86
```

A binary search tree

CustomerDatabase.cpp | Customer.cpp | CustomerDatabase.h | Customer.h | bst.h

(Global Scope) | display_contact(CustomerDatabase & d

```
#include <iostream>
#include <cstdlib>
#include <stdexcept>

#include "Customer.h"
#include "CustomerDatabase.h"

#define DISPLAY "1"
#define QUIT "2"

using namespace std;

void display_menu(void);
string get_choice(void);
void display_contact(CustomerDatabase& db);

int main(int argc, char* argv[]) {

    try {

        if (argc != 2) {
            cerr << "Usage: " << argv[0] << " contact_filename\n";
            return EXIT_FAILURE;
        }

        CustomerDatabase db(argv[1]);
        display_menu();
        string choice = get_choice();

        while (choice != QUIT) {

            if (choice == DISPLAY) {
                display_contact(db);
            } else {
                cout << "Invalid entry\n\n";
            }

            display_menu();
            choice = get_choice();
        }

        return EXIT_SUCCESS;
    } catch (exception& e) {
        cerr << e.what() << endl;
    } catch (...) {
        cerr << "Unknown exception caught!" << endl;
    }

    return EXIT_FAILURE;
}
```

main.cpp | CustomerDatabase.cpp | Customer.cpp | CustomerDatabase.h | Customer.h | bst.h

(Global Scope) | get_choice(void)

```
53  void display_contact(CustomerDatabase& db) {
54
55      cout << "Enter contact's email: ";
56      string email;
57      cin >> email;
58
59      Customer key("", email, "");
60      const Customer* const p = db.search(key);
61
62      if (p == NULL) {
63          cout << "\nSorry, contact not found\n";
64      } else {
65          cout << endl << *p << endl;
66      }
67  }
68
69  void display_menu(void) {
70
71      cout << "Customer Contact Database\n";
72      cout << "-------------------------\n\n";
73      cout << "Commands:\n";
74      cout << "1) Display a contact\'s information\n";
75      cout << "2) Quit program\n\n";
76
77  }
78
79  string get_choice(void) {
80
81      cout << "Enter choice: ";
82
83      string choice;
84      getline(cin, choice);
85
86      return choice;
87  }
```

main.cpp | CustomerDatabase.cpp | Customer.cpp | CustomerDatabase.h | Customer.h | bst.h

(Global Scope)

```
#include <vector>
#include <algorithm>
#include <stdexcept>

#include "CustomerDatabase.h"

CustomerDatabase::CustomerDatabase(const string& filename) {
    load_database(filename);
}

void CustomerDatabase::load_database(const string& filename) {

    // To optimize our database, we first read the
    // customers into a vector. We then sort and then
    // insert during a reverse binary
    // search to ensure that our inherited bst is as
    // completely balanced as it can be.

    vector<Customer> v;
    read_datafile(v, filename);
    sort(v.begin(), v.end());
    reverse_binary_insert(v, 0, v.size() - 1);
    tree.insert(v[v.size() - 1]);
}
```

Dijkstra's Algorithm

```
#include <iostream>

using namespace std;

const int V = 6;
void Dijkstra(int GR[V][V], int st)
{
    int distance[V], count, index, i, u, m = st + 1;
    bool visited[V];
    for (i = 0; i < V; i++)
    {
        distance[i] = INT_MAX; visited[i] = false;
    }
    distance[st] = 0;
    for (count = 0; count < V - 1; count++)
    {
        int min = INT_MAX;
        for (i = 0; i < V; i++)
        if (!visited[i] && distance[i] <= min)
        {
            min = distance[i]; index = i;
        }
        u = index;
        visited[u] = true;
        for (i = 0; i < V; i++)
        if (!visited[i] && GR[u][i] && distance[u] != INT_MAX &&
            distance[u] + GR[u][i] < distance[i])
            distance[i] = distance[u] + GR[u][i];
    }
    cout << "Cost of a way from initial vertex to remaining:\t\n";
    for (i = 0; i < V; i++) if (distance[i] != INT_MAX)
        cout << m << ">" << i + 1 << " = " << distance[i] << endl;
    else cout << m << ">" << i + 1 << " = " << "the route is unavailable" << endl;
}
void main(){

        int start, GR[V][V] = {
            { 0, 1, 4, 0, 2, 0 },
            { 0, 0, 0, 9, 0, 0 },
            { 4, 0, 0, 7, 0, 0 },
            { 0, 9, 7, 0, 0, 2 },
            { 0, 0, 0, 0, 0, 8 },
            { 0, 0, 0, 0, 0, 0 } } ;
            cout << "Initial vertex >> "; cin >> start;
            Dijkstra(GR, start - 1);
            system("pause>>void");
            }
```

Bibliography

1. Alfred V.Aho, Jone E. Hopkroft, Jeffrey D. Ullman. Data Structures and Algorithms. BellLaboratories, Cornell University, Stanford University. July 3-5, 2003.
2. Mark Allen Weiss. Data Structures and Algorithm Analysis in C++ Florida International University, Boston.Pearson Education, Inc., publishing as Addison-Wesley. 2014.
3. Frank M. Carrano, Timothy Henry. Data Abstraction & Problem Solving with C++: Walls and Mirrors. Copyright, Pearson Education, Inc., publishing as Addison-Wesley.2013.
4. Robert Sedgewick. Algorithms in C++. Princeton University. Massachusetts l Menlo park, California. 2001.
5. Jin Y. Yen. An algorithm for Finding Shortest Routes from all Source Nodes to a Given Destination in General Network, Quart. Appl. Math., 27, 1970.
6. Richard Bellman: On a Routing Problem, in Quarterly of Applied Mathematics, 16(1), pp.87-90, 1958.
7. Lestor R. Ford, D. R. Fulkerson: Flows in Networks, Princeton University Press, 1962.
8. Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001.
9. Niklaus Wirth. Algorithms and Data Structures 1985 (Oberon version: August 2004).
10. Alan Parker.Algorithms and Data Structures in C++. CRC Press LLC ISBN: 0849371716,Pub Date: 08.01.1993.
11. Douglas Baldwin, Greg W. Scragg. Algorithms and Data Structures: The Science of Computing. Charles River Media, ISBN:1584502509,2004.
12. Robert Sedgewick Algorithms in Java: Parts 1-4, Third Edition: Addison Wesley Pub Date: July 23, 2002. ISBN: 0-201-36120-5, 768 pages.
13. Paul M. Embree.C Algorithms For Real Time DSP. ISBN 0-13-337353-3.QA76.73.C15E63. 1995.
14. P.S. Deshpande, O.G. Kakde. C & Data Structures. ISBN:1584503386. 2004.
15. NellDale Jones,C++ Plus Data Structures, Third Edition University of Texas, Austin. Bartlett Publishers ISBN:0763704814, 2003.
16. Adam Drozdek, Data structures and algorithms in C++. ISBN 0 534-37597-9 (text). 2nd ed.2001.
17. Robert Lafore. Data Structures & Algorithms in Java. SAMS. ISBN: 1571690956,1998.

18. Ron Penton. Data Structure For Game Programers. Premier Press, a division of Course Technology.2003.

19. Kurt Mehlhorn. Data Structures and Algorithms 3. EATCS Monographs on Theoretical Computer Science. 1984.

20. Peter Drake.Data Structures and Algorithms in Java.PublisherPrentice HallDecember 19, 2005.

21. Michael McMillan.Data structures and algorithms using visual basic. net. PulaskiTechnical College. 2005.

22. Bruno R. Preiss. Data Structures and Algorithmswith Object-Oriented Design Patterns in C++. University of Waterloo, Waterloo, Canada 2000.

23. Robert L. Kruse, Alexander J. Ryba. Data Structures and Program Design in C++. Upper Saddle River, New Jersey. ISBN 0–13–087697–6, 2000.

24. Jim Keogh, Ken Davidson. McGraw-Hill Data Structures Demystified. Osborne.ISBN:0072253592. 2004.

25. Ellis Horowitz, Sartaj Sahni. Fundamentals of Data Structures.–Macmillan Computer Publishing. October 1981.

26. Richard Wiener, Lewis J. Pinson. Fundamentals of OOP and Data Structures in Java. University of Colorado, Colorado Springs. Cambridge University Press, 2000.

27. Richard Jones, Rafael Lins.Algorithms for Automatic Dynamic Memory Management. University of Kent at Canterbury, UK. - 2001.

28. A D McGettrick, J van Leeuwen. Handbook of Algorithms and Data Structures in Pascal and C. 2nd Ed - G.H. Gonnet 2004.

29. William B. Frakes, Ricardo Baeza-Yates.Information Retrieval:Data Structures & Algorithms –Match 7, 2004.

30. Julian Bucknall.The Tomes of Delphi Algorithms and Data Structures. ISBN 1-55622-736-1, 2001.

31. Adil J.Seyketov. Data structures and algorithms. Educational manual - Almaty. KazNU, ISBN 978-601-04-0426-7, 2014.

32. АхоА. - Структуры данных и алгоритмы – Москва, С-Петербург 2003.

33. А.А. Кубенский. Структуры и алгоритмы обработки данных. Объектно-ориентированный подход и реализация на С++ . СПб: ISBN 5-94157-506-8, 2004.

34. Джулиан Бакнелл. Фундаментальные алгоритмы и структуры данных в Delphi. СПб: ISBN 5-93772-087-3, 2003.

A. SEIKETOV

ALGORITHMS, DATA STRUCTURES AND PROGRAMMING

Textbook

Format 60×90$^{1/16}$. Offset paper. Font type "Times New Roman".
Volume –18 p.p. Edition –1300 pcs.

Order №50

The printing house "BookPrint" LLP.

Tel.: 386-58-80.

sd.bookprint@mail.ru