C++ Tutorial
Java 1.5 Based

University of Waterloo

Version 1.0

Peter A. Buhr ©[*]2005

last update: December 4, 2007

Good code has good design elements;
good code also uses the dominant metaphors in a language
to make it easy for other programmers to understand.

Contents

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 Java programming experience and familiarity with basic programming language concepts. By working through the exercises, core C++ concepts can be learned and 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 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 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 often used to write 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 C/C++ Source File

A C++ source file should be created in a UNIX file with the suffix .cc, .cxx, .cpp, .cp, .c++ or .C. (Suffix .C is used in this document.) A C source file should be created with suffix .c. Any text editor may be used to create a source file. Many text editors use a file's suffix to infer the kind of data within the file. Then the editor provides language-specific operations, such as colourization, indentation, searching, etc. As well, the C++ compiler uses the file suffix to decide which files are used for specific compilation steps.

4 Compilation

The C++ compilation command performs the following steps: preprocessing, compilation, assembly and linkage to produce an executable file called a.out. (The file a.out is overwritten for each compilation.) By using appropriate command-line options, individual steps of the compilation and actions within a step can be controlled. This tutorial uses the GNU g++ compiler; 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 often only one source file.

An option starts with a "-" (minus character) followed by a single-character name and possibly a value depending on the option. In most cases, the option value can immediately follow the option name or be separated from it by spaces. In general, no options are required; some useful options are (see man g++ for a complete list of options):

-E Perform only the preprocessor step, printing the output on standard output (see Section 11, p. 21).

-c Perform only the preprocessor and compilation steps (see Section 24, p. 56).

-o *name* Name the executable file to the specified name instead of a.out, e.g., g++ -o assn1 *yourprogram*.C creates an executable called assn1 instead of a.out.

-O Optimize the code generated by the compiler so the program runs faster.

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

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

5 Execution

Once an executable file is created from a C++ source program, it is presented to a shell to be loaded and run, like built-in 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 or relative directory path. The full (absolute) 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.

Like built-in commands, a C/C++ executable may have command-line options; accessing these shell arguments is discussed in Section 16, p. 33.

6 Program Structure

A C++ program is composed of two components: comments strictly for people, and statements for both people and the preprocessor/compiler. A source file contains a mixture of comments and statements. The C/C++ preprocessor/compiler only reads the statements and ignores the comments.

6.1 Comment

Comments are essential to document what a program does and how it does it. Like Java, a comment may be placed anywhere a whitespace (space, tab, newline) is allowed, and there are two kinds of comments in C/C++, which are identical to those in Java:

	Java/C/C++
1	/* ... */
2	// remainder of line

The first form begins with the start symbol, /*, and ends with the terminator symbol, */, and hence, can extend over multiple lines. Like Java, *this form cannot be nested one within another*:

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

Here, the first terminator, */, ends the comment and the remaining comment text is treated as statements. Hence, be extremely careful in using this form of comment to elide/comment-out code:

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

The second form begins with the start symbol, //, and continues to the end of the line, i.e., only one line long. Like Java, this form of comment can be nested one within another:

```
// ... // ... nested comment
```

so it can be used to comment-out code:

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

Section 11.3, p. 23 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; these comments provide additional explanation and never affect the program's execution. In fact, all the code for each work assignment (labelled ExN) is available online; ask for the specific location.

6.2 Statement

C++ is actually composed of 3 languages:

1. The preprocessor language modifies (text-edits) the program *before* compilation (see Section 11, p. 21).
2. The template (generic) language adds new types and routines *during* compilation (see Section 21, p. 51).
3. The programming language specifies declarations and control flow to be executed *after* compilation.

A programmer uses the three programming languages in the following way:

user edits → preprocessor edits → templates expand → compilation (→ linking/loading → execution)

The syntax for a preprocessor statement is a **#** character, followed by a series of tokens separated by whitespace, which is usually a single line and not terminated by punctuation. The syntax for a C/C++ statement (both template and regular) is a series of tokens separated by whitespace and terminated by a semicolon.[1]

7 First Program

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

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

Java	C++
	`// First C++ program by: YourFirstName YourLastName`
	`// Print "Hello World!" to the screen.`
`import java.lang.*; // implicit`	`#include <iostream>      // import I/O facilities`
`class hello {`	`using namespace std;     // direct naming of I/O facilities`
`    public static void main( String[] args ) {`	`int main() {             // program starts here`
`        System.out.println("Hello World!");`	`    cout << "Hello World!" << endl;`
`        System.exit( 0 );`	`    return 0;            // return 0 to shell`
`    }`	`}`
`}`	

Several important points are illustrated in this program:

1. When writing programs, there should be comments at the beginning identifying the programmer 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.

2. The **#include** <iostream> imports the basic input and output (I/O) facilities for C++, which facilitates reading and writing of values (no equivalent in Java).

3. The **using namespace** std allows imported I/O names to be accessed directly, i.e., *without* qualification, like Java **import** java.lang.*.

4. The routine header **int** main() is the default location to begin execution when an executable file is called from a shell. There is only one main routine per program and it returns an integer code to its invoking shell.

5. Like Java, the curly braces, { ... }, denote the start and end of a block of code, and this block is the body of routine main.

6. The statement cout << "Hello World!" << endl prints the text "Hello World!" to standard output, called cout, which is usually the terminal screen (like System.out in Java). Think of the information as cascading from right to left, as indicated by the chevron, <<, to be printed on cout. endl ensures the string "Hello World!" appears on its own line, like println in Java.

7. The **return** 0 returns zero to the shell indicating successful completion of the program; non-zero usually indicates an error. In many C/C++ programs, routine main does not return a value; in this case, an implicit value of 0 is returned. Only routine main has this special property. The routine exit, like Java System.exit, can also be used to stop a program at any location and return a code to the invoking shell, e.g., exit(0).

[1] The exception is a block, denoted with { }, which forms a complete statement so it is not terminated with a semicolon (see Section 10.1, p. 18).

⇒ Compile the first 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` in the shell.
⇒ Edit file `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

A declaration defines new variables and types in a program. Variables and types may be named or be anonymous.

8.1 Identifier

An identifier is a name used to refer to a variable or type. Like Java, identifiers in C/C++ may be arbitrarily long, and the first character must be a letter (upper or lower case) or an underscore “_”; characters other than the first can be any of the previous, or a digit. An identifier is *case-sensitive*; i.e., 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:

```
VeryLongVariableName      Page1     Income_Tax     _75
```

Some identifiers are reserved as they denote keywords (and appear in **bold** font in this document); see a C++ textbook for a complete list of reserved identifiers. (Identifiers beginning with an underscore are reserved for the C++ implementation.)

8.2 Basic Types

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

Java	C/C++	
boolean	**bool**[a]	
char	**char**/**wchar_t**	
byte	**char**/**wchar_t**	integral types
int	**int**	
float	**float**	floating-point types
double	**double**	

[a] C requires `<stdbool.h>`

Unlike Java, C/C++ treat **char** and **wchar_t** (for unicode characters) as an integral type for computation. Java types **short** and **long** are created using type qualifiers (see Section 8.4).

8.3 Variable Declaration

A simple variable declaration in C/C++ is the same as in Java: a type followed by a list of identifiers.

Java/C/C++

char a, b, c, d;
int i, j, k;
double x, y, z;

Declarations in C/C++ can be global to a source file (unlike Java), and local to a block forming a routine body or any of its containing (nested) blocks. Declarations can be intermixed among executable statements within a block; variable names can be reused in nested blocks, i.e., hide (*override*) names in a containing block. All global variables of the basic types are zero-initialized, while similar local variables in a block are *not* initialized. Unlike Java, C/C++ do not check for uninitialized variables.[2] A C/C++ declaration may have an initializing assignment (except for a **struct**/**class** member, see Section 8.6.3, p. 13):

```
int i = 3;
```

⇒ Edit file `hello.C`.

[2] Using the `-Wall` *and* `-O` compilation flags (see Section 4, p. 4) does check for uninitialized variables in g++, which are not optimized away.

⇒ Enter the following program:

```
#include <iostream>                  // Ex01
using namespace std;
bool x;                              // global declaration in a source file
int main () {
    short y;                         // local (automatic) declaration in block
    cout << "x:" << x << endl;       // use of global variable
    cout << "y:" << y << endl;       // use of local variable
    int z;                           // local declaration anywhere in block
    cout << "z:" << z << endl;       // use of local variable
    {
        long y;                      // nested local redeclaration, hide previous y
        cout << "y:" << y << endl;   // use of local variable
        double z;                    // nested local redeclaration, hide previous z
        cout << "z:" << z << endl;   // use of local variable
    }
}
```

⇒ Compile with compilation flags -Wall -O and run the program.

Since variables y and z are uninitialized, the values printed may not be zero and could vary each time the program is run. Why is variable x initialized?

8.4 ⋈ Type Qualifier

C/C++ provide only two basic integral types **char** and **int**; other integral types, like Java **short** and **long**, are generated using type qualifiers. Like Java, C/C++ provide signed (positive/negative) integral types; unlike Java, C/C++ also provide unsigned (positive only) integral types.

integral types	range
signed char or **char**	at least -127 to 127
unsigned char	at least 0 to 255
signed short int or **short**	at least -32767 to 32767
unsigned short int or **unsigned short**	at least 0 to 65535
signed int or **int**	at least -32767 to 32767
unsigned int	at least 0 to 65535
signed long int or **long**	at least -2147483647 to 2147483647
unsigned long int or **unsigned long**	at least 0 to 4294967295
signed long long int or **long long**, g++ only	at least -9223372036854775807 to 9223372036854775807
unsigned long long int or **unsigned long long**, g++	at least 0 to 18446744073709551615

Unlike Java, the range of values for **int** is machine specific; usually 2 bytes for 16-bit computers and 4 bytes for 32/64-bit computers. Similarly, **long** is usually 4 bytes for 16-bit computers and 8 bytes for 32/64-bit computers.

Like Java, C/C++ support constant variables that are write-once/read-only. Java uses type qualifier **final**, while C/C++ use type qualifier **const**, applicable in any variable declaration context. Unlike Java, a C/C++ **const** identifier *must* be assigned a value on its declaration (or through a constructor's declaration); the value can be the result of a runtime expression:

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

A constant variable can appear in read-only contexts after it is initialized.

⇒ Edit file hello.C
⇒ Enter the following program:

```
#include <iostream>                              // Ex02
using namespace std;
int main () {
    long int x;
    x = 10000000000;
    unsigned short int y = -1;
    const int z = y + 3;
    z = 4;
}
```

⇒ Compile the program.
⇒ Read the messages from the compiler and do not proceed until you understand why each is generated.

8.5 Constants

Java and C/C++ share almost all the same constants for the basic types (except for unsigned). A *designated* constant indicates its type with suffixes L/l for long, LL/ll for long long, U/u for unsigned, and F/f for float. Unlike Java, there is no D/d suffix for **double** constants. An *undesignated* integral constant (octal/decimal/hexadecimal) is the smallest **int** type that holds the value, and a floating-point constant is of type **double**.

boolean	**false**, **true**
decimal	123, -456L, 789u, 21UL
octal, prefix 0	0144, -045l, 0223U, 067ULL
hexadecimal, prefix 0X or 0x	0xfe, -0X1fL, 0x11eU, 0xffUL
floating-point	.1, 1., -1., -7.3E3, -6.6e-2F use E/e for exponent
character, single character	'a', '\''
string, multi-character	"abc", "\"\""

Care must be taken to use the right kind of constant with the right kind of character or string variable. Like Java, an escape sequence for special characters can appear any number of times in a string constant. An escape sequence starts with a backslash, \. The most common escape sequences are (see a 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 (not in Java)

Unlike Java, a C/C++ string constant is 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, p. 31.)

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

```
int main () {                                    // Ex03
    cout << 12 << endl << 014 << endl << 0xc << endl;
    cout << 1234.5 << endl << 1.2345e3 << endl;
    cout << 'w' << '\\' << '\'' << '"' << '\n' << endl;
    cout << "w\\'\"\n" << endl;
}
```

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

Some of the printed values are different from the constants in the output statements. Section 12.2, p. 25 explains how to precisely control the format of printed values.

8.6 Type Constructor

A type constructor is a declaration that builds a more complex type from the basic types.

constructor	**Java**	**C/C++**
enumeration	**enum** Colour { R, G, B }	**enum** Colour { R, G, B }
pointer		*any-type* *p;
reference	*class-type* r;	*any-type* &r; (C++ only)
structure	**class**	**struct** or **class**

array	**int** v[ ] = **new int**[10];
	int m[ ][ ] = **new int**[10][10];
		int v[10];
		int m[10][10];

type aliasing		**typedef char** name[25];
		name first, last;

Like Java, C/C++ use *name equivalence* to decide if two types are the same:

```
class T1 {                  class T2 {  // identical structure
    int i, j, k;                int i, j, k;
    double x, y, z;             double x, y, z;
}                           }
T1 t1 = new T1();
T2 t2 = t1;      // incompatible types
```

Here the types T1 and T2 have identical structure (same fields in the same places) but have different names so the initialization of variable t2 fails, even though technically it could work. An *alias* is a different name for the same type, so alias types are equivalent.

8.6.1 Enumeration

An *enumeration* is a type defining a set of named constants with only comparison, assignment and cast to integer operations:

```
enum Names { John, Mary, Fred, Jane }; // declare type and its constants
Names name = Mary;    // only assignment operation
```

The Java enumeration capabilities are more sophisticated than in C/C++. A C/C++ enumeration can only give names to integral values, whereas a Java enumeration can give names (and operations) to any value. Like Java, an enumeration in C++ denotes a new type; in C an enumeration is an alias for **int**. The names in an enumeration are called *enumerators*. In Java, the enumerator names are contained in the scope of the enumeration and must always be qualified. In C/C++, the enumerator names are contained in the scope where the enumeration is declared and are not qualified; hence, enumerator names must be unique in a declaration scope. Like Java, the enumerators can be numbered explicitly.

⇒ Edit file hello.C

⇒ Make the following modification to routine main:

```
int main() {                                    // Ex04
    enum Day {Mon,Tue,Wed,Thu,Fri,Sat,Sun};     // type declaration, implicit numbering
    Day day = Sat;                              // variable declaration, initialization
    enum {Yes, No} vote = Yes;                  // 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 << " vote:" << vote << " colour:" << colour << endl;
}
```

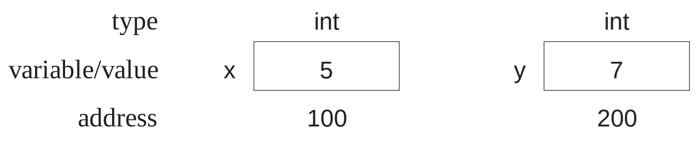
⇒ Compile the program, run it, check the output, and make sure you understand it.

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

```
enum Day day = Sat;    // repeat "enum" on variable declaration
```

8.6.2 Pointer/Reference

A *pointer/reference* is an indirect (versus direct) mechanism to access a type instance. To understand pointers/references it is necessary to know that *all* variables have an address in memory, e.g., **int** x = 5, y = 7:



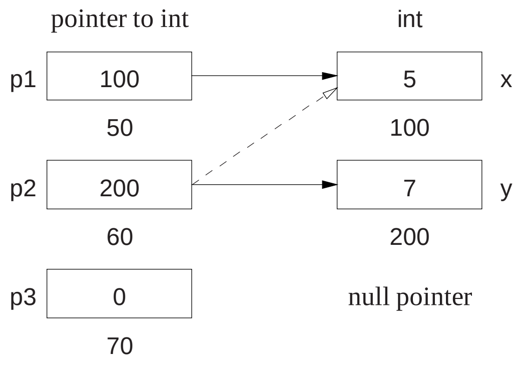
The value of a pointer/reference is simply the address of a variable; access to this address is different depending on whether it is a pointer or reference.

There are two basic pointer/reference operations:

1. *referencing*: obtain the address of a variable; unary operator & in C++:
   ```
   &x → 100
   &y → 200
   ```
2. *dereferencing*: retrieve the value at an address; unary operator * in C++:
   ```
   *(&x) → *(100) → 5
   *(&y) → *(200) → 7
   ```

In addition, there is a special address no variable can have called the *null pointer* (null in Java, 0 in C++).

A pointer/reference variable has as its value either the memory address of another variable (called *indirection*) or the null pointer (or an undefined address if the pointer variable is uninitialized):



```
&p1 → 50
&p2 → 60
&p3 → 70
*(&p1) → 100
*(&p2) → 200
*(&p3) → 0
*(*(&p1)) → 5
*(*(&p2)) → 7
*(*(&p3)) → error
```

A pointer/reference may point to the same memory address as another pointer/reference (dashed line). Also, dereferencing the null pointer is an error because no variable is allocated at address 0.

Explicit dereference is an operation usually associated with a pointer:

```
*p2 = *p1;        ≡   y = x;   // value assignment
*p1 = *p2 * 3;    ≡   x = y * 3;
```

In the first expression, the value pointed to by p2 is assigned the value pointed to by p1, which is an indirect way to perform y = x. In the second expression, the value pointed to by p1 is assigned the value pointed to by p2 times 3, which is an indirect way to perform x = y * 3. Note, the unary and binary use of operator * for deference and multiplication, respectively. Address assignment does not require dereferencing:

```
p2 = p1;          // address assignment
```

Here, p2 is assigned the same memory address as p1, i.e., p2 points at x; the values of x and y do not change.

When pointers are used frequently, having to perform explicit dereferencing can be tedious and error prone. For example, in:

```
p1 = p2 * 3;      // implicit deference
```

it is *unreasonable* to interpret this expression as p1 is assigned the address in p2 times 3, because there is no multiplication operation for address values and there may not be a valid integer variable at memory location 600. Instead, it is reasonable to interpret this expression as the value pointed to by p1 is assigned the value pointed to by p2 times 3, as both pointers refer to integer variables and there is a multiplication operation for integers. A pointer that provides implicit dereferencing is a *reference*. However, implicit dereferencing generates an ambiguous situation for:

```
p2 = p1;
```

Should this expression perform address or value assignment, and how are both cases specified? Disambiguating this expression is discussed next.

C provides only a pointer; C++ provides a pointer and a restricted reference; Java provides only a general reference.

1. C/C++ pointer: created using the * type-constructor, may point to any type (i.e., basic or object type) in any storage location (i.e., global, stack or heap storage), and no implicit referencing or dereferencing.

   ⇒ Edit file hello.C

   ⇒ Make the following modification to routine main:

```
int main() {                    // Ex05
    int x = 5, y = 7;           // basic type
    int *p1, *p2;               // pointer to basic type
    p1  = &x;                   // point to x, explicit referencing
    p2  = &y;                   // point to y, explicit referencing
    p1  = p2;                   // address assignment
    *p2 = *p1;                  // value assignment, explicit dereferencing
    *p1 = *p2 * 3;              // explicit dereferencing
    cout << "p1:" << p1 << " *p1:" << *p1 << endl;
    cout << "p2:" << p2 << " *p2:" << *p2 << endl;
}
```

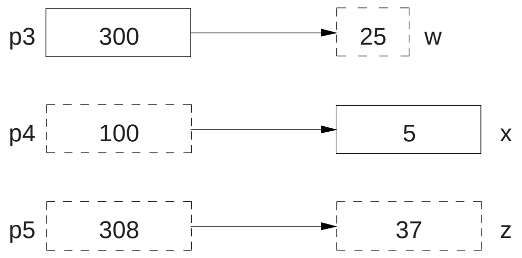
⇒ Compile the program, run it, check the output, and make sure you understand it.

Type qualifiers (see Section 8.4, p. 8) can be used to modify pointer types:

```
const short int w = 25;
const short int *p3 = &w;

int * const p4 = &x;
( int &p4 = x; )

const long int z = 37;
const long int * const p5 = &z;
```



Pointer p3 may point at any **const short int** variable. In this case, the pointer can change to point at different variables, but the value of the variables cannot be changed through the pointer because each is **const**. Pointer p4 may only point at variable x. In this case, the pointer cannot change to point at a different variable because it is **const**, but the value of the variable can be changed through the pointer. Pointer p5 may only point at variable z. In this case, the pointer cannot change to point at a different variable because it is **const**, and the value of the variable z cannot be changed through the pointer because it is also **const**.

2. C++ reference: created using the & type-constructor, may point to any type (i.e., basic or object type) in any storage location (i.e., global, stack or heap storage), restricted to a constant pointer to user created (non-temporary/non-constant) storage, and always has implicit dereferencing.

   ⇒ Edit file hello.C

   ⇒ Make the following modification to routine main:

   ```
   int main() {                    // Ex06
       int x, y;                   // basic type
       int &r1 = x, &r2 = y;       // restricted reference to basic type
       r1 = 5;                     // initialize x, implicit dereferencing
       r2 = 7;                     // initialize y, implicit dereferencing
       r2 = r1;                    // value assignment, implicit dereferencing
       r1 = r2 * 3;                // implicit dereferencing
       cout << "&r1:" << &r1 << " r1:" << r1 << endl;
       cout << "&r2:" << &r2 << " r2:" << r2 << endl;
   }
   ```

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

   ⇒ Change the initialization of r1 to x + y and then to 3.

   ⇒ Compile the program for each change and explain the error message.

   Due to the constant-pointer restriction, a C++ reference is equivalent to a Java **final** reference or * **const** pointer with implicit dereferencing (see previous diagram). A Java reference can vary what it points to, but it can only point to objects in heap storage (see Section 13, p. 26). The C++ constant-pointer restriction has two implications:

   - A C++ reference must be initialized at the point of declaration. Note, the initializing expression has implicit referencing because an address is *always* required; hence, putting an & before the initializing expression is an error because there is implicitly one there:

     ```
     int &r1 = &x; // error, unnecessary & before x
     ```

   - There is no need for address assignment after a C++ reference declaration because the address cannot change. Whereas, a Java reference always interprets r2 = r1 as address assignment and provides no mechanism to perform value assignment between reference types, i.e., no assignment of the member values in one object to the corresponding members in another.

Finally, the pointer/reference type-constructor in C/C++ is *not distributed across the identifier list*, e.g.:

```
int * p1, p2;  // only p1 is a pointer, p2 is an integer, should be    int *p1, *p2;
int & rx, ry;  // only rx is a reference, ry is an integer, should be  int &rx, &ry;
```

8.6.3 Aggregation (structure/array)

Like Java, C++ is object-oriented, but it does not subscribe to the Java notion that everything is a basic type or an object. Instead, aggregation is performed by structures and arrays, and computation is performed by routines; an object type is the composition of a structure and routines (see Section 17, p. 34). As a consequence, in C++, a routine can exist without being embedded in a **struct**/**class** (see Section 14, p. 29).

Structure is a mechanism to group together heterogeneous values, including (nested) structures:

Java	C/C++

class foo {
    public int i = 3;
    . . . *// more fields*
}
	struct foo {
	    int i; *// no initialization*
	    . . . *// more members*
	}; *// semi-colon terminated*

The components of a structure are called *members*[3] in C++. Like Java, all members of a structure are accessible (public) by default (excluding Java **package** visibility). Unlike Java, a structure member cannot be directly initialized (see Section 8.7 and 17.3, p. 37), and a structure is terminated with a semicolon.

As for enumerations, a structure can be defined and instances declared in a single statement.

⇒ Edit file hello.C

⇒ Make the following modification to routine main:

```
int main() {                                  // Ex07
    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.

⇒ While the messages from the compiler are cryptic, why are they generated (think 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 variable:

```
struct complex a, b;      // repeat “struct” on variable declaration
```

Recursive types, like lists and trees, can be defined using a pointer in a structure:

```
struct node {
    . . .            // data members
    node *link;      // pointer to another node
};
```

Array is a mechanism to group together homogeneous values. The array in C/C++ is primitive in comparison to Java because dimension information is not stored with an array object. Therefore, there is no equivalent to Java's length member for arrays, *no subscript checking*, and no array assignment. (See Section 21, p. 51 for the C++ vector type,

[3] Java subdivides members into fields (data) and methods (routines).

which is similar to a Java array.) Unlike Java, array variables in C/C++ can have dimensions specified on declaration and all the array elements are implicitly allocated:

```
int x[10];          // int x[] = new int[10]
int y[10][20];      // int y[][] = new int[10][20]
```

Be careful not to write (explained in Section 9):

```
int b[10, 20];      // not int b[10][20]
```

C++ only supports a compile-time dimension value; g++ allows a runtime expression.

```
int r, c;
cin >> r >> c;      // input dimensions (reading is explained later)
int array[r];       // dynamic dimension, g++ only
int matrix[r][c];   // dynamic dimension, g++ only
```

Like Java, an array is subscripted starting at 0.

⇒ Edit file hello.C

⇒ Make the following modification to routine main:

```
int main() {                                           // Ex08
    char c1[3], c2[3];
    c1[0] = 'T'; c1[1] = 'o'; c1[2] = 'm';             // initialization
    c2[2] = c1[0]; c2[1] = c1[1]; c2[0] = c1[2];       // array copy
    cout << c1[0] << c1[1] << c1[2] << endl;
    cout << c2[0] << c2[1] << c2[2] << endl;
    int v[3];
    v[0] = 93; v[1] = 67; v[2] = 77;                   // initialization
    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 and prints an undefined value!

8.6.4 ⋈ Type Aliasing

Java provides no mechanism to rename types; C/C++ provides **typedef**

```
typedef short int shrint1;   // shrint1 => short int
typedef shrint1 shrint2;     // shrint2 => short int
typedef short int shrint3;   // shrint3 => short int
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 among the variables s1, s2 and s3, because they have the same type name "**short int**" (see "name equivalence" in Section 8.6, p. 9).

8.7 ⋈ Type-Constructor Constant

enumeration	enumerators
pointer	0 or NULL indicates a null pointer
structure	**struct** { **double** r, i; } c = { 3.0, 2.1 };
array	**int** v[3] = { 1, 2, 3 };

C/C++ use 0 to initialize pointers versus null in Java. Certain system include-files define the preprocessor variable NULL as 0 (see Section 11, p. 21).

Structure and array initialization can only occur as part of a declaration. (g++ allows type-constructor constants in executable statements, see Section 9.1, p. 17.) 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:

```
struct { int i; struct { double r, i; } s; } d = { 1, { 3.0, 2.1 } };  // nested structure initialization
int m[2][3] = { {93, 67, 72}, {77, 81, 86} };  // multidimensional array initialization
```

String constants 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 constants in that dimension:

```
char s[] = "abcde";   // first dimension inferred as 6 (Why 6?)
int  v[] = { 0, 1, 2, 3, 4 } // first dimension inferred as 5
int  m[][3] = { {93, 67, 72}, {77, 81, 86} };    // first dimension inferred as 2
```

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

```
int main() {                                    // Ex09
    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 Expression

	Java	C/C++	priority
unary	., (), [], call	., ->, (), [], call, **dynamic_cast**	high
	cast, +, -, !, ~, **new**	cast, +, -, !, ~, &, *, **new**, **delete**, **sizeof**	
binary	*, /, %	*, /, %	
	+, -	+, -	
bit shift	<<, >>, >>>	<<, >>	
	<, <=, >, >=, **instanceof**	<, <=, >, >=	
	==, !=	==, !=	
	&	&	
exclusive-or	^	^	
	|	|	
	&&	&&	
	||	||	
	?:	?:	
	=, +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, ^=, |=	=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=	
		,	low

Like algebra, both Java and C/C++ prioritize operators and perform the operations in an expression from highest to lowest priority. If two operators have the same priority, they are done left to right, except for unary, ?:, and assignment operators, which associate right to left. In Java, the order of evaluation of subexpressions of an operator and argument evaluation is from left to right; however, in C/C++, it is unspecified:

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

Both the referencing (address-of), &, and dereference, *, operators (see Section 8.6.2, p. 10) do not exist in Java because access to storage is restricted. Note, it is possible to determine the address of any variable in any storage context, e.g., &x is the address of x, &s.d is the address of member d in structure s, and &v[5] is the address of array element v[5].

The arrow operator, ->, is unique to C/C++ and is an anomaly among programming languages. It exists solely because the priority of the selection operator "." is incorrectly higher than the dereference operator "*", so *p.f executes as *(p.f) instead of (*p).f. Rather than correct this mistake, the special -> operator exists to perform a dereference and member selection in the correct order, i.e., p->f is implicitly rewritten as (*p).f.

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

```
int main() {                              // Ex10
    struct node { int val; node *link; };
    node x = { 5, NULL }, y = { 3, &x }, *n = &y;
    cout << n->val << endl;               // print first node
    n = n->link;                          // n = (*n).link, advance to next node
    cout << n->val << endl;               // print second node
}
```

⇒ Compile the program, run it, and check the output (drawing a picture of the linked list is helpful).

The pseudo-routine **sizeof** does not exist in Java because it is related to explicit storage management. 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 references.

⇒ Edit file hello.C

⇒ Make the following modification to routine main:

```
int main() {                              // Ex11
    struct node { int val; node *link; };
    node x = { 5, NULL }, y = { 3, &x }, *n = &y;
    cout << sizeof(node) << " " << sizeof(x) << " " << sizeof(n) << " " << sizeof(*n) << endl;
    cout << sizeof("abc") << endl;
}
```

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

The bit-shift operators, << (left), and >> (right), are identical to those in Java. (Notice, the bit-shift operators are overloading with the input and output operators in C++.) They are used to shift bits in integral variables left and right. Shifting left is defined to be equivalent to multiplying by 2, modulus the variable's 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. The Java >>> operator does not exist in C/C++ and handles shifting right for positive *and* negative values.

Like Java, assignment in C/C++ is an operator, which is useful for *cascade assignment* to initialize multiple variables of the same type to a common value:

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

Other uses of assignment in an expression are discouraged because changing variables during the evaluation of an expression can cause unknown side-effects. Except for cascade assignment, good programming practice is to have only one assignment in an expression on the left-hand side, i.e., one side-effect after an expression is evaluated. Finally, unlike Java, C/C++ allows any expression to appear as a statement:

```
3;    j + i;    ( i + j ) * ( k + j );    sin(x);
```

Other assignment operators, such as lhs += rhs, are implicitly rewritten as:

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

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

```
v[ rand() % 5 ] += 1;       // only calls random once
v[ rand() % 5 ] = v[ 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 is a series of expressions separated by commas; the expressions are evaluated left to right with the value of the rightmost expression returned as the result of the comma expression. The comma expression allows multiple expressions to be evaluated in a context where only a single expression is allowed (see page 21). Note, the earlier dimension problem with b[10, 20] (see page 14) actually means b[20] because 10, 20 is a comma expression not a dimension list. The same problem occurs with subscripting as b[3, 4] means b[4], the 4th row of the matrix.

This tutorial strongly discourages the general use of the increment/decrement operators ++ and -- even though both are standard idioms in Java and C/C++ (see page 54). Having a special operator to increment/decrement by one is largely superfluous and is an anomaly among programming languages. It is more general to use `i += 1` rather than `i++` because the former can be trivially changed to add any amount and the latter cannot.

9.1 Conversion

Conversion is transforming a value from one type to another type, which can be performed implicitly or explicitly (see Section 17.3.2, p. 39). Conversions are divided into two kinds:

- *widening* conversion, no information is lost:

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

- *narrowing* conversion, information can be lost:

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

Java only supports implicit widening conversions; C/C++ support both implicit widening and narrowing conversions. 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 floating-point to integer, but r is assigned 3.5 in the first expression and 3.0 in the second because its value is first narrowed to 3 and then widened to 3.0. *Be careful!*

Like Java, C/C++ support explicit narrow conversions using the *cast* operator. Due to potential loss of information, it is good programming practice in C/C++ to use an explicit narrowing conversion rather than an implicit one:

```
int i;
double x, y;
i = (int) x;              // explicit narrowing conversion
i = (int) x / (int) y;    // explicit narrowing conversions to get integer division
i = (int)(x / y);         // alternative technique
```

C/C++ supports casting among the basic types and user defined types (see Section 17, p. 34).

⇒ Edit file hello.C

⇒ Make the following modification to routine main:

```
int main() {                                    // Ex12
    char c;
    short int si;
    long int li;
    double d;
    d = li = si = c = '\x41';                   // 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, run it, check the output for anomalies, and make sure you understand it.

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 constants in the assignment statements.

As mentioned in Section 8.7, p. 14, g++ has a cast extension allowing construction of structure and array constants in executable statements not just declarations:

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

In both cases, a cast is used to indicate the meaning and structure of the constant.

10 Control Structure

	Java	**C/C++**
block	{ *intermixed decls/stmts* }	{ *intermixed decls/stmts* }
selection	**if** (*bool-expr1*) *stmt1*	**if** (*cond-expr1*) *stmt1*
	else if (*bool-expr2*) *stmt2*	**else if** (*cond-expr2*) *stmt2*
	...	...
	else *stmtn*	**else** *stmtn*
	switch (*integral-expr*) {	**switch** (*integral-expr*) {
	case c1: *stmt1*; **break**;	**case** c1: *stmt1*; **break**;
	...	...
	case cn: *stmtn*; **break**;	**case** cn: *stmtn*; **break**;
	default: *stmt0*;	**default**: *stmt0*;
	}	}
looping	**while** (*bool-expr*) *stmt*	**while** (*cond-expr*) *stmt*
	do *stmt* **while** (*bool-expr*) ;	**do** *stmt* **while** (*cond-expr*) ;
	for (*init-expr*; *bool-expr*; *incr-expr*) *stmt*	**for** (*init-expr*; *cond-expr*; *incr-expr*) *stmt*
transfer	**break** *[label]*	**break**
	continue *[label]*	**continue**
		goto *label*
	return *[expr]*	**return** *[expr]*
label	*label* : *stmt*	*label* : *stmt*

10.1 Block

A *block* is a series of statements bracketed by braces, {...}, which can be nested one within another. (As opposed to a comma expression, see page 16, which only contains expressions.) A block serves two purposes: bracket several statements into a single statement and introduce local declarations. For control structures requiring a statement, a good programming practice is to always use a block as it allows easy insertion and removal of statements to or from that block. Putting local declarations precisely where they are needed can help reduce declaration clutter at the beginning of an outer block; however, it can also make locating them more difficult.

10.2 ⋈ Conditional

Java uses a "boolean" expression in control structures that causes conditional transfer based on the result of the expression, e.g., in **if**, **while**, **do**, and **for** control structures. C/C++ uses a "conditional" expression in the same context, which is evaluated and implicitly tested for not equal to zero, i.e., *cond-expr* ≡ *expr* != 0. Boolean expressions are converted to 0 for **false** and 1 for **true** before comparison to zero, e.g.:

if (x > y) ... implicitly rewritten as **if** ((x > y) != 0) ...

As a result, other expressions are allowed in a conditional, giving the following C/C++ idiom:

if (x) ... implicitly rewritten as **if** ((x) != 0) ...
while (x) ... **while** ((x) != 0) ...

Watch for the common mistake in a conditional:

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

which assigns y to x and tests x != 0.

⇒ Explain the one situation in Java where this mistake also occurs. (Think about the type of the operands.)

10.3 Selection

The C/C++ selection statements, **if** and **switch**, are the same as in Java, except for the difference between boolean and conditional expression (see Section 10.2).

An **if** statement selectively executes one of two alternatives based on the result of a comparison, e.g.:

```
if ( x > y ) max = x;
else max = y;
```

Like Java, C/C++ has the *dangling else* problem of correctly associating an **else** clause with its matching **if** in nested **if** statements. 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.

Dangling Else	Fix Using Null Else	Fix Using Blocks

`if ( sales < 10000 )`
`    if ( sales < 5000 )`
`    income -= penalty;`
`else // incorrect match!!!`
`    income += bonus;`
	`if ( sales < 10000 )`
	`    if ( sales < 5000 )`
	`        income -= penalty;`
	`    else ;    // null statement`
	`else`
	`    income += bonus;`
		`if ( sales < 10000 ) {`
		`    if ( sales < 5000 ) {`
		`        income -= penalty;`
		`    }`
		`} else {`
		`    income += bonus;`
		`}`

The solution using blocks is preferred because it allows easy addition or removal of statements.

A **switch** statement selectively executes one of N alternatives based on matching an integral value with a series of case clauses, e.g.:

```
switch ( day ) {        // integral expression
  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
}
```

Once a case clause is matched, its statements are executed, and control continues to the *next* statement. Like Java, a **break** statement is used at the end of a case clause to exit the **switch** statement. (It is a common error to forget the **break**.) If no case clause is matched and there is a **default** clause, its statements are executed, and control continues to the *next* statement; otherwise, the **switch** statement does nothing. Only one label is allowed for each **case** clause but a list of **case** clauses is possible.

10.4 Conditional Expression Evaluation

Conditional expression evaluation is used to perform partial evaluation of expressions. These are control structures, not true operators because both operands may not be evaluated, as for real 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 conjunctive 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 both operands. In many situations with boolean operands, the corresponding operators are interchangeable.

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

```
int main() {                                          // Ex13
    if ( ( cout << "a", true ) | ( cout << "b", true ) ) cout << endl; // note, comma expression
    if ( ( cout << "a", true ) || ( cout << "b", true ) ) cout << endl; // note, comma expression
}
```

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

Conditional ?: 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 file hello.C

⇒ Make the following modification to routine main:

```
int main() {                                          // Ex14
    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 Looping

The C/C++ looping statements are the same as in Java, except for the difference between boolean and conditional expression (see Section 10.2, p. 18). A **while** statement executes its statement zero or more times, a **do** statement executes its statement one or more times, and a **for** statement is a specialized **while** statement for iterating using an index. As for Java, beware of accidental infinite loops:

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

⇒ Edit file hello.C

⇒ Make the following modification to routine main:

```
int main() {              // Ex15
    int val = 1;          // initialize
    while ( val ) {       // conditional
        cout << val << endl;
        val <<= 1;        // shifting left 1 bit position
    }
}
```

⇒ Compile the program and run it.

⇒ Explain the last value printed and why the loop stopped. (Think of the internal representation of integer values and how a conditional works.)

The **for** statement is a specialized **while** statement with an index:

```
init-expr;
while ( cond-expr ) {         for ( init-expr; cond-expr; incr-expr ) {
    stmt;                         stmt;
    incr-expr;
}                             }
```

There are many ways to use the **for** statement to construct iteration:

```
for ( i = 1; i <= 10; i += 1 ) {          // count up
    // loop 10 times
} // i has the value 11 on exit
for ( i = 10; 1 <= i; i -= 1 ) {          // count down
    // loop 10 times
} // i has the value 0 on exit
```

```
for ( p = l; p != NULL; p = p->link ) {      // pointer index
    // loop through list structure
} // p has the value NULL on exit
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 (see page 16) to initialize and increment 2 indices in a context where normally only a single expression is allowed. While the loop control variable can be modified in the loop body, it is discouraged. A default value of **true** is inserted if no conditional is specified for a **for** statement.

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

⇒ Edit file hello.C

⇒ Make the following modification to routine main:

```
int main() {                              // Ex16
    int val;
    for ( val = 1; val; val <<= 1 ) {
        cout << val << endl;
    }
}
```

⇒ Compile the program and run it.

A common example of short-circuit expression evaluation (see Section 10.4, p. 19) 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 worry about array subscript problems, even when C/C++ does not perform subscript checking? The reason is that the invalid subscript can result in other errors, such as addressing outside the program's memory, which is called a *segment fault*.

Finally, the **continue**/**break** statements can be used in all iteration constructs to cause immediate advancement to the next loop iteration or termination of the loop construct, respectively. 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 cannot occur. Unlike Java, C/C++ does not support labelled **continue**/**break** for transferring among multiple levels of nested control structures.

The **goto** statement is not discussed (see a C++ textbook), but it is similar to the Java labelled **break** and **continue**.

11 ⋈ Preprocessor

The preprocessor manipulates the text of the program *before* compilation (see Section 6.2, p. 6). 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, p. 4) to print the output of the preprocessor to understand why the compiler is generating error messages.

The three most commonly used preprocessor facilities are substitution, file inclusion, and conditional 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 spaces, and then a preprocessor statement; *no semi-colon*!

11.1 Substitution

The **#define** statement declares a preprocessor variable, and its value is all the text after the name up to the end of line.

⇒ Edit file hello.C

⇒ Enter the following program (**including comments**):

```
#define Integer int              // Ex17
#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: }
```

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

The initial lines starting with **#** inform the compiler of the source-file name and other information 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 are removed. Finally, the preprocessed statements appear, without comments, which the compiler sees and compiles.

As the example shows, 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. Finally, it is possible to define and initialize preprocessor variables from the compilation command (see Section 4, p. 4):

```
g++ -Dxxx=2 -Dyyy ... source-file1.C
```

which creates two preprocessor: variables xxx, which is initialized to 2, and yyy, which is uninitialized; both variables exist in the compilation of each source file in the compilation command. Finally, a C/C++ compiler may have predefined preprocessor-variables identifying the kind of hardware the compiler is generating code for, e.g., variable mcpu is assigned the kind of CPU.

Traditionally, textual substitution was used to give names to constants; this is better done using **const** declarations (**final** in Java):

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

#define can also be used to declare macros with parameters, which expand inline during compilation, textually substituting arguments for parameters, 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/C++ (see a C++ textbook for details).

11.2 File Inclusion

File inclusion is used to copy a block of text from a file into a C/C++ program; an included file may contain anything. In effect, file inclusion is a shorthand for retyping the same text into a program. Most commonly, an include file contains preprocessor and C/C++ declarations for library routines used in a program. All included text goes through every compilation step, i.e., preprocessor, compiler, etc. Java does implicit inclusion by matching class names with file names in CLASSPATH directories, and extracting and including necessary declarations.

The **#include** statement specifies the file to be included. C convention uses the suffix ".h" for include files containing C declarations; C++ convention drops the suffix ".h" for its standard libraries and uses special file names for equivalent C files (e.g., cstdio versus stdio.h).

```
#include "user.h"
#include <system.h>     // C style
#include <system>       // C++ style
```

The file name can be enclosed in "" or <>. "" means 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 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 similar to:

```
#include <...> search starts here:
/usr/include/c++/3.3
/usr/include/c++/3.3/i486-linux
/usr/include/c++/3.3/backward
/usr/local/include
/usr/lib/gcc-lib/i486-linux/3.3.5/include
/usr/include
```

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

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

⇒ Edit 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.

11.3 Conditional Inclusion

The preprocessor has an **if** statement, which may be nested, to conditionally add/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 character values (no float or string values).

```
#define DEBUG 0             // declare and initialize preprocessor variable
...
#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 included into the compilation.

A simple 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 an **#include** file to ensure its contents are only expanded into a program once (see Section 24, p. 56). 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.

12 Input/Output

Input/Output (I/O) is divided into two kinds: formatted and unformatted. Formatted I/O transfers data with implicit conversion of internal values to/from human-readable form; conversion is based on the type of variables and format codes. Unformatted I/O transfers data without conversion, e.g., internal integer and floating-point values. Only formatted I/O is discussed as it is the most common (see a C++ textbook for unformatted I/O).

C++ provides one kind of formatted I/O library and C provides another. While C++ can use both libraries, only the C++ library is discussed in detail (see a C textbook for its I/O library).

Java	C	C++
File, Scanner PrintStream	FILE FILE	ifstream ofstream
Scanner in = **new** Scanner(**new** File("f"))	fopen("f", "r");	ifstream in("f");
PrintStream out = **new** PrintStream("g")	out = fopen("g", "w")	ofstream out("g")
in.close()	close(in)	scope ends
out.close()	close(out)	scope ends
nextInt()	fscanf(in, "%d", &i)	in >> T
nextFloat()	fscanf(in, "%f", &f)	
nextByte()	fscanf(in, "%c", &c)	
next()	fscanf(in, "%s", &s)	
hasNext()	feof(in)	in.eof()
hasNextT()	fscanf return value	in.fail()
		in.clear()
skip("regexp")	fscanf(in, "%*[regexp]")	in.ignore(n, c)
out.print(String)	fprintf(out, "%d", i)	out << T
	fprintf(out, "%f", f)	
	fprintf(out, "%c", c)	
	fprintf(out, "%s", s)	

Formatted I/O occurs to/from a *stream file* in both Java and C++. Java has three implicit stream files in class System: in, out and err, which are automatically declared and opened; similarly, C++ has equivalent stream files: cin, cout and cerr, which are automatically declared and opened. (C uses stdin, stdout and stderr.) As in previous examples, the system include-file iostream provides all necessary declarations to use stream files cin, cout and cerr. Like Java streams in and out, C++ stream cin normally reads input from the keyboard (unless redirected by the shell), and cout writes to the terminal screen (unless redirected by 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 always be written to* cerr *because it is normally not redirected by the shell, but more importantly, it is unbuffered so output appears immediately.*

To use stream files other than the 3 implicit ones requires declaring each file *object*:

```
#include <fstream>  // required for explicit stream-file declarations
ifstream infile( "myinfile" );      // input file
ofstream outfile( "myoutfile" );    // output file
```

The include file fstream is necessary for declaring stream files. Like Java, 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 fail, e.g., infile.fail(), which returns **true** if the open failed and **false** otherwise (see Figure 1, p. 35). Like Java, the type of the file, ifstream or ofstream, indicates whether the file can be read or written. The connection between the file name in the program and the actual operating-system file-name is done at the declaration. 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.

The C++ I/O library uses overloading (see Section 19, p. 44) with operators << and >> to perform I/O (also used for bit shift, see page 16). C I/O library uses fscanf(outfile,. . .) and fprintf(infile,. . .), which have short forms scanf(. . .) and printf(. . .) for stdin and stdout. Parameters in C are always passed by value (see Section 14.1, p. 30), so arguments to fscanf must be preceded with & (except arrays) so they can be changed. Both I/O libraries can cascade multiple I/O operations, i.e., input or output multiple values in a single expression.

12.1 Input

Java formatted input requires *explicit* specification of character conversion for all basic types using a Scanner attached to an input file. C/C++ formatted input has *implicit* character conversion for all basic types and is extensible to user-defined types. Valid input values for a stream file are C/C++ constants: 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, p. 31 for reading entire lines). As mentioned, the >> operator is overloaded to work with different types of operands. The type of the operand indicates the kind of constant expected in the stream file, e.g., an integer operand means an integer constant is expected. Stream cin starts reading where the last cin left off. When all the input values on the current line are read, cin proceeds to the next line. Hence, the placement of input values on lines of a file is often arbitrary.

Unlike Java, C/C++ must attempt to read *before* end-of-file is set and can be tested for. End of file can be detected in two ways: cin and fscanf return 0 and EOF when eof is reached, respectively; C++ member eof and the C routine feof return true when eof is reached.

Java	C	C++

import java.io.*;
import java.util.Scanner;
Scanner in = **new** Scanner(**new** File("f"));
PrintStream out = **new** PrintStream("g");
int i, j;
while (in.hasNext()) {
   i = in.nextInt(); j = in.nextInt();
   out.println("i:" + i + " j:" + j);
}
in.close();
out.close();
	#include <stdio.h>
	FILE *in = fopen("f", "r");
	FILE *out = fopen("g", "w");
	int i, j;
	for (;;) {
	   fscanf(in, "%d%d", &i, &j);
	 if (feof(in)) **break**;
	   fprintf(out, "i:%d j:%d\n", i, j);
	}
	close(in);
	close(out);
		#include <fstream>
		ifstream in("f");
		ofstream out("g");
		int i, j;
		for (;;) {
		   in >> i >> j;
		 if (in.eof()) **break**;
		   out << "i:" << i
		        << "j:" << j << endl;
		}
		// in/out closed implicitly

Note, there is no end-of-file character; end-of-file is the detection of the physical end of a file. When reading from the keyboard, a special indicator is required to cause the shell to close the current input file marking its physical end. The indicator, normally <ctrl>-d (press the <ctrl> and d keys simultaneously), is a signal to the shell and *not* read by cin.

⇒ Edit file hello.C
⇒ Enter the following program:

```
#include <iostream>                    // Ex18
using namespace std;
int main() {
    int n;
    for ( ;; ) {
        cout << "Enter a number: ";
        cin >> n;
      if ( cin.eof() ) break;          // eof ?
        if ( ! cin.fail() ) {          // number ?
            cout << "n = " << n << endl;
        } else {
            cout << "Not a number. ";
            cin.clear();               // reset stream failure
            cin.ignore( numeric_limits<int>::max(), '\n' );  // skip until newline
        }
    }
    cout << endl;
}
```

⇒ Compile and run the program, entering some integer and non-integer values.
⇒ End the input and the program by entering <ctrl>-d

After reading, it is possible to check for a successful read using the stream routine fail, e.g., cin.fail(), which returns **true** if the read failed and **false** otherwise. After an unsuccessful read, a call to clear() is necessary to reset the stream. The ignore member skips either n characters, e.g., cin.ignore(5) or until a specified character, as above.

12.2 Output

Java output style converts values to strings, concatenates these strings, and prints the final long string:

```
System.out.println( i + " " + j );     // build a string and print it
```

Whereas, C/C++ output style supplies a list of formats and values, and the output operation generates the strings:

```
cout << i << " " << j << endl;          // print each string as it is formed
```

As such, there is no implicit conversion from the basic types to string in C++ (but one can be constructed). While it is possible to use the Java string-concatenation style in C++, it is an incorrect style.

Many examples of output have already been presented, so the discussion here is on how to control the format of output (and input). The main mechanism to control input/output format is via *manipulators*, which appear in a cascaded input/output expression and apply to all constants/variables after it (except for setw). The following manipulators are available by including iomanip:

oct	print values in octal
dec	print values in decimal
hex	print values in hexadecimal
left / right (default)	print values with padding after / before values
boolapha / noboolapha (default)	print bool values as false/true instead of 0/1
showbase / noshowbase (default)	print values with / without prefix 0 for octal & 0x for hex
fixed (default) / scientific	print float-point values without / with exponent
setprecision(N)	print fraction of float-point values in maximum of N columns
setw(N)	print NEXT VALUE ONLY in minimum of N columns
setfill('ch')	padding character before/after value within a fixed width (default blank)
endl	flush current output buffer and start a new line (output only)
skipws (default) / noskipws	skip whitespace characters (input only)

Note, endl is not the same as '\n'; only the former is guaranteed to flush the buffer for interactive output.

⇒ Edit file hello.C
⇒ Enter the following program:

```
#include <iostream>              // Ex19
#include <iomanip>               // manipulators
using namespace std;
int main() {
    bool b = true;
    int i = 27;
    double d = 3.5;
    char c = 'a';
    char s[] = "abc";
    cout << showbase << right << boolalpha << setprecision(2)
         << " b:" << b
         << " i:" << setw(3) << i
         << " d:" << fixed << setw(7) << d
         << " i:" << i
         << " c:" << c
         << " d:" << d
         << " s:" << s
         << oct << " i:" << setw(5) << i
         << " i:" << i
         << endl;
}
```

⇒ Compile and run the program.
⇒ Try some of the other manipulators to vary the format of the output. e.g., change right to left and fixed to scientific, add some other manipulators in different places, etc.

Notice manipulator setw only applies to the next value in the I/O expression while the other manipulators apply to all values after it and even to the next I/O expression for a specific stream file.

13 ⋈ Dynamic Storage Management

C++ operator **new** is like Java **new**; both take a type operand and return a pointer to new storage of that type allocated in an area called the *heap*. Unlike Java, C/C++ allow *all* types to be dynamically allocated not just object types, e.g., **new int**. However, C/C++ do not have *garbage collection* of dynamically allocated storage after the variables using it no longer need it; therefore, there is an additional dynamic storage-management operation to *free* storage. C++ provides one kind of dynamic storage-management operations, **new**/**delete** and C provides another, malloc/free (see a textbook for the C form). *Do not mix the two forms in a C++ program.*

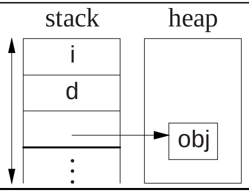
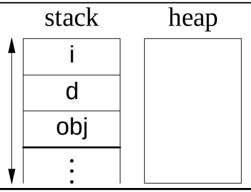
Java	**C/C++**
`class foo {`	`struct foo {`
`    char a, b, c;`	`    char a, b, c;`
`}`	`};`
`class test {`	
`    public static void main( String[] args ) {`	`int main() {`
`        foo f = new foo();`	`    foo *f = new foo();    // optional parenthesis`
`        f.c = 'R';`	`    f->c = 'R';`
`    }`	`    delete f;              // explicitly free storage`
`}`	`}`

In C++, the parenthesis after the type name in the **new** operation are optional. As well, once storage is no longer needed it must be explicitly deleted as there is no implicit garbage collection. After storage is deleted, it should not be used:

```
delete f;
f->c = 'S';        // result of dereference is undefined
```

Unlike Java, aggregate types can be allocated on the stack, i.e., local variables of a block:

Java		**C++**	

`{ // basic & reference types only`
`    int i;`
`    double d;`
`    ObjType obj = new ObjType();`
`    . . .`
`} // obj garbage collected`
		`{ // all types`
		`    int i;`
		`    double d;`
		`    ObjType obj;`
		`    . . .`
		`} // obj implicitly deleted`
			

Because stack allocation is more efficient than heap allocation and does not require explicit storage management, use it whenever possible; hence, *there is significantly less dynamic allocation in C++* . In general, dynamic allocation in C++ should be used only when:

- a variable's storage must outlive the block in which it is allocated:

  ```
  ObjType *rtn(. . .) {
      ObjType *obj = new ObjType();
      . . . // use obj
      return obj; // storage outlives block
  } // obj deleted later
  ```

  The storage for variable obj is passed *outside* of the block associated with a call to rtn, and hence, its storage must outlive the block in which it is created.

- when each element of an array of objects needs initialization (see Section 17.3, p. 37):

  ```
  ObjType *v[10]; // array of object pointers
  for ( int i = 0; i < 10; i += 1 ) {
      v[i] = new ObjType( i ); // each element has different initialization
  }
  ```

Declaration of a pointer to an array is complex in C/C++; *pay special attention*. Because C/C++ do not maintain array-size information, the dimension value for an array pointer is often unspecified:

```
int *arr = new int[10];     // think arr[], pointer to an array with 10 elements
```

The Java notation:

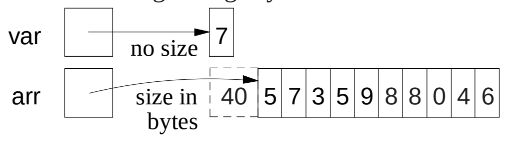
```
int arr[ ] = new int[10];
```

cannot be used because **int** arr[ ] is actually rewritten as **int** arr[N], where N is the size of the initializer value (see Section 8.7, p. 14). Note, the lack of dimension information for an array means there is no subscript checking.

As well, no dimension information results in the following ambiguity:

```
int *var = new int;

int *arr = new int[10];   // think arr[]
```



Here, variables var and arr have the same type but one is an array, which poses a problem when deleting a dynamically allocated array. To solve the problem, special syntax is used to distinguish these cases:

```
delete var;       // single element
delete [] arr;    // multiple elements
```

The second syntax indicates the variable has multiple elements (but unknown number and size of dimensions) and the total array-size is stored with the array for deletion purposes.

⇒ What do you think happens if you forget to put [] when deleting an array?

Never do this:

```
delete [] arr, var; // => (delete [] arr), var;
```

which is an incorrect use of a comma expression; var is not deleted.

⇒ Edit file hello.C

⇒ Make the following modification to routine main:

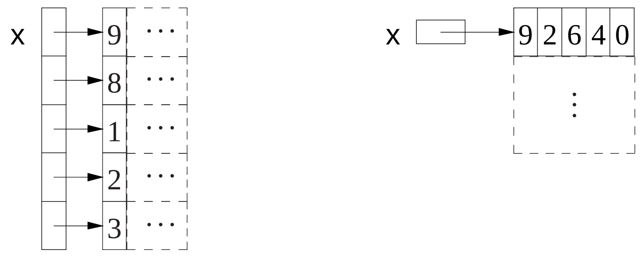
```
int main() {                                    // Ex20
    int i, size;
    cin >> size;                                // read array dimension
    int vals[size];                             // g++ only
    for ( i = 0; i < size; i += 1 ) {           // read values
        cin >> vals[i];
    }
    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.

Declaration of a pointer to a matrix is equally complex in C/C++. The matrix declaration **int** *x[5] could mean:



On the left is an array of 5 pointers to an array of unknown number of integers, and the right is a pointer to a matrix of unknown number of rows with 5 columns of integers. The question is whether the * or [] is applied first. In fact, dimension has higher priority (as for subscript, see Section 9, p. 15), so the declaration is interpreted as **int** (*(x[5])) (left example), where parenthesis indicate the ordering of the type qualifiers. In general, to read a C/C++ 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:

```
int *(m1[5]);     // array of 5 pointers to array of unknown number of integers
int (*m2)[5];     // pointer to a matrix of unknown number of rows and 5 columns
```

⇒ Write out in words the meaning of this declaration: **int** (*(x[5]))[10].
Answer: *array of 5 pointers to array of 10 integers*

Unfortunately, only the left example (above) of declaring a matrix can be generalized to allow a dynamically-sized matrix; the right example cannot be generalized because the second dimension must be a constant.

⇒ Edit file hello.C

⇒ Make the following modification to routine main:

```
int main() {                                   // Ex21
    int *m[5];                                 // 5 rows
    for ( int r = 0; r < 5; r += 1 ) {
        m[r] = new int[4];                     // 4 columns per row
        for ( int c = 0; c < 4; c += 1 ) {     // initialize matrix
            m[r][c] = r + c;
        }
    }
    for ( int r = 0; r < 5; r += 1 ) {         // print matrix
        for ( int c = 0; c < 4; c += 1 ) {
            cout << m[r][c] << ", ";
        }
        cout << endl;
    }
    for ( int r = 0; r < 5; r += 1 ) {
        delete [] m[r];                        // delete each row
    }
}                                              // implicitly delete array "m"
```

⇒ Compile and test the program.

14 ⋈ Routine

As mentioned in Section 8.6.3, p. 13, C++ provides aggregation and routines separately (versus combined in an object), e.g., routine main is not defined in a type. The general form of a routine is:

C	C++

```
void p(   or   T f( // parameters
    T1 a      // pass by value



    )
{   // routine body
    // intermixed decls/stmts
}
```
	```
	void p(   or   T f( // parameters
	    T1 a,       // pass by value
	    T2 &b,      // pass by reference
	    T3 c = 3    // optional, default value
	    )
	{   // routine body
	    // intermixed decls/stmts
	}
	```

Like Java, C/C++ divides routines into a *procedure* or a *function* based on the existence of a return type at the beginning of the routine. A procedure is a routine not returning a value, 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
```

Like Java, routines in C/C++ cannot be nested in other routines, so all routine names are at the same scope level in a source file. Therefore, the only routine scope is between the global scope of the source file and a routine body:

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

⇒ Edit file hello.C

⇒ Make the following modification to routine main:

```
int i = 3;              // Ex22
int main() {
    cout << i << endl;
    int i = 4;
    cout << i << endl;
}
```

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

Like Java, a C/C++ procedure terminates when either control runs off the end of the routine body or a **return** statement is executed; a function terminates when a **return** statement is executed.

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

A **return** statement can appear anywhere in a routine body, and multiple return statements are possible.

⇒ Edit file hello.C

⇒ Make the following modification to routine main:

```
int main() {          // Ex23
    return 7;         // return value 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.

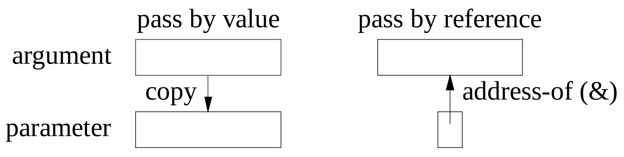
While it is possible to return the address of a local variable:

```
int *rtn() {
    int n;
    return &n;
}
```

the use of the returned pointer is undefined because the local storage for n is implicitly freed when the routine returns.

14.1 Argument/Parameter Passing

The two most common forms of *parameter passing* are value and reference. In *value passing*, the parameter is initialized by the argument (often by a bit-wise copy). In *reference passing*, the parameter is a reference to the argument and is initialized to the argument's address.



In Java and C, parameter passing is by value, i.e., basic types and object references are copied. In C++, parameter passing is by value or reference depending on the type of the parameter. For C/C++, when a routine is called, all the expressions in the argument list are evaluated *in any order* (see Section 9, p. 15), then the routine's local variables, including parameters, are allocated on the stack. For value parameters, each argument-expression result is used to initialize the corresponding parameter, *which may involve an implicit conversion*. For reference parameters, each argument-expression result is referenced (address of) and this address is assigned to the corresponding parameter.

⇒ Edit file hello.C

⇒ Enter the following program:

```
#include <iostream>                    // Ex24
using namespace std;
struct complex { double r, i; };
void r( int i, int &ri, complex c, complex &rc ) {
    ri = i = 3;
    rc = c = (complex){ 3.0, 3.0 };
}
int main() {
    int i1 = 1, i2 = 2;
    complex c1 = { 1.0, 1.0 }, c2 = { 2.0, 2.0 };
    r( i1, i2, c1, c2 );
    cout << i1 << " " << i2 << " " << endl
         << c1.r << " " << c1.i << " " << c2.r << " " << c2.i << endl;
}
```

⇒ Compile the program and run it.

⇒ Explain why the arguments passed by value are not changed, while arguments passed by reference are changed.

⇒ Change the routine call to r(i1, i1+i2, c1, c2).
⇒ Compile the program and explain the error message (see Section 8.6.2, p. 10).

Value passing is most efficient for basic and small structures because the values are accessed directly in the routine (versus indirectly through a reference). Reference passing is most efficient for large structures and arrays because the values are not duplicated in the routine.

Type qualifiers can be used to create 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, the security of pass by value because the argument cannot change, and allows temporary variables and constants as arguments:

```
void r( const int &i, const complex &c, const int v[5] ) {
    i = 3;          // assignments disallowed, read only!
    c.r = 3.0;
    v[0] = 3;
}
r( i + j, (complex){ 1.0, 7.0 }, (int [5]){ 3, 2, 7, 9, 0 } );  // allow temporary variables and constants
```

The reason v is not declared a reference parameter is discussed in Section 14.2.

Unlike Java, 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. In a call, once an argument is omitted for a parameter with a default value, no more arguments can be specified to the right of it.

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

14.2 Array Parameter

Like Java, array copy is unsupported (see Section 8.6, p. 9) so arrays cannot be passed by value only by reference. Therefore, all array parameters are implicitly reference parameters, and hence, the reason why parameter v above does not have a reference symbol. Interestingly, a parameter declaration can specify the first dimension with a dimension value, [10] (where the dimension is ignored), an empty dimension list, [ ], or a pointer, *; the declarations within each row are equivalent:

```
double sum( double v[5] );     double sum( double v[ ] );     double sum( double *v );
double sum( double *m[5] );    double sum( double *m[ ] );    double sum( double **m );
```

Good programming practice uses the middle form because it clearly indicates the variable is going to be subscripted. Note, only a formal (parameter) declaration can use the empty dimension; an actual declaration must use *:

```
double sum( double v[ ] ) {  // formal declaration
    double *cv;                  // actual declaration, think cv[]
    cv = v;                      // address assignment
```

Given the above declarations, it is possible to write a routine to add up the elements of an arbitrary-sized array or matrix by passing the dimensions explicitly:

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

```
double sum( int rows, int cols, double *m[ ] ) {
    int total = 0.0;
    for ( int r = 0; r < rows; r += 1 )
        for ( int c = 0; c < cols; c += 1 )
            total += m[r][c];
    return total;
}
```

15 String

Strings are supported in C by a combination of language and library facilities. The language facility ensures all string constants are terminated with a character value '\0'. For example, the string constant "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 constant is a sentinel value used by the C string routines to locate the end of a character string by searching through the individual characters for '\0'. Unfortunately, this approach suffers from three drawbacks. First, a string cannot contain a character with the value '\0' as that character immediately marks the end of the string. Second, string operations needing the length of a string must perform a linear search for the character '\0', which is expensive for long strings. Third, the management of variable-sized strings is the programmer's responsibility, making complex string operations a storage management problem.

Like Java, C++ solves these problems by providing a string type using a length member at the beginning of each string and managing all of the storage for the variable-sized strings. Unlike Java, instances of the C++ string type are not constant; values can change so a companion type like StringBuffer in Java is unnecessary. While C++ can use both C and C++ strings, only C++ strings are discussed (see a C textbook for C strings). The most important point to remember about a string value is that it can vary in length dynamically, and powerful operations are available to manipulate the characters of the string and search through them. *Therefore, it is seldom necessary to iterate through the characters of a string variable.*

Java String methods	**C char** [] routines	**C++** string members
	strcpy, strncpy	=
+, concat	strcat, strncat	+
compareTo	strcmp, strncmp	==, !=, <, <=, >, >=
length	strlen	length
charAt	[]	[]
substring		substr
replace		replace
indexOf, lastIndexOf	strstr	find, rfind
	strcspn	find_first_of, find_last_of
	strspn	find_first_not_of, find_last_not_of

All of the C++ string find members return string::npos if a search is unsuccessful.

```
string a, b, c;                   // declare string variables
cin >> c;                         // read white-space delimited sequence of characters
getline( cin, c, '\n' );          // read remaining characters until newline (newline is default)
cout << c << endl;                // print string
a = "abc";                        // set value, a is "abc"
b = a;                            // copy value, b is "abc"
c = a + b;                        // concatenate strings, c is "abcabc"
if ( a == b )                     // compare strings, lexigraphical ordering
string::size_type 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", must be character constant
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"
string::size_type p = c.find( "ax" ); // search for 1st occurrence of string "ax", p is 3
p = c.rfind( "ax" );              // search for last occurrence of string "ax", p is 5
p = c.find_first_of( "aeiou" );   // search for first vowel, p is 0
p = c.find_first_not_of( "aeiou" ); // search for first consonant (not vowel), p is 1
p = c.find_last_of( "aeiou" );    // search for last vowel, p is 5
p = c.find_last_not_of( "aeiou" ); // search for last consonant (not vowel), p is 7
```

⇒ Edit file hello.C
⇒ Enter the following program:

```
#include <iostream>                           // Ex25
#include <string>
using namespace std;
int main() {
    string line, word;
    string::size_type p, words = 0;
    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 == string::npos ) 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 -w hello.C

16 Shell Argument

Up to now, 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/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. Because the executable file-name is included, *the count is one greater than in Java*. argv is an array of pointers to the character strings that make up the arguments. For example, if the executable is called in the following way:

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

the arguments to main have the values:

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

Notice, the call of main by the shell is inconsistent with a normal routine call in C/C++ because the arguments are passed as strings not values of or references to variables. Hence, a shell argument of "32" may have to converted to an integer.

Like Java, routine main usually begins by checking argc for shell arguments. But unlike Java, the C/C++ arguments are processed in the range argv[1] through argv[argc-1], i.e., starting one greater than Java.

Java	C/C++

`class prog {`
`    public static void main( String[] args ) {`
`        switch ( args.length ) {`
`          case 0: . . .     // no args`
`            break;`
`          case 1: . . . args[0] . . . // 1 arg`
`            break;`
`          case . . .        // others args`
`            break;`
`          default: . . .    // usage message`
`            System.exit( -1 );`
`        }`
`        . . .`
	
	`int main( int argc, char *argv[] ) {`
	`    switch( argc ) {`
	`      case 1: . . .     // no args`
	`        break;`
	`      case 2: . . . args[1] . . . // 1 arg`
	`        break;`
	`      case . . .        // others args`
	`        break;`
	`      default: . . .    // usage message`
	`        exit( -1 );`
	`    }`
	`    . . .`

⇒ Edit file hello.C

⇒ Enter the program in Figure 1(b) but modify it so the input file is optional and defaults to cin if unspecified.

⇒ Test your program to ensure it is correct.

17 Object

Object-oriented programming is not a new programming methodology; it was developed in the mid-1960s by Dahl and Nygaard and first implemented in a programming language called SIMULA. The following is a short review of the notion of an object.

Objects are based on the notion of a structure, used for organizing logically related data (see Section 8.6.3, p. 13):

unorganized	organized

`int people_age[30];`
`bool people_sex[30];`
`char people_name[30][50];`
	`struct person {`
	`    int age;`
	`    bool sex;`
	`    char name[50];`
	`} people[30];`

Notice, both code fragments create an identical amount of information; the difference is solely in the way the information is organized (and laid out in memory). 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 a programming language.

The organizational capabilities of the structure are 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. Note, a routine member is a constant, and hence, cannot be assigned (e.g., like a **const** member). The following compares the structure and object form for complex numbers, containing a real and imaginary value.

structure form	object form

`struct complex {`
`    double re, im;`
`};`
`double abs( complex *This ) { // name "This" is arbitrary`
`    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 access? A normal C/C++ routine's scope is the global scope of the source file (see Sections 8.3, p. 7 and 14, p. 29). Interestingly, a

```
/*******************
  Read/Write integers
          java test input-file [ output-file ]

  Example usage:
          java test inputfile
          java test inputfile outputfile
*******************/
import java.io.*;
import java.util.Scanner;

public class test {
    public static void main( String [] args ) {
          Scanner infile = null;
          PrintStream outfile = new PrintStream(System.out);
          int i;

          switch ( args.length ) {
            case 2:
              try {
                   outfile = new PrintStream( args[1] );
              } catch ( FileNotFoundException e ) {
                   System.out.println( "Open failure \""
                                        + args[1] + "\"" );
                   System.exit( -1 );    // TERMINATE!
              }
              // FALL THROUGH
            case 1:
              try {
                   infile = new Scanner( new File( args[0] ) );
              } catch ( FileNotFoundException e ) {
                   System.out.println( "Open failure \""
                                        + args[0] + "\"" );
                   System.exit( -1 );    // TERMINATE!
              }
              break;
            default:
              System.out.println(
                "Usage: input-file [ output-file ]" );
              System.exit( -1 );         // TERMINATE!
          }

          while ( infile.hasNext() ) {
              i = infile.nextInt();

              outfile.println( i );
          }
          infile.close();
          outfile.close();
    }
}
```

(a) Java

```
/*******************
  Read/Write integers
          ./a.out input-file [ output-file ]

  Example usage:
          ./a.out inputfile
          ./a.out inputfile outputfile
*******************/
#include <iostream>      // Ex26
#include <fstream>
using namespace std;

int main( int argc, char *argv[] ) {
    istream *infile;
    ostream *outfile = &cout;
    int i;

    switch ( argc ) {
      case 3:

        outfile = new ofstream( argv[2] );
        if ( outfile->fail() ) {
            cerr << "Open failure \""
                 << argv[2] << "\"" << endl;
            exit( -1 );          // TERMINATE!
        } // if
        // FALL THROUGH
      case 2:

        infile = new ifstream( argv[1] );
        if ( infile->fail() ) {
            cerr << "Open failure \""
                 << argv[1] << "\"" << endl;
            exit( -1 );          // TERMINATE!
        } // if
        break;
      default:
        cerr << "Usage: " << argv[0] <<
          " input-file [output-file]" << endl;
        exit( -1 );              // TERMINATE!
    }

    for ( ;; ) {
        *infile >> i;
      if ( infile->eof() ) break;
        *outfile << i;
    }
    delete infile;
    if ( outfile != &cout ) delete outfile;
}
```

(b) C++

Figure 1: Processing Shell Arguments

structure also creates a scope, and therefore, a routine member can access the structure members. 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 the global scope. A simple model for understanding scoping is that each routine member is implicitly pulled out of the structure and rewritten as a routine that takes 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. So except for the syntactic differences, the two forms are identical.

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 does abs have no arguments in the call; where does abs get a parameter to calculate a result? The answer is 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 file complex.C **(Note the name change for the source file.)**

⇒ Enter the following program:

```
#include <iostream>              // Ex27
#include <cmath>                 // needed to use routine sqrt (square root)
using namespace std;
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 member values. Like Java, use of the implicit parameter **this** is seldom necessary 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.

17.1 ⋈ Operator Members

The previous syntax for adding complex values does not look like adding integer or floating-point values, where the built-in operator + is used. In C++, it is possible to use operator symbols for routine names:

```
struct complex {
    . . .
    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++ also allows a call to an operator member to be written using infix notation, and rewrites this notation back to member selection notation; thus, x + y is allowed and implicitly rewritten as x.**operator**+(y).

⇒ Edit file complex.C

⇒ Make the following modifications to complex and main:

```
struct complex {                    // Ex28
    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.

17.2 Nesting

C++ supports syntactic nesting of object types, but unlike Java *the nesting does not imply scoping*:

```
struct foo {
    int g;
    int r(. . .) { . . . }
    struct bar {                        // nested object type
        int s(. . .) { g = 3; r(. . .); }   // references to g and r fail
    };
};
```

In effect, C++ flattens structure scoping. As a result, the references in routine s to members g and r in foo fail because there is no scope relationship between types bar and foo. Because nested syntax is allowed but there is no scoping, it is discouraged except for controlling visibility for types (see Section 23, p. 55), such as:

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

The enumeration Colour is nested in foo to control visibility, and references to it outside the object must be qualified with "foo::". Note, the new type operator "::" for the qualification. Unlike Java, the C++ selection operator ".", e.g., foo.Colour, is inappropriate because it requires an object instance not a type.

17.3 Constructor

A *constructor* is a special member used to perform *initialization* after object allocation to ensure the object is in a valid state before use. Constructors are called implicitly at local declaration of a variable, dynamic allocation of a variable, and creation of a parameter variable for a routine call. Unlike Java, C++ does not initialize all object members to default values. When a C++ constructor executes, the constructor is responsible for all necessary initializing of it members not already initialized via other constructors. Because a constructor is a routine, arbitrary execution can be performed (e.g., loops, routine calls, etc.) to perform initialization.

Like Java, the name of a constructor is unusual because it is overloaded with the type name of the structure in which it is defined. A constructor may have parameters but does not have a return type (not even **void**). The constructor without parameters is called the *default constructor*.

Java	C++

class complex {
    double re, im;
    complex() { re = 0.; im = 0.; }
    . . . *// other fields and methods*
};
	struct complex {
	    double re, im;
	    complex() { re = 0.; im = 0.; } *// default constructor*
	    . . . *// other members*
	};

When present, the default constructor is implicitly called after storage is allocated for a variable:

```
complex x;                                          complex x;  x.complex();
complex *y = new complex;   implicitly rewritten as complex *y = new complex;  y->complex();
```

When declaring a local object in C++, never put parenthesis to invoke the default constructor:

```
complex x();      // x is a routine taking no parameters and returning a complex
```

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

```
complex x = { 3.2 };          // disallowed
complex y = { 3.2, 4.5 };     // disallowed
```

Like Java, this form of initialization is replaced using 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; }
    . . .
};
```

Unlike Java, constructor argument(s) can be specified *after* a variable for local declarations:

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

This syntax is used in Section 12, p. 23 for declaring stream files, e.g., ifstream infile("myinfile"). The more familiar Java dynamic allocation is:

```
complex *x = new complex(); // parenthesis optional
complex *y = new complex(1.0);
complex *z = new complex(6.1, 7.2);
```

Unlike Java, a C++ constructor cannot be called explicitly at the start of another constructor, so constructor reuse must be done through a separate member:

Java	C++

class foo {
    int i, j;
 	
    foo() { **this**(2); } *// explicit constructor call*
    foo(**int** p) { i = p; j = 1; }
}
	struct foo {
	    int i, j;
	    void common(**int** p) { i = p; j = 1; }
	    foo() { common(2); }
	    foo(**int** p) { common(p); }
	};

17.3.1 Constant

Constructors can also be used to create object constants, like g++ type-constructor constants in Section 9.1, p. 17:

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

In fact, the previous operator + for complex (see page 37) has to be changed because type-constructor constants are disallowed for a type with constructors; the change is to use a complex constructor to create the return value:

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

17.3.2 Conversion

By default, constructors are used to perform implicit conversions (see Section 9.1, p. 17):

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

which is a powerful feature allowing built-in constants and types to interact seamlessly with user-defined types. Note, two implicit conversions are performed on variable i in x + i: **int** to **double** and then **double** to complex. Implicit conversion via a constructor is turned off by qualifying it with **explicit**:

```
struct complex {
    . . .
    explicit complex( double r ) { re = r; im = 0.; }   // turn off implicit conversions
    explicit complex( double r, double i ) { re = r; im = i; }
    . . .
};
```

While implicit conversion allows built-in constants and types to be used directly with user defined types, it fails for commutative binary operators. For example, 3.2 + x, fails because it is conceptually rewritten as (3.0).**operator**+(x), and there is no member **double operator**+(complex) in the built-in type **double**. To solve this problem, the operator + is moved out of the object type and made into a routine, which can also be called in infixed form:

```
struct complex { . . . };    // same as before, except operator + removed
complex operator+( complex a, complex b ) { // 2 parameters
    return complex( a.re + b.re, a.im + b.im );
}
x + y;                                            +(x, y)
3.0 + x;            implicitly rewritten as       +(complex(3.0), x)
x + 3.0;                                          +(x, complex(3.0) )
```

The compiler first checks for an appropriate operator defined in the object, and if found, applies conversions only on the second operand. If there is no appropriate operator in the object type, the compiler checks for an appropriate routine (it is ambiguous to have both), and if found, applies applicable conversions to *both* operands. In general, communicative binary operators should be written as routines to allow implicit conversion on both operands.

17.3.3 Copy

The constructor with a **const** reference parameter to the object type, e.g.:

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

is called the *copy constructor*, and has special meaning for two important initialization contexts: declarations and parameters. A declaration initialization:

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

The use of operator "=" in the declaration is misleading because it does not call the assignment operator but rather the copy constructor. The value on the right-hand side of the assignment is the argument to the copy constructor.

Similarly, each parameter of a routine is initialized using the copy constructor. For example, given the declarations:

```
complex foo( complex a, complex b );
complex x, y;
```

the call foo(x, y) results in the following implicit action in foo:

```
complex foo( complex a, complex b ) {
    a.complex( x ); b.complex( y ); // initialize parameters with arguments
```

If a copy constructor is not specified, an implicit one is generated that copies all the values from its parameter into the object, i.e., bit-wise copy.

Why does C++ differentiate between copy and assignment? For the copy situation (and constructors in general), after allocation, an object's members contain undefined values (unless a member has a constructor) and a constructor initializes appropriate members. For assignment, lhs = rhs, the left-hand variable may contain values and assignment only needs to copy a subset of values from the right-hand variable. For example, if an object type has a member variable to count the number of assignments, the counter is set to zero on initialization and incremented on assignment. In most languages, assignment means copy all the "bits" from one variable to another, which is also the default behaviour in C++; however, assignment in C++ can be redefined to selectively modify the "bits" (see Section 24, p. 56).

17.3.4 const Member

Unlike Java, a C/C++ **const** member of a structure must be initialized at the declaration:

```
struct foo {
    const int i; ...
} x = { 3 };        // const 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 foo {
    const int i; ...
    foo() { i = 3; } // attempt to initialize const member
};
```

However, this fails because it is assignment not initialization, and a **const** variable can only be initialized to 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:

Java	C++

```
class bar {}
class foo {
    final int i;

    final bar rp;
    foo ( bar b ) {
        i = 3;

        rp = b;
        ...
    }
}
```
	```
	class bar {};
	class foo {
	    const int i;
	    bar * const p;   // explicit const pointer
	    bar &rp;         // implicit const reference
	    foo ( bar b ) :  // syntax for initializing const members
	        i( 3 ),
	        p( &b ),     // explicit referencing
	        rp( b ) {    // implicit referencing
	        ...
	    }
	};
	```

In the example, member i is initialized to 3, and p and r are initialized to point at argument b, for the object's lifetime. In fact, this syntax can also be used to initialize non-**const** members.

17.4 Destructor

A *destructor* (finalize in Java) is a special member used to perform uninitialization at object deallocation, *which is only necessary if an object changes its environment*, e.g., closing communication channels or files, freeing dynamically allocated storage, etc. A self-contained object, like a complex object, requires no destructor. (See Section 24, p. 56 for a version of complex requiring a destructor.) There is only one destructor for an object type, and its name is the character "~" followed by the type name (like a constructor), versus the keyword finalize in Java; a destructor has no parameters nor return type (not even **void**):

Java	C++

class foo {
    . . .
    finalize() { . . . }
}
	struct foo {
	    . . .
	    ~foo() { . . . } *// destructor*
	};

A destructor is invoked immediately *before* an object is deallocated, either implicitly at the end of a block or explicitly by a **delete**:

```
{                                                    {
    complex x, y;                                        complex x;  x.complex();
                                                         complex y;  y.complex();
    complex *z = new complex;                            complex *z = new complex;  z->complex();
    . . .                    implicitly rewritten as     . . .
    delete z;                                            z->~complex();  delete z;
    . . .                                                . . .
                                                         y.~complex();  x.~complex();
} // deallocate local storage                        }
```

For local variables in a block, destructors are called in *reverse* order to constructors (independent of explicit **delete**).

A destructor is more common in C++ than a finalize in Java due to the lack of garbage collection in C++. If an object type performs dynamic allocation of storage, it needs a destructor to free the storage:

```
struct foo {
    int *i;    // think int i[]
    foo( int size ) { i = new int[size]; } // allocate dynamic sized array
    ~foo() { delete [] i; }     // must free storage
    . . .
};
```

Also, a destructor in C++ is invoked at a deterministic time (block termination or **delete**), ensuring prompt cleanup of the execution environment. In Java, a finalize is invoked at a non-deterministic time during garbage collection or *not at all*, so cleanup of the execution environment is unknown.

18 ⋈ Forward Declaration

Most programming languages have the notion of *Declaration Before Use* (DBU), e.g., a variable declaration must appear before its usage in a block:

```
{
    i += 1;        // no prior declaration of i
    int i;         // declaration after usage
}
```

While it is conceivable for a compiler to handle this situation, it makes other cases ambiguous:

```
int i;
{
    i += 1;        // now which i should be used?
    int i;         // declaration after usage
}
```

However, there are some cases where DBU can be allowed without causing ambiguity. C always requires DBU. C++ requires DBU in a block and among types but not within a type. Java only requires DBU in a block, but not for declarations in or among classes.

A language with DBU has a fundamental problem specifying *mutually recursive* referencing:

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

The problem is that the compiler cannot type-check the call to g in f to ensure the correct number and type of arguments and that the return value is used correctly because the actual definition of g, specifying the necessary type-checking information, occurs after the call. Clearly, interchanging the two routines does not solve the problem. The solution is a *forward declaration* to introduce a routine's type before its actual declaration:

```
int f( int i, double );          // routine prototype: parameter names optional and no routine body
. . .
int f( int i, double d ) {       // type repeated and checked with the prototype
    . . .
}
```

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 file hello.C
⇒ Enter the following program:

```
#include <iostream>               // Ex29
using namespace std;
void g( int i );                  // forward declaration with parameter name
void f( int i ) {
    cout << "f(" << i << ")" << endl;
    if ( i > 0 ) g( i - 1 );      // recursion
    cout << "f(" << i << ")" << endl;
}
void g( int i ) {                 // check for match with prototype
    cout << "g(" << i << ")" << endl;
    if ( i > 0 ) f( i - 1 );      // recursion
    cout << "g(" << i << ")" << endl;
}
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 also useful for organizing routines in a source file, e.g., allowing the main routine to appear first, and for separate compilation (see Section 24, p. 56):

```
void g( int );                   // forward declarations without parameter names
void f( int );
int main() {                     // appears first rather than last
    f( 5 );                      // actual declarations later
    g( 4 );
}
void g( int i ) { . . . }        // actual declarations
void f( int i ) { . . . }
```

Like Java, C++ does not require DBU for mutually-recursive routines within a type:

```
struct T {
    void f( int i ) { . . . g(. . .); . . . }   // g is not defined but it works!
    void g( int i ) { . . . f(. . .); . . . }
};
```

⇒ Edit file hello.C
⇒ Enter the following program:

```
#include <iostream>                    // Ex30
using namespace std;
struct T {
    void f( int i ) {
        cout << "f(" << i << ")" << endl;
        if ( i > 0 ) g( i - 1 );       // no forward declaration needed
        cout << "f(" << i << ")" << endl;
    }
    void g( int i ) {
        cout << "g(" << i << ")" << endl;
        if ( i > 0 ) f( i - 1 );
        cout << "g(" << i << ")" << endl;
    }
};
int main() {
    T x;
    x.f( 5 );
    cout << endl;
    x.g( 4 );
}
```

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

Unlike Java, C++ requires a forward declaration for mutually-recursive declarations among types:

Java	C++
	`struct T2;` `// forward declaration, no body`
`class T1 {`	`struct T1 {` `// T1 referencing T2`
`    final T2 t2;`	`    T2 &t2;` `// know about T2 from forward`
`    T1( final T2 t2 ) { this.t2 = t2; }`	`    T1( T2 &t2 ) : t2( t2 ) {} // constructor initialize`
`    void g( int i ) { ... t2.f(...) ... }`	`    void g( int i ) { ... t2.f(...); ... } // FAILS!!!`
`}`	`};`
`class T2 {`	`struct T2 {` `// T2 referencing T1`
`    final T1 t1`	`    T1 &t1;`
`        = new T1( this );`	`    T2() : t1( *this ) {} // constructor initialize`
`    void f( int i ) { ... t1.g(...) ... }`	`    void f( int i ) { ... t1.g(...); ... }`
`}`	`};`

The forward declaration of T2 allows the declaration of variable T1::t2. Note, a forward declaration only introduces the name of a type. Given just a type name, the only declarations possible are pointers/references to the type, which only allocate storage for an address rather than an actual object. An actual object declaration and usage requires the object's size and members so storage can be allocated, initialized, and usages type-checked. As a result, the C++ usage t2.f in T1::g fails because the information about type T2's members is defined later.

⇒ It is possible to change the declaration of T2::t1 from T1 &t1 to T1 t1, i.e., from a reference to an actual object?

Java's solution to this problem is to find the definition of T2 to obtain needed information (not DBU). C++'s solution involves forward declarations and a syntactic trick (DBU). First, a member containing the non-DBU reference is replaced by a forward declaration:

```
struct T1 {                // T1 referencing T2
    ...                    // as above
    void g( int i );       // forward
};
```

and second, a syntactic trick allows the actual member definition to be placed *after both types are defined*:

```
void T1::g( int i ) { ... t2.f(...); ... }
```

Now the compiler knows all the information about the types to verify usage in T1::g. Note, the trick use of qualified names T1::g to specify this is actually a member logically declared in T1 but physically located after the types (also see Section 24, p. 56).

⇒ Edit file hello.C

⇒ Enter the following program:

```
#include <iostream>                          // Ex31
using namespace std;
struct T2;                                   // forward declaration, no body
struct T1 {                                  // T1 referencing T2
    T2 &t2;                                  // know about T2 from forward
    T1( T2 &t2 ) : t2( t2 ) {}               // constructor initialize
    void g( int i );                         // forward declaration
};
struct T2 {                                  // T2 referencing T1
    T1 t1;
    T2() : t1( *this ) {}                    // constructor initialize
    void f( int i ) {
        cout << "T2::f(" << i << ")" << endl;
        if ( i > 0 ) t1.g( i - 1 );
        cout << "T2::f(" << i << ")" << endl;
    }
};
void T1::g( int i ) {                        // placed after both structure declarations
    cout << "T1::g(" << i << ")" << endl;
    if ( i > 0 ) t2.f( i - 1 );
    cout << "T1::g(" << i << ")" << endl;
}
int main() {
    T2 t2;
    t2.f( 5 );
    cout << endl;
    T1 t1( t2 );
    t1.g( 4 );
}
```

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

19 ⋈ Overloading

All programming languages have some form of overloading, where a name has multiple meanings in the same context. Overloading is possible if the compiler can disambiguate among identical names based on some criteria; the criterion 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.

```
int i;                   // variable overloading disallowed
double i;
void r( int ) {}         // routine overloading allowed
void r( double ) {}
```

For example, most built-in operators are overloaded to work with both integral and floating-point operands, i.e., the + operator is different for 1 + 2 than for 1.0 + 2.0. The power of overloading occurs when the type of a variable changes: operations on the variable are implicitly reselected to the variable's new type, e.g., after changing a variable's type from **int** to **double**, all operations implicitly change from integral to floating-point.

Like Java, C++ overloads the built-in operators for the basic types and allows users to overload members in a type. As well, C++ allows routines to be overloaded including operators, e.g., **operator**+ in Section 17.3.2, p. 39. 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
```

⇒ Edit file hello.C

⇒ Enter the following program:
```
#include <iostream>            // Ex32
using namespace std;
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 can cause problems with overloaded routines, e.g., given the above overloaded declarations of r, this call is ambiguous:

```
r( 1, 2.0 );      // ambiguous, could be either 1st or 2nd r
```

because either argument can be converted to integer or double. Use an explicit cast to provide sufficient information to disambiguate, e.g., r(1, (**int**)2.0) or r((**double**)1, 2.0).

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.

⇒ Edit file hello.C
⇒ Enter the following program:
```
int r( int i ) { }             // Ex33
int r( int i, int j = 2 ) { }
int main() {
    r( 3 );
}
```
⇒ Compile the program and explain the error message.

Another example of overloaded routines are the I/O operators << and >> for user types:

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

In this case, the compiler uses the << operator in object cout to first print a string value, but used the overloaded 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 file complex.C
⇒ Make the following modifications:
```
#include <iostream>            // Ex34
#include <cmath>
using namespace std;
struct complex {
    double re, im;
    double abs() { return sqrt( re * re + im * im ); }
    complex() { re = 0.; im = 0.; } // overloaded constructors
    complex( double r ) { re = r; im = 0.; }
    complex( double r, double i ) { re = r; im = i; }
};
```

```
// overloaded routines
complex operator+( complex a, complex b ) { return complex( a.re + b.re, a.im + b.im ); }
ostream &operator<<( ostream &os, complex c ) { return os << c.re << "+" << c.im << "i"; }
int main() {
    complex x, y, z;
    x = 3.2;
    y = x + complex( 1.3, 7.2 );
    z = y + x;
    cout << "x:" << x << " y:" << y << " z:" << z << endl;
}
```

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

20 Inheritance

The "oriented" part of object-oriented refers to an additional notion called *inheritance*, which is useful for writing general, reusable program components.

Java	**C++**

class base { . . . }
class derived **extends** base { . . . }
	struct base { . . . }
	struct derived : **public** base { . . . };

Inheritance has two orthogonal sharing concepts: implementation and type; each is discussed separately.

20.1 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 {
    int i;
    int r(...) { ... }
    base() { ... }
};
struct derived : public base {    // implicit inclusion

    int s(...) { i = 3; r(...); ... }
    derived() { ... }
} d;
d.i = 3;      // reference member in included member
d.r(...);     // reference member in included member
d.s(...);     // s can access i and r in included member
```
	```
	struct base {
	    int i;
	    int r(...) { ... }
	    base() { ... }
	};
	struct derived {
	    base b; // explicit inclusion
	    int s(...) { b.i = 3; b.r(...); ... }
	    derived() { ... }
	} d;
	d.b.i = 3;
	d.b.r(...);
	d.s(...)
	```

In the example, object type derived states it is inheriting from a base object, via the "**public** base" clause. 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. The inclusion analogy involves explicit creation of an object member, b, to aid in the implementation.

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;                                base b;                 // implicit, hidden declaration
...                                       derived d; b.base(); d.derived();
                  implicitly rewritten as ...
                                          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 blocks: a name in the inner scope hides (overrides) a name at the outer scope (see Section 8.3, p. 7). However, it is still possible to access these members by using "::" qualification (see Section 17, p. 34) to specify the particular nesting level that contains the member.

Java	C++
class base1 {	**struct** base1 {
int i;	**int** i;
}	};
class base2 **extends** base1 {	**struct** base2 : **public** base1 {
int i;	**int** i; *// hides base1::i*
}	};
class derived **extends** base2 {	**struct** derived : **public** base2 {
int i;	**int** i; *// hides base2::i*
void s() {	**void** r() {
int i = 3;	**int** i = 3; *// hides derived::i*
this.i = 3;	derived::i = 3; *// this.i*
((base1)**this**).i = 3; *// super.i*	base2::i = 3;
((base2)**this**).i = 3;	base2::base1::i = 3;
}	}
}	};

⇒ Edit file hello.C

⇒ Enter the following program:

```
#include <iostream>                    // Ex35
using namespace std;
struct base1 {
    void r() { cout << "base1::r" << endl; }
};
struct base2 : public base1 {
    void r() { cout << "base2::r" << endl; }
};
struct derived : public base2 {
    void r() {
        cout << "derived::r" << endl;
        base2::r();
        base2::base1::r();
    }
};
int main() {
    derived d;
    d.r();
}
```

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

Implementation inheritance is used to write reusable program components by composing a new object's implementation from an existing object, making it possible to take advantage of previously written and tested code to substantially reduce the time in composing and debugging a new object type. Unfortunately, having to inherit all of the members is not always desirable; some members may be inappropriate 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, routines provide smaller units for reuse than entire objects.)

20.2 Type Inheritance

Type inheritance extends name equivalence (see Section 8.6, p. 9) to allow routines to handle multiple types, called *polymorphism*, e.g.:

```
struct foo {                      struct bar {
    int i;                            int i;
    double d;                         double d;
} f;                              } m;
void r( foo f ) { ... }
r( f );    // valid call
r( m );    // should also work
```

Since types foo and bar are identical, instances of either type can work as arguments to routine r. Even if type bar has more members at the end, routine r only accesses the common ones at the beginning as its parameter is type foo. However, Java and C++ both use name equivalence to compare types for equality; hence, the call r(m) fails even though m is structurally identical to f. Type inheritance relaxes name equivalence by aliasing the derived name with all of its base-type names:

```
struct foo {                     struct bar : public foo { // inheritance
    int i;                           // no members
    double d;
} f;                             } m;
void r( foo f ) { . . . }
r( f );   // valid call, derived name matches
r( m );   // valid call because of inheritance, base name matches
```

For example, create a new type mycomplex that counts the number of times abs is called for each mycomplex object. Use both implementation and type inheritance to simplify building type mycomplex.

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

Derived type mycomplex uses all the implementation of the base type complex, adds new members, and overrides abs to count each call. The power of type inheritance is the reuse of complex's addition and output operation for mycomplex values, which can be used because of the relaxed name equivalence provided by type inheritance between argument and parameter.

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

Now variables of type complex are redeclared to mycomplex, and member calls returns the current number of calls to abs for any mycomplex object.

⇒ Edit file complex.C

⇒ Make the following modifications:

```
. . . // same as before until the end of the complex output operator
struct mycomplex : public complex { // Ex36
    int cntCalls;
    mycomplex() : cntCalls(0) {}
    double abs() { cntCalls += 1; return complex::abs(); }
    int calls() { return cntCalls; }
};
int main() {
    mycomplex x, y, z;
    cout << "x:" << x.abs() << " y:" << y.abs() << " z:" << z.abs() << endl;
    cout << "x:" << x.calls() << " y:" << y.calls() << " z:" << z.calls() << endl;
}
```

⇒ Compile the program and run it.

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 that manipulates the base type also manipulates the derived type.

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

⇒ Edit file complex.C

⇒ Make the following modification to routine main:

```
int main() {                          // Ex37
    mycomplex x;
    x = x + x;
}
```

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

Like the previous example, the complex routine **operator+** is used to add the mycomplex values because of the relaxed name 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 are beyond the level of this tutorial.

The second problem is illustrated by:

⇒ Edit file complex.C
⇒ Make the following modifications:

```
. . . // same as before until the end of the declaration of mycomplex
void r( complex &c ) { c.abs(); }      // Ex38
int main() {
    mycomplex x;
    x.abs();       // direct call of abs
    r( x );        // indirect call of abs
    cout << "x:" << x.calls() << 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 next.

20.3 ⋈ Virtual Routine

In general, when a member is called, it is obvious which one is invoked even with overriding, e.g.:

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

However, it is not obvious for arguments/parameters and pointers/references:

```
void s( base &b ) { b.r(); }
s( d );            // call allowed because of inheritance; does s call base::r or derived::r ?
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, but both calls should invoke derived::r because argument b and reference bp currently pointing at an object of type derived; likewise, if variable d is replaced with b, the calls should invoke base::r. However, there are situations where a programmer may want to access members in base even if the actual object is of type derived. Notice, this is never a problem because derived *contains* a base.

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. To set the call default to invoke the routine defined in the referenced object, qualify the member routine with **virtual**. To set the call default to invoke the routine defined by the type of the pointer/reference, do not qualify the member routine with **virtual**. C++ uses non-virtual as the default because it is more efficient. Java sets the call default to virtual for all calls to objects, and does not suppose the second form of object call. Finally, once a base type qualifies a member as virtual, *it is virtual in all derived types regardless of the derived type's qualification for that member*. The following example shows how to access *all* routine members in the base and derived type regardless of how the routines are qualified:

Java	C++
class base {	**struct** base {
public void f() {} // *virtual*	**void** f() {} // *non-virtual*
public void g() {} // *virtual*	**void** g() {} // *non-virtual*
public void h() {} // *virtual*	**virtual void** h() {} // *virtual*
}	};
class derived **extends** base {	**struct** derived : **public** base {
public void g() {} // *virtual*	**void** g() {}; // *non-virtual*
public void h() {} // *virtual*	**void** h() {}; // *virtual*
}	};
final base bp = **new** derived();	base &bp = ***new** derived(); // *polymorphic assignment*
bp.f(); // *base.f*	bp.f(); // *base::f, use pointer type*
((base)bp).g(); // *derived.g*	bp.g(); // *base::g, use pointer type*
bp.g(); // *derived.g*	((derived &)bp).g(); // *derived::g, use pointer type*
((base)bp).h(); // *derived.h*	bp.base::h(); // *base::h, explicit selection*
bp.h(); // *derived.h*	bp.h(); // *derived::h, use object type*

Notice, casting in Java does not provide access to the base-type's member routines.It is important to understand that virtual members are *only* necessary to access derived members through a base type reference or pointer. *Therefore, if a type is never involved in inheritance (***final** *class in Java), it never needs virtual members, and hence, can take advantage of more efficient calls to its members.*

When a type is involved in inheritance, one problem with virtual members in C++ is that the qualification is made in the base type as opposed to the derived type. Hence, C++ requires the base-type definer to look into the future and guess how derived definers might want the call default to work. Therefore, like Java, good programming practice is to make all routine members virtual for types involved in inheritance. Finally, any type with virtual members and a destructor should make the destructor virtual, to ensure the most derived destructor is called through a base-type pointer/reference.

⇒ Edit file complex.C.

⇒ Modify the program so *all* calls to member abs are counted.

20.4 Down Cast

Type inheritance can mask the actual type of an object through a pointer/reference (see Section 20.2, p. 47). Like Java, C++ provides a mechanism to dynamically determine the actual type of a pointer/reference. The Java operator **instanceof** and the C++ operator **dynamic_cast** perform a dynamic check of the object addressed by a pointer/reference:

Java	C++
base bp = **new** derived();	base *bp = **new** derived();
if (bp **instanceof** derived)	**if** (**dynamic_cast**<derived *>(bp) != 0)
((derived)bp).rtn();	((derived *)bp)->rtn();

To use **dynamic_cast** on a type, *the type must have at least one virtual member*.

20.5 Constructor/Destructor

Like Java, C++ constructors are *implicitly* executed top-down, from base to most derived type. This order is mandated by the scope rules, which allow a derived-type constructor to use a base type's variables so the base type must be initialized first. Unlike Java, C++ destructors are *implicitly* executed bottom-up, from most derived to base type. Again, this order is mandated by the scope rules, which allow a derived-type constructor to use a base type's variables so the base type must be uninitialized last. Java finalize must be *explicitly* called from derived to base type.

Unlike Java, C++ disallows calls to other constructors at the start of a constructor (see Section 17.3.4, p. 40). To pass arguments to other constructors, use the same syntax as for initializing **const** members (see Section 17.3.4, p. 40).

Java	C++

`class base {`
`    base( int i ) { ... }`
`};`
`class derived extends base {`
`    derived() { super( 3 ); ... }`
`    derived( int i ) { super( i ); ... }`
`};`
	`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`
	`};`

⇒ Edit file Hello.C.

⇒ Enter the following program:

```
#include <iostream>                 // Ex39
using namespace std;
struct base {
    int i;
    base( int i ) { cout << "base, i:" << i << endl; }
    ~base() { cout << "~base" << endl; }
};
struct derived : public base {
    derived() : base( 3 ) { cout << "derived" << endl; }
    derived( int i ) : base( i ) { cout << "derived, i:" << i << endl; }
    ~derived() { cout << "~derived" << endl; }
};
int main() {
    base b( 2 );          cout << "=====" << endl;
    derived d1;           cout << "=====" << endl;
    derived d2( 7 );      cout << "=====" << endl;
}
```

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

20.6 Abstract Interface

Like Java, C++ supports a mechanism to create an abstract interface from which actual types can be defined:

Java	C++

`interface shape {`
`    void move( int x, int y );`
`};`
`class circle implements shape {`
`    public void move( int x, int y ) {}`
`};`
	`struct shape {`
	`    virtual void move( int x, int y ) = 0; // strange initialization`
	`};`
	`struct circle : public shape {`
	`    void move( int x, int y ) {}  // must define this member`
	`};`

In the C++ example, note the strange initialization of member shape::move to 0, which actually means this member *must* be defined by any derived type of shape. Unlike Java, C++ allows the abstract interface to contain actual members, which results in a combination of implementation inheritance and abstract description.

21 ⋈ Template

Inheritance handles reuse where types are organized into a hierarchy to extend name equivalence. There is another kind of reuse for situations where there is no type hierarchy and types are not equivalent. For example, the overloaded abs routines defined in Section 19, p. 44 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. A template routine allows types to become compile-time parameters (Java does not support template routines), e.g.:

```
template<typename 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 types for abs. 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 file hello.C

⇒ Enter the following program:

```
#include <iostream>                    // Ex40
using namespace std;
template<typename 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.

A template type is also possible, when a type has identical code but manipulates different types (Java does support template types but using a different technique). 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<typename T, int N = 10> struct Stack {
    T elems[N]; // maximum N elements
    int size;
    Stack() { size = 0; }
    void push( T e ) { elems[size] = e; size += 1; }  // use N to check for overflow
    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 the member routines. The integer parameter denotes the maximum stack size. For template types, the compiler cannot infer the type parameter, so it is explicitly specified.

```
Stack<int, 20> 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 declaration, e.g., Stack< Stack<**int**> >; there must be a space between the two ending chevrons or >> is parsed as **operator**>>.

Container types are a common use of templates. A container template forms a specific container type storing programmer defined nodes. The C++ Standard Template Library (STL) provides different kinds of containers (vector, stack, queue, list, deque, set, map). Figure 2 shows the STL vector container as an alternative to C/C++ arrays (see Section 4, p. 13). Like a Java array, vector can be dynamically sized, has a member routine to obtain the size, has subscript checking, and also supports assignment. The example in Figure 2(a) is a conversion of the program to read values and print them in reverse order on page 28. The declaration of a vector may specify an initial size, e.g., vector<**int**> vals(size), like a dimension. While the size of a vector may increase (or decrease) dynamically, it is more efficient to dimension, when the size is known. At any time, it is possible to query a vector's size, e.g., vals.size(). The subscript operator, [], and member "at" both perform subscripting of array elements; the difference is *only member "*at*" performs subscript checking.* As well, one vector can be assigned to another, copying each element. The example in Figure 2(b) is a conversion of the program to initialize and print a matrix on page 29. The declaration of a matrix is a vector of vectors, e.g., vector< vector<**int**> > m(5), which specifies 5 rows. Before values can be assigned into a row, each row is dimensioned to the specific size, m[r].resize(4). All loop bounds are controlled by using the dynamic size of the row or column.

Figure 3 shows a list-container example. In general, 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 default and/or copy constructor so instances can be created without having to know arguments to constructors. The C++ STL uses copying and requires a node type to have a default constructor. The implication of linking is that the node type must inherit from a particular list type to ensure it has appropriate link members. Like the

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    int i, size;
    cin >> size;
    vector<int> vals(size);     // think int vals[size]
    for ( i = 0; i < vals.size(); i += 1 ) {
        cin >> vals.at(i);      // think vals[i]
    }
    vector<int> v;              // think: int v[]
    v = vals;                   // array assignment
    for ( i = v.size() - 1; 0 <= i; i -= 1 ) {
        cout << v.at(i) << " ";
    }
    cout << endl;
}
```

(a) Array (with subscript checking using member at)

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector< vector<int> > m( 5 ); // 5 rows
    for ( int r = 0; r < m.size(); r += 1 ) {
        m[r].resize( 4 );         // 4 columns per row
        for ( int c = 0; c < m[r].size(); c += 1 ) {
            m[r][c] = r+c;        // or m.at(r).at(c)
        }
    }
    for ( int r = 0; r < m.size(); r += 1 ) {
        for ( int c = 0; c < m[r].size(); c += 1 ) {
            cout << m[r][c] << ", ";
        }
        cout << endl;
    }
}
```

(b) Matrix (with no subscript checking using operator [])

Figure 2: STL Vector Container

```
#include <iostream>
#include <list>
using namespace std;
struct node {
    char c;
    int i;
    double d;
    node() {}                                           // must have a basic constructor to copy
    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 3: STL Doubly Linked-List Container

Stack container-type, a list must specify the type of the list nodes, e.g., list<node>.

An additional concept introduced by containers is the *iterator*, which is used to traverse a container without knowing how the container is implemented. The capabilities of an iterator depend on the kind of container, e.g., a singly linked list only allows traversing the list unidirectionally while a doubly linked list allows bidirectional traversal. Each container in the C++ STL provides an appropriate iterator as a nested object type (see the end of Section 17, p. 34); hence the declaration type of the iterator for list<node> is list<node>::iterator.

In Figure 3, the first loop initializes a node with values and calls push_back, which copies the node to the end (back) of the list. (push_back can also be used with vector to incrementally extend a vector's size.) The second loop traverses the list using an iterator index, ni, starting at the beginning of the list and stepping through the list one node at a time until ni is past the end of the list (end() is not the last node but past the end node). Note, iterator ni is like a pointer to a node stored in the list so the node is accessed with operator ->. As well, the operator "++" is used to advance to the next node in the list. The final loop destroys the list by repeatedly erasing 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 to print the nodes in reverse order. (Stopping the loop is tricky.)

An alternate mechanism to iterate through a container is using the STL template-routine for_each, which uses a container's iterator to traverse a data structure, applying an action to each node:

```
#include <iostream>
#include <list>
#include <vector>
using namespace std;
void print( int i ) { cout << i << " "; }          // print node
int main() {
    list< int > int_list;
    vector< int > int_vec;
    for ( int i = 0; i < 10; i += 1 ) {            // create lists
        int_list.push_back( i );
        int_vec.push_back( i );
    }
    for_each( int_list.begin(), int_list.end(), print ); // print each node
    for_each( int_vec.begin(), int_vec.end(), print );
}
```

The action routine to for_each is called for each node in the container, passing the node to the routine for processing. In general, the type of the action routine is **void** rtn(T), where T is the type of the container node. In this example, print must have an **int** parameter matching the type of the node in each container.

If a more complex action is necessary, it can be constructed by a "function object", called a *functor*, using the routine-call operator. For example, to have an action routine that prints values on a specified stream, a type is created to store the stream, and **operator**() is defined to allow the object to behave like a function:

```
struct print {
    ostream &stream;                // stream used for output
    print( ostream &stream ) : stream( stream ) {}
    void operator()( int i ) { stream << i << " "; }
};
int main() {
    list< int > int_list;
    vector< int > int_vec;
    . . .
    for_each( int_list.begin(), int_list.end(), print(cout) ); // print on different streams
    for_each( int_vec.begin(), int_vec.end(), print(cerr) );
}
```

The expression print(cout) creates a constant print object, and for_each calls **operator**()(node) in the object.

22 Namespace

Like Java, C++ has a mechanism to organize complex programs and libraries composed of multiple types and declarations. For example, this tutorial relies on namespace std, containing all the I/O declarations and container types. The names in a namespace form a declaration region, like the scope of block. Unlike Java, C++ allows multiple namespaces to be defined in a file; types and declarations do not have to be added consecutively.

Java source file	C++ source file

package foo; *// one package / file*
// types / declarations
	namespace foo {
	    // types / declarations
	};
	namespace bar {
	    // types / declarations
	};
	namespace foo {
	    // more types / declarations
	};

Like Java, the contents of a namespace can be accessed using full-qualified names:

Java	C++
foo.T t = **new** foo.T();	foo::T *t = **new** foo::T();

or by importing individual items or all of the namespace content.

Java	C++

import foo.T;
import foo.*;
	using foo::T; *// import individual*
	using namespace foo; *// import all*

23 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
    . . . // interface routine members
};
```

so the implementation 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 24). C++ features work both within and among source files.

Like Java, C++ provides 3 levels of visibility control for object types:

Java	C++

class foo {
    private . . .
    . . .
    protected . . .
    . . .
    public . . .
    . . .
};
	struct foo {
	  private: *// visible within and to friends*
	    // private members
	  protected: *// visible within, to friends and inherited types*
	    // protected members
	  public: *// visible within, to friends, inherited types and users*
	    // public members
	};

Java requires encapsulation specification for each member, while C++ groups members with the same encapsulation, i.e., all members after a label, **private**, **protected** or **public**, have that visibility. Visibility labels can occur in any order and multiple times in an object type. Only the object type can access the private members, so the implementation members are normally private. Inherited object types can access and modify public and protected members, which may allow access to some of an object's implementation. The public members 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 abstraction. 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 kind of structure called a **class**, which is the same as **struct**, except the default is that all members are private.

⇒ Edit file hello.C
⇒ Enter the following program:

```
class base {                          // Ex41
  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.

Encapsulation introduces a new problem for routines used to implement binary operations for an object: a 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 routine is allowed access to its implementation, called *friendship* (similar to package visibility in Java).

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

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

```
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 operator+( complex a, complex b ) { return complex( a.re + b.re, a.im + b.im ); }
ostream &operator<<( ostream &os, complex c ) { return os << c.re << "+" << c.im << "i"; }
```

24 ⋈ Separate Compilation

Like Java's package access, a C/C++ *source file* provides another mechanism for encapsulation. By default, all global variables and routines in a source file are exported outside the file (package). To encapsulate declarations in a source file, the declaration must be qualified with **static**.

```
// file.C
int i;                      // public (exported)
void f(. . .) {}            // public (exported)
static int j;               // private
static void g(. . .) {}     // private
```

Like Java, a type is encapsulated in a source file, unless explicitly denoted as public.

Unlike Java, which has automatic access to the public contents of a source file, C/C++ require the use of the preprocessor (see Section 11, p. 21) and forward declarations (see Section 18, p. 41) to access public contents, which is accomplished by dividing declarations into two parts, and into two (or more) files. Each declaration is divided into its interface and implementation. The interface is usually composed of the prototype declaration(s) (but possibly some implementation), and the implementation is composed of the actual declarations and code. Second, the interface is entered into one or more include files (.h files), and the implementation is entered into one or more source files (.C files). Encapsulation is provided by giving a user access to only the include file(s) and the compiled source file(s), which is sufficient to use an abstraction, but not the implementation in the source file(s). Most software supplied from software vendors comes this way. Prototypes in the include files for exported variables and routines (not types) must be qualified with **extern** to indicate the implementation appears elsewhere:

```
// file.h
extern int i;         // public, implementation elsewhere
extern void f(. . .); // public, implementation elsewhere (extern optional for routines)
```

For example, the complex type can be divided into file complex.h, which users include in their programs, and file complex.C (see Figure 4). The .C file normally includes the .h file so that there is only one copy of the constants, 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, but each constructor in the source file can increment the counter when a complex object is created. *All static variables, whether in a class or file, must be explicitly initialized in the* .C *file.* For example, variable cplxObjCnt is set to 0. The exported routine complexStats can be called by users at any time to print the number of complex objects created so far in a program. Notice, all the member routines of complex are separated into a forward declaration and an implementation after the object type, allowing the implementation to be placed in the .C file (see Section 18, p. 41).

However, by reading the .h, it may be possible to determine the implementation technique used, so there is only a partial 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 5, p. 59). Essentially, how much information goes into .h file depends on the amount of encapsulation; but the amount of encapsulation may affect the implementation. Note, because the compiler requires a template definition for each usage, both the interface and implementation of a template must be in a .h file, which precludes certain forms of encapsulation.

Notice the use of a copy constructor and assignment operator in Figure 5, p. 59 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.

⇒ Edit file complex.h
⇒ Enter the program in Figure 4(a).
⇒ Edit file complex.C
⇒ Enter the program in Figure 4(b).
⇒ Compile the program to create the executable complex.o.

To use an encapsulated object, a program specifies the necessary include file(s) to access the object's interface, and then links with any necessary executables.

⇒ Edit file hello.C
⇒ Enter the following program:

```
#ifndef __COMPLEX_H__                 // Ex42
#define __COMPLEX_H__                 // protect against multiple inclusion
#include <iostream>                   // access: ostream
using std::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:
        complex();
        complex( double r );
        complex( double r, double i );
        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"                  // Ex43
#include <cmath>                      // access: sqrt
using namespace std;
// private declarations
static int cplxObjCnt = 0;            // must be initialized
// 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 ) { return os << c.re << "+" << c.im << "i"; }
```

(b) complex.C

Figure 4: Partially Encapsulated Abstract Type

```
#include "complex.h"
#include <iostream>
using namespace std;
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;
}
```

Notice, iostream is included twice, once in this program and once in complex.h, which is why each include file needs to prevent multiple inclusions.

⇒ Compile the program with command:

```
g++ hello.C complex.o
```

⇒ Redo the last two work sections replacing the code in complex.h and complex.C with the code from Figure 5(a) and Figure 5(b), respectively.

```
#ifndef __COMPLEX_H__              // Ex44
#define __COMPLEX_H__              // protect against multiple inclusion
#include <iostream>                // access: ostream
using std::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;           // indirection 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"               // Ex45
#include <cmath>                   // access: sqrt
using namespace std;
// private declarations
static int cplxObjCnt = 0;
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(const 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 ) { return os << c.impl.re << "+" << c.impl.im << "i"; }
```

(b) complex.C

Figure 5: Fully Encapsulated Abstract Type

25 Acknowledgments

The following people made suggestions to improve the tutorial: Glen Ditchfield, Rob Holte, Caroline Kierstead, Steve Mann, Richard Bilson and John-Paul Pretti.

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>
#include <iomanip>
#include <fstream>
#include <string>
#include <list>
using namespace std;

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

     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