C++ Tutorial

University of Waterloo

Version 1.3

Peter A. Buhr

August 15, 2000

Contents

1 Introduction **4**

2 A Brief History of C/C++ **4**

3 Creating a C++ Source File **4**

4 Compiling a C++ Program **4**

5 Running a C++ Program **5**

6 Structure of a C++ Program **5**
6.1 Comment 5
6.2 Statement 5

7 First Program **6**

8 Declaration **6**
8.1 Identifier 6
8.2 Basic Types 7
8.3 Variable Declaration 7
8.4 Literals 7
8.5 Type Modifier 8
8.6 Type Constructor 9
8.7 Type Constructor Literal 12

9 Expressions **12**
9.1 Type Operator 14
9.2 Conversion 14

10 Control Structures **16**
10.1 Block 16
10.2 Conditional 16
10.3 Selection Statement 17
10.4 ⋈ Conditional Expression Evaluation 17
10.5 Iteration 18

11 Routine **20**
11.1 Argument/Parameter Passing 21
11.2 Array Parameters 21
11.3 Prototype 22
11.4 ⋈ Overloading 23

12 ⋈ Preprocessor **24**
12.1 Textual Substitution 24
12.2 Textual Inclusion 25
12.3 Conditional Textual Inclusion 25

13 Input/Output **26**
13.1 Input 27
13.2 Output 27

14 Dynamic Storage Management **28**

15 ⋈ Objects **30**
15.1 Constructor/Destructor . . . 33
15.2 ⋈ String . . . 37

16 Inheritance **38**
16.1 Implementation Inheritance . . . 38
16.2 Type Inheritance . . . 40
16.3 Virtual Routines . . . 42

17 Templates **43**

18 ⋈ Shell Argument **45**

19 Encapsulation **46**
19.1 Separate Compilation . . . 48

20 Acknowledgments **51**

A Pulling It All Together **51**

Index **53**

1 Introduction

This tutorial is designed to give a working knowledge of C++ (and indirectly parts of C) as quickly as possible for people with Pascal programming experience and familiarity with basic programming language concepts. By working through the exercises, basic C++ concepts can be practiced. This tutorial is not a substitute for a good C++ textbook; in several places, you are referred to a textbook for more complete information. This tutorial also assumes familiarity with the UNIX[1] operating system and a text editor. Any corrections or suggestions about this tutorial can be sent to pabuhr@uwaterloo.ca.

Throughout the tutorial, the following symbols are used:

⇒ This symbol indicates that you are to perform the action marked by the arrow.

⋈ This symbol indicates that the section explains a concept that may be unfamiliar even if you have some previous programming experience. Make sure you understand this concept before advancing in the tutorial.

NOTE: A particular programming style is used in this tutorial. However, each course may have its own programming style; always follow that particular style.

2 A Brief History of C/C++

C was designed for and implemented on the UNIX operating system of a DEC PDP-11 by Dennis Ritchie, starting in 1970. The intent was to create a language more powerful than assembly language but still allow direct access to the machine. C is used for writing system software, such as UNIX.

C++ was designed by Bjarne Stroustrup, starting in 1980, to add object-oriented capabilities, along with other important extensions to C. C++ is *mostly* a superset of C. While C++ made important improvements, it did not fix existing problems with C, to maintain backwards compatibility. Therefore, C++ has both its problems and C's problems.

3 Creating a C++ Source File

A C++ source file should be created in a UNIX file with the suffix ".C" or ".cc". Any text editor may be used to create a source file. Certain text editors, such as emacs, perform automatic indentation of C++ statements.

4 Compiling a C++ Program

The C++ compilation command performs the following steps: preprocessing, compilation, assembly and linkage, and produces an executable file called a.out. (The file a.out is overwritten so be careful.) By using appropriate options, individual steps of the compilation and actions within a step can be controlled. In this tutorial, the GNU g++ compiler is used; any g++ specific material is always identified. The general form of a C++ compilation command is:

g++ *-option* *-option* ... *source-file1*.C *source-file2*.C ...

There is usually only one source file.

Options start with a "-" (minus character) followed by a single character and possibly a value depending on the option. In general, no options are required; some useful options are:

-g Produce additional symbol table information for a symbolic debugger (dbx or gdb).

-E Perform only the preprocessor step, printing the output on standard output. (used to check for preprocessor errors)

-c Perform only the preprocessor and compilation steps. (used to separately compile a program)

-o *name* Name the executable file to the specified name instead of a.out, e.g.,

g++ -o fred *yourprogram*.C

creates an executable called fred instead of a.out.

-Wall Print useful warning messages. (g++ only)

[1] UNIX is a Trademark of X/OPEN

5 Running a C++ Program

Once an executable file is created from a C++ source program, it is presented to a shell to be loaded and run, like other commands (e.g., ls, emacs, rm). Unless the directory where the executable resides is in the shell's search path, the shell cannot find the executable file. It then becomes necessary to specify the location of the executable using an absolute directory path. The full directory path can be specified, but if the executable is located in the current directory, its location can be specified using a relative path, such as "./". For example, if working in directory /u/userid/work, the executable a.out can be run by specifying either /u/userid/work/a.out or ./a.out on the shell command line.

6 Structure of a C++ Program

6.1 Comment

Comments are essential to document what a program does and how it does it. A comment may be placed anywhere a whitespace (blank, tab, newline) is allowed. There are two kinds of comments: C and C++ style.

Pascal	C	C++
{ ... } or (* ... *)	/* ... */	/* ... */ // remainder of line

Like Pascal, C style comments begin with a start symbol, /*, and end with a terminator symbol, */, and hence, can extend over multiple lines. *Like Pascal, C comments cannot be nested one within another*:

```
/* ... /* ... */ ... */
```

Here, the first terminator, */, ends the comment and the compiler treats the remaining comment text as program. Hence, be extremely careful in using comments to elide/comment-out code:

```
/*    attempt to comment-out a number of statements
while ( ... ) {
    /* ... */    nested comment causes errors
    if ( ... ) {
        /* ... */    nested comment causes errors
    }
}
*/
```

C++ style comments begin with the start symbol, //, and continue to the end of the line (i.e., only one line long). C++ style comments can be nested one within another:

```
// ... // ...
```

so they can be used to comment-out code:

```
// while ( ... ) {
//     /* ... */    nested comment does not cause errors
//     if ( ... ) {
//         // ...    nested comment does not cause errors
//     }
// }
```

Section 12 presents another way to comment-out code.

When asked to enter or modify a program in this document, it is unnecessary to enter comments in the program (save time typing); these comments are additional explanation and never affect the program's execution.

6.2 Statement

The syntactic structure of C/C++ statements is a series of tokens separated by whitespace. This flexibility makes it easy to make code readable. Tokens are grouped together into two kinds of statements: preprocessor (see Section 12) and C/C++. Preprocessor statements are a single line and not terminated by punctuation. C/C++ statements are terminated by a semicolon.[2] A source-file contains a mixture of preprocessor statements and C/C++.

Unlike Pascal, there is no enclosing program declaration of all the source code. The routine main is the default location to begin execution when an executable file is called from a shell. The code in routine main is equivalent to the

[2] A block, denoted with { }, forms a complete statement so it is not terminated with a semicolon (see Section 10.1).

body of the Pascal program statement. main returns an integer code to its invoking shell. The routine exit can also be used to stop a program at any location and return a code to the invoking shell.

7 First Program

The standard first C++ program prints "Hello World!" to the screen.

⇒ Edit a file called hello.C
⇒ Enter the following program (**including comments**):

```
// First C++ program by: YourFirstName YourLastName
// Print "Hello World!" to the screen.
#include <iostream.h>
int main() {
    cout << "Hello World!" << endl;
    return 0;
}
```

Several important points are illustrated in this program; each is discussed separately. First, when writing programs, there should be comments at the beginning identifying you and what the program does. Additional comments should appear within the source code to explain how the program works. Different courses have different documentation guidelines; it is your responsibility to follow those guidelines. Second, the line **#include** <iostream.h> makes input and output (I/O) possible in a program. Third, the routine header:

```
int main ()
```

defines the execution starting point. The curly braces, { ... }, denote the start and end of a C++ block of code, like Pascal begin and end (as opposed to comment characters), and this block is the body of routine main. Fourth, the output statement:

```
cout << "Hello World!" << endl;
```

prints the message to standard output, called cout, which is usually the terminal screen (like file Output in Pascal). Think of the information as cascading from right to left, as indicated by the chevrons, <<, to be printed on cout. endl ensures the string "Hello World!" appears on its own line. Finally, **return** 0 returns 0 (zero) to the shell indicating successful completion of the program; non-zero usually indicates an error. Most C/C++ programs do not return a value, which technically means the program has *undefined behaviour*; a correct C/C++ program always returns a value. Not returning a value implies checking the return code from such a program in the shell is meaningless.

⇒ Save the program.
⇒ Compile the program with the command: g++ hello.C
⇒ If the compilation produces error messages, read the messages and make appropriate changes to the code.
⇒ Once the program compiles properly, run it by issuing the command ./a.out.
⇒ Edit hello.C.
⇒ Remove "<< endl" from the output statement (**but not the semicolon**).
⇒ Compile and run the program again.

Notice the difference in output between the two programs.

8 Declaration

Like Pascal, C/C++ requires a declaration to appear before use, called the *declaration-before-use* rule.

8.1 Identifier

Identifiers may be arbitrarily long. Like Pascal, the first character must be a letter, with the addition of the underscore "_" in C/C++. Characters other than the first can be any of the above, or a digit. Unlike Pascal, identifiers are case-sensitive in C/C++; an identifier written in upper-case is not the same as one in lower case or in mixed case (both upper and lower case). Examples of valid identifiers are:

```
VeryLongVarName   Page1   Income_Tax   _75
```

Some identifiers are reserved as they denote keywords (and appear in **bold** font in this document); see a textbook for a complete list of reserved identifiers.

8.2 Basic Types

The basic types in C/C++ are like those in Pascal:

Pascal	C	C++
Boolean	**int** (0,1)	**bool**
Integer	**int**	**int**
Char	**char**	**char**
Real	**double**	**double**

Like Pascal, the range of values for **int** is machine specific; usually 2 bytes for small computers and 4 or more bytes for medium and large computers.

8.3 Variable Declaration

All variable declarations in C/C++ are done in the reverse manner from Pascal: first the type, then the list of identifiers.

Pascal	C/C++
var a, b, c, d, e, f : Integer	**int** a, b, c, d, e, f;

Declarations in Pascal are restricted to the beginning of routines. Declarations in C/C++ can be global to the source file, or local to a routine (within blocks), where C only allows declarations at the beginning of a block, and in C++, mixed among executable statements within a block. All global variables of the basic types are zero-initialized, while similar local variables in a block are *NOT* zero-initialized.

⇒ Edit hello.C.

⇒ Make the following modification to routine main:

```
int x;                  // C/C++ allow global declarations in a source file
int main () {
    int y;              // C only allows local declarations at beginning of a block
    cout << "x:" << x << endl;   // use of a global variable
    cout << "y:" << y << endl;   // use of a local variable
    int z;              // C++ allows local declarations anywhere in the body of the block
    cout << "z:" << z << endl;   // use of a local variable
}
```

⇒ Compile and run the program.

Since variables y and z are not initialized, the values printed might vary each time the program is run.

8.4 Literals

		Pascal	C/C++
ordinal	boolean	**false**, **true**	C: (0, 1), C++: **false**, **true**
	decimal	346	346
	octal	N/A	0144, begins with zero
	hexadecimal	N/A	0xfe, begins with zero, use X or x
real		0.1, 1.0, -1.0, -7.3E3, -6.6E-2	.1, 1., -1., -7.3E3, -6.6E-2, use E or e
character		'a', ''''	'a', '\''
string		'abc', ''''''	"abc", "\"\""

Pascal has only base 10 integer literals, while C/C++ has base 8, 10 and 16. Real literals in C/C++ do not require a leading or trailing zero before or after the decimal point. Unlike Pascal, C/C++ has two kinds of string literals, one for single character strings and one for multi-character strings, enclosed by ' and ", respectively. Care must be taken to use the right kind of string literal with the right kind of character or string variable. Also, the escape sequence for special characters in a string is different between Pascal and C/C++. Pascal has only one character that needs escaping: the quote character, which is escaped by entering two quotes to mean one quote. C/C++ provides a general facility using the backslash, \, which allows all characters in the character set to appear in a character or string constant. The most common escape sequences are (see a C/C++ textbook for others):

`'\\'`	backslash
`'\''`, `"\""`	single and double quote
`'\t'`, `'\n'`	tab, newline
`'\0'`	zero, string termination character
`'\ooo'`	octal character value, where `ooo` is up to 3 octal digits
`'\xhh'`	hexadecimal character value, where `hh` is up to 2 hexadecimal digits

String literals are implicitly terminated with a character containing the value 0. For example, the string `"abc"` is actually 4 characters composed of `'a'`, `'b'`, `'c'`, `'\0'`. (The reason is given in Section 15.2.) As well, adjacent string literals are concatenated, which allows long strings to be divided across multiple lines.

⇒ Edit hello.C
⇒ Make the following modification to routine main:

```
int main () {
    cout << 12 << endl << 014 << endl << 0xc << endl;
    cout << 1234.5 << endl << 1.2345e3 << endl;
    cout << 'w' << '\\' << '\'' << '"' << '\n' << endl;
    cout << "w\\'\"\n"        // concatenated string literal
            "w\\'\"\n" << endl;
}
```

⇒ Compile and run the program.
⇒ Check the output carefully.

Some of the printed values are different from the literals in the output statements. Section 13.2 explains how to precisely control the format of printed values.

8.5 Type Modifier

Unlike Pascal, C/C++ has no general subrange type. Instead, there are several type modifiers available to specialize the basic types into particular subranges:

Pascal	C/C++
at least -2147483648. .2147483647	**long int**
at least -32768. .32767	**short int**
at least 0. .65535	**unsigned short int**
at least 0. .4294967295	**unsigned long int**

Both Pascal and C/C++ support constant identifiers that are write-once/read-only. Pascal has a **const** section in each routine, while C/C++ has a type modifier, **const**, applicable in any variable declaration context; hence, C/C++ **const** is more general than Pascal **const**. A Pascal **const** identifier can only be assigned a compile-time value; a C/C++ **const** identifier can be assigned an expression calculated at runtime. Unlike Pascal, where the type of a constant identifier is inferred from the value assigned to it, a C/C++ **const** identifier must have a type specified, e.g.:

Pascal	C/C++
const x = 3; y = 7; c = 'x';	**const short int** x = 3, y = x + 7; **const char** c = 'x';

In both cases, the identifier must be initialized immediately after it is declared (write once), after which the identifier can appear in read-only contexts. In fact, any C/C++ declaration can have an initializing assignment:

```
short int i = 3;
```

where C/C++ uses the token "=" for assignment as opposed to ":=" in Pascal.

⇒ Edit hello.C
⇒ Make the following modification to routine main:

```
int main () {
    short int x;
    x = 65536;
    unsigned long int y = -1;
    const int z = y + 3;
    z = 4;
}
```

⇒ Compile the program.
⇒ Read the messages from the compiler until you understand why each is generated.

8.6 Type Constructor

Type constructors are declarations that build more complex types from the basic types. Like Pascal, C/C++ use name equivalence to decide if two types are the same.

	Pascal	C/C++
enumeration	type Colour = (R, G, B)	**enum** Colour { R, G, B }
subrange	1..5	N/A
set	set of	≈ bitset *// C++ only*
pointer reference	var p : ^Integer; ≈ var parameter	**int** *p; **int** &ip; *// C++ only*

structure	type info = record
	c : Char;
	i : Integer;
	d : Real;
	end
		struct info {
		char c;
		int i;
		double d;
		};

array	var a : array[0..9] of Integer; var b : array[0..9] of array[0..9] of Integer;	**int** a[10]; **int** b[10][10];
type (declaration)	type name = packed array[0..24] of Char; var first, last : name;	**typedef char** name[25]; name first, last;

Like Pascal, an enumeration in C++ denotes a new type; in C an enumeration is a synonym for **int**. The names in an enumeration are called enumerators. Like Pascal, enumerator names must be unique in a declaration scope. Unlike Pascal, the enumerators can be numbered explicitly, as well as implicitly. In addition, it is possible in C/C++ to combine type and var in a single declaration.

⇒ Edit hello.C
⇒ Make the following modification to routine main:

```
int main() {
    enum Day {Mon,Tue,Wed,Thu,Fri,Sat,Sun};         // type declaration, implicit numbering
    Day day = Sat;                                  // variable declaration, initialization
    enum {Yes, No} vote;                            // anonymous type and variable declaration
    enum Colour {R=0x1, G=0x2, B=0x4} colour;       // type and variable declaration, explicit numbering
    colour = B;                                     // assignment
    cout << "day:" << day << " colour:" << colour << endl;
}
```

⇒ Compile the program and run it.
⇒ Check the output and make sure you understand it.

In C, the keyword **enum** must always be specified when declaring an enumeration:

```
enum Day day = Sat;    // repeat "enum"
```

There is no direct subrange support in C/C++. The closest facility to subranges is type modifiers (see Section 8.5).

An approximation of Pascal set is provided by the C++ standard template library (STL) class bitset. Since sets are used infrequently, bitset is not discussed in this tutorial (see a C++ textbook).

The pointer/reference capabilities of C++ are a superset of those in Pascal. In a declaration, the pointer/reference type modifier in C/C++ is *not distributed across the identifier list*, e.g.:

int *a, b; is not var a, b : ^Integer;   **int** *a, *b; is var a, b : ^Integer;

Also, the dereference operator, *, appears to the left of the pointer variable not the right as in Pascal: a^ ≡ *a. Unlike Pascal, a pointer (or reference) can point at both local and dynamic variables. A C++ reference is like a Pascal var parameter, but more general; a var parameter is only available in a parameter list, but a C++ reference variable can be declared anywhere. In both cases, the initializing expression is automatically referenced (take the address of), and the variable is automatically dereferenced in executable statements. For example, the declaration **int** &rp = rhs is implicitly rewritten as **int** * **const** rp = *address-of*(rhs), while the expression rp = rp is implicitly rewritten as *rp = *rp. This rewrite is exactly what occurs in Pascal when an argument is assigned to a var parameter, and when the var parameter is used in the body of a routine. Notice, both var parameter and reference are constant pointers (never change what they point at, i.e., write-once/read-only), and hence, must be initialized on declaration.

⇒ Edit hello.C

⇒ Make the following modification to routine main:

```
int main() {
    int *ip, *jp;           // pointers to integer
    ip  = new int;          // allocate dynamic storage
    jp  = new int;          // allocate dynamic storage
    *ip = 3;                // assign value to dynamic storage
    *jp = *ip;              // copy value
    ip  = jp;               // copy pointer
    cout << "ip:" << ip << " *ip:" << *ip << endl;
    cout << "jp:" << jp << " *jp:" << *jp << endl;
    int i = 3;
    int &rp = i;            // reference to local storage (not allowed in Pascal)
    rp  = 4;                // change value of i through rp
    cout << "i:" << i << " rp:" << rp << endl;
}
```

⇒ Compile the program and run it.

⇒ Check the output carefully and make sure you understand it.

The operator **new** is discussed in Section 14 (as is operator **delete** to free allocated storage).

Records, called *structures* in C/C++, are syntactically different but semantically the same as records in Pascal, including nested structures. The fields of a structure are called *members* in C++. As for enumerations, a C++ structure can be defined and instances declared in a single statement.

⇒ Edit hello.C

⇒ Make the following modification to routine main:

```
int main() {
    struct complex { double r, i; };        // type declaration
    complex a, b;                           // variable declaration
    struct { double r, i; } c, d;           // anonymous type and variable declaration
    struct Complex { double r, i; } e;      // type and variable declaration
    a.r = 3.0;          // . (period) is used for member selection and decimal point
    a.i = 2.1;
    b = a;              // copies both members r and i
    cout << "a=" << a.r << "+" << a.i << "i" << endl;
    cout << "b=" << b.r << "+" << b.i << "i" << endl;
    c = a; d = b; e = a;        // assignments allowed ?
}
```

⇒ Compile the program and run it.

⇒ While the messages from the compiler may be cryptic, try to understand why they are generated (think about name equivalence for types).

⇒ Comment-out the last line, and compile and run the program.

In C, the keyword **struct** must always be specified when declaring a structure:

```
struct complex a, b;        // repeat "struct"
```

Recursive data types, like lists and trees, can be defined as in Pascal:

```
struct node {
    int val;
    node *link;             // pointer to instance of node
};
```

Arrays in C/C++ are primitive in comparison to Pascal. In Pascal, the value for each array dimension specifies the subscript range, which indirectly indicates the number of elements in that particular dimension. In C/C++, the lower subscript bound is always zero, the dimension value is the *number* of elements in that particular dimension, making the upper subscript bound one less than the dimension value, e.g.:

```
int a[10];       // array[0..9] of Integer
int b[10][20];   // array[0..9] of array[0..19] of Integer
```

The Pascal abbreviation for multidimensional arrays should *NOT* be used, as it means something completely different.

```
int b[10,20];    // NOT array[0..9] of array[0..19] of Integer
```

(See the use of the comma operator at the end of Section 9.) For declarations, the dimension value must be a compile-time value. (g++ allows a runtime expression.) Unlike Pascal, there is *no bounds checking* for invalid subscripts; it is the programmer's responsibility to ensure that correct subscripts are used for each dimension. Array assignment is not supported in C/C++ because in many contexts the exact size of the array is unknown (see Section 11.2). (g++ does supported limited array assignment where the array size is known.)

⇒ Edit hello.C

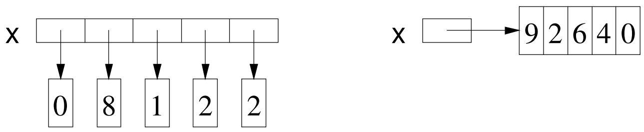
⇒ Make the following modification to routine main:

```
int main() {
    char n1[3], n2[3];
    n1[0] = 'T'; n1[1] = 'o'; n1[2] = 'm';
    n2 = n1;                  // array assignment, g++ only
    cout << n1[0] << n1[1] << n1[2] << endl;
    cout << n2[0] << n2[1] << endl;
    int v[3];
    v[0] = 93; v[1] = 67; v[2] = 77;
    cout << v[0] << " " << v[1] << " " << v[2] << " " << v[3] << endl;
}
```

⇒ Compile the program, run it, and check the output.

Notice the invalid subscript, v[3], does not generate an error!

Declaration of pointers to arrays and arrays of pointers is confusing in C/C++; *pay special attention*. (Pointers to routines are even more complex; see a C/C++ textbook.) The declaration **int** *x[5] could mean one of two things:



On the left is an array of 5 pointers to integers and the right is a pointer to an array of 5 integers. The question is whether the * or [] is applied first. In fact, dimension has higher priority (as for subscript, see Section 9), so the declaration in interpreted as **int** (*(x[5])) (left example), where parenthesis indicate the ordering of the type qualifiers. To read a declaration, parenthesize all the type qualifiers, and read from inside the parenthesis outwards, starting with the variable name and ending with the type name on the left, rather than left to right as in Pascal:

```
int *(x[5]);     // x : array[0..4] of ^Integer
int (*x)[5];     // x : ^array[0..4] of Integer
```

⇒ Write the Pascal declaration for: **int** (*(x[5]))[10]

Unlike Pascal, **typedef** in C/C++ is a synonym for a type, whereas type in Pascal creates a new type. In other words, **typedef** is more like Pascal **const**, but for declarations rather than executable statements, e.g.:

```
typedef short int shrint1;   // NOT type shrint1 = -32768..32767;
typedef shrint1 shrint2;     // NOT type shrint2 = shrint1;
typedef short int shrint3;   // NOT type shrint3 = -32768..32767;
shrint1 s1;      // implicitly rewritten as: short int s1
shrint2 s2;      // implicitly rewritten as: short int s2
shrint3 s3;      // implicitly rewritten as: short int s3
```

All possible combinations of assignments are allowed in C/C++ among the variables s1, s2 and s3, because they are the same type. In Pascal, variables s1, s2 and s3, have different types and assignment among them is disallowed.

8.7 Type Constructor Literal

pointer	0 or NULL indicates an empty/null/nil pointer (NULL is defined in Section 12)
structure	complex c = { 3.0, 2.1 };
array	**int** a[3] = { 1, 2, 3 };

Structure and array initialization can only occur as part of a declaration. (g++ allows type constructor literals in executable statements, see Section 9.2.) Values in the initialization list are placed into a variable starting at the beginning of the structure or array, but not all the members/elements have to be initialized. A nested structure or multidimensional array is initialized by creating corresponding nesting levels using braces:

```
int m[2][3] = { {93, 67, 72}, {77, 81, 86} };
```

String literals can be used as a shorthand array initializer value:

```
char s[6] = "abcde";      implicitly rewritten as      char s[6] = { 'a', 'b', 'c', 'd', 'e', '\0' };
```

When initializing, it is possible to leave out the first dimension, and the compiler infers its value from the number of literals in that dimension:

```
char s[] = "abcde";   // first dimension inferred as 6
int  m[][3] = { {93, 67, 72}, {77, 81, 86} };    // first dimension inferred as 2
```

⇒ Edit hello.C

⇒ Make the following modification to routine main:

```
int main() {
    char n[] = "Tom";
    int  m[][2] = { {93, 67}, {77, 81} };
    struct complex { double r, i; } c = { 3.4 };    // not all members initialized
    cout << n[0] << n[1] << n[2] << endl;
    cout << m[0][0] << " " << m[0][1] << " " << m[1][0] << " " << m[1][1] << endl;
    cout << c.r << " " << c.i << endl;
}
```

⇒ Compile the program, run it, and check the output.

9 Expressions

	Pascal	C/C++	priority
	(), ., [], ., routine call	(), ., [], ->, routine call	highest
(unary)	N/A, ^, +, -, not, N/A, N/A	&, *, +, -, !, ~, cast (explicit conversion)	
	*, div, /, mod	*, /, /, %	
(binary)	+, -	+, -	
	N/A	<<, >> (bit shift)	
	<, <=, >, >=	<, <=, >, >=	
	=, <>	==, !=	
	and	&	
	N/A	^ (exclusive-or)	
	or	|	
	N/A	&&	
	N/A	||	
	N/A	?:	
	:=, N/A ...	=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=	
	N/A	,	lowest
	Abs round, trunc Ord, Chr Exp, Ln, Sqrt, ArcTan, Sin	abs (**int**), fabs (**double**) N/A, conversion from **double** to **int** conversion to/from **int**/**enum** to **char** exp, log, sqrt, atan, sin	

The priority of operators in the table is for C/C++; Pascal operators have a slightly different order of priority, e.g., and before >. Only those operators that are new or different from their Pascal counterpart are discussed.

Like Pascal, operators with the highest priority in an expression are done first. If two operators have the same priority, they are done left to right, except for unary, ?:, and assignment operators, which are done right to left. However, order of parenthesized expressions and routine calls is undefined:

```
(i + j) * (k + j);        // either + may be done first
g(i) + f(k) + h(j);       // g, f, or h may be called in any order
```

In Pascal, there are two division operators, DIV for integers and / for reals. In C/C++, there is one overloaded operator /, which acts like Pascal DIV if both operands are integers and like Pascal / if one or more operands is a real.

⇒ Edit hello.C
⇒ Make the following modification to routine main:

```
int main() {
    cout << 5 / 2 << " " << 5 / 2.0 << " " << 5.0 / 2.0  << endl;
}
```

⇒ Compile the program, run it, and check the output.

The address-of operator, &, does not exist is Pascal (some extended Pascals use @). It returns a pointer to a variable regardless of storage class, i.e., it is possible to determine the address of global and local variables. For example, &x is the address of x, &s.d is the address of member d in structure s, and &a[5] is the address of array element a[5].

The arrow operator, ->, does not exist in Pascal (or any other computer language). It exists solely because the priority of operator "." is incorrectly higher than unary "*", which means *p.f executes as *(p.f) instead of (*p).f. The -> operator performs a dereference and member selection in the correct order, i.e., p->f is implicitly rewritten as (*p).f.

⇒ Edit hello.C
⇒ Make the following modification to routine main:

```
int main() {
    struct node { int val; node *link; };
    node x = { 5, NULL }, y = { 3, &x }, *n = &y;
    cout << n->val << endl;
    n = n->link;        // n = n^.link
    cout << n->val << endl;
}
```

⇒ Compile the program, run it, and check the output.

While it is possible to build a recursive data type without dynamically allocating storage by using the address-of operator, it is discouraged. It is possible to pass pointers to local variables as an argument to other routines, e.g., n, but returning it results in undefined behaviour because the local storage for the nodes is freed when the routine returns.

C/C++ allows arithmetic on pointer variables, making it possible to write low-level storage management operations. For example, subscripting an array is just pointer arithmetic, which calculates the address of a particular element in an array. When a literal is used in a pointer calculation, it is implicitly scaled to the size of the pointer type. For example, given the array declaration **int** v[3], the subscript v[3] is implicitly rewritten as *(v+3), where v is a pointer to the start of the array, the + 3 adds 3×4 to v because the subscript is scaled by the pointer size (4 bytes for **int**), and the pointer sum is then dereferenced to obtain the value of element v[3].

The bit-shift operators, << (left), and >> (right), do not exist in Pascal. (Notice the bit-shift operators are overloaded with the input and output operators.) They are used to shift bits in integer variables left and right. Shifting left is defined to be equivalent to multiplying by 2, modulus the integer size; shifting right is equivalent to dividing by 2 if the integer is unsigned or positive, and undefined otherwise. For example, 1 << 3 shifts the value 1, 3 bits left, giving 8, while 8 >> 3, shifts the value 8, 3 bits right, giving 1.

Unlike Pascal, the symbol for equality in C/C++ is == not =, and the symbol for not equals is != not <>.

Unlike Pascal, the symbol for assignment in C/C++ is = not :=, and assignment is an operator not a statement, which means assignment yields a value, the left-hand operand after the assignment, so it can occur anywhere and multiple times in an expression:

```
i = 3; j = 6;
t = ( i = j ) + ( j = i );        // value of t?
```

The value of t is undefined because the order of parenthesized expressions is undefined. If the first parenthesized expression is done first, the value of t is 12, otherwise 6. Changing variables during the evaluation of an expression causes side-effects. Good programming practice is to have only one assignment in an expression on the left-hand side of the expression, i.e., one side-effect after the expression is evaluated. One exception to this rule is *cascade assignment*, which is useful for initializing multiple variables to a common value:

```
x = y = j + 4;          // cascade assignment
a = b = c = 0;
```

Operators, such as lhs += rhs, are implicitly rewritten as:

```
temp = &(lhs); *temp = *temp + rhs;
```

Hence, the left-hand side, lhs, is evaluated only once:

```
a[rand() % 5] += 1;          // only calls random once
a[rand() % 5] = a[rand() % 5] + 1;     // calls random twice
```

The first expression increments a random element of the array, while the second increments a random value of the array, and assigns the incremented value to a random and probably different element of the array.

The comma expression list does not exist in Pascal, and is used largely by automatic translators, not programmers. It is a series of expressions separated by commas; the expressions are evaluated left to right, and the value of the right most expression is returned as the result of the comma expression. The comma expression allows multiple expressions to be evaluated in a context that only allows a single expression. In effect, a comma expression is an anonymous routine, whose body is the series of expressions and whose result is the value of the last expression.

```
f( 3, (t=a+b, c+t), x );          // same as: t = a+b; f( 3, c+t, x );
```

(The parenthesis are necessary around the comma expression to differentiate it from elements of the argument list.) This style of coding is discouraged and unnecessary; however, there is one situation where the comma expression is commonly used (see Section 10.5). The earlier dimension problem with b[10,20] (see page 11) is now explained, as it means b[20] because 10,20 is a comma expression not a dimension list (g++ only). The same problem occurs with subscripting as b[3,4] means b[4], which is the 4th row of the matrix.

The control structures &&, || and ?: are discussed in Section 10.4.

9.1 Type Operator

The pseudo-routine **sizeof** does not exist is Pascal. It returns the number of bytes for a type or variable:

```
long int i;
sizeof(long int);     // type, at least 4
sizeof(i);            // variable, at least 4
```

The **sizeof** a pointer (type or variable) is the size of the pointer on that particular computer and NOT the size of the type the pointer refers to.

⇒ Edit hello.C

⇒ Make the following modification to routine main:

```
int main() {
    struct node { int val; node *link; };
    node x = { 5, NULL }, y = { 3, &x }, *n = &y;
    cout << sizeof(node) << " " << sizeof(x) << " " << sizeof(n) << endl;
    cout << sizeof("abc") << endl;
}
```

⇒ Compile the program, run it, and check the output.

9.2 Conversion

Conversion is transforming a value from one type to another type, which can be performed implicitly or explicitly. Pascal has only one implicit conversion: from integer to real; C/C++ has a large number of implicit conversions, which can be divided into two kinds:

- *widening* conversion, no information is lost, for example:

char	→	**short int**	→	**long int**	→	**double**
'\x7'		7		7		7.000000000000000

- *narrowing* conversion, information can be lost, for example:

double	→	**long int**	→	**short int**	→	**char**
77777.77777777777		77777		12241		´\xd1´

Clearly implicit narrowing conversions can cause problems, such as:

```
int i;  double r;
i = r = 3.5;        // value of r?
r = i = 3.5;        // value of r?
```

In both expressions, i is assigned the value 3 because of the implicit conversion of real to integer, but r is assigned 3.5 in the first expression and 3.0 in the second because the conversion from 3.5 to i is a narrowing conversion. *Be careful!*

Pascal has routines for explicit conversion, e.g., trunc and round, from real to integer. C/C++ has similar routines, but they are invoked explicitly using a special unary operator syntax, called a *cast*.

```
int i;  double x;
i = (int)x;                       // explicit conversion, like trunc(x)
x = (double)i / (double)i;        // explicit conversion to get correct division, one conversion is sufficient
```

The syntax indicates a conversion routine is needed for converting from the type of the cast operand to the type specified in the parenthesis of the cast; the conversion routine is implicitly called with the cast operand as its argument. C/C++ has a fixed set of builtin conversion routines, so it is impossible to arbitrarily convert among types. Writing:

```
struct complex { double re, double im; } c;
int i;
c = (complex)i;    // try converting int to complex
```

fails because there is no builtin conversion routine that takes an integer value and converts it to a complex value. (Section 15 shows how to define one.) Good programming practice is to have as few conversions as possible due to potential narrowing problems. If a conversion is necessary, use an explicit cast, so the next programmer reading the program can easily see a conversion is occurring.

⇒ Edit hello.C
⇒ Make the following modification to routine main:

```
int main() {
    char c;
    short int si;
    long int li;
    double d;
    d = li = si = c = ´\x7´;                       // implicit widening conversions
    cout << (int)c << " " << si << " " << li << " " << d << endl;
    c = si = li = d = 77777.77777777777;        // implicit narrowing conversions
    cout << (int)c << " " << si << " " << li << " " << d << endl;
}
```

⇒ Compile the program and run it.
⇒ Check the output carefully for anomalies.

The cast "(**int**)" of variable c forces c's value to be printed as an integer and not a character. (Try running the program without the cast.) Again, some of the printed values are different from the literals in the assignment statements.

As mentioned in Section 8.7, g++ has a cast extension allowing construction of structure and array literals in executable statements not just declarations.

```
int m[2][3];
struct complex { double r, i; } c;
m = (int [2][3]){ {93, 67, 72}, {77, 81, 86} };        // g++ only
c = (complex){ 2.1, 3.4 };                              // g++ only
```

In both assignments, a cast is used to indicate the meaning and structure of the initialization value.

10 Control Structures

Pascal	C/C++
N/A	{ d1; d2; ... stmt1; stmt2; ... }
if cond1 then stmt1 **else if** cond2 then stmt2 ... **else** stmtn	**if** (cond1) stmt1 **else if** (cond2) stmt2 ... **else** stmtn

case e of
    c1: stmt1;
    ...
    cn: stmtn;
    otherwise: stmt0; (extended)
end
	switch (e) {
	    case c1: stmt1; **break**;
	    ...
	    case cn: stmtn; **break**;
	    default: stmt0;
	}

while cond **do**
    stmt
	while (cond)
	    stmt

repeat
    stmt1; stmt2; ...
until cond
	do {
	    stmt1; stmt2; ...
	} **while** (! cond) ;

for i := f to l **do**
    stmt
	for (i = f; i <= l; i += 1)
	    stmt

break (extended)	**break**
with	N/A
goto	**goto**

10.1 Block

A *block* is a series of statements bracketed by braces, { and }, which can be nested one within another. (As opposed to a comma expression list, which only contains expressions.) A C/C++ block is more general than Pascal begin/end because it can appear in any context that allows a statement not just as the body of a routine. A block serves two purposes: bracket several statements into a single statement, like begin/end in Pascal, and introduce local declarations, unlike Pascal. The latter point is not as relevant in C++ because declarations can occur anywhere within a block.

```
if ( a > b ) {
    int temp = a;    // local declaration in block
    a = b;
    b = temp;
}
```

(The **if** statement is discussed next.) Putting declarations precisely where they are needed can help reduce declaration clutter at the beginning of a block; however, it can also make locating them more difficult. Like Pascal, each control structure takes a single statement or a block. Good programming practice is to always use a block as it allows easy insertion and removal of statements in a program.

10.2 Conditional

A conditional occurs in control structures that optionally transfer control based on the result of a conditional expression, e.g., condition of **if**, **while**, **do**, and **for** control structures. In C/C++, a conditional must be surrounded with parenthesis (). In Pascal, a conditional expression must evaluate to a boolean result, which indicates if control transfers. However, in C/C++, the conditional expression is evaluated and implicitly tested for not equal to zero (or NULL for pointers), i.e., cond != 0; hence, 0 $\Rightarrow$ false and any other value $\Rightarrow$ true for a C/C++ conditional. It is common is C/C++ programs to see:

```
if ( p ) ...       implicitly rewritten as    if ( (p) != 0 ) ...
while ( p ) ...    implicitly rewritten as    while ( (p) != 0 ) ...
```

If the conditional returns a boolean result, **false** is converted to 0 and **true** to 1, which causes the implicit test to generate the original boolean value. Watch for the following mistake in a conditional:

```
if ( x = y ) . . .        // implicitly rewritten as ( (x = y) != 0 )
```

which assigns y to x and tests x != 0. Remember the test for equality in C/C++ is ==, not =, and assignment is an operator not a statement.

10.3 Selection Statement

Like Pascal, there are two selection statements in C/C++: **if** and **switch**.

An **if** statement selectively executes one of two alternatives, and is the same as in Pascal, including the *dangling else* problem for associating **if** with **else**. For example, reward the WIDGET salesperson who sold more than $10,000 worth of WIDGETS and dock the pay of those who sold less than $5,000 worth.

Dangling Else	Fix Using Blocks	Fix Using Null Else

<pre>if (sales < 10000)
    if (sales < 5000)
        income -= penalty;
else   // dangling else
    income += bonus;</pre>
	<pre>if (sales < 10000) {
	    if (sales < 5000) {
	        income -= penalty;
	    }
	} else {
	    income += bonus;
	}</pre>
		<pre>if (sales < 10000)
		    if (sales < 5000)
		        income -= penalty;
		    else ; // null statement
		else
		    income += bonus;</pre>

The solution using blocks is preferred because it allows subsequent statements to be easily added and removed.

C/C++ **switch** statement is different from Pascal **case**.

```
switch ( day ) {
  case MON: case TUE: case WED: case THU: // list of case values
    cout << "PROGRAM" << endl;
    break;        // exit switch
  case FRI:
    wallet += pay;
    // fall through !!!!!
  case SAT:
    cout << "PARTY" << endl;
    wallet -= party;
    break;        // exit switch
  case SUN:
    cout << "REST" << endl;
    break;        // exit switch
  default:
    cerr << "ERROR" << endl;
    exit( -1 );   // terminate program
}
```

Unlike Pascal, once a case clause is selected in C/C++, control continues to the *next* case clause unless there is a **break** statement to exit the **switch** statement. (It is common to forget a **break**.) Unlike Pascal, which allows a list of values for a single case clause, in C/C++ only one label is allowed for each **case** clause, but a list of **case** clauses is allowed. Unlike Pascal, if no case clause is matched, it is not an error; the statement simply does nothing. As well, in C/C++, there is a **default** clause that is executed if no case clause matches (some extended Pascals have an equivalent otherwise clause).

10.4 ⋈ Conditional Expression Evaluation

Conditional expression evaluation is used to perform partial evaluation of expressions (some extended Pascals provide this capability). These are control structures, not true operators because both operands are not evaluated, as for other operators.

Symbol	Meaning
&&	short-circuit logical and: only evaluates the right operand if the left operand is true
||	short-circuit logical or: only evaluates the right operand if the left operand is false
?:	if statement in an expression: only evaluates one of two alternative parts of an expression

Conditional && and || (often referred to as *short-circuit*), are similar to logical & and | for boolean operands, i.e., both produce a logical conjunction or disjunctive result. However, conditional && and || evaluate operands lazily until a result is determined, short-circuiting the evaluation of other operands, while logical & and | evaluate operands eagerly, evaluating all operands (like Pascal and and or). In many situations with boolean operands, the corresponding operators are interchangeable.

⇒ Edit hello.C
⇒ Enter the following program:

```
#include <iostream.h>
int main() {
    if ( ( cout << "a", true )  |  ( cout << "b", true ) ) cout << endl;
    if ( ( cout << "a", true )  || ( cout << "b", true ) ) cout << endl;
}
```

⇒ Compile the program and run it.
⇒ Check the output and make sure you understand it.

Notice that for logical | both operands of the conditional are evaluated, while for short-circuit || only the first operand is evaluated.

Conditional ?: conditionally evaluates one of two expressions, and returns the result of the evaluated expression, i.e., it acts like an **if** statement in an expression, e.g., abs2 = (a < 0 ? -a : a) + 2. Like the comma expression, this operator is used infrequently.

⇒ Edit hello.C
⇒ Enter the following program:

```
#include <iostream.h>
int main() {
    int w = 1;
    cout << "There " << (w>1 ? "are ":"is ") << w << " widget" << (w>1 ? "s":"") << endl;
    w = 2;
    cout << "There " << (w>1 ? "are ":"is ") << w << " widget" << (w>1 ? "s":"") << endl;
}
```

⇒ Compile the program and run it.

10.5 Iteration

The C/C++ **while** statement is the same as in Pascal, except for the differences mentioned previously about the conditional. As for Pascal, beware of accidental infinite loops:

```
x = 0;                                  x = 0;
while (x < 5)      // missing block     while (x < 5);     // extra semicolon!
    y = y + x;                              x = x + 1;
    x = x + 1;
```

⇒ Edit hello.C
⇒ Enter the following program:

```
#include <iostream.h>
int main() {
    int val = 1;            // initialize
    while ( val ) {         // conditional
        cout << val << endl;
        val <<= 1;          // shifting left 1 bit position
    }
}
```

⇒ Compile the program and run it.
⇒ Can you explain the last value printed and why the loop stopped? (Think of the internal representation of integer values and how a conditional works.)

Unlike the Pascal repeat statement, the C/C++ **do** statement only allows a single statement for the loop body, and the loop body is executed while the conditional is true, not while the conditional is false.

```
do {
    // use a block for multiple statements
} while ( a != x & b );    // execute while conditional is true
```

Like the Pascal repeat statement, the C/C++ **do** is only used when the loop body must be executed at least once.

The C/C++ **for** statement is significantly more general than its Pascal counterpart:

```
for ( initialization; conditional; increment )
    stmt;
```

which is implicitly rewritten as:

```
initialization;
while ( conditional ) {
    stmt;
    increment;
}
```

Some examples of **for** loops are:

```
for ( i = 1; i <= 10; i += 1 ) {            // for i := 1 to 10 do
    // loop 10 times
} // i has the value 11 on exit, not 10

for ( i = 10; 1 <= i; i -= 1 ) {            // for i := 10 downto 1 do
    // loop 10 times
} // i has the value 0 on exit, not 1

for ( p = l; p != NULL; p = p->link ) {
    // loop through list structure
}
for ( i = 1, p = l; i <= 10 & p != NULL; i += 1, p = p->link ) {    // 2 indices
    // loop until 10th node or end of list encountered
}
```

The last example illustrates the use of the comma expression to initialize and increment 2 indices in a context where normally only a single expression is allowed. Unlike Pascal, the last increment is performed so the last value of the loop control variable is different from Pascal. Also, the control variable in C/C++ can be modified in the loop body (discouraged). Magically, a default value of **true** is inserted, if no conditional is specified for a **for** statement.

```
for ( ; ; )        // rewritten as: for ( ; true ; )
```

⇒ Edit hello.C

⇒ Make the following modification to routine main:

```
int main() {
    int val;
    for ( val = 1; val; val <<= 1 ) {
        cout << val << endl;
    }
}
```

⇒ Compile the program and run it.

Another example of short-circuit expression evaluation (see Section 10.4) is a linear search of an array for a key, where the loop index indicates the position of the key in the array if the key is found, or the array size plus 1 if not found:

```
for ( i = 0; i < size && list[i] != key; i += 1 );    // no loop body
```

The short-circuit && only evaluates the second operand of the conditional if the first operand is true, otherwise there is a potential subscript error when the key is not found; i is equal to size on the last loop iteration, and if both operands are evaluated, it results in list[size], which is one past the end of the array (subscripts have origin zero). Therefore, using logical & would be incorrect because it evaluates both operands. Why must you worry about array subscript problems, even when C/C++ does not perform subscript checking? The reason is that the invalid subscript can result is other errors, such as addressing outside the program's memory, which is called a *segment fault*.

Finally, the **break** statement can be used in all looping constructs to cause immediate termination of the loop. The previous linear search using short-circuit && can be rewritten using loop exits.

```
for ( i = 0; ; i += 1 ) {              // infinite loop, conditional defaults to "true"
    if ( i == size ) break;            // exit if not found
    if ( list[i] == key ) break;       // exit if found
}
```

Since the loop exits when i is equal to size a subscript error does not occur.

There is no equivalent to the Pascal with statement in C/C++, and **goto** is not discussed here (see a C/C++ textbook).

11 Routine

Pascal	C	C++

procedure p(or function f(
    a : t1;
    VAR b : t2;
    N/A
) : t4
    declarations
begin
    statements
end
	void p(or t4 f(
	    t1 a,
	    N/A,
	    N/A
	)
	{
	    declarations
	    statements
	}
		void p(or t4 f(
		    t1 a,
		    t2 &b = v1,
		    const t3 &c = v2
		)
		{
		    declarations
		    statements
		}

Unlike Pascal, routines in C/C++ cannot be nested, so all routine names are at the same scope level in a source file. Therefore, the only routine scoping occurs between the global scope of the source file and routine bodies.

```
int i = 1;          // global scope
int main() {
    int i = 2;      // local scope, hides previous variable i
}
```

⇒ Edit hello.C

⇒ Enter the following program:

```
#include <iostream.h>
int i = 3;
int main() {
    cout << i << endl;
    int i = 4;
    cout << i << endl;
}
```

⇒ Compile the program and run it.

⇒ Check the output and make sure you understand it.

Unlike Pascal, C/C++ distinguishes between a procedure or function based solely on the existence of a return type, which is specified at the beginning of the routine header rather than at the end. Procedures are routines that do not return a value, which is indicated with a return type of **void**:

```
void r( ... ) { ... }
```

A routine with no parameters is specified with parameter **void** in C and an empty parameter list in C++:

```
... r( void ) { ... }     // C: no parameters
... r() { ... }           // C++: no parameters
```

Unlike Pascal, which terminates a procedure or function only when control runs off the end of the routine body, C/C++ has a **return** statement to indicate termination of the routine body. A **return** statement can appear anywhere in the routine body, and multiple return statements are possible (discouraged). Unlike Pascal, the routine name is not a variable in the body of the routine used to return a value to the call site. In C/C++, the **return** statement is used to return a value to the call site.

```
return;           // for procedure, no value returned
return a+b;       // for function, value returned is the expression a+b
```

⇒ Edit hello.C
⇒ Enter the following program:

```
int main() {
    return 0;    // indicate successful completion to the shell
}
```

⇒ Compile the program and run it.
⇒ Print the return value with the command echo $status from the csh/tcsh shell or echo $? from the sh/bash shell.
⇒ Try returning a different return value and check that the shell receives it.

11.1 Argument/Parameter Passing

Unlike Pascal, it is impossible to declare lists of parameters for a specific type. In C/C++, each parameter must have a type specified.

```
void f( int i, j, k, double f, g, h ) { ... }    // not allowed
void f( int i, int j, int k, double f, double g, double h ) { ... }    // allowed
```

Like Pascal, C++ arguments can be passed by value or reference (variable). C arguments are only passed by value; passing arguments by reference is accomplished by passing an explicit pointer, which is awkward and not discussed here (see a C textbook). In addition, C++ has read-only reference parameters so the corresponding argument is guaranteed not to be changed by the routine, which provides the efficiency of pass by reference for large variables but the security of pass by value because the argument cannot change. (A read-only value parameter is possible in C/C++, but provides no advantage over a normal value parameter. Why?)

```
void f( const complex &c, const int v[10], const int &i ) {
    c.r = 3.0;    // assignment not allowed
    v[0] = 3;     // assignment not allowed
    i = 3;        // assignment not allowed
}
```

The reason v is not declared a reference parameter is discussed in Section 11.2.

As well, a C++ parameter can have a default value, which is passed as the argument value if no argument is specified at the call site. In a routine, once a parameter has a default value, all parameters to the right of it must have default values. At a call site, once an argument is omitted for a parameter with a default value, no more arguments can be specified to the right of it.

```
int f( int i, double g, char c = '*', double h = 3.5 ) { ... }
f( 1, 2.0, 'b', 9.3 );    // maximum arguments
f( 1, 2.0, 'b' );         // h defaults to 3.5
f( 1, 2.0 );              // c defaults to '*', h defaults to 3.5
```

When a routine is called, all the expressions in the argument list are evaluated *in any order*. For value parameters, the result of an expression is assigned to the corresponding parameter, *which may involve an implicit conversion*, and for reference parameters, the expression result is referenced (take the address of) and this address is assigned to the corresponding parameter. Because the argument list is evaluated in any order, side-effects in an argument list can produce unpredictable results:

```
double g( int a, int b, int c ) { ... }
int i = 0;
g( i += 1, i += 1, i += 1 );
```

Never do this as the possible arguments passed to routine g are (1, 2, 3), (2, 3, 1), (3, 2, 1), ...

11.2 Array Parameters

Because array copy is unsupported (see Section 8.6), arrays cannot be passed by value only by reference. Therefore, all array parameters are implicitly call by reference parameters, and hence, the reason why parameter v does not have a reference symbol in the previous section. A parameter declaration can specify the first dimension with a dimension value, [10] (where the value is ignored), an empty dimension list, [ ], or a pointer, *; the following rows are equivalent:

```
double sum( double v[10] );        double sum( double v[] );       double sum( double *v );
double sum( double m[10][10] );    double sum( double m[][10] );   double sum( double (*m)[10] );
```

(The extra parenthesis are needed in the second right-most example because dimension has higher priority than pointer.) Good programming practice is to use the middle form because it clearly indicates that the variable is going to be subscripted, and that one dimension is free while the other dimension is fixed.

A limited conformant array capability for the first dimension makes it possible to write a routine to add up the elements of an arbitrary sized vector or a matrix with an arbitrary number of rows *but a fixed number of columns*.

```
double sum( double v[ ], int cols ) {
    int total = 0.0;
    for ( int i = 0; i < cols; i += 1 )
        total += v[i];
    return total;
}
```

```
double sum( double m[ ][10], int rows ) {
    int total = 0.0;
    for ( int i = 0; i < rows; i += 1 )
        for ( int j = 0; j < 10; j += 1 )
            total += m[i][j];
    return total;
}
```

⇒ Edit hello.C

⇒ Enter the following program:

```
int r( int *(v[ ]), int (*p)[5] ) { }
int main() {
    int *(x[5]);        // array of 5 pointers to int
    int (*y)[5];        // pointer to array of 5 int
    r( x, y );
}
```

⇒ Compile the program to ensure it is correct.

⇒ Remove the dimension value 5 from parameter p for routine r.

⇒ Compile the program again, and explain the error message. (This is a trick question; look at the previous programs for a hint).

11.3 Prototype

A language with a declaration-before-use rule has a fundamental problem specifying mutually recursive routines:

```
void f() {
    g();        // g is not defined and it is being used
}
void g() {
    f();        // f is defined and can be used
}
```

Clearly, interchanging the two routines does not solve the problem. The problem is solved using a *forward* declaration, called a *prototype* in C/C++, to introduce a routine's type before its actual declaration.

Pascal	C/C++

function f(. . .) : . . .; Forward;
. . .
function f; (* type not repeated *)
begin
    . . .
end;
	. . . f(. . .); // parameter names optional and no routine body
	. . .
	. . . f(. . .) // type repeated and checked with the prototype
	{
	    . . .
	}

Unlike Pascal, the prototype parameter names in C/C++ are optional (but usually specified for documentation reasons), and the actual routine declaration repeats the routine type and the repeated type must match the prototype.

⇒ Edit hello.C

⇒ Enter the following program:

```
#include <iostream.h>
void f( int );                // forward, no parameter name
void g( int i ) {
    cout << "g(" << i << ")" << endl;
    if ( i > 0 ) f( i - 1 );
}
```

```
void f( int i ) {             // check for match with prototype
    cout << "f(" << i << ")" << endl;
    if ( i > 0 ) g( i - 1 );
}
int main () {
    f( 5 );
    cout << endl;
    g( 4 );
}
```

⇒ Compile the program, run it, and check the output.
⇒ Remove the forward declaration for f.
⇒ Compile the program and read the message from the compiler.

Routine prototypes are useful for organizing routines in a source file, e.g., allowing the main routine to appear first in a source file, and for separate compilation (see Section 19.1).

```
int g( double );                // forward declarations
int f( double );
int main() {
    f( 7.5 );                   // declarations later
    g( 9.1 );
}
int g( double x ) { ... }       // declarations
int f( double x ) { ... }
```

11.4 ⋈ Overloading

All languages have some form of overloading, where a name has multiple meanings when used in different ways in the same context. In Pascal/C/C++, most of the operators are overloaded to work with both integer and real operands, e.g., the + operator is different for 1 + 2 than for 1.0 + 2.0 in a block. In Pascal, the builtin routine Abs is overloaded to work with integer and real arguments, e.g., Abs(-3) and Abs(-3.0). However, as mentioned on page 13, the / operator is not overloaded in Pascal; there is DIV for integers and / for reals.

Overloading a name is possible as long as the compiler can disambiguate among the name's meanings based on some criterion; the criteria normally used is type information. In general, overloading is done on operations not variables, so each variable name is distinct in a block but a routine name may have multiple meanings. Pascal/C only allows builtin operations to be overloaded; C++ allows programmer defined overloaded operations. The criteria used to select among a name's different meanings are the number and types of the parameters *but not the return type*.

```
int r(int i, int j) { ... }       // overload name r three different ways
int r(double x, double y) { ... }
int r(int k) { ... }
r( 1, 2 )          // invoke 1st r based on integer arguments
r( 1.0, 2.0 )      // invoke 2nd r based on double arguments
r( 3 )             // invoke 3rd r based on number of arguments
```

Hence, it is possible to write the two Pascal Abs routines in C++, and use them like the builtin Pascal Abs.

⇒ Edit hello.C
⇒ Enter the following program:

```
#include <iostream.h>
int Abs( int val ) { return val >= 0 ? val : -val; }
double Abs( double val ) { return val >= 0 ? val : -val; }
int main () {
    cout << Abs( 1 ) << " " << Abs( -1 ) << endl;
    cout << Abs( 1.1 ) << " " << Abs( -1.1 ) << endl;
}
```

⇒ Compile the program, run it, and check the output.

Implicit conversions between arguments and parameters at call sites can cause problems with overloaded routines.

```
int r( int i, double d ) { ... }
int r( double d, int i ) { ... }
double x, y;
r( x, y );        // convert which double argument to int ?
```

Because either real argument can be converted to integer, it is impossible to disambiguate between the overloaded r routines. Use an explicit cast to provide sufficient information to disambiguate, e.g., r((**int**)x, y).

Notice there is overlap between overloading and default arguments when the parameters have the same type.

Overloading	Default Argument
int r(**int** i, **int** j) { ... } **int** r(**int** i) { **int** j = 2; ... } r(3); *// 2nd overloaded declaration of r*	**int** r(**int** i, **int** j = 2) { ... } r(3); *// default argument of 2*

If the overloaded routine bodies are essentially the same, use a default argument, otherwise use overloaded routines.

⇒ Enter the following program:

```
int r( int i ) { }
int r( int i, int j = 2 ) { }
int main() { r( 3 ); }
```

⇒ Compile the program and explain the error message.

12 ⋈ Preprocessor

C/C++ are actually composed of two languages: a preprocessor language, which manipulates the text of the program *before* compilation, and the C/C++ language, which specifies how the computer is to execute *after* compilation. Hence, the program you see is not what the compiler sees; the compiler sees the program after it is changed by the preprocessor. Occasionally it is necessary to use the -E flag (see Section 4) to print the output of the preprocessor to understand why the compiler is generating unusual error messages.

The three most commonly used preprocessor facilities are textual substitution, textual inclusion, and conditional textual inclusion (see a C textbook for other preprocessor facilities). The syntax of a preprocessor statement is a **#** at the start of a line, followed by optional blanks, and then a preprocessor statement; no semi-colon!

12.1 Textual Substitution

Textual substitution is like **const** in Pascal, except the constant name can be any string of text, not just literals. The **#define** statement declares a preprocessor variable, and its value is all the text after the name up to the end of the line.

```
#define Integer int
#define begin {
#define end }
#define PI 3.14159
#define X 1 +
#define Y Fred =
Integer main() begin        // same as: int main() {
    Integer x = 3;          // same as: int x = 3;
    Y X PI;                 // same as: Fred = 1 + 3.14159;
    X Y PI;                 // same as: 1 + Fred = 3.14159;
end                         // same as: }
```

As these examples show, the preprocessor can transform the basic syntax of a C/C++ program (discouraged). It is also possible to make mistakes that are difficult to locate, because what you see is not what the compiler sees.

⇒ Edit hello.C
⇒ Enter the above program.
⇒ Compile the program with the command: g++ -E hello.C
⇒ Look carefully at the output.

The first line informs the compiler of the original source file name before preprocessing occurred so the compiler can generate meaningful error messages. Then there is an empty space where the preprocessor **#define**s used to be;

since preprocessor statements are not understood by the compiler, they must be removed. Finally, the preprocessed statements appear, without comments, which the compiler sees, and compiles.

Traditionally, textual substitution was used to give names to literals, like Pascal **const**; now, this is better done using **const** declarations:

```
const double PI = 3.14159;
const int arraySize = 100;
```

#define can also be used to declare macros, which are one line routines expanded inline during compilation, e.g.:

```
#define MAX(a,b) ((a > b) ? a : b)
z = MAX( x, y );        // implicitly rewritten as: z = ((x > y) ? x : y)
```

however, this capability is better handled by **inline** routines in C++ (see a C++ textbook for details).

12.2 Textual Inclusion

Textual inclusion is used to bring a block of text from a file into a C/C++ program. The file that is included may contain anything. Most commonly it contains prototype declarations for library routines that are used in a program, as well as **#define**s for literals needed in using the library routines. In effect, textual inclusion is a shorthand for typing in the same prototype declarations and **#define**s into your program. C/C++ convention uses the suffix ".h" for files containing C/C++ prototypes and **#define**s.

The **#include** statement specifies the file to be included. The file name can be enclosed in `""` or <>.

```
#include "user.h"
#include <system.h>
```

`""` means that the preprocessor starts looking for the file in the same directory as the file being compiled, then it looks in the system include directories. <> means that the preprocessor only looks in the system include directories. With g++, it is possible to determine which system include directories are searched.

- ⇒ Enter the command: g++ -v Hello.C
- ⇒ Look carefully at the output for something like:

```
#include <...> search starts here:
 /software/egcs+p-1.1/distribution/include/g++
 /software/egcs+p-1.1/distribution/sparc-sun-solaris2.6/include
 /software/egcs+p-1.1/distribution/lib/gcc-lib/sparc-sun-solaris2.6/egcs-2.91.55/include
 /usr/include
```

The list of UNIX path names are the directories in which the compiler searched for files.

The system include files limits.h and stddef.h contains many useful **#define**s, like the null pointer literal NULL.

- ⇒ Edit the file: /usr/include/limits.h
- ⇒ Look carefully at the file. While not all of the file may make sense, notice some of the useful **#define**s.

12.3 Conditional Textual Inclusion

The preprocessor has an **if** statement, which may be nested, to conditionally remove code from a program. The conditional of the **if** uses the same relational and logical operators as C/C++, but the operands can contain only integer or char values (no real or string values).

```
#define DEBUG 0
...
#if DEBUG == 1          // level 1 debugging
#   include "debug1.h"
    ...
#elif DEBUG == 2        // level 2 debugging
#   include "debug2.h"
    ...
#else                   // non-debugging code
    ...
#endif
```

By changing the value of the preprocessor variable DEBUG, different parts of the program can be excluded from compilation. The simplest way to exclude code (comment-out) is to have a 0 conditional because 0 implies false.

```
#if 0
    . . .                       // code commented out
#endif
```

It is also possible to check if a preprocessor variable is defined or not defined by using **#ifdef** or **#ifndef**, respectively:

```
#ifndef __MYDEFS_H__       // if not defined
#define __MYDEFS_H__ 1     // make it so
. . .
#endif
```

This technique is used in a **#include** file to ensure its contents are only expanded into a program once. Notice the difference between checking if a preprocessor variable is defined and checking the value of the variable. The former capability does not exist in most programming languages, i.e., checking if a variable is declared before trying to use it.

13 Input/Output

Like Pascal, two kinds of files can be accessed: data and text files. Data files contain data stored in machine representation, while text files contain data in human readable form. Since most I/O is to and from text files, only text files are discussed; see a C/C++ text book for I/O to data files. Furthermore, C provides one kind of I/O library and C++ provides another. While C++ can use both libraries, only the C++ library is discussed; see a textbook for the C library.

Pascal	C++
Input	cin
Output	cout
N/A	cerr
infile : Text	ifstream infile("*UNIX-file-name*")
outfile : Text	ofstream outfile("*UNIX-file-name*")
eof(infile)	infile.eof()
Reset(infile)	infile.seekg(0) *// move to 1st character*
Rewrite(outfile)	outfile.seekg(0) *// move to 1st character*
Read(infile, v1, v2, . . .);	infile >> v1 >> v2 >> . . . ;
Readln(infile, v1, v2, . . .);	N/A
Write(outfile, e1, e2, . . .);	outfile << e1 << e2 << . . .;
Writeln(outfile, e1, e2, . . .);	outfile << e1 << e2 << . . . << endl;

A text file is called a *stream file* in C++. Pascal has two implicit stream (text) files: Input and Output, which are automatically declared and opened. C++ has three implicit stream files: cin, cout and cerr, which are automatically declared and opened. As seen in all previous examples, the system include file iostream.h declares all necessary types and prototypes to use stream files. Like Pascal Input and Output, cin normally reads input from the keyboard (unless redirected from the shell), and cout writes to the terminal screen (unless redirected from the shell). In addition, stream cerr writes to the terminal screen even when cout output is being redirected from the shell. *Error and debugging messages should be written to* cerr *because it is normally not redirected by the shell, and furthermore, it is not buffered so output appears immediately.*

Like Pascal, all other stream files must be explicitly declared:

```
#include <iostream.h>     // implicit text files
#include <fstream.h>      // explicit text files
ifstream infile( "myinfile" );
ofstream outfile( "myoutfile" );
```

(The unusual declaration syntax infile(. . .) is explained in Section 15.) The include file fstream.h is necessary for explicitly accessing stream files. Like Pascal, each file is declared, and the declaration *opens* the file, making it accessible through the variable name, e.g., infile and outfile are used for file access. After declaration, it is possible to check for successful opening of a file using the stream routine bad, as in infile.bad(), which returns **true** if the open failed and **false** otherwise. (The unusual routine call infile.bad() and calls like it are explained in Section 15.) Unlike Pascal, the type of the file, ifstream or ofstream, indicates whether the file can be read or written; hence only reading occurs from infile and writing to outfile. Unlike Pascal, the connection between the file name in the program and the actual file name is done at the declaration (some extended Pascals use routine assign); hence infile reads from file myinfile and outfile writes to file myoutfile, where both files are located in the directory where the program is run.

13.1 Input

Other than syntax, the major semantic difference between Pascal and C++ input is how end-of-file is treated. In Pascal, there is an implicit pre-read when a file is opened (except for interactive input), which makes it possible to check for end-of-file *before* actually performing an input operation on a text file. In C/C++, there is no implicit pre-read, so end-of-file can be detected only *after* an input operation is attempted and encounters end-of-file.

Pascal	C++

while not Eof(Input) **do** begin
    Read(Input, i);
 	
    Writeln(Output, i)
end
	for (; ;) {
	    cin >> i; *// input fails when eof encountered*
	  if (infile.eof()) **break**;
	    cout << i << endl;
	}

Valid values in a stream file are C/C++ literals: 3, 3.5e-1, etc., separated by whitespace, except for characters and character strings, which are not in quotes. Unfortunately, this exception precludes reading strings containing white spaces (see Section 15.2 for reading entire lines). Like Pascal Read, the >> operator is overloaded to work with different types of operands. The type of the operand indicates the kind of literal expected in the stream file, e.g., an integer operand means an integer literal is expected. Like Pascal Read, cin starts reading where the last cin left off. When all the values on the current line have been read, cin proceeds to the next line.

When reading from the keyboard, a special indicator is required to cause the equivalent of end-of-file. This indicator takes the form of a special character, which is normally <ctrl>-d (press the <ctrl> and d keys at the same time on a line by itself). Note, this character is *not* an input character; it is not read by cin.

⇒ Edit hello.C

⇒ Enter the following program:

```
#include <iostream.h>
int main() {
    int count = 0, a, b;
    for ( ;; ) {
        cout << "enter 2 numbers: ";
        cin >> a >> b;
      if ( cin.eof() ) break;
        count += 1;
        cout << a << "+" << b << "=" << a + b << endl;
    }
    cout << endl << "you entered " << count << " pairs of numbers" << endl;
}
```

⇒ Compile and run the program, entering some integer literals.

⇒ End the input and the program by entering <ctrl>-d

13.2 Output

Other than syntax, the major semantic difference between Pascal and C++ output is how format codes are specified.

Pascal	C++

Writeln(
    d:10:2,
)
	cout <<
	    setw(10) << setprecision(2) << d <<
	    endl;

In Pascal, format codes are specified beside each variable name in the Write variable list. In C++, format codes are placed *before* the variable name in the output cascade, using special routines called *manipulators*, which apply to all I/O that come after it. The following manipulators are available by including iomanip.h:

oct	print values in octal
dec	print values in decimal
hex	print values in hexadecimal
setw(N)	print values in minimum of N columns
setprecision(N)	print fraction in maximum of N columns
setiosflags(F)	set flags that further control printing
resetiosflags(F)	reset flags that further control printing
endl	flush current output buffer and start a new line

Two useful flags (F) are ios::fixed and ios::scientific, which control whether real numbers are printed with or without exponents; ios::fixed is the default. (The unusual syntax ios::fixed is explained in Section 15.)

⇒ Edit hello.C
⇒ Enter the following program:

```
#include <iostream.h>
#include <iomanip.h>
int main() {
    int i, j;
    cout << setiosflags( ios::fixed );          // no exponent for subsequent output
    for ( i = 1; i <= 10; i += 1 ) {
        for ( j = 1; j <= 4; j += 1 ) {
            cout << "(" << setw(2) << i << "," << j << "):"
                << setw(i) << setprecision(j) << 123.789 << endl;
        }
    }
}
```

⇒ Compile and run the program.
⇒ Change ios::fixed to ios::scientific, and compile and run the program.

14 Dynamic Storage Management

Like Pascal, there are two dynamic storage management operations to allocate and free storage. C provides one kind of dynamic storage management and C++ provides another. While C++ can use both kinds, only the C++ form is discussed; see a textbook for the C form.

Pascal	C++
type info = record	**struct** info {
a, b, c : Char;	**char** a, b, c;
end;	};
var infop : ^info;	info *infop;
New(infop);	infop = **new** info;
infop^.c = "R";	infop->c = 'R';
Dispose(infop);	**delete** infop;

In Pascal, allocation is performed by the pseudo-routine New, which takes a pointer argument, infers from the pointer's type the amount of storage, and returns in the argument a pointer to the new storage. In C++, allocation is performed by the pseudo-operator **new**, which takes a type operand and returns a pointer to new storage of that type. Disposing or deleting of allocated storage is done using routine Dispose in Pascal and the operator **delete** in C++. As in Pascal, after storage is deallocated, the storage can no longer be used.

```
delete infop;
infop->c = 'R';          // result of dereference is undefined
```

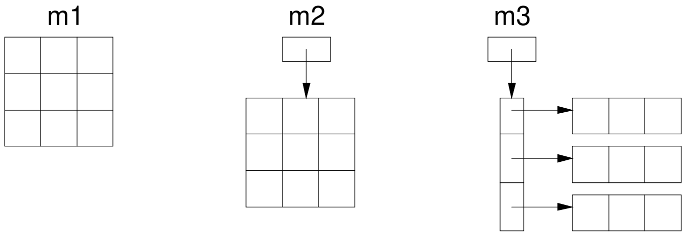
Dynamic array allocation is complex in C++. The first dimension of a multidimensional array can be specified at run-time; the rest must be given as compile-time constants so the compiler can determine the stride amount for subscripting.

```
int r;
int *vec = new int [10];             // think: int vec[ ]
cin >> r;                            // read a row dimension
int (*pmr6)[6] = new int [r][6];     // think: int pmr6[ ][6]
int *((*px)) = new (int *([r]));     // think int *(px[ ])
int (*(*py))[5] = new (int (*)[5]);
```

The last two dynamic allocations require parenthesis around the entire **new** operand to avoid parsing problems.

⇒ Explain what the last two dynamic allocations do. (Draw a picture for px and py like that on page 11.)

Because C needs to statically know the size of all dimensions but the first, dynamically declared multi-dimensional array are impossible. Here are three options for a dynamically declared multi-dimensional array, but all depend on g++ dynamic array capabilities. The storage layout for each option and the code to create it follows:




```
void fred( int row, int col ) {
    double m1[row][col];                    // g++ only, matrix of double on the stack
    double (*m2)[col]=new double[row][col]; // g++ only, pointer to matrix of double in the heap
    double **m3 = new (double *[row]);      // g++ only, pointer to array of pointers to array of double in the heap
    for ( int r = 0; r < row; r += 1 ) {
        m3[r] = new double[col];            // rows of doubles
    } // for

    for ( int r = 0; r < row; r += 1 ) {
        for ( int c = 0; c < col; c += 1 ) {
            m1[r][c] = r + c;
            cout << "m1[" << r << "][" << c << "]:" << m1[r][c] << "  ";
            m2[r][c] = r + c;
            cout << "m2[" << r << "][" << c << "]:" << m2[r][c] << "  ";
            m3[r][c] = r + c;
            cout << "m3[" << r << "][" << c << "]:" << m3[r][c] << endl;
        } // for
    } // for

    delete [ ] m2;                          // delete matrix of double in heap
    for ( int r = 0; r < row; r += 1 ) {    // delete rows
        delete m3[r];
    } // for
    delete [ ] m3;                          // delete column
}
```

The first option uses dynamic-length automatic arrays, but with variable dimension values to allocate a matrix, m1, on the stack. The second option uses the same technique for dynamically allocating a matrix, m2, on the heap. In both these options, the amount of storage allocated is $N \times M$ elements. The third option creates an array of pointers for each row of the matrix and then creates an array of elements for each row of the matrix. The resulting data structure is not a matrix but can be subscripted identically to one. The amount of storage allocated is N pointers $+ N \times M$ elements.

Because an array can be declared in a form that hides the dimension information, e.g., **int** *vec, deallocation has to be told explicitly if a pointer is to a single element or an array of elements (the number of dimensions is not required):

```
delete [ ] vec;  delete [ ] pmr6;  delete [ ] pmrc;     // indicate array pointers
```

Consider the following:

```
info *infov = new info [10];        // think: info infov[ ]
infov[2].b = 'W';                   // why . and not ->
```

To understand why the member selection operator is used and not the pointer dereference operator, it is necessary to rewrite the subscript using pointer arithmetic, (*(infov+2)).b, which means the dereference has already occurred because of the subscript, and hence, operator -> cannot be used.

⇒ Edit hello.C

⇒ Enter the following program:

```
#include <iostream.h>
int main() {
    int i, size;
    cin >> size;            // read array dimension
    int vals[size];         // g++ only
    for ( i = 0; i < size; i += 1 ) {      // read values
        cin >> vals[i];
    }
    for ( i = size - 1; 0 <= i; i -= 1 ) {     // print values in reverse
        cout << vals[i] << " ";
    }
    cout << endl;
}
```

which reads a set of data values of the form:

```
5 0 1 2 3 4
```

where the first value (5) indicates the number of values in the set (0, 1, 2, 3, 4). The program then prints the values out in reverse order from that read in: 4 3 2 1 0

⇒ Compile and test the program.

⇒ Change the program to dynamically allocate and free the array instead of using the g++ variable dimension size.

⇒ Change the program to read in a row and column size and dynamically allocate a matrix; write the program first using automatic and then dynamically allocation of the matrix.

15 ⋈ Objects

Objects and object-oriented programming are not available in standard Pascal, so comparison is impossible in this section (some extended Pascals provide limited objects). Object-oriented programming is not a new programming methodology; it was developed in the mid-1960s by Dahl and Nygaard. Objects extend the notion of a record/structure, available in most programming languages. A structure is a mechanism for organizing logically related data, e.g.:

<table>
<tr><th>unorganized</th><th>organized</th></tr>
<tr><td><pre>int people_age[30];
bool people_sex[30];
char people_name[30][50];</pre></td><td><pre>struct person {
    int age;
    bool sex;
    char name[50];
} people[30];</pre></td></tr>
</table>

Notice that both code fragments create an identical amount of information; the difference is solely in the way the information is organized. In essence, a structure is irrelevant from the computer's perspective because the information and its manipulation is largely the same. Nevertheless, a structure is an important administrative tool for helping programmers organize information for easier understanding and convenient manipulation in the programming language.

The organizational capabilities of the structure can be extended by allowing routine members, instances of such a structure are *objects*. Hence, the idea of associating routines with structures is the basis of objects. The power behind objects is that each object provides both data and the operations necessary to manipulate that data in one self-contained package. (Unfortunately, this approach also has a number of limitations, which are not discussed.) Note, a routine member is a constant, and hence, cannot be assigned to (e.g., like a **const** member). The following compares the structure to the object form for representing complex numbers:

structure form	object form

struct complex {
    double re, im;
};
double abs(complex *****this**) {
    return sqrt(**this**->re * **this**->re + **this**->im * **this**->im);
}
complex x; *// structure*
abs(x); *// call abs*
	struct complex {
	    double re, im;
	    double abs() {
	        return sqrt(re * re + im * im);
	    }
	};
	complex x; *// object*
	x.abs(); *// call abs*

Structure complex, on the right, now generates objects because it has a routine member, abs, which calculates the absolute value of a complex number (distance from the origin).

What is the scope of a routine defined in a structure, i.e., what variables can a routine member reference? A normal routine's scope is the static contexts (blocks) containing it (only one level in C/C++ because routines cannot be nested). Interestingly, a structure also creates a static context, and therefore, a routine member can access the structure members and the static context in which the structure is defined. In other words, scope rules allow the body of abs, in the right example, to refer to members re and im, plus any other members in enclosing scopes (nested structures). A simple model for understanding this static context is that each routine member is implicitly pulled out of the structure and rewritten as a *free routine* (i.e., not bound in an object) to take the structure as an explicit parameter, as in the left example above. As well, all implicit references to members of the structure are rewritten to explicit references to members of the parameter, as in the body of abs on the left. In fact, C++ provides this implicit parameter through the keyword **this**, which is available in each routine member.

How is abs called? Normally a routine is invoked like abs(x). However, because abs is a member in a structure, it must be accessed like other members, using member selection: x.abs(). This form of routine call is one of the first peculiarities of objects, and has been used already with file objects, e.g., cin.eof(). The next question is why abs has no arguments in the call; where does abs get a parameter to calculate a result? The answer is the static context of the structure in which it is nested (or the implicit parameter). abs can make references to variables re and im by virtue of the fact that it is nested in structure complex. Hence, the call x.abs() is invoked in the context of object x, so members re and im of x are accessed in abs. This form of supplying parameters to a routine is the second peculiarity of objects. Once these two peculiarities are mastered, objects are straightforward to use and understand.

⇒ Edit complex.C **(Note the name change for the source file.)**

⇒ Enter the following program:

```
#include <iostream.h>
#include <math.h>     // needed to use routine sqrt (square root)
struct complex {
    double re, im;    // real and imaginary Cartesian coordinates
    double abs() { return sqrt( re * re + im * im ); }
};
int main() {
    complex x = { 3.0, 5.2 }, y = { -9.1, 7.4 };
    cout << "x:" << x.re << "+" << x.im << "i" << endl;
    cout << "y:" << y.re << "+" << y.im << "i" << endl;
    cout << "xd:" << x.abs() << endl;
    cout << "yd:" << y.abs() << endl;
}
```

⇒ Compile and run the program.

⇒ Change routine abs to be:

```
double abs() { return sqrt( this->re * this->re + this->im * this->im ); }
```

⇒ Compile and run the program.

The change to abs illustrates the hidden parameter to all routine members and the fact that the type of the implicit parameter **this** is a pointer to the structure instance, requiring operator -> to access values in the fields. Use of the implicit parameter **this** is rare in C++.

The type complex needs additional arithmetic operations, like addition:

```
struct complex {
    double re, im;
    double abs() { return sqrt( re * re + im * im ); }
    complex add( complex c ) {
        complex sum = { re + c.re, im + c.im };
        return sum;
    }
};
```

To sum x and y, write x.add(y). Because addition is a binary operation, add needs a parameter as well as the implicit context in which it executes. In add, the members of the implicit operand, x, are added to the explicit ones of the parameter, y, and a new complex value is returned.

Unfortunately, the syntax for adding complex values does not look like adding integers or reals, where the builtin operator + is used. In C++, it is possible to use operator symbols for routine names:

```
struct complex {
    double re, im;
    . . .
    complex operator+( complex c ) {
        complex sum = { re + c.re, im + c.im };
        return sum;
    }
};
```

The addition routine is now called +, and x and y can be added by x.**operator**+(y) or y.**operator**+(x), which is only slightly better. In fact, C++ allows all operators to be written using prefix or infix notation, and rewrites the notation back to member selection notation if necessary; thus, x + y is allowed and rewritten as x.**operator**+(y).

⇒ Edit complex.C

⇒ Make the following modifications:

```
#include <iostream.h>
#include <math.h>
struct complex {
    double re, im;
    double abs() { return sqrt( re * re + im * im ); }
    complex operator+( complex c ) {
        complex sum = { re + c.re, im + c.im };
        return sum;
    }
};
int main() {
    complex x = { 3.0, 5.2 }, y = { -9.1, 7.4 };
    cout << "x:" << x.re << "+" << x.im << "i" << endl;
    cout << "y:" << y.re << "+" << y.im << "i" << endl;
    complex sum = x + y;
    cout << "sum:" << sum.re << "+" << sum.im << "i" << endl;
}
```

⇒ Compile and run the program.

As mentioned in Section 11.3, the declaration-before-use rule causes problems with mutually recursive references. This problem also occurs with routines in objects, and is handled in a similar way using prototypes.

```
struct info {
    . . .
    void f() { g(); }     // g is not defined
    void g() { f(); }
};
```

```
struct info {
    . . .
    void f();     // prototype
    void g();
};
void info::f() { g(); }     // declaration outside of object
void info::g() { f(); }
```

The declaration for the routines f and g are replaced with prototypes in the object, and the declarations are moved *outside* the object declaration. Unfortunately, having the routine declarations in the object along with the prototypes is disallowed. (This restriction is not a problem because most prototypes are used for organizing routine declarations in a source file and for separate compilation, see Section 19.1, which require routine declarations outside the object type.) The special qualifier “info::” indicates the routine is not a free routine but a member of object type info. The member selection operator “.”, e.g., **void** info.f() { . . . }, is inappropriate because it requires an object instance not an object type.

⇒ Edit hello.C
⇒ Enter the following program:

```
#include <iostream.h>
struct info {
    void f( int i );            // prototype
    void g( int i );
};
void info::f( int i ) {
    cout << "f(" << i << ")" << endl;
    if ( i > 0 ) g( i - 1 );
}
void info::g( int i ) {
    cout << "g(" << i << ")" << endl;
    if ( i > 0 ) f( i - 1 );
}
int main() {
    info x;
    x.f( 5 );  cout << endl;  x.g( 4 );
}
```

⇒ Compile the program, run it, and check the output.

This qualifier syntax can be used to reference type declarations inside an object, such as:

```
struct info {
    enum Colour { R, G, B };      // nested type declaration
    . . .
};
info::Colour colour = info::R;
```

The enumeration Colour is nested in info, and references to it outside the object must be qualified with the object type, like ios::fixed from Section 13.2.

Finally, C++ supports syntactic nesting of object types (largely for encapsulation, see Section 19), *but the nesting does not imply scoping*:

```
struct fred {
    int i;
    int r(. . .) { . . . }
    struct mary {                           // nested object type
        int s(. . .) { i = 3; r(. . .); }   // references to i and r fail
    };
};
fred f
fred::mary m;
```

In effect, C++ flattens the scoping structure. As a result, the references in mary::s to members i and r in fred fail because there is no scope relationship between types mary and fred! Because nested syntax is allowed but there is no scoping, it is confusing and discouraged. This restriction results from the basic lack of nested routines because member routines in nested classes have the same scope requirements as nested routines.

15.1 Constructor/Destructor

C++ subdivides traditional assignment into two subtly different but important concepts: *initialization*, which initializes an object immediately after allocation, hence an object contains no value(s) prior to initialization, e.g., **int** x = 3, and *assignment*, which is assignment to a (usually) initialized object, e.g., x = 5. In initialization, most (all) of the object members are given values, whereas, in assignment only some of the members may be copied from one object to another. For example, if an object type has a member variable to count the number of times each instance is assigned, the counter is set to zero on initialization, but not copied on assignment only incremented. In most languages, assignment means copy all the "bits" from one variable to another, which is also the default behaviour in C++; assignment in C++ also allows selective copying of the "bits". This section discusses initialization (and uninitialization); see a C++ textbook for a discussion of the assignment operator.

As stated, initialization is defined to occur immediately *after* an object is allocated when its members contain undefined values. Places where this occurs are a declaration of a variable, creation of a parameter for a routine call, and dynamic allocation of a variable. C++ also provides a companion capability, uninitialization, which is defined to occur immediately *before* an object is deallocated, either implicitly at the end of a block or explicitly by a **delete**. Having both capabilities allows an object designer to ensure pre- and post-conditions for an object.

A constructor is a member routine that is implicitly called *after* an object is allocated to perform initialization. When the body of the constructor begins execution, all the object members contain *undefined* values. A destructor is a member routine that is implicitly called *before* an object is deallocated. When the body of the destructor begins execution, all the object members (usually) have *defined* values, and the destructor uninitializes them. For simple objects, there is usually nothing to uninitialize; for advanced objects, there may be files or communication channels to close, and/or dynamically allocated storage to free. The name of a constructor/destructor is unusual because it is overloaded with the type name of the structure the routines are defined in, neither constructor or destructor has a return type (not even **void**), and the destructor name is preceded by the character "~".

```
struct complex {
    double re, im;
    complex() { re = 0.; im = 0.; }      // constructor (default, no parameters)
    ~complex() { }                       // destructor
    . . .
};
```

Because a constructor and destructor are routines, arbitrary execution can be performed (e.g., loops, routine calls, etc.) after allocation and before deallocation. For complex, the constructor ensures that both members are initialized to zero, while the destructor does nothing (same as not having one).

All allocations and deallocations of variables implicitly call constructors after the storage is allocated and destructors before storage is deallocated.

{ // allocate local storage
    complex x;
    . . .
 	
} // deallocate local storage
	implicitly rewritten as	{
		    complex x; x.complex();
		    . . .
		    x.~complex();
		}

This rewrite occurs for each local variable in a block, with destructors called in reverse order to constructors. Similarly, for dynamically allocated variables:

complex *x = **new** complex;
. . .
delete x;
	implicitly rewritten as	complex *x = **new** complex; x->complex();
		. . .
		x->~complex(); **delete** x;

Once a constructor is specified, the old style structure initialization is disallowed:

```
complex x = { 3.2, 4.5 };         // not allowed
```

Instead, this form of initialization is replaced with overloaded constructors with parameters:

```
struct complex {
    double re, im;
    complex() { re = 0.; im = 0.; }
    complex(double r) { re = r; im = 0.; }
    complex(double r, double i) { re = r; im = i; }
    ~complex() { }
    . . .
};
```

What is unusual about constructors with parameters is that arguments are specified *after* the variable in a declaration.

```
complex x, y(1.0), z(6.1, 7.2);   implicitly rewritten as   complex x; x.complex();
                                                            complex y; y.complex(1.0);
                                                            complex z; z.complex(6.1, 7.2);
```

This syntax is used for declaring text files, e.g., ifstream infile("myinfile"). Note, never explicitly invoke the default constructor in a declaration, as in:

```
complex x(); // attempt to call default constructor
```

This declaration actually defines a routine prototype called x taking no parameters and returning a complex value.

A special constructor, called the *copy constructor*, with a **const** reference parameter to the object type, e.g.:

```
complex( const complex &c ) { ... }
```

is used in two important contexts for initialization: declaration and parameter assignment.

```
complex x(3.2);   implicitly rewritten as   complex x; x.complex(3.2)
complex y = x     implicitly rewritten as   complex y; y.complex(x);    // copy constructor
```

Notice the difference in the two initialization cases: the first is an explicit constructor call to initialize x, while the second uses the operator "=" to perform declaration initialization (see Section 8.5), which is implicitly rewritten as an implicit call to the copy constructor. As well, for each complex argument passed by value to a complex parameter, the copy constructor is called to initialize the parameter with the argument value. For example, in:

```
void r( complex a, complex b ) { ... }
r( x, y );      // copy constructor called twice for each value parameter
```

the copy constructor is invoked twice, a.complex(x) and b.complex(y), to initialize both parameters, before r begins execution.

Constructors can also be used to create object literals, like g++ type constructor literals in Section 9.2.

```
complex x, y, z;
x = complex( 3.2 );               // create complex literal with value 3.2+0.0i
y = x + complex( 1.3, 7.2 );      // create complex literal with value 1.3+7.2i
z = complex( 2 );                 // 2 widened to 2.0, create complex literal with value 2.0+0.0i
```

In fact, operator + for complex must now be rewritten to use a complex constructor to create the return value because type constructor literals are not allowed for objects.

```
complex operator+( complex c ) {
    return complex( re + c.re, im + c.im );   // use constructor to create new complex value
}
```

Finally, constructors are used automatically to perform implicit conversions:

```
complex x, y;
x = 3.2;          implicitly rewritten as   x = complex(3.2);
y = x + 1.3;      implicitly rewritten as   y = x.operator+( complex(1.3) );
```

Unfortunately, the commutative form, 3.0 + x, fails because it is conceptually rewritten as (3.0).**operator**+(x), and there is no method **double operator**+(complex) in the builtin type **double**. To solve this problem requires moving the operator + out of the object and making it an overloaded free routine called +, which can be written in infixed form:

```
struct complex { ... };     // same as before, except operator + removed
complex operator+( complex a, complex b ) {   // overloaded free routine
    return complex( a.re + b.re, a.im + b.im );
}
x + y;        // implicitly rewritten as: +(x, y)
3.0 + x;      // implicitly rewritten as: +(complex(3.0), x)
x + 3.0;      // implicitly rewritten as: +(x, complex(3.0) )
```

The compiler first checks for an appropriate operator defined in the object, and then for an appropriate free routine (it is ambiguous to have both); it then applies applicable conversions. In general, binary operators should be written as overloaded free routines to allow implicit conversion on both operands, not just the second operand. Another example of overloaded free routines is the I/O operators << and >>, which are overloaded for reading and printing.

```
ostream &operator<<( ostream &os, complex c ) {
    os << c.re << "+" << c.im << "i";
    return os;
}
cout << "x:" << x;   // implicitly rewritten as: <<( cout.operator<<("x:"), x )
```

In this case, the compiler uses the << operator in object cout to print a string value, but used the overloaded free routine << to print the complex variable x. There is a standard convention for all I/O operators to take and return a stream reference to allow cascading with other stream operators.

⇒ Edit complex.C

⇒ Make the following modifications:

```
#include <iostream.h>
#include <math.h>
struct complex {
    double re, im;
    double abs() { return sqrt( re * re + im * im ); }
    complex() { re = 0.; im = 0.; }
    complex(double r) { re = r; im = 0.; }
    complex(double r, double i) { re = r; im = i; }
    ~complex() { }
};
complex operator+( complex a, complex b ) {        // overloaded free routine
    return complex( a.re + b.re, a.im + b.im );
}
ostream &operator<<( ostream &os, complex c ) {   // overloaded free routine
    os << c.re << "+" << c.im << "i";
    return os;
}
int main() {
    complex x, y, z;
    x = complex( 3.2 );
    y = x + complex( 1.3, 7.2 );
    z = complex( 2 );
    cout << "x:" << x << " y:" << y << " z:" << z << endl;
}
```

⇒ Compile the program, run it, and check the output.

Finally, there is a problem with initializing **const** members of an object:

```
struct info {
    const int i;  . . .
} x = { 3 };        // first member must be initialized because it is write-once/read-only
```

As mentioned, this form of initialization is disallowed for objects, and must be replaced with a constructor:

```
struct info {
    const int i;  . . .
    info() { i = 3; }
};
```

However, this fails because it is assignment not initialization, and a **const** variable can only be initialized, otherwise the compiler cannot ensure a read does not occur before the initial write. Therefore, a special syntax is used for initializing **const** members of an object *before* the constructor is executed:

```
struct info {
    const int i;
    int j, *const cp, &rp;
    info( int x ) : i(3), cp(&j), rp(j) {          // special syntax for initializing const members
        j = x;    // initialize non-const members
    }
};
```

In the example, member i is initialized to 3, and cp and rp are initialized to point at member j. In fact, this syntax can be used to initialize any variable in a class, hence non-**const** members may be initialized either way.

```
info( int x ) : i(3), cp(&j), rp(j), j(x) { }          // initialize j with special syntax
```

15.2 ⋈ String

Strings are supported in C by a combination of language and library facilities. The language facility ensures that all string literals are terminated with a character value '\0' or zero. For example, the string literal "abc" is actually an array of the 4 characters: 'a', 'b', 'c', and '\0', which occupies 4 bytes of storage. The zero value at the end of a string literal conforms to a convention used by the C string routines. This convention allows the C string routines to locate the end of a character string value by checking through the individual character values for '\0'. Unfortunately, this convention suffers from three important drawbacks. First, a string cannot contain a character with the value '\0' as that character immediately marks the end of the string. Second, many string operations need the length of a string, and finding the length of a string requires a linear search for the character '\0', which is an expensive operation for long strings. Third, the management of variable sized strings is the programmer's responsibility, making complex string operations a storage management nightmare.

To deal with these drawbacks, C++ provides a string object type that uses a length member at the beginning of each string, and manages all of the variable sized storage for the variable sized strings. The string object type uses all the ideas presented in the previous discussion of objects in its implementation, and is part of the C++ STL. While C++ can use both C and C++ strings, only C++ strings are discussed; see a textbook for C strings.

Pascal (extended)	C	C++
String	**char** [ ]	string
:=	strcpy, strncpy	=
Concat	strcat, strncat	+
=, <>, <, <=, >, >=	strcmp, strncmp	==, !=, <, <=, >, >=
Length	strlen	length
[ ]	[ ]	[ ]
Copy	strstr	substr
Insert	N/A	replace
Pos	strchr	find

In general, a string type has powerful operations to manipulate the characters of the string and search through them. The most important point to remember about a string value is that it can vary in length dynamically from 0 to 65535 characters, e.g:

```
string a, b, c;                   // declare string variables
a = "abc";                        // set value
b = a;                            // copy value
c = a + b;                        // concatenate strings, c is "abcabc"
if ( a == b )                     // compare strings, lexigraphical ordering
    cout << c << endl;            // print string
int l = c.length();               // string length, l is 6
char ch = c[4];                   // subscript, ch is 'b', zero origin
c[4] = 'x';                       // subscript, c is "abcaxc"
string d = c.substr(2,3);         // extract starting at position 2 (zero origin) for length 3, d is "cax"
c.replace(2,1,d);                 // replace starting at position 2 for length 1 and insert d, c is "abcaxaxc"
int p = c.find("xc");             // search for 1st occurrence of string "xc" in c, p is 6
p = c.find_first_of("cx");        // search for first character 'c' or 'x', p is 2
p = c.find_first_not_of("abc");   // search for first character not 'a', 'b' or 'c', p is 4
cin >> c;                         // read white-space delimited sequence of characters
getline(cin, c, '\n');            // read characters until newline (newline is the default)
```

⇒ Edit hello.C
⇒ Enter the following program:

```
#include <iostream.h>
#include <iomanip.h>
#include <string>
int main() {
    string line, word;
    int p, words = 0;
    cin >> resetiosflags( ios::skipws );          // turn off white space skipping during input
    for ( ;; ) {                                   // scan lines from a file
        getline( cin, line );                      // read entire line, but not newline
      if ( cin.eof() ) break;                      // end-of-file ?
        line += "\n";                              // add newline character as sentinel character
        for ( ;; ) {                               // scan words off line
            p =line.find_first_not_of(" \t\n");    // find position of 1st non-whitespace character
          if ( p == -1 ) break;                    // any characters left ?
            line = line.substr( p );               // remove leading whitespace
            p = line.find_first_of(" \t\n");       // find position of 1st whitespace character
            word = line.substr( 0, p );            // extract word from start of line
            words += 1;                            // count word
            line = line.substr( p );               // delete word from line
        } // for
    } // for
    cout << "words: " << words << endl;
}
```

⇒ Examine the program to determine what it does.
⇒ Compile the program and run it with the command: ./a.out < hello.C
⇒ Check the results with the command: wc hello.C

16 Inheritance

The "oriented" part of object-oriented refers to an additional notion called inheritance, which is useful for writing general, reusable program components. Inheritance encompasses two orthogonal concepts: implementation and type sharing. Each concept is discussed separately.

16.1 Implementation Inheritance

Implementation inheritance allows one object to reuse existing declarations to build another object. One way to understand this technique is to model it via explicit inclusion, e.g.:

Inheritance	Inclusion
struct base {	**struct** base {
int i;	**int** i;
int r(...) { ... }	**int** r(...) { ... }
base() { ... }	base() { ... }
};	};
struct derived : **public** base { *// implicit inclusion*	**struct** derived {
	base b; *// explicit inclusion*
int s(...) { i = 3; r(...); ... }	**int** s(...) { b.i = 3; b.r(...); ... }
derived() { ... }	derived() { ... }
} d;	} d;
d.i = 3; *// reference member in included member*	d.b.i = 3;
d.r(...); *// reference member in included member*	d.b.r(...);
d.s(...); *// s can access i and r in included member*	d.s(...)

Inheritance implicitly creates an anonymous object member and "opens" the scope of the anonymous member so that its members are accessible without qualification, both inside and outside the inheriting object type, similar to the with statement in Pascal. The inclusion model analogy involves explicit creation of an object member, b, that aids in the implementation. In the example, object type derived states that it is including a base object, via the "**public** base" clause, which is opened in the context of derived.

For implementation inheritance to work, a derived declaration first implicitly creates an invisible base object in a derived object, like the explicitly created member in the inclusion model, otherwise the implicit references to base::i and base::r in derived::s would fail. As well, constructors and destructors must be invoked for all implicitly declared objects in the inheritance hierarchy as would be done for an explicit member in the inclusion model.

derived d;	implicitly rewritten as	base b; // implicit, hidden declaration
...		derived d; b.base(); d.derived();
		...
		d.~derived();b.~base(); // reverse order of construction

In the case where the included object type has members with the same name as the including type, it works like nested Pascal with clauses: a name in the inner scope hides (overrides) a name at the outer scope. However, it is still possible to access these members by using "::" qualification (see Section 15) to specify the particular nesting level that contains the member.

```
struct base {
    int i;
    int r() { }
};
struct derived : public base {
    int i;              // hides base::i
    int r() { }         // hides base::r
    int s() {
        i = 3;          // derived::i
        r();            // derived::r
        base::i = 3;    // qualify scope
        base::r();      // qualify scope
    }
};
```

This qualification can also be used in a routine member to access a hidden name in the containing object type:

```
struct derived : public base {
    int i;
    derived( int i ) {
        derived::i = i      // copy from local to object scope
    }
    ...
};
```

⇒ Edit hello.C

⇒ Enter the following program:

```
#include <iostream.h>
struct base {
    int i;
    base() { i = 1; cout << "base::base" << endl; }
    ~base() { cout << "base::~base, i:" << i << endl; }
    int r() { cout << "base::r, i:" << i << endl; }
};
```

```
struct derived : public base {
    int i;                  // hides base::i
    derived() { i = 10; cout << "derived::derived" << endl; }
    ~derived() { cout << "derived::~derived, i:" << i << endl; }
    int r() { cout << "derived::r, i:" << i << endl; }
    int s() {
        i = 3;              // derived::i
        cout << "derived::s, i:" << i << endl;
        r();                // derived::r
        base::i = 3;        // qualify scope
        base::r();          // qualify scope
    }
};
int main() {
    base b;
    b.r();
    derived d;
    d.r();
    d.s();
}
```

⇒ Compile the program and run it.
⇒ Check the output and make sure you understand it.

What happens if the included object requires an argument for its constructor?

Inheritance	Inclusion

`struct base {`
`    base( int i ) { ... }          // requires argument`
`};`
`struct derived : public base(3) {   // argument not allowed!`
 	
`} d;`
	`struct base {`
	`    base( int i ) { ... }`
	`};`
	`struct derived {`
	`    base b(3);   // explicit inclusion`
	`} d;`

C++ does not allow the argument to the inherited object type to appear as part of the inheritance specification. Instead, the argument must appear with each constructor using the syntax for initializing **const** variables (see Section 15.1).

```
struct base {
    base( int i ) { ... }                // requires argument
};
struct derived : public base {
    derived() : base(3) { ... }          // argument for base type
    derived( int i ) : base(i) { ... }   // argument for base type
} d;
```

How is implementation inheritance used to write reusable program components? By composing a new object's implementation from an existing object, it is possible to take advantage of previously written and tested code, which can substantially reduce the time to compose and debug a new object. Unfortunately, having to inherit all of the members of an object is not always desirable; some members are not appropriate for the new type. As a result, both the inherited and inheriting object must be very similar to have so much common code. (In general, free routines provide smaller units for reuse than entire objects.)

16.2 Type Inheritance

Type inheritance takes advantage of the implicitly inherited members and routines. Clearly, the type of the inheriting object is *derived* from the inherited base type because it has all the members of the base type, plus any it adds or overrides. Type inheritance means that objects of the derived type can behave just like objects of the base type.

```
struct fred {                 struct mary {
    int i;                        int i;
    char c;                       char c;
    double d                      double d;
} f;                          } m;
void r( fred f ) { . . . }
r( f );    // valid call
r( m );    // invalid call
```

In this example, it is clear that the type of variable f is identical to the type of m, and therefore, it is safe to use m in any context that allows f. However, f and m have different types because C++, like Pascal, uses name equivalence to compare types; therefore, m cannot be an argument to routine r, even though it structure is identical to f. Type inheritance relaxes strict name equivalence so that a variable's type matches if its type *or any of its base types* match:

```
struct fred {                 struct mary : public fred {
    int i;                        // no members
    char c;
    double d
} f;                          } m;
void r( fred f ) { . . . }
r( f );    // valid call
r( m );    // valid call because of inheritance
```

For example, to count the number of times abs is called for each complex object, a specialized version of complex is created with a modified abs member:

```
struct mycomplex : public complex {
    int cntCalls;
    mycomplex() : cntCalls(0) {}
    double abs() { cntCalls += 1; return complex::abs(); }    // reuse complex's abs routine
    int calls() { return cntCalls; }
};
```

⇒ Explain why the qualification complex:: is necessary in mycomplex::abs.

The idea is to change only the type of the complex variables to mycomplex, and then invoke member-routine calls for each object to obtain the number of calls to abs.

⇒ Edit complex.C
⇒ Make the following modifications:

```
. . . // same as before until the end of the complex output operator
struct mycomplex : public complex {
    int cntCalls;
    mycomplex() : cntCalls(0) {}
    double abs() { cntCalls += 1; return complex::abs(); }
    int absCalls() { return cntCalls; }
};
int main() {
    mycomplex x, y, z;
    cout << "x:" << x.abs() << " y:" << y.abs() << " z:" << z.abs() << endl;
    cout << "x:" << x.absCalls() << " y:" << y.absCalls() << " z:" << z.absCalls() << endl;
}
```

⇒ Compile the program and run it.

The key point, with regard to type inheritance, is the use of the complex output operation, <<, to print mycomplex values. The complex free routine << can be used because of the relaxed type equivalence provided by type inheritance.

How is type inheritance used to write reusable program components? While implementation inheritance provides reuse *inside* the object type, type inheritance provides reuse *outside* the object type by taking advantage of existing code that manipulates the base type. In other words, any routine (member or free) that manipulates the base type also manipulates the derived type.

Finally, the previous example can be used to illustrate two important problems with type inheritance. The first problem is illustrated by:

⇒ Edit complex.C
⇒ Make the following modifications:

```
. . . // same as before until the end of the declaration of mycomplex
int main() {
    mycomplex x;
    x = x + x;
}
```

⇒ Compile the program and read the message from the compiler.

Like the previous example, the complex free routine **operator**+ is used to add the mycomplex values because of the relaxed type equivalence provided by type inheritance. However, the result type from **operator**+ is complex, not mycomplex. Now, it is impossible to assign a complex (base type) to mycomplex (derived type) because the complex value is missing the cntCalls member! In other words, a mycomplex can mimic a complex but not vice versa. This fundamental problem of type inheritance is called *contra-variance*; C++ provides various solutions, all of which have problems and all beyond the level of this tutorial.

The second problem is illustrated by:

⇒ Edit complex.C
⇒ Make the following modifications:

```
. . . // same as before until the end of the declaration of mycomplex
void fred( complex x ) {
    x.abs();
}
int main() {
    mycomplex x;
    x.abs();        // direct call of abs
    fred( x );      // indirect call of abs
    cout << "x:" << x.absCalls() << endl;
}
```

⇒ Compile the program, run it, and check the output.

While there are two calls to abs on object x, only one is counted. This peculiarity is resolved in the next section.

16.3 Virtual Routines

In general, when a routine member is called, it is obvious which routine is invoked:

```
struct base {
    void r() { . . . }
};
struct derived : public base {
    void r() { . . . }
};
base b; derived d;
b.r();    // call base::r
d.r();    // call derived::r
```

However, it is not obvious for pointers/references to objects:

```
base &bp = d;   // assignment allowed because of inheritance
bp.r();         // call base::r or derived::r ?
```

In essence, inheritance masks the actual type of the object the pointer references. In general, the call bp.r() should invoke derived::r because bp is currently pointing at an object of type derived; likewise, if bp is pointing at an object of type base, the call should invoke base::r. However, there are situations where, regardless of what bp is pointing at, the programmer wants base::r called because bp is declared as a pointer to a base object. Notice, this is never a problem, even if bp is pointing at a derived object, because every derived contains a base. The latter situation is rare, but useful.

To handle both cases, C++ provides a facility to specify the default form for a member routine call, and to override the default at the call site. The routine member qualifier **virtual** indicated that a call to this member through a pointer invokes the routine defined in the referenced object. If there is no **virtual** qualifier, a call to this member always invokes this member, even through a pointer referencing a derived object. Therefore, the object type designer decides ahead of time how each call to a member routine behaves.

```
struct base {
    int r(int);                 // NO virtual qualifier
    virtual int s(int);         // virtual qualifier
};
struct derived : public base {
    int r(int);                 // NO virtual qualifier
    int s(int);                 // NO virtual qualifier
};
base &bp = *new derived;        // assignment allowed because of inheritance
bp.r(3);                        // call base.r, because r in "base" is NOT virtual
bp.s(3);                        // call derived.s, because s in "base" is virtual
delete &bp;                     // destructor must be virtual
```

Notice, virtual members are *only* necessary to access derived members through a base type reference or pointer. The problem with **virtual** is that it is made in the base class as opposed to the derived class (Modula-3 is the opposite). Hence, C++ requires the base-class definer to look into the future and guess how derived definers might want the reference to work. Therefore, good programming practice is to make all methods virtual because reference parameters occur frequently. In this case, a derived object can be passed to a base type parameter and the routine accesses the derived members not the ones associated with the parameter type.

⇒ Why does the destructor for bp have to be virtual for the **delete** &bp to work correctly?
⇒ Why is always making routine members **virtual** a good idea for inheritance?

As mentioned, the best default form of call to a member routine is a **virtual** call. However, it is still possible to override the **virtual** specification at the call site using "::" qualification.

```
bp.base::s(3);                  // call base.s
```

The reverse situation of calling a member routine of the actual object type referenced by the pointer for a non-virtual the member is possible. However, it requires runtime type checking to be accomplished safely (see a C++ textbook).

⇒ Edit complex.C.
⇒ Make the following modification to routine main:

```
int main() {
    mycomplex x;
    complex &xp = x;
    cout << "x:" << xp.abs() << endl;
    cout << "x:" << x.absCalls() << endl;
}
```

⇒ Compile the program, run it, and check the output.
⇒ Modify the program (other than main) to correct the problem in the output.

17 Templates

Inheritance handles reuse where types have commonality, but implementations are different. There is another kind of reuse for situations where there is little or no commonality among the types. For example, the overloaded Abs routines defined in Section 11.4 both have identical code but different types.

```
int Abs( int val ) { return val >= 0 ? val : -val; }
double Abs( double val ) { return val >= 0 ? val : -val; }
```

Instead of duplicating the code, a different form of reuse is used, called a *template*, in which types become parameters in the construction of routines and objects, e.g., a template Abs routine:

```
template<class T>  T Abs( T val ) { return val >= 0 ? val : -val; }
```

The keyword **template** introduces the type parameter T, which is subsequently used to declare the return and parameter type for Abs. (The keyword **class** must be used in these examples, but it means the same as **struct**; **class** is defined in Section 19.) When Abs is used, e.g., Abs(-1.1), the compiler infers the type T from the argument, -1.1, to be **double**, and constructs a specific Abs routine with T replaced by **double**.

⇒ Edit hello.C

⇒ Enter the following program:

```
#include <iostream.h>
template<class T> T Abs( T val ) { return val >= 0 ? val : -val; }
int main() {
    cout << Abs( 1 ) << " " << Abs( -1 ) << endl;
    cout << Abs( 1.1 ) << " " << Abs( -1.1 ) << endl;
}
```

⇒ Compile and run the program.

Template object types are also possible, when object types have identical code but different types. For example, a stack data-structure, implemented using an array, has common code to manipulate the array, but the type of the array elements varies.

```
template<class T> struct Stack {
    T elems[10];
    int size;
    Stack() { size = 0; }
    void push( T e ) { elems[size] = e; size += 1; }
    T pop() { size -= 1; return elems[size]; }
};
```

The type parameter, T, is used to declare the element type of the array elems, as well as return and parameter types of member routines. For template object types, the compiler cannot infer the type parameter, so it is explicitly specified.

```
Stack<int> si;                  // stack of int
Stack<double> sd;               // stack of double
Stack< Stack<int> > ssi;        // stack of stack of int
si.push(3);
sd.push(3.0);
ssi.push( si );
int i = si.pop();
double d = sd.pop();
si = ssi.pop();
```

Beware the syntax problem for nested template instantiation, e.g., Stack< Stack<**int**> >; there must be a space between the two ending chevrons or >> is parsed as **operator**>>.

Data structures are the most common place where programmers use templates. Templates are used to form a specific list structure from a general template that manipulates programmer defined list nodes. The C++ STL provides a basic set of list structures, and advanced list libraries are available. Figure 1 shows a simple list example. List libraries are divided into two kinds: those that copy the user nodes into the list and those that link the nodes directly into the list. The implication of copying is that the node type must have a basic constructor (i.e., one with no parameters) so instances can be created without having to know arguments to constructors. The implication of linking is that the node type must inherit from a particular list type to ensure it has appropriate link members. The C++ STL uses copying. The declaration of a list also specifies the type of the list nodes, list<node>. The instance created, l in the example, is conceptually a pointer to an empty list of nodes. The template parameter allows the creation of different kinds of lists in the same program, such as list<string>, list<node>, and list<**int**>.

An additional concept introduced by lists is the *iterator*, which is used to traverse a list without knowing how the list is implemented. The capabilities of an iterator depends on the kind of list, e.g., a singly linked list only allows traversing the list unidirectionally while a doubly linked list allows bidirectional traversal. Each list in the C++ library provides an appropriate iterator as a nested object type (see the end of Section 15); hence the declaration type of the iterator for list<node> is list<node>::iterator.

In Figure 1, the first loop initializes a node with values and passes it to push_back, which copies it at the end (back) of the list. The secnd loop traverses the list using the iterator to start a pointer at the beginning of the list and steps through the list one node at a time until the pointer is past the end of the list (end() is not the last node but past the end node). At each iteration, the iterator index, ni, is *almost* a pointer to the data stored in the list (explaining *almost* is beyond this tutorial), and therefore, the operator "->" cannot be used to access members. Instead, operators "*" and "." must be used with parenthesis to achieve the correct order of evaluation (see Section 9). As well, the operator "++" is used to advance to the next node in the list. (Operator "++" is an old form of the operator "+=" and its use is

```
#include <iostream.h>
#include <list>
struct node {
    char c;
    int i;
    double d;
    node() {}                                          // must have a basic constructor
    node( char c, int i, double d ) : c(c), i(i), d(d) {}
};
int main() {
    list<node> top;                                    // doubly linked list
    for ( int i = 0; i < 10; i += 1 ) {                // create list nodes
        node n( 'a'+i, i, i+0.5 );                     // node to be added
        top.push_back( n );                            // copy node at end of list
    }
    list<node>::iterator ni;                           // iterator for doubly linked list
    for ( ni = top.begin(); ni != top.end(); ++ni ) {  // traverse list nodes
        cout << "c:" << (*ni).c << " i:" << (*ni).i << " d:" << (*ni).d << endl;
    }
    cout << endl;
    while ( 0 < top.size() ) {                         // destroy list nodes
        node n = top.front();                          // copy node at front of list
        top.erase( top.begin() );                      // remove first node
        cout << "c:" << n.c << " i:" << n.i << " d:" << n.d << endl;
    }
    if ( top.empty() ) {                               // verify list nodes destroyed
        cout << endl << "list is empty" << endl;
    }
}
```

Figure 1: STL Doubly Linked List

discouraged other than this context.) The loop that destroys the list repeatedly erases the first node from the list until the number of nodes is zero.

⇒ The iterator operator "--" moves in the reverse direction to "++", and the last node in a list is defined to be --end() (one back from past the end). Write a loop that prints the nodes of a list in reverse order. (Stopping the loop is tricky.)

18 ⋈ Shell Argument

Up to now, the routine main has been written without parameters. However, it actually has two parameters, which are passed as arguments when the executable file is invoked from the shell. The shell takes the command line tokens and transforms them into C++ arguments. The prototype for main in this case is:

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

argc is the number of tokens in the shell command, including the name of executable file. argv is an array of pointers to the character strings that make up the arguments. If the executable is called in the following way:

```
./a.out -option infile.C outfile.C
```

the arguments have the values:

```
argc=4  argv[0] = "./a.out\0"
        argv[1] = "-option\0"
        argv[2] = "infile.C\0"
        argv[3] = "outfile.C\0"
        argv[4] = 0 // mark end of variable length list, not really necessary
```

Notice, the call of main by the shell is not consistent with a normal routine call in C/C++, because the arguments are passed as strings not values of or references to variables.

The routine main usually begins by checking argc for a value greater than 1, and then processing the values in argv[1] through argv[argc-1]. Figure 2 illustrates how to deal with command arguments to a program.

⇒ Modify the program in Figure 2, so that the input file is optional and defaults to cin if not specified.
⇒ Test your program to ensure it is correct.

19 Encapsulation

Abstraction is the separation of interface and implementation, which allows an object's implementation to change without affecting usage. Abstraction is essential for reuse and maintenance. For example, the complex type provides an interface that does not require a user to directly access the implementation to perform operations, e.g.:

```
struct complex {
    double re, im;    // implementation data structures
    . . . // interface routine members
};
```

so the implementation members and the code for routine members can change from Cartesian to polar coordinates, while the user interface remains constant. Developing good interfaces for objects is the skill of abstraction.

Encapsulation is hiding the implementation for security or financial reasons, called access-control. Encapsulation is neither essential nor required to develop software, assuming users follow a convention of not directly accessing the implementation; however, relying on users to follow conventions is dangerous. Encapsulation is provided by a combination of C and C++ features. C features work largely among source files, and are indirectly tied into separate compilation (see Section 19.1). C++ features work both within and among source files.

A simple, and yet powerful, encapsulation feature in C++ are 3 levels of visibility control for object types:

```
struct info {
  private:      // visible within object type and to friends
    // members and routines
  protected:    // visible within object type, to friends and inherited object types
    // members and routines
  public:       // visible within object type, to friends, inherited object types and to object type users
    // members and routines
};
```

All members and routines after a label, **private**, **protected** or **public**, have that particular visibility. Visibility labels can occur in any order and multiple times in an object type. For a **struct**, the label **public** is implicitly inserted at the beginning of the structure, i.e., the default is that all members are public. C++ provides another structure constructor called **class**, which is the same as **struct**, except the default is that all members are private. In both cases, the public members and routines define an object type's interface, i.e., what a user can access. A user can still see private and protected parts but cannot access them, and therefore, cannot write code that depends on or violates the implementation. Inherited object types can access and modify public and protected members, which may allow access to some of an object's implementation. Only the object type can access the private members and routines, so in general, most or all of the implementation members are private.

Encapsulation introduces a new problem for free routines used to implement binary operations for an object: a free routine may need to access an object's implementation, but it is treated the same as a user routine so it cannot access private members. To solve this problem, C++ provides a mechanism to state that a free routine is allowed access to its implementation, called *friendship*.

```
struct complex {
    friend complex operator+(complex a, complex b);    // prototype of friend free routine
    . . .
};
complex operator+(complex a, complex b) { . . . }
```

The **friend** prototype indicates that any routine with the specified name and type may access this object's implementation, i.e., the free routine is in the set of private members for the object. Thus, an encapsulated complex type looks like:

```
/******************
  Copy text file.

  Command line syntax is:

     ./a.out input-file [output-file]

  input-file must be specified. The output-file is optional and defaults to cout if not specified.
  Example usage:
     ./a.out inputfile
     ./a.out inputfile outputfile
******************/

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>

int main( int argc, char *argv[ ] ) {
    istream  *infile;                          // pointer to input stream
    ostream *outfile = &cout;                  // pointer to output stream; default to cout
    char     ch;

    switch ( argc ) {
      case 3:
        outfile = new ofstream( argv[2] );     // open the outfile file
        if ( outfile->bad() ) {
            cerr << "Error! Could not open output file \"" << argv[2] << "\"" << endl;
            exit( -1 );                        // TERMINATE!
        } // if
        // fall through to handle input file
      case 2:
        infile = new ifstream( argv[1] );      // open the first input file
        if ( infile->bad() ) {
            cerr << "Error! Could not open input file \"" << argv[1] << "\"" << endl;
            exit( -1 );                        // TERMINATE!
        } // if
        break;
      default:                                 // must specify an input file
        cerr << "Usage: " << argv[0] << " input-file [output-file]" << endl;
        exit( -1 );                            // TERMINATE!
    }
    *infile >> resetiosflags( ios::skipws );   // turn off white space skipping during input

    for ( ;; ) {                               // copy input file to output file
        *infile >> ch;                         // read a character
      if ( infile->eof() ) break;              // finished reading ?
        *outfile << ch;                        // write a character
    }
    delete infile;
    if ( outfile != &cout ) delete outfile;    // do not delete cout!
}
```

Figure 2: Processing Shell Arguments

```
class complex {
    friend complex operator+(complex a, complex b);
    friend ostream &operator<<( ostream &os, complex c );
    double re, im;
  public:
    double abs() { return sqrt( re * re + im * im ); }
    complex() { re = 0.; im = 0.; }
    complex(double r) { re = r; im = 0.; }
    complex(double r, double i) { re = r; im = i; }
    ~complex() { }
};
complex operator+( complex a, complex b ) {
    return complex( a.re + b.re, a.im + b.im );
}
ostream &operator<<( ostream &os, complex c ) {
    os << c.re << "+" << c.im << "i";
    return os;
}
```

⇒ Edit hello.C
⇒ Enter the following program:

```
class base {
  private:
    int x;
  protected:
    int y;
  public:
    int z;
};
class derived : public base {
  public:
    derived() { x; y; z; };
};
int main() {
    derived d;
    d.x; d.y; d.z;
}
```

⇒ Compile the program, removing invalid references, until there is successful compilation.

19.1 Separate Compilation

The source file provides an additional mechanism for encapsulation. The approach is to divide an object into two parts, and into two (or more) files. An object is divided into an interface and implementation. The interface is usually composed of only prototype declarations (but possibly some implementation), and goes into one or more .h files, and the implementation is composed of the necessary declarations, which goes into one or more .C files. Encapsulation is provided by giving a user access to only the .h file and the compiled .C executable file(s), which is sufficient to use an abstraction, but not the source in the .C file(s). Most software supplied from software vendors comes this way.

While a .h file normally includes only literals, declarations and prototypes needed to use an object, facilities are still required to export declarations from the .C file, and hide others. By default, all global variables and routines are exported, and hence visible outside the source file. The prototypes in the .h files for exported declarations must be qualified with **extern** to indicate that the implementation appears elsewhere. To encapsulate declarations in a source file, the declaration must be qualified with **static**. For example, the complex type can be divided into file complex.h:

```
#ifndef __COMPLEX_H__     // protect against multiple inclusion
#define __COMPLEX_H__
#include <iostream.h>      // access: ostream
extern void complexStats();
class complex {
    friend complex operator+(complex a, complex b);
    friend ostream &operator<<( ostream &os, complex c );
    double re, im;          // exposed implementation
  public:
    double abs();
    complex();
    complex(double r);
    complex(double r, double i);
};
extern complex operator+( complex a, complex b );
extern ostream &operator<<( ostream &os, complex c );
#endif  // __COMPLEX_H__
```

which users include in their programs, and file complex.C:

```
#include "complex.h"
#include <math.h>           // access: sqrt
// private declarations
static int cplxObjCnt = 0;  // private counter
// interface declarations
void complexStats() { cout << cplxObjCnt << endl; }
complex::complex() { re = 0.; im = 0.; cplxObjCnt += 1; }
complex::complex(double r) { re = r; im = 0.; cplxObjCnt += 1; }
complex::complex(double r, double i) { re = r; im = i; cplxObjCnt += 1; }
double complex::abs() { return sqrt( re * re + im * im ); }
complex operator+( complex a, complex b ) {
    return complex( a.re + b.re, a.im + b.im );
}
ostream &operator<<( ostream &os, complex c ) {
    os << c.re << "+" << c.im << "i";
    return os;
}
```

The .C file normally includes the .h file so that there is only one copy of the literals, declarations, and prototype information. The variable cplxObjCnt is qualified with **static** to make it a private variable to this source file, i.e., no user can access it, and each constructor increments the counter when a complex object is created. The exported free routine complexStats can be called to print the number of complex objects created so far in a program. Notice, also, all the member routines of complex are declared outside of the object type so they can be placed in the .C file.

Note by reading the .h, it may be possible to determine the implementation technique used, so there is only a certain level of encapsulation. It is possible to provide complete encapsulation in C/C++, using more expensive references rather than values in the implementation (see Figure 3). Essentially, how much information goes into .h file depends on the amount of encapsulation; the amount of encapsulation may affect the implementation. (Currently g++ **template**s are implemented by expansion so they must appear in the .h files so the compiler can expand them at the point of usage.)

Notice the use of a copy constructor and assignment operator in Figure 3 because complex objects now contain a reference pointer to the implementation, and a reference pointer cannot be copied on initialization or assignment without generating storage management problems. For example, copying the reference pointer can result in two complex objects pointing at the same complex value and both may eventually attempt to delete it. As well, overwriting a reference pointer may lose the only pointer to the storage so it can never be freed.

An encapsulated object is compiled using the -c compilation flag and subsequently linked with other compiled source files to form a program:

```
g++ -c complex.C
```

which creates a file called complex.o containing a compiled version of the source code.

```
#ifndef __COMPLEX_H__          // protect against multiple inclusion
#define __COMPLEX_H__
#include <iostream.h>          // access: ostream
extern void complexStats();
class complex {
    friend complex operator+(complex a, complex b);
    friend ostream &operator<<( ostream &os, complex c );
    struct complexImpl;        // hidden implementation, nested class
    complexImpl &impl;         // pointer to implementation
  public:
    complex();
    complex(double r);
    complex(double r, double i);
    ~complex();
    complex(const complex &c);   // copy constructor
    complex &operator=(const complex &c); // assignment operator
    double abs();
};
extern complex operator+( complex a, complex b );
extern ostream &operator<<( ostream &os, complex c );
#endif   // __COMPLEX_H__
```

(a) complex.h

```
#include "complex.h"
#include <math.h>              // access: sqrt
// private declarations
static int cplxObjCnt = 0;     // private counter
struct complex::complexImpl {  // actual implementation, nested class
    double re, im;
};
// interface declarations
void complexStats() { cout << cplxObjCnt << endl; }
complex::complex() : impl(*new complexImpl) { impl.re = 0.; impl.im = 0.; cplxObjCnt += 1; }
complex::complex(double r) : impl(*new complexImpl) { impl.re = r; impl.im = 0.; cplxObjCnt += 1; }
complex::complex(double r, double i) : impl(*new complexImpl) { impl.re = r; impl.im = i; cplxObjCnt += 1; }
complex::~complex() { delete &impl; }
complex::complex(complex &c) : impl(*new complexImpl) {
    impl.re = c.impl.re; impl.im = c.impl.im; cplxObjCnt += 1;
}
complex &complex::operator=(const complex &c) {
    impl.re = c.impl.re; impl.im = c.impl.im; return *this;
}
double complex::abs() { return sqrt( impl.re * impl.re + impl.im * impl.im ); }
complex operator+( complex a, complex b ) {
    return complex( a.impl.re + b.impl.re, a.impl.im + b.impl.im );
}
ostream &operator<<( ostream &os, complex c ) {
    os << c.impl.re << "+" << c.impl.im << "i";
    return os;
}
```

(b) complex.cc

Figure 3: Encapsulated Abstract Type

To use an encapsulated object, a program includes the .h file and uses the declarations:

```
#include <iostream.h>
#include "complex.h"
int main() {
    complex x, y, z;
    x = complex( 3.2 );
    y = x + complex( 1.3, 7.2 );
    z = complex( 2 );
    cout << "x:" << x << " y:" << y << " z:" << z << endl;
}
```

which is similar to using the encapsulated types iostream and string is previous examples. Notice, iostream.h is included twice, once in this program and once in complex.h, which is why each .h file needs to prevent multiple inclusions. The compilation command specifies the source file(s) that use the encapsulated type and any compiled file(s) for the encapsulated type(s):

```
g++ yourprogram.C complex.o
```

20 Acknowledgments

Special thanks to Glen Ditchfield and Rob Holte for reading and commenting on the initial drafts of the tutorial. Additional thanks to Caroline Kierstead and Steve Mann for commenting on the final draft.

A Pulling It All Together

```
/******************
  Words are read in and written out in reverse order. A word contains only alphabetic characters.
  Command line syntax is:

  ./a.out [input-file [output-file]]

  input-file is the optional name of the input file (defaults to cin). If output-file is specified,
  the input file must also be specified. The output file defaults to cout if not specified.

  Examples:
     ./a.out
     ./a.out inputfile
     ./a.out inputfile outputfile
******************/

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <string>
#include <list>

int main( int argc, char *argv[ ] ) {
    istream  *infile = &cin;                        // pointer to input stream; default to cin
    ostream *outfile = &cout;                       // pointer to output stream; default to cout

    switch ( argc ) {
      case 3:
        outfile = new ofstream( argv[2] );          // open the outfile file
        if ( outfile->bad() ) {
            cerr << "Error! Could not open output file \"" << argv[2] << "\"" << endl;
            exit( -1 );                             // TERMINATE!
        } // if
        // fall through to handle input file
      case 2:
        infile = new ifstream( argv[1] );           // open the first input file
        if ( infile->bad() ) {
            cerr << "Error! Could not open input file \"" << argv[1] << "\"" << endl;
            exit( -1 );                             // TERMINATE!
        } // if
```

```
            break;
        case 1:
            // use cin and cout
            break;
        default:                                    // too many arguments
            cerr << "Usage: " << argv[0] << " [input-file [output-file]]" << endl;
            exit( -1 );                             // TERMINATE!
    }

    string line, alpha = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
    list<string> words;                             // list of words in document

    *infile >> resetiosflags( ios::skipws );        // turn off white space skipping during input
    for ( ;; ) {                                    // scan lines from a file
            getline( *infile, line );               // read entire line, but not newline
        if ( infile->eof() ) break;                 // end-of-file ?
            line += "\n";                           // add newline character as sentinel character
            for ( ;; ) {                            // scan words off line
                int posn = line.find_first_of(alpha);   // find position of 1st alphabetic character
              if ( posn == -1 ) break;              // any characters left ?
                line = line.substr( posn );         // remove leading whitespace
                posn = line.find_first_not_of(alpha);   // find position of 1st non-alphabetic character
                string word = line.substr( 0, posn );   // extract word from start of line
                words.push_back( word );            // add word to end of list
                line = line.substr( posn );         // delete word from line
            } // for
    } // for

    *outfile << "The words in reverse order:" << endl;

    while ( 0 < words.size() ) {                    // traverse list in reverse order
            *outfile << *(--words.end()) << endl;   // print last node
            words.erase( --words.end() );           // remove last node
    } // whille

    if ( infile != &cin ) delete infile;            // do not delete cin
    if ( outfile != &cout ) delete outfile;         // do not delete cout
} // main

// Local Variables: //
// tab-width: 4 //
// End: //
```

Index

!, 12, 18
!=, 12, 13
"", 25
#define, 24
#elif, 25
#else, 25
#endif, 25
#if, 25
#ifdef, 26
#ifndef, 26
#include, 25
⇒, 4
⋈, 4
&, 12, 13, 18, 19
&&, 12, 14, 17, 19
&=, 12
*, 12, 13
*/, 5
*=, 12
+, 12
+=, 12
,, 12, 14
-, 12
-=, 12
->, 12, 13
., 12
., 13
./, 5
.C, 4
.cc, 4
.h, 25
/, 12, 13
/*, 5
//, 5
/=, 12
::, 33, 39
;, 5
<, 12
<<, 12, 13
<<=, 12
<=, 12
<>, 25
=, 8, 12, 13, 17
==, 12, 13, 17
>, 12
>=, 12
>>, 12, 13
>>=, 12
?:, 12, 14, 17, 18
%, 12
%=, 12
{}, 16
^, 12
^=, 12
_ , 6
|=, 12
||, 12, 14, 17
~, 12

a.out, 4, 5, 45
abstraction, 46
access control, 46
argc, 45, 46
argument, 21
argv, 45, 46
array, 9, 12
    dynamic multi-dimensional, 29
array parameter, 21
assignment, 34
    cascade, 14
assignment operator, 49

basic types, 7, 9
    bool, 7
    char, 7
    double, 7
    int, 7
bitset, 9
block, 6, 16
    {}, 16
bool, 7
break, 17, 20

cascade assignment, 14
case-sensitive, 6
    lower-case, 6
    mixed case, 6
    upper-case, 6
cast, 12, 15, 24
cerr, 26
char, 7
chevrons, 6
class, 46
comment, 5, 6
    */, 5
    /*, 5
    //, 5
    C++ style, 5
    C style, 5
    nesting, 5
    out, 25
compilation options, 4
compiling a C++ program, 4

a.out, 4
compilation options
-E, 4
-Wall, 4
-c, 4, 49
-g, 4
-o, 4
g++, 4
conditional, 16
conditional expression evaluation, 17
&&, 17
?:, 17
||, 17
partial evaluatio, 17
short-circuit, 18
conformant array, 22
const, 8, 25
constructor, 34
contra-variance, 42
control structures, 16
block, 16
{}, 16
conditional, 16
conditional expression evaluation, 17
&&, 17
?:, 17
||, 17
partial evaluation, 17
short-circuit, 18
iteration, 18
break, 20
do, 18
for, 19
while, 18
selection, 17
break, 17
default, 17
else, 17
if, 17
switch, 17
short-circuit expression evaluation, 18
conversion, 14
cast, 12
explicit, 15
implicit, 14, 16
narrowing, 15
widening, 14
copy constructor, 35, 49
cout, 26
creating a C++ source file, 4

dangling else, 17
declaration, 6
basic types, 7
constants
const, 25
type constructor, 9
type modifier, 8
variable, 7
declaration-before-use rule, 6, 22, 32
default, 17
delete, 28
derived type, 40
destructor, 34
do, 18
documentation, 5
double, 7
dynamic storage management, 28
multi-dimensional array, 29

else, 17
encapsulation, 46
end-of-file, 27
enumeration, 9
escape sequence, 7
exit, 6
explicit conversion, 15
explicit inclusion, 38
expression, 12
extern, 48

file
.C, 4, 48
.cc, 4
.h, 48
.o, 49
for, 19
forward declaration, 22
free routine, 31, 33, 35, 40–42, 46, 49
friendship, 46

g++, 4

identifier, 6
if, 17
?:, 17
dangling else, 17
else, 17
implementation inheritance, 38
implicit conversion, 14, 16
inclusion
explicit, 38
inheritance, 38
implementation, 38
type, 40
initialization, 12, 33, 49
array, 12
string, 12

- structure, 12
input, 26, 27
- cin, 26

input and output
- cerr, 26
- cin, 26
- cout, 26
- iostream.h, 26

int, 7
iostream.h, 26
iteration, 18
iteration statement
- **break**, 20
- **do**, 18
- **for**, 19
- **while**, 18

iterator, 44

keywords, 6

literals, 7
- **bool**, 7
- **char**, 7
- **double**, 7
- escape sequence, 7
- **int**, 7
- string, 7

long, 8

macros, 25
main, 5, 23, 45
manipulators, 27
members, 10

name equivalence, 9, 10, 41
narrowing conversion, 15
nesting
- blocks, 16
- comments, 5
- initialization, 12
- preprocessor, 25
- routines, 20
- structures, 31
- types, 33

new, 28
NULL, 25

object, 30
- type
  - nesting, 33, 44

object-oriented, 4, 30, 38
operators
- arithmetic, 12
- assignment, 12
- bit shift, 12
- bit-wise, 12
- cast, 12
- control structures, 12
- expression list, 12
- logical, 12
- pointer, 12
- relational, 12
- **struct**, 12
- type, 14

output, 26, 27
- cerr, 26
- cout, 26

overloading, 13, 23

parameter, 21
- pass by reference, 21
- pass by value, 21

passed by reference, 21
passed by value, 21
pointer, 9, 12
- arithmetic, 13

preprocessor, 5, 24
- **#define**, 24
- **#elif**, 25
- **#else**, 25
- **#endif**, 25
- **#if**, 25
- **#ifdef**, 26
- **#ifndef**, 26
- **#include**, 25
- macros, 25

private, 46
programming style, 4
protected, 46
prototype, 22
public, 46

reserved identifiers, 6
return, 20
return type, 20
routine, 20
- argument/parameter passing, 21
- array parameter, 21
- parameter, 20
  - pass by reference, 21
  - pass by value, 21
- prototype, 22
- **return**, 20
- return type, 20
- routine overloading, 23
- routine prototype
  - forward declaration, 22

running a C++ program, 5

./, 5
a.out, 5

segment fault, 19
selection statement, 17
break, 17
default, 17
else, 17
if, 17
switch, 17
semicolon, 5
separate compilation, 48
-c, 49
shell, 5
shell arguments, 45
argc, 45, 46
argv, 45, 46
main, 45
short, 8
short-circuit expression evaluation
&&, 18
||, 18
side-effect, 14, 21
sizeof, 14
software development
.C, 48
.h, 48
.o, 49
separate compilation, 48
-c, 49
source file, 4, 20, 23, 33, 46, 48
standard template library, 9, 37, 44
statement, 5
basic forms, 5
static, 48
STL, 9, 37, 44
stream file, 26
string, 7, 37
structure, 10
members, 10
struct, 9, 12
structure of a C++ program, 5
basic structure
main, 45
comment
out, 25
comments, 5
program structure
main, 5
statement, 5
switch, 17
break, 17
default, 17
syntactic structure, 5
template, 9, 43, 49
this, 31
token, 5
type constructor, 9
array, 9
enumeration, 9
literal, 12
pointer, 9
struct, 9
typedef, 9
type constructor literal, 12
array, 12
pointer, 12
struct, 12
type equivalence, 9, 41, 42
type inheritance, 40
type modifier, 8
const, 8
extern, 48
long, 8
short, 8
static, 48
unsigned, 8
type operator, 14
type sharing, 38
typedef, 9

undefined behaviour, 6
unsigned, 8

variable declarations
type modifier, 8
virtual, 42
void, 20

while, 18
whitespace, 5
widening conversion, 14