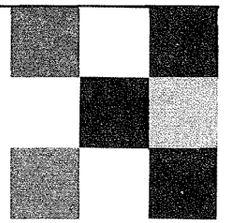


Chapter 13

Moving from C to C++

This chapter gives an overview of the C++ programming language. It also provides an introduction to C++'s use as an object-oriented programming language. In the chapter, a series of programs is presented, and the elements of each program are carefully explained. The programs increase in complexity, and the examples in the later sections illustrate some of the concepts of object-oriented programming.

The examples in this chapter give simple, immediate, hands-on experience with key features of the C++ language. The chapter introduces the reader to stream I/O, operator and function overloading, reference parameters, classes, constructors, destructors, templates, and inheritance. Mastery of the individual topics requires a thorough reading of a companion book such as either Pohl, *C++ for C Programmers*, 2d ed (Redwood City, CA: Benjamin/Cummings, 1993) or Pohl, *Object-Oriented Programming Using C++*, 2d ed (Reading, MA: Addison-Wesley, 1997).

Object-oriented programming is implemented by the `class` construct. The `class` construct in C++ is an extension of `struct` in C. The later examples in this chapter illustrate how C++ implements OOP (object-oriented programming) concepts, such as data hiding, ADTs, inheritance, and type hierarchies.

13.1 Output

Programs must communicate to be useful. Our first example is a program that prints on the screen the phrase "C++ is an improved C." The complete program is

In file improved.cpp

```
// A first C++ program illustrating output.
//    Title:  Improved
//    Author: Richmond Q. Programmer

#include <iostream.h>

int main()
{
   cout << "C++ is an improved C.\n";
}
```

The program prints on the screen

```
C++ is an improved C.
```

Dissection of the *improved* Program

■ `// A first C++ program illustrating output.`

The double slash // is a new comment symbol. The comment runs to the end of the line. The old C bracketing comment symbols /* */ are still available for multiline comments.

■ `#include <iostream.h>`

The *iostream.h* header introduces I/O facilities for C++.

■ `int main()`

In C++, the empty parentheses always mean `main(void)`, never `main(...)`. C++ style is not to use the redundant `void` for declaring the empty argument list.

■ `cout << "C++ is an improved C.\n";`

This statement prints to the screen. The identifier `cout` is the name of the standard output stream. The operator << passes the string `"C++ is an improved C.\n"` to standard out. Used in this way the *output operator* << is referred to as the *put to* or *insertion* operator.

We can rewrite our first program as follows:

```
// A first C++ program illustrating output.

#include <iostream.h>

main()
{
   cout << "C++ is an improved C." << endl;
}
```

Although it is different from the first version, it produces the same output. This version drops the explicit declaration of `main()` as returning an `int` and uses the fact that this `return` type is implicit. Here we use the output operator << *put to* twice. Each time the << is used with `cout`, printing continues from the position where it previously left off. In this case, the identifier `endl` forces a new line followed by a *flush*. The `endl` is called a *manipulator*.

13.2 Input

We will write a program to convert to kilometers the distance in miles from the Earth to the moon. In miles this distance is, on average, 238,857 miles. This number is an integer. To convert miles to kilometers, we multiply by the conversion factor 1.609, a real number.

Our conversion program will use variables capable of storing integer values and real values. In C++, all variables must be declared before their use, but unlike in C, they need not be at the head of a block. Declarations may be mixed in with executable statements. Their scope is from the point of the declaration to the end of the block within which they are declared. Identifiers should be chosen to reflect their use in the program. In this way, they serve as documentation, making the program more readable.

These programs assume a 4-byte `int`, but on some machines these variables should be declared `long`. You can check the constant `INT_MAX` in *limits.h*.

In file moon.cpp

```
// The distance to the moon converted to kilometers.
//    Title: moon

#include <iostream.h>


int main()
{
   const int moon = 238857;

   cout << "The moon's distance from Earth is " << moon;
   cout << " miles." << endl;
   int moon_kilo = moon * 1.609;
   cout << "In kilometers this is " << moon_kilo;
   cout << " km." << endl;
}
```

The output of the program is

```
The moon's distance from Earth is 238857 miles.
In kilometers this is 384320 km.
```

Dissection of the *moon* Program

- `const int moon = 238857;`

The keyword `const` is new in C++. It replaces some uses of the preprocessor command `define` to create named literals. Using this type modifier informs the compiler that the initialized value of `moon` cannot be changed. Thus, it makes `moon` a symbolic constant.

- `cout << "The moon's distance from Earth is " << moon;`

The stream I/O in C++ can discriminate among a variety of simple values without needing additional formatting information. Here the value of `moon` is printed as an integer.

- `int moon_kilo = moon * 1.609;`

Declarations can occur after executable statements. This allows declarations of variables to be nearer to their use.

Let us write a program that will convert a series of values from miles to kilometers. The program will be interactive. The user will type in a value in miles, and the program will convert this value to kilometers and print it out.

In file mi_km.cpp

```
// Miles are converted to kilometers.
//   Title: mi_km

#include <iostream.h>

const double m_to_k = 1.609;

inline int convert(int mi) { return (mi * m_to_k); }


int main()
{
   int miles;

   do {
      cout << "Input distance in miles: ";
      cin >> miles;
      cout << "\nDistance is " << convert(miles) << " km."
           << endl;
   } while (miles > 0);
}
```

This program uses the input stream variable `cin`, which is normally standard input. The *input operator* >> is called the *get from* or *extraction* operator, which assigns values from the input stream to a variable. This program illustrates both input and output.

Dissection of the *mi_km* Program

- `const double m_to_k = 1.609;`

C++ reduces C's traditional reliance on the preprocessor. For example, instead of having to use `define`, special constants, such as the conversion factor 1.609, are simply assigned to variables specified as constants.

- `inline int convert(int mi) { return (mi * m_to_k); }`

The new keyword `inline` specifies that a function is to be compiled, if possible as inline code. This avoids function call overhead and is better practice than C's use of `define` macros. As a rule, `inline` should be done sparingly and only on short functions. Also note how the parameter `mi` is declared within the function parentheses. C++ uses *function prototypes* to define and declare functions. This will be explained in the next section.

-
```
do {
   cout << "Input distance in miles: ";
   cin >> miles;
   cout << "\nDistance is " << convert(miles) << " km."
        << endl;
} while (miles > 0);
```

The program repeatedly prompts the user for a distance in miles. The program is terminated by a zero or negative value. The value placed in the standard input stream is automatically converted to an integer value assigned to `miles`.

13.3 Functions

The syntax of functions in C++ inspired the new function prototype syntax found in Standard C compilers. Basically, the types of parameters are listed inside the header parentheses. By explicitly listing the type and number of arguments, strong type checking and assignment-compatible conversions are possible in C++.

C++ allows functions to have arguments directly called by reference. Call-by-reference parameters are declared using the syntax

type& identifier

Also, C++ function parameters can have default values. These are given in the function declaration inside the parameter list by

= *expression*

added after the parameter.

The following example illustrates these points:

In file add3.cpp

```
// Use of a default value

#include <iostream.h>

inline void add3(int& s, int a, int b, int c = 0)
{
   s = a + b + c;
}

inline double average(int s) { return s / 3.0; }
```

```
int main()
{
   int   score_1, score_2, score_3, sum;

   cout << "\nEnter 3 scores: ";
   cin >> score_1 >> score_2 >> score_3;
   add3(sum, score_1, score_2, score_3);
   cout << "\nSum = " << sum;
   cout << "\nAverage = " << average(sum) << endl;
   add3(sum, 2 * score_1, score_2);  // use of default value 0
   cout << "\nWeighted Sum = " << sum << ".";
   cout << "\nWeighted Average = " << average(sum) << ".\n";
}
```

Dissection of the *add3* Program

- ```
  inline void add3(int& s, int a, int b, int c = 0)
  {
     s = a + b + c;
  }
  ```

The variable s is call-by-reference. An actual argument passed in must be an lvalue, because that will be the actual address used when the procedure is called.

- ```
  add3(sum, score_1, score_2, score_3);
  ```

The variable sum is passed-by-reference. Therefore, it is directly manipulated and can be used to obtain a result from the function's computation.

- ```
  add3(sum, 2 * score_1, score_2);  // use of default value 0
  ```

Here only three actual arguments are used in calling `add3()`. The fourth argument defaults to value zero.

13.4 Classes and Abstract Data Types

What is novel about C++ is its aggregate type `class`. A `class` is an extension of the idea of `struct` in traditional C. A `class` provides the means for implementing a user-defined data type and associated functions and operators. Therefore, a `class` can be used to implement an ADT. Let us write a `class` called `string` that will implement a restricted form of string.

In file my_string.cpp

```
// An elementary implementation of type string.

#include <string.h>
#include <iostream.h>

const int max_len = 255;


class string {
public:          // universal access
   void assign(const char* st)
      { strcpy(s, st); len = strlen(st); }
   int  length() { return len; }
   void print() { cout << s << "\nLength: " << len << "\n"; }
private:         // restricted access to member functions
   char s[max_len];    // implementation by character array
   int  len;
};
```

Two important additions to the structure concept of traditional C are found in this example: first, it has members that are functions, such as `assign`, and second, it has both public and private members. The keyword `public` indicates the visibility of the members that follow it. Without this keyword, the members are private to the class. Private members are available for use only by other member functions of the class. Public members are available to any function within the scope of the class declaration. Privacy allows part of the implementation of a class type to be "hidden." This restriction prevents unanticipated modifications to the data structure. Restricted access, or *data hiding*, is a feature of object-oriented programming.

The declaration of member functions allows the ADT to have particular functions act on its private representation. For example, the member function `length` returns the length of the string defined to be the number of characters up to but excluding the first
```

zero value character. The member function `print()` outputs both the string and its length. The member function `assign()` stores a character string into the hidden variable `s` and computes and stores its length in the hidden variable `len`.

We can now use this data type `string` as if it were a basic type of the language. It obeys the standard block structure scope rules of C. Other code that uses this type is a *client*. The client can use only the public members to act on variables of type `string`.

```
// Test of the class string.

int main()
{
   string  one, two;
   char    three[40] = {"My name is Charles Babbage."};

   one.assign("My name is Alan Turing.");
   two.assign(three);
   cout << three;
   cout << "\nLength: " << strlen(three) << endl;
   // Print shorter of one and two.
   if (one.length() <= two.length())
      one.print();
   else
      two.print();
}
```

The variables `one` and `two` are of type `string`. The variable `three` is of type pointer to `char` and is not compatible with `string`. The member functions are called using the dot operator or "structure member operator." As is seen from their definitions, these member functions act on the hidden private member fields of the named variables. One cannot write inside `main` the expression `one.len` expecting to access this member. The output of this example program is

```
My name is Charles Babbage.
Length: 27
My name is Alan Turing.
Length: 23
```


13.5 Overloading

The term *overloading* refers to the practice of giving several meanings to an operator or a function. The meaning selected depends on the types of the arguments used by the operator or function. Let us overload the function `print` in the previous example. This will be a second definition of the `print` function.

```
class string {
public:          // universal access
   .....
   void print() { cout << s << "\nLength: " << len << "\n"; }
   void print(int n)
   {
      for(int i = 0; i < n; ++i)
         cout << s << endl;
   }
   .....
}
```

This version of `print` takes a single argument of type `int`. It will print the string `n` times.

```
three.print(2);     // print string three twice
three.print(-1);    // string three is not printed
```

It is also possible to overload most of the C operators. For example, let us overload `+` to mean concatenate two strings. To do this we need two new keywords: `friend` and `operator`. The keyword `operator` precedes the operator token and replaces what would otherwise be a function name in a function declaration. The keyword `friend` gives a function access to the private members of a class variable. A `friend` function is not a member of the class but has the privileges of a member function in the class in which it is declared.

In file ovl_string.cpp

```
// Overloading the operator +

#include <string.h>
#include <iostream.h>

const int max_len = 255;

class string {
public:
   void assign(const char* st) { strcpy(s, st); len = strlen(st); }
   int  length() { return len; }
   void print() { cout << s << "\nLength: " << len << endl; }
   friend string operator+(const string& a, const string& b);
private:
   char s[max_len];
   int  len;
};

string operator+(const string& a, const string& b)    // overload +
{
   string temp;

   temp.assign(a.s);
   temp.len = a.len + b.len;
   if (temp.len < max_len)
      strcat(temp.s, b.s);
   else
      cerr << "Max length exceeded in concatenation.\n";
   return temp;
}

void print(const char* c)    // file scope print definition
{
   cout << c << "\nLength: " << strlen(c) << "\n";
}
```



```
int main()
{
   string   one, two, both;
   char     three[40] = {"My name is Charles Babbage."};

   one.assign("My name is Alan Turing.");
   two.assign(three);
   print(three);          // file scope print called
   // Print shorter of one and two.
   if (one.length() <= two.length())
      one.print();        // member function print called
   else
      two.print();
   both = one + two;      // plus overloaded to be concatenate
   both.print();
}
```

Dissection of the operator+ Function

- `string operator+(const string& a, const string& b)`

Plus is overloaded. The two arguments it will take are both strings. The arguments are call-by-reference. Use of `const` indicates that the arguments cannot be modified.

- `string temp;`

The function needs to return a value of type `string`. This local variable will be used to store and return the concatenated string value.

-
```
temp.assign(a.s);
temp.len = a.len + b.len;
if (temp.len < max_len)
   strcat(temp.s, b.s);
```

The string `a.s` is copied into `temp.s` by calling the `strcpy()` library function. The length of the resulting concatenated string is tested to see that it does not exceed the maximum length for strings. If the length is acceptable, the standard library function `strcat()` is called with the hidden string members `temp.s` and `b.s`. The references to `temp.s`, `a.s`, and `b.s` are allowed because this function is a `friend` of class `string`.

- `cerr << "Max length exceeded in concatenation.\n";`

The standard error stream `cerr` is used to print an error message, and no concatenation takes place. Only the first string will be returned.

- `return temp;`

The operator was given a return type of `string`, and `temp` has been assigned the appropriate concatenated string.

13.6 Constructors and Destructors

A *constructor* is a member function whose job is to *initialize* a variable of its class. In OOP terms, such a variable is an *object*. In many cases, this involves dynamic storage allocation. Constructors are invoked any time an object of its associated class is created. A *destructor* is a member function whose job is to deallocate, or *finalize*, a variable of its class. The destructor is called implicitly when an automatic object goes out of scope.

Let us change our `string` example by dynamically allocating storage for each `string` variable. We will replace the private array variable by a pointer. The remodeled class will use a constructor to allocate an appropriate amount of storage dynamically using the `new` operator.

```
// An implementation of dynamically allocated strings.

class string {
public:
   string(int n) { s = new char[n + 1]; len = n; }  // constructor
   void assign(const char* st)
      { strcpy(s, st); len = strlen(st); }
   int  length() { return len; }
   void print() { cout << s << "\nLength: " << len << "\n"; }
   friend string operator+(const string& a, const string& b);
private:
   char* s;
   int   len;
};
```


A constructor is a member function whose name is the same as the class name. The keyword `new` is an addition to the C language. It is a unary operator that takes as an argument a data type that can include an array size. It allocates the appropriate amount of memory from free store to store this type, and returns the pointer value that addresses this memory. In the preceding example, `n + 1` bytes would be allocated from free store. Thus, the declaration

```
string a(40), b(100);
```

would allocate 41 bytes for the variable `a`, pointed at by `a.s` , and 101 bytes for the variable `b`, pointed at by `b.s`. We add 1 byte for the end-of-string value 0. Storage obtained by `new` is persistent and is not automatically returned on block exit. When storage return is desired, a destructor function must be included in the class. A destructor is written as an ordinary member function whose name is the same as the class name preceded by the tilde symbol ~ . Typically, a destructor uses the unary operator `delete` or `delete[]`, another addition to the language, to automatically deallocate storage associated with a pointer expression. The `delete[]` is used anytime `new` *`type[size]`* was used for allocation.

```
// Add as a member function to class string.
~string() { delete []s; }   // destructor
```

It is usual to overload the constructor, writing a variety of such functions to accommodate more than one style of initialization. Consider initializing a string with a pointer to `char` value. Such a constructor is

```
string(const char* p)
{
   len = strlen(p);
   s = new char[len + 1];
   strcpy(s, p);
}
```

A typical declaration invoking this version of the constructor is

```
char*   str = "I came on foot.";
string  a("I came by bus."), b(str);
```

It would also be desirable to have a constructor of no arguments:

```
string() { len = 255; s = new char[255]; }
```

This would be invoked by declarations without parenthesized arguments and would, by default, allocate 255 bytes of memory. Now all three constructors would be invoked in the following declaration:

```
string  a, b(10), c("I came by horse.");
```

The overloaded constructor is selected by the form of each declaration. The variable a has no parameters and so is allocated 255 bytes. The variable b has an integer parameter and so is allocated 11 bytes. The variable c has a pointer parameter to the literal string "I came by horse." and so is allocated 17 bytes, with this literal string copied into its private s member.

13.7 Object-oriented Programming and Inheritance

A novel concept in OOP is the *inheritance* mechanism. This is the mechanism of *deriving* a new class from an existing one called the *base class.* The derived class adds to or alters the inherited base class members. This is used to share code and interface and to create a hierarchy of related types.

Hierarchy is a method for coping with complexity. It imposes classifications on objects. For example, the periodic table of elements has elements that are gases. These have properties that are shared by all elements in that classification. Inert gases are an important special class of gases. The hierarchy here is as follows: An inert gas, such as argon, is a gas, which, in turn, is an element. This hierarchy provides a convenient way to understand the behavior of inert gases. We know they are composed of protons and electrons, as this is shared description with all elements. We know they are in a gaseous state at room temperature, as this behavior is shared with all gases. We know they do not combine in ordinary chemical reactions with other elements, as this is shared behavior with all inert gases.

Consider designing a data base for a college. The registrar must track different types of students. The base class we need to develop captures a description of "student." Two main categories of student are graduate and undergraduate.

Here is the OOP design methodology:

OOP Design Methodology

1 Decide on an appropriate set of types.

2 Design in their relatedness.

3 Use inheritance to share code.

An example of deriving a class is as follows:

```
enum support { ta, ra, fellowship, other };
enum year { fresh, soph, junior, senior, grad };

class student {
public:
   student(char* nm, int id, double g, year x);
   void  print();
private:
   int       student_id;
   double    gpa;
   year      y;
   char      name[30];
};

class grad_student: public student {
public:
   grad_student
      (char* nm, int id, double g, year x, support t,
       char* d, char* th);
   void print();
private:
   support   s;
   char      dept[10];
   char      thesis[80];
};
```

In this example, grad_student is the derived class, and student is the base class. The use of the keyword public following the colon in the derived class header means that the public members of student are to be inherited as public members of grad_student. Private members of the base class cannot be accessed in the derived class. Public inheritance also means that the derived class grad_student is a subtype of student.

An inheritance structure provides a design for the overall system. For example, a data base that contained all the people at a college could be derived from the base class person. The student-grad_student relation could be extended to extension students, as a further significant category of objects. Similarly, person could be the base class for a variety of employee categories.

13.8 Polymorphism

A *polymorphic* function has many forms. An example in Standard C is the division operator. If the arguments to the division operator are integral, then integer division is used. However, if one or both arguments are floating-point, then floating-point division is used.

In C++, a function name or operator is overloadable. A function is called based on its *signature*, defined as the list of argument types in its parameter list.

```
a / b        // divide behavior determined by native coercions
cout << a    // overloading << the shift operator for output
```

In the division expression, the result depends on the arguments being automatically coerced to the widest type. So if both arguments are integer, the result is an integer division. If one or both arguments are floating-point, the result is floating-point. In the output statement, the shift operator << is invoking a function that is able to output an object of type a.

Polymorphism localizes responsibility for behavior. The client code frequently requires no revision when additional functionality is added to the system through ADT-provided code improvements.

In C, the technique for implementing a package of routines to provide an ADT shape would rely on a comprehensive structural description of any shape.

```
struct shape {
   enum{CIRCLE, .....} e_val;
   double center, radius;
.....
};
```

would have all the members necessary for any shape currently drawable in our system, plus an enumerator value so that it can be identified. The area routine would then be written as

```
double area(shape* s)
{
  switch(s -> e_val) {
  case CIRCLE:  return (PI * s -> radius * s -> radius);
  case RECTANGLE: return (s -> height * s -> width);
  .....
}
```


Question: What is involved in revising this C code to include a new shape? *Answer:* An additional case in the code body and additional members in the structure. Unfortunately, these would have ripple effects throughout our entire code body. Each routine so structured has to have an additional case, even when that case is just adding a label to a preexisting case. Thus, what is conceptually a local improvement requires global changes.

OOP coding techniques in C++ for the same problem use a shape hierarchy. The hierarchy is the obvious one where circle and rectangle are derived from shape. The revision process is one in which code improvements are provided in a new derived class, so additional description is localized. The programmer overrides the meaning of any changed routines—in this case, the new area calculation. Client code that does not use the new type is unaffected. Client code that is improved by the new type is typically minimally changed.

C++ code following this design uses `shape` as an *abstract base class.* This is a class containing one or more pure virtual functions. A pure virtual function does not have a definition. The definition is placed in a derived class.

```
// shape is an abstract base class

class shape {
public:
   virtual double area() = 0;     // pure virtual function
};

class rectangle: public shape {
public:
   rectangle(double h, double w): height(h), width(w) {}
   double area() { return (height * width); }    // overridden
private:
   double height, width;
};

class circle: public shape {
public:
   circle(double r): radius(r) {}
   double area() { return ( 3.14159 * radius * radius); }
private:
   double radius;
};
```

Client code for computing an arbitrary area is polymorphic. The appropriate `area()` function is selected at run-time.

```
shape*  ptr_shape;
   .....
   cout << " area = " << ptr_shape -> area();
   .....
```

Now imagine improving our hierarchy of types by developing a square class.

```
class square: public rectangle {
public:
   square(double h): rectangle(h,h) {}
   double area() { return rectangle::area(); }
};
```

The client code remains unchanged. This would not have been the case with the non-OOP code.

13.9 Templates

C++ uses the keyword `template` to provide *parametric polymorphism*. Parametric polymorphism allows the same code to be used with respect to different types, where the type is a parameter of the code body. The code is written generically to act on `class T`. The template is used to generate different actual classes when `class T` is substituted for with an actual type.

An especially important use for this technique is in writing generic *container classes*. A container class is used to contain data of a particular type. Stacks, vectors, trees, and lists are all examples of standard container classes. We shall develop a stack *container* class as a parameterized type.

In file stack.cpp

```
// template stack implementation

template <class TYPE>
class stack {
public:
   stack(int size = 1000) :max_len(size)
      { s = new TYPE[size]; top = EMPTY; }
   ~stack() { delete []s; }
   void reset() { top = EMPTY; }
   void push(TYPE c) { s[++top] = c; }
   TYPE pop() { return s[top--]; }
   TYPE top_of() { return s[top]; }
   bool empty() { return top == EMPTY; }
   bool full() { return top == max_len - 1; }
private:
   enum  {EMPTY = -1};
   TYPE* s;
   int   max_len;
   int   top;
};
```

The syntax of the class declaration is prefaced by

```
template <class identifier >
```

This identifier is a template argument that essentially stands for an arbitrary type. Throughout the class definition, the template argument can be used as a type name. This argument is instantiated in the actual declarations. An example of a `stack` declaration using this is

```
stack<char> stk_ch;                // 1000 element char stack
stack<char*>  stk_str(200);        // 200 element char* stack
stack<complex> stk_cmplx(100);     // 100 element complex stack
```

This mechanism saves us rewriting class declarations where the only variation would be type declarations.

When processing such a type, the code must always use the angle brackets as part of the declaration. Here are two functions using the stack template:

```
// Reversing a series of char* represented strings

void reverse(char* str[], int n)
{
   stack<char*> stk(n);   // this stack holds char*
   for (int i = 0; i < n; ++i)
      stk.push(str[i]);
   for (i = 0; i < n; ++i)
      str[i] = stk.pop();
}
```

In function reverse(), a stack<char*> is used to insert n strings and then pops them in reverse order.

```
// Initializing a stack of complex numbers from an array
void init(complex c[], stack<complex>& stk, n)
{
   for (int i = 0; i < n; ++i)
      stk.push(c[i]);
}
```

In function init(), a stack<complex> variable is passed by reference, and n complex numbers are pushed onto this stack. Notice that we used bool in this example. It is a basic type in C++ whose values are true and false.

13.10 C++ Exceptions

C++ introduces an exception-handling mechanism that is sensitive to context. The context for raising an exception will be a try block. Handlers declared using the keyword catch are found at the end of a try block.

An exception is raised by using the throw expression. The exception will be handled by invoking an appropriate *handler* selected from a list of handlers found immediately after the handler's try block. A simple example of all this is


```
// stack constructor with exceptions

stack::stack(int n)
{
   if (n < 1)
      throw (n);       // want a positive value
   p = new char[n];    // create a stack of characters
   if (p == 0)         // if new returns 0 when it fails
      throw ("FREE STORE EXHAUSTED");
}

void g()
{
   try {
      stack  a(n), b(n);
      .....
   }
   catch (int n) { ... }           // an incorrect size
   catch (char* error) { ... }     // free store exhaustion
}
```

The first throw() has an integer argument and matches the catch(int n) signature. This handler is expected to perform an appropriate action where an incorrect array size has been passed as an argument to the constructor. For example, an error message and abort are normal. The second throw() has a pointer to char argument and matches the catch(char* error) signature.

Modern ANSI C++ compilers throw the standard exception bad_alloc if new fails. Older systems returned the null pointer value 0, when new failed.

13.11 Benefits of Object-oriented Programming

The central element of OOP is the encapsulation of an appropriate set of data types and their operations. The class construct, with its member functions and data members, provides an appropriate coding tool. Class variables are the *objects* to be manipulated.

Classes also provide data hiding. Access privileges can be managed and limited to whatever group of functions needs access to implementation details. This promotes modularity and robustness.

Another important concept in OOP is the promotion of code reuse through the *inheritance* mechanism. This is the mechanism of *deriving* a new class from an existing one

called the *base* class. The base class can be added to or altered to create the derived class. In this way a hierarchy of related data types can be created that share code.

Many useful data structures are variants of one another, and it is frequently tedious to produce the same code for each. A derived class inherits the description of the base class. It can then be altered by adding additional members, overloading existing member functions, and modifying access privileges. Without this reuse mechanism, each minor variation would require code replication.

The OOP programming task is frequently more difficult than normal procedural programming as found in C. There is at least one extra design step before getting to the coding of algorithms involving the hierarchy of types that is useful for the problem at hand. Frequently, one is solving the problem more generally than is strictly necessary.

The belief is that OOP will pay dividends in several ways. The solution will be more encapsulated and thus, more robust and easier to maintain and change. Also, the solution will be more reusable. For example, where the code needs a stack, that stack is easily borrowed from existing code. In an ordinary procedural language, such a data structure is frequently "wired into" the algorithm and cannot be exported.

All these benefits are especially important for large coding projects that require coordination among many programmers. Here, the ability to have header files specify general interfaces for different classes allows each programmer to work on individual code segments with a high degree of independence and integrity.

OOP is many things to many people. Attempts at defining it are reminiscent of the story of the blind sages attempting to describe an elephant. We will offer one more equation:

OOP = type-extensibility + polymorphism

Summary

1 The double slash // is a new comment symbol. The comment runs to the end of the line. The old C bracketing comment symbols /* */ are still available for multiline comments.

2 The *iostream.h* header introduces I/O facilities for C++. The identifier `cout` is the name of the standard output stream. The operator << passes its argument to standard out. Used in this way, the << is referred to as the *put to* operator. The identifier `cin` is the name of the standard input stream. The operator >> is the input operator, called *get from*, that assigns values from the input stream to a variable.

3 C++ reduces C's traditional reliance on the preprocessor. Instead of using `define`, special constants are assigned to variables specified as `const`. The new keyword `inline` specifies that a function is to be compiled inline to avoid function call overhead. As a rule, this should be done sparingly and only on short functions.

4 The syntax of functions in C++ inspired the new function prototype syntax found in Standard C compilers. Basically, the types of parameters are listed inside the header parentheses—for example, `void add3(int&, int, int, int)`. Call-by-reference is available as well as default parameter values. By explicitly listing the type and number of arguments, strong type checking and assignment-compatible conversions are possible in C++.

5 What is novel about C++ is the aggregate type `class`. A `class` is an extension of the idea of `struct` in traditional C. Its use is a way of implementing a data type and associated functions and operators. Therefore, a `class` is an implementation of an abstract data type (ADT). There are two important additions to the structure concept: first, it includes members that are functions, and second, it employs access keywords `public`, `private`, and `protected`. These keywords indicate the visibility of the members that follow. Public members are available to any function within the scope of the class declaration. Private members are available for use only by other member functions of the class. Protected members are available for use only by other member functions of the class and by derived classes. Privacy allows part of the implementation of a class type to be "hidden."

6 The term *overloading* refers to the practice of giving several meanings to an operator or a function. The meaning selected will depend on the types of the arguments used by the operator or function.

7 A constructor is a member function whose job is to initialize a variable of its class. In many cases, this involves dynamic storage allocation. Constructors are invoked any time an object of its associated class is created. A destructor is a member function whose job is to finalize a variable of its class. The destructor is invoked implicitly when an automatic object goes out of scope.

8 The central element of object-oriented programming (OOP) is the encapsulation of an appropriate set of data types and their operations. These user-defined types are ADTs. The class construct, with its member functions and data members, provides an appropriate coding tool. Class variables are the *objects* to be manipulated.

9 Another important concept in OOP is the promotion of code reuse through the *inheritance* mechanism. This is the mechanism of *deriving* a new class from an existing one, called the *base class.* The base class can be added to or altered to create the derived class. In this way, a hierarchy of related data types can be created that share code. This typing hierarchy can be used dynamically by `virtual` functions. Virtual member functions in a base class are overloaded in a derived class. These functions allow for dynamic or run-time typing. A pointer to the base class can also point at objects of the derived classes. When such a pointer is used to point at the overloaded virtual function, it dynamically selects which version of the member function to call.

10 A *polymorphic* function has many forms. A `virtual` function allows run-time selection from a group of functions overridden within a type hierarchy. An example in the text is the area calculation within the `shape` hierarchy. Client code for computing an arbitrary area is polymorphic. The appropriate `area()` function is selected at run-time.

11 C++ uses the keyword `template` to provide *parametric polymorphism.* Parametric polymorphism allows the same code to be used with respect to different types, where the type is a parameter of the code body. The code is written generically to act on `class T`. The template is used to generate different actual classes when `class T` is substituted for with an actual type.

12 C++ introduces an exception-handling mechanism that is sensitive to context. The context for raising an exception will be a `try` block. Handlers declared using the keyword `catch` are found at the end of a `try` block. An exception is raised by using the `throw` expression. The exception will be handled by invoking an appropriate *handler* selected from a list of handlers found immediately after the handler's `try` block.

Exercises

1 Using stream I/O, write on the screen the words

```
she sells sea shells by the seashore
```

(a) all on one line, (b) on three lines, (c) inside a box.

2 Write a program that will convert distances measured in yards to distances measured in meters. The relationship is 1 meter equals 1.0936 yards. Write the program to use `cin` to read in distances. The program should be a loop that does this calculation until it receives a zero or negative number for input.

3 The following program reads in three integers and prints their sum. Observe that the expression `!cin` is used to test whether input into `a`, `b`, and `c` succeeded. To exit the `for` loop, the user can type `bye` or `exit`, or anything else that cannot be converted to an integer. Experiment with the program so that you understand its effects.

```
#include <iostream.h>

int main()
{
   int   a, b, c, sum;

   cout << "---\n"
           "Integers a, b, and c will be summed.\n"
           "\n";
   for ( ; ; ) {
      cout << "Input a, b, and c: ";
      cin >> a >> b >> c;
      if (!cin)
         break;
      sum = a + b + c;
      cout << "\n"
              "   a + b + c = " << sum << "\n"
              "\n";
   }
   cout << "\nBye!\n\n";
}
```

4 Some C++ systems provide a "big integer" type. In GNU C++, for example, this type is called `Integer`. We will write a program that uses this type to compute factorials.

```
#include <assert.h>
#include <iostream.h>
#include <Integer.h>              // valid for GNU g++\

int main()
{
   int       i;
   int       n;
   Integer   product = 1;

   cout << "The factorial of n will be computed.\n"
           "\n"
           "Input n: ";
   cin >> n;
   assert(cin && n >= 0);
   for (i = 2; i <= n; ++i)
      product *= i;
   cout << "\n"
           "factorial(" << n << ") = " << product << "\n"
           "\n";
}
```

Observe that the program looks similar to a factorial program written in C. However, when we execute this program, we learn, for example, that

```
factorial(37) = 13763753091226345046315979581580902400000000
```

Using a type such as `int` or `double` we cannot generate such large integers, but with a big integer type we can. If GNU C++ is available to you, try this program. Otherwise, find out if your C++ system has a big integer type, and if so, write a factorial program similar to this one. If your program computes 100 factorial correctly, it should end with 24 zeros. Does it?

5 Take a working program, omit each line in turn, and run it through your compiler. Record the error messages each such deletion causes. For example, use the code given in exercise 3.

6 Write a program that asks interactively for your *name* and *age* and responds with

`Hello` *name*`, next year you will be` *next_age*.

where *next_age* is *age* + 1.

7 Write a program that prints out a table of squares, square roots, and cubes. Use either tabbing or strings of blanks to get a neatly aligned table.

```
i      i * i     square root     i * i * i
-------------------------
1          1     1.00000                 1
2          4     1.41421                 8
.....
```

8 The C swapping function is

```
void swap(int *i, int *j)
{
   int  temp;

   temp = *i;
   *i = *j;
   *j = temp;
}
```

Rewrite this using reference parameters and test it.

```
void swap(int& i, int& j);
```

9 In traditional C, but not in ANSI C or C++, the following code causes an error:

```
#include <math.h>
#include <stdio.h>

int main()
{
   printf("%f is the square root of 2.\n", sqrt(2));
   return 0;
}
```

Explain the reason for this and why function prototypes in C++ avoid this problem. Rewrite using *iostream.h*.

10 Add to the class `string` in Section 13.4, "Classes and Abstract Data Types," on page 601, a member function `reverse`. This function reverses the underlying representation of the character sequence stored in the private member `s`.

11 Add to the class `string` in Section 13.4, "Classes and Abstract Data Types," on page 601, a member function `void print(int pos, int k)`. This function overloads `print()` and is meant to print the `k` characters of the string starting at position `pos`.

12 Overload the operator * in class `string`. Its member declaration will be

```
string string::operator*(int n);
```

The expression `s * k` will be a string that is *k* copies of the string `s`. Check that this does not overrun storage.

13 Write a `class person` that would contain basic information such as name, birthdate, and address. Derive `class student` from `person`.

14 Write a `class triangle` that inherits from `shape`. It needs to have its own `area()` member function.

15 The function `reverse()` can be written generically as follows:

```
// generic reversal

template <class T>
void reverse(T v[], int n)
{
   stack<T> stk(n);
   for (int i = 0; i < n; ++i)
      stk.push(v[i]);
   for (i = 0; i < n; ++i)
      v[i] = stk.pop();
}
```

Try this on your system, using it to reverse an array of characters and to reverse an array of `char*`.

16 (S. Clamage.) The next three programs behave differently:

```
// Function declarations at file scope

int f(int);
double f(double);   // overloads f(int)

double add_f()
{
   return (f(1) + f(1.0));   // f(int) + f(double)
}
```


Now we place one function declaration internally.

```
// Function declaration at local scope

int f(int);

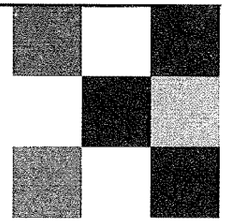
double add_f()
{
   double f(double);            // hides f(int)
   return (f(1) + f(1.0));      // f(double) + f(double)
}
```

Now we place the other function declaration internally.

```
double f(double);

double add_f()
{
   int f(int);
   return (f(1) + f(1.0));      // What is called here?
}
```

Write some test programs that clearly show the different behaviors.



Chapter 14

Moving from C to Java

This chapter gives an overview of the Java programming language. It also provides an introduction to Java's use as an object-oriented programming language. It is organized in the manner of the C++ chapter with a series of programs presented, and the elements of each program explained. The programs increase in complexity, and the examples in the later sections illustrate some of the concepts of object-oriented programming. It can be read independently of the C++ chapter.

The examples in this chapter give simple, immediate, hands-on experience with key features of the Java language. The chapter introduces the reader to Java I/O, classes, inheritance, graphics, threads, and exceptions. Mastery of the individual topics requires a thorough reading of a companion book such as Arnold and Gosling, *The Java Programming Language* (Reading, MA: Addison-Wesley, 1996).

Object-oriented programming is implemented by the `class` construct. The `class` construct in Java is an extension of `struct` in C. The later examples in this chapter illustrate how Java implements OOP (object-oriented programming) concepts, such as data hiding, ADTs, inheritance, and type hierarchies. Java has been designed to be used on the World Wide Web. It has special libraries designed for graphics and communication across the Net. It is designed to run in a machine- and system-independent manner. This means the Java program will execute with the same results on a PC running Windows 95 or a workstation running SUN Solaris. It does this by defining its semantics completely in terms of a virtual machine. The job for a system that wants to run Java is to port the virtual machine. This is a trade-off between portability and efficiency. There inevitably is additional overhead in a machine running a simulator of a different architecture. Some of this inefficiency can be overcome by the use of just-in-time compilers or the use of native code written in C that is used where efficiency is crucial. On many platforms, it is also possible to employ a direct-to-native code compiler for maximum run-time efficiency.

14.1 Output

Programs must communicate to be useful. Our first example is a program that prints on the screen the phrase "Java is an improved C." The complete program is

In file Improved.java

```
// A first Java program illustrating output.
//    Title:  Improved
//    Author: Richmond Q. Programmer

class Improved {
   public static void main (String[] args)
   {
      System.out.println("Java is an improved C.");
   }
}
```

The program prints on the screen

```
Java is an improved C.
```

This program is compiled using the command *javac improved.java*; resulting in a code file named *Improved.class*. This can be run using the command *java Improved*.

Dissection of the *improved* Program

- `// A first Java program illustrating output.`

The double slash // is a new comment symbol. The comment runs to the end of the line. The old C bracketing comment symbols /* */ are still available for multiline comments. Java also provides /** */ bracketing comment symbols for a doc comment. A program *javadoc* uses doc comments and generates an HTML file.

- `class Improved {`

Java programs are classes. A `class` has syntactic form that is derived from the C `struct`, which is not in Java. In Java, class identifier names, such as `Improved`, are by convention capitalized. Data and code are placed within classes.

- `public static void main (String[] args )`

When a class is executed as a program, it starts by calling the member function `main()`. In this case, `main()` is a member of `Improved`. In Java, command line arguments are passed in an array of `String`'s. In C, we need an `argc` variable to tell the program the number of command line arguments. In Java, this array length is found by using `args.length`.

- `System.out.println("Java is an improved C.");`

This statement prints to the screen. The `System.out` object uses the member function `println()` to print. The function prints the string and adds a new line, which moves the screen cursor to the next line. Unlike `printf()` in C, `println()` does not use format controls.

In Java, all functions are contained in classes. In this case, the function `main()` is a member of `class Improved`. A member function is called a *method*.

14.2 Variables and Types

We will write a program to convert to kilometers the distance in miles from the Earth to the moon. In miles this distance is, on average, 238,857 miles. This number is an integer. To convert miles to kilometers, we multiply by the conversion factor 1.609, a real number.

Our conversion program will use variables capable of storing integer values and real values. The variables in the following program will be declared in `main()`. Java cannot have variables declared as `extern` (in other words, as global or file scope variables).

The primitive types in a Java program can be `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. These types are always identically defined regardless of the machine or system they run on. For example, the `int` type is always a signed 32-bit inte-

ger, unlike in C, where this can vary from system to system. The boolean type is not an arithmetic type and cannot be used in mixed arithmetical expressions. The `char` type use 16-bit Unicode values. The `byte`, `short`, `int`, and `long` are all signed integer types, whose length in bits is 8, 16, 32, and 64 respectively. Unlike in C, unsigned types are not provided. The floating types comply with IEEE754 standards and are `float`, a 32-bit size, and `double`, a 64-bit size. The non-primitive types are class types and array types, and variables of these types take references as their value.

In file Moon.java

```
// The distance to the moon converted to kilometers.
//    Title: moon

public class Moon {
   public static void main(String[] s) {
      int moon = 238857;
      int moon_kilo;

      System.out.println("Earth to moon = " + moon + " mi.");
      moon_kilo = (int)(moon * 1.609);
      System.out.println("Kilometers = " + moon_kilo +"km.");
   }
```

The output of the program is

```
Earth to moon = 238857 mi.
Kilometers = 384320 km.
```

Dissection of the *Moon* Program

```
int moon = 238857;
```

Variables of type int are signed 32-bit integers. They can be initialized as in C.

```
System.out.println("Earth to moon = " + moon + " mi.");
```

The `println()` method can discriminate among a variety of simple values without needing additional formatting information. Here, the value of `moon` will be printed as an integer. The symbol `+` represents string concatenation. Using "plus" `println()` can print a list of arguments. What is happening is that each argument is converted from its

specific type to an output string that is concatenated together and printed along with a newline.

```
moon_kilo = (int)(moon * 1.609);
```

The mixed expression `moon * 1.609` is a `double`. It must be explicitly converted to `int`.

Note that narrowing conversions that are implicit in C are not done in Java.

14.3 Classes and Abstract Data Types

What is novel about Java is its aggregate type `class`. A `class` is an extension of the idea of `struct` in traditional C. A `class` provides the means for implementing a user-defined data type and associated functions. Therefore, a `class` can be used to implement an ADT. Let us write a `class` called `Person` that will be used to store information about people.

In file Person.java

```
// An elementary implementation of type Person.

class Person {
   private String name;
   private int age;
   private char gender;          //male == 'M' , female == 'F'

   public void assignName(String nm) { name = nm; }
   public void assignAge(int a) { age = a; }
   public void assignGender(char b) { gender = b; }
   public String toString(
       return (name + " age is " + age +
       " sex is " + gender );
   }
};
```

Two important additions to the structure concept of C are found in this example: first, it has members called *class methods* that are functions, such as `AssignAge()`, and second, it has both public and private members. The keyword `public` indicates the visibility of the members that follow it. Without this keyword, the members are private to the class. Private members are available for use only by other member functions of the class. Public members are available anywhere the class is available. Privacy allows part of the implementation of a class type to be "hidden." This restriction prevents unanticipated modifications to the data structure. Restricted access, or *data hiding*, is a feature of object-oriented programming.

The declaration of methods inside a class allows the ADT to have actions or behaviors that can act on its private representation. For example, the member function `toString()` has access to private members and gives `Person` a string representation used in output. This method is common to many class types.

We can now use this data type `Person` as if it were a basic type of the language. Other code that uses this type is a *client*. The client can use only the public members to act on variables of type `Person`.

```
//PersonTest.java uses Person.

public class PersonTest {
    public static void main (String[] args )
    {
        System.out.println("Person test:");
        Person p1 = new Person();    //create a Person object
        p1.assignAge(20);
        p1.assignName("Alan Turing");
        p1.assignGender(false);
        System.out.println(p1.toString());
    }
}
```

The output of this example program is

```
Person test:
Alan Turing age is 20 sex is M
```

Notice the use of `new Person()` to create an actual instance of `Person`. The `new` operator goes off to the heap, much as `malloc()` does in C and obtains memory for creating an actual instance of object `Person`. The value of `p1` is a reference to this object. In effect, this is the address of the object.

14.4 Overloading

The term *overloading* refers to the practice of giving several meanings to a method. The meaning selected depends on the types of the arguments passed to the method called the method's *signature*. Let us overload the function `assignGender()` in the previous example. This will be a second definition of the `assignGender()` method.

```
class Person {
.....
      public void assignGender(char b) { gender = b; }
      public void assignGender(String b)
       { gender = ((b == "M")? 'M': 'F'); }
}
```

This version of `assignGender()` takes a single argument of type `String`. It will convert and store this properly as a `gender` character value. Now a user can use either a `char` or a `String` value in assigning gender.

14.5 Construction and Destruction of Class Types

A *constructor* is a function whose job is to *initialize* an object of its class. Constructors are invoked after the instance variables of a newly created class object have been assigned default initial values and any explicit initializers are called. Constructors are frequently overloaded.

A constructor is a member function whose name is the same as the class name. The constructor is not a method and does not have a return type. Let us change our `Person` example to have constructors to initialize the name instance variable.

```
//constructor to be placed in Person

public Person() {name = "Unknown";}
public Person(String nm) { name =nm;}
public Person(String nm, int a, char b)
   { name =nm; age =a; gender = b;}
```

These would be invoked when new gets used to associate a created instance with the appropriate type reference variable. For example,

```
p1 = new Person();                      //creates "unknown 0 M
p1 = new Person("Laura Pohl");          //creates Laura Pohl 0 M
p1 = new Person("Laura Pohl" 9, 'F');   //creates Laura Pohl 9 F
```

The overloaded constructor is selected by the set of arguments that matches the constructors parameter list.

Destruction is done automatically by the system using automatic garbage collection. When the object can no longer be referenced, for example, when the existing reference is given a new object, the now inaccessible object is called garbage. Periodically, the system sweeps through memory and retrieves these "dead" objects. The programmer need not be concerned with such apparent memory leaks.

14.6 Object-oriented Programming and Inheritance

In Java, inheritance is the mechanism of *extending* a new class from an existing one called the *superclass*. The extended class adds to or alters the inherited superclass methods. This is used to share interface and to create a hierarchy of related types.

Consider designing a data base for a college. The registrar must track different types of students. The superclass we start with will be `Person1`. This class will be identical to `Person`, except that the `private` instance variables will be changed to have access `protected`. This access allows their use in the subclass, but otherwise acts like `private`.

Here is an example of deriving a class:


```
// Note Person1 is Person with private instance variables
// made protected

class Student extends Person1 {
   private String college;
   private byte year;          //1 = fr, 2 = so, 3 = jr, 4 = sr
   private double gpa;         //0.0 to 4.0
   public void assignCollege(String nm) { college = nm; }
   public void assignYear(byte a) { year = a; }
   public void assignGpa(double g) { gpa = g; }
   public String toString()
      { return (super.toString() + "  College is " + college); }
   public Student()
      {super.assignName("Unknown"); college = "Unknown";}
   public Student(String nm)
       { super(nm); college = "Unknown"; }
   public Student(String nm, int a, char b)
       { name =nm; age =a; gender = b; }
};
```

In this example, `Student` is the subclass, and `Person1` is the superclass. Notice the use of the keyword `super`. It provides a means of accessing the instance variables or methods found in the superclass.

The inheritance structure provides a design for the overall system. The superclass `Person1` leads to a design where the subclass `Student` is derived from it. Other subclasses such as `GradStudent` or `Employee` could be added to this inheritance hierarchy.

14.7 Polymorphism and Overriding Methods

In Java, polymorphism comes from both method overloading and method overriding. Overloading has already been discussed. Overriding occurs when a method is redefined in the subclass. The `toString()` method is in `Person1` and is redefined in `Student` extended from `Person1`.

```
//Overriding the printName() method
class Person1 {
   protected String name;
   protected int age;
   protected char gender;          //male == 'M' , female == 'F'
   public toString() {
       return(name + " age is " + age +
              " sex is " + (gender == 'F' ? "F": "M") );
   }
.....
};

class Student extends Person1 {
   private String college;
   private byte year;
   private double gpa;             //0.0 to 4.0
   public toString()
       { return(super.toString() + "  College is " + college); }
.....
};
```

The overridden method `toString()` has the same name and signature in both the superclass `Person1` and the subclass `Student`. Which one gets selected depends at runtime on what is being referenced. For example, in the code

```
//StudentTest.java use Person1

public class StudentTest {
    public static void main (String[] args )
    {
        Person1 q1;
        q1 = new Student();
        q1.assignName("Charles Babbage");
        System.out.println(q1.toString());
        q1 = new Person1();
        q1.assignName("Charles Babbage");
        System.out.println(q1.toString());
    }
}
```

The variable `q1` can refer to either `Person1` object or the subtype `Student` object. At runtime the correct `toString()` will be selected. The `assignName()` method is known at compile time since it is the superclass `Person1` method.

14.8 Applets

Java is known for providing applets on Web pages. A browser is used to display and execute the applet. Typically, the applet provides a graphical user interface to the code. The next piece of code will be an applet for computing the greatest common divisor for two numbers.

```
//GCD applet implementations

import java.applet.*; //gets the applet superclass
import java.awt.*;    //abstract windowing toolkit
import java.io.*;

//derived from the class Applet

public class wgcd extends Applet {
   int x, y, z, r;
   TextField a = new TextField(10);        //input box
   TextField b = new TextField(10);        //input box
   TextField c = new TextField(10);        //output box
   Label l1 = new Label("Value1: ");
   Label l2 = new Label("Value2: ");
   Button gcd = new Button("   GCD: ");

//draws the screen layout such as the TextFields

   public void init() {
      setLayout(new FlowLayout());
      c.setEditable(false);
      add(l1); add(a);
      add(l2); add(b);
      add(gcd); add(c);
   }

//computes the greatest common divisor

   public int gcd(int m, int n) {
      while (n !=0) {
         r = m % n;
         m = n;
         n = r;
      }
      return m;
   }
```

```
//looks for screen events to interact with

   public boolean action(Event e, Object o) {
      if ("   GCD: ".equals(o)) { //press button
         x = Integer.parseInt(a.getText());
         y = Integer.parseInt(b.getText());
         z = gcd(x,y);
        //place answer in output TextField
         c.setText(Integer.toString(z));
      }
      return true;
   }
};
```

The code uses the graphics library awt and the applet class to draw an interactive interface that can be executed either by a special program called the *appletviewer* or by a Java-aware browser such as Microsoft Explorer or Netscape Navigator. Unlike ordinary Java programs, this program does not use a `main()` method to initiate the computation. Instead, the `init()` method draws the screen. Further computation is event-driven and processed by the `action()` method. The user terminates the applet by clicking on the Quit command in the applet pull-down menu.

14.9 Java Exceptions

Java has an exception-handling mechanism that is integral to the language and is heavily used for error detection at runtime. It is similar to the one found in C++. An exception is thrown by a method when it detects an error condition. The exception will be handled by invoking an appropriate *handler* selected from a list of handlers called catches. These explicit catches occur at the end of an enclosing `try` block. An uncaught exception is handled by a default Java handler that issues a message and terminates the program. An exception is itself an object, which must be derived from the superclass `Throwable`. As a simple example of all this, we will add an exception `NoSuchNameException` to our `Person` example class.

```
class NoSuchNameException extends Exception {
   public String str() { return name; }
   public String name;
   NoSuchNameException(String p) { name = p; }
};
```


The purpose of this exception is to report an incorrect or improperly formed name. In many cases, exceptions act as assertions would in the C language. They determine whether an illegal action has occurred and report it. We now modify the Person code to take advantage of the exception.

```
//Person2.class: Person with exceptions added

class Person2 {
   private String name;
   public Person2(String p)throws NoSuchNameException {
      if (p == "")
         throw new NoSuchNameException(p);
      name = p;  }
   public String toString(){ return name;}
   public static void main(String[] args)
         throws NoSuchNameException
   {
       try{
          Person2 p = new Person2("ira pohl");
          System.out.println("PERSONS");
          System.out.println(p.toString());
          p = new Person2("");
       }
      catch(NoSuchNameException t)
         { System.out.println(" exception with name " + t.str()); }
      finally
         { System.out.println("finally clause"); }
};
```

The `throw()` has a `NoSuchNameException` argument and matches the `catch()` signature. This handler is expected to perform an appropriate action where an incorrect name has been passed as an argument to the `Person2` constructor. As in this example, an error message and abort are normal. The `finally` clause is shown here. It is code that is done regardless of how the try block terminates.

14.10 Benefits of Java and OOP

Java shares with C++ the use of classes and inheritance to build software in an object-oriented manner. Also, both use data hiding and have methods that are bundled within the class.

Unlike C++, Java does not allow for conventional programming. Everything is encapsulated in some class. This forces the programmer to think and design everything as an object. The downside is that conventional C code is not as readily adapted to Java as it is to C++. Java avoids most of the memory pointer errors that are common to C and C++. Address arithmetic and manipulation are done by the compiler and system—not the programmer. Therefore, the Java programmer writes safer code. Also, memory reclamation is automatically done by the Java garbage collector.

Another important concept in OOP is the promotion of code reuse through the *inheritance* mechanism. In Java, this is the mechanism of *extending* a new class called a *subclass* from an existing one called the *superclass*. Methods in the extended class override the superclass methods. The method selection occurs at runtime and is a highly flexible polymorphic style of coding.

Java, in a strict sense, is completely portable across all platforms that support it. Java is compiled to byte code that is run on the Java virtual machine. This is typically an interpreter—code that understands the Java byte code instructions. Such code is much slower than native code on most systems. The trade-off here is between universally consistent behavior versus loss of efficiency.

Java has extensively developed libraries for performing Web-based programming. It has the ability to write graphical user interfaces that are used interactively. It also has a thread package and secure web communication features that let the coder write distributed applications.

Java is far simpler than C++ in the core language and its features. In some ways, this is deceptive in that much of the complexity is in its libraries. Java is far safer because of very strict typing, avoidance of pointer arithmetic, and well-integrated exception handling. It is system-independent in its behavior, so one size fits all. This combination of OOP, simplicity, universality, and Web-sensitive libraries makes it the language of the moment.

Summary

1 The double slash // is a new comment symbol. The comment runs to the end of the line. The old C bracketing comment symbols /* */ are still available for multiline comments. Java also provides /** */ bracketing comment symbols for a doc comment. A program *javadoc* uses doc comments and generates an HTML file.

2 Java programs are classes. A `class` has syntactic form that is derived from the C `struct`, which is not in Java. Data and code are placed within classes. When a class is executed as a program, it starts by calling the member function `main()`.

3 The term *overloading* refers to the practice of giving several meanings to a method. The meaning selected depends on the types of the arguments passed to the method called the method's *signature*.

4 A *constructor* is a function whose job is to *initialize* an object of its class. Constructors are invoked after the instance variables of a newly created class object have been assigned default initial values and any explicit initializers are called. Constructors are frequently overloaded. Destruction is done automatically by the system using automatic garbage collection.

5 Inheritance is the mechanism of *extending* a new class from an existing one called the *superclass*. The extended class adds to or alters the inherited superclass methods. This is used to share interface and to create a hierarchy of related types.

6 In Java, polymorphism comes from method overloading and method overriding. Overriding occurs when a method is redefined in the subclass. The selection of the appropriate overridden method definition is decided at runtime depending on the object's type.

7 Java is known for providing applets on Web pages. A browser is used to display and execute the applet. Typically, the applet provides a graphical user interface to the code. In applets, an `action()` method picks up an event, such as a mouse click, and decides on a next action.

8 An exception is thrown by a method when it detects an error condition. The exception will be handled by invoking an appropriate *handler* selected from a list of handlers called catches. These explicit catches occur at the end of an enclosing try block. An uncaught exception is handled by a default Java handler that issues a message and terminates the program.

9 Unlike C++, Java does not allow for conventional programming. Everything is encapsulated in some class. This forces the programmer to think and design everything as an object. The downside is that C code is not as readily adapted to Java as it is to C++. The upside is that the Java programmer writes safer code.

Exercises

1 Using Java I/O, output (a) all on one line, (b) on three lines, (c) inside a box:

```
she sells sea shells by the seashore
```

2 Write a program that will convert distances measured in yards to distances measured in meters. The relationship is 1 meter equals 1.0936 yards. If you can, write the program to read in distances; otherwise, do it by simple assignment to an instance variable inside the method `main()`.

3 Write an applet that asks interactively for your *name* and *age* and responds with

`Hello` *name*`, next year you will be` *next_age*`.`

where *next_age* is *age* + 1.

4 Write a program that prints out a table of squares, square roots, and cubes. Use either tabbing or strings of blanks to get a neatly aligned table.

```
i      i * i     square root     i * i * i
-------------------------
1          1     1.00000                 1.....
```

5 Write a class that can perform complex arithmetic. Unlike C++, in Java you cannot overload operators. Write method `Complex.plus(Complex)` and method `Complex.minus(Complex)`, such that they return an appropriate `Complex` result.

6 Write a `class GradStudent()` that extends `Student`. Add to Student additional information that includes the graduate student's thesis topic and routines for assigning to this part of the class and printing out this information.

7 Add exceptions to the `class GradStudent` so that improperly initializing `GradStudent` objects will result in a run-time exception.

Appendix A

The Standard Library

The standard library provides functions that are available for use by the programmer. Associated with the library are standard header files provided by the system. These header files contain prototypes of functions in the standard library, macro definitions, and other programming elements. If a programmer wants to use a particular function from the library, the corresponding header file should be included. Here is a complete list of the header files:

C header files			
<assert.h>	<limits.h>	<signal.h>	<stdlib.h>
<ctype.h>	<locale.h>	<stdarg.h>	<string.h>
<errno.h>	<math.h>	<stddef.h>	<time.h>
<float.h>	<setjmp.h>	<stdio.h>	

These files may be included in any order. Also, they may be included more than once, and the effect will be the same as if they were included only once. In this appendix, we organize our discussion by header file.

A.1 Diagnostics: <assert.h>

This header file defines the `assert()` macro. If the macro `NDEBUG` is defined at the point where `<assert.h>` is included, then all assertions are effectively discarded.

- `void assert(int expr);`

If `expr` is zero (*false*), then diagnostics are printed and the program is aborted. The diagnostics include the expression, the file name, and the line number in the file.

A.2 Character Handling: <ctype.h>

This header defines several macros that are used to test a character argument. In addition, there are function prototypes for two functions used to map a character argument.

Testing a Character

```
int isalnum(int c);      /* is alphanumeric */
int isalpha(int c);      /* is alphabetic */
int iscntrl(int c);      /* is control */
int isdigit(int c);      /* is digit: 0-9 */
int isgraph(int c);      /* is graphic */
int islower(int c);      /* is lowercase */
int isprint(int c);      /* is printable */
int ispunct(int c);      /* is punctuation */
int isspace(int c);      /* is white space */
int isupper(int c);      /* is uppercase */
int isxdigit(int c);     /* is hex digit 0-9, a-f, A-F */
```

These character tests are typically implemented as macros; see *ctype.h* in your installation for details. If the argument `c` satisfies the test, then a nonzero value (*true*) is returned; otherwise, zero (*false*) is returned. These macros should also be available as functions.

The printing characters are implementation-defined, but each occupies one printing position on the screen. A graphic character is any printing character, except for a space. Thus, a graphic character puts a visible mark on a single printing position on the screen. A punctuation character is any printing character other than a space or a character `c` for which `isalnum(c)` is true. The standard white space characters are space, form feed (`'\f'`), newline (`'\n'`), carriage return (`'\r'`), horizontal tab (`'\t'`), and vertical tab (`'\v'`). The control characters are the audible bell (`'\a'`), backspace (`'\b'`), any character `c` for which `isspace(c)` is true other than space, and control-c, control-h, and so on.

Mapping a Character

The two functions `tolower()` and `toupper()` are used to map a character argument. *Caution:* Early versions of many ANSI C compilers did not implement these functions correctly.

- `int tolower(int c);`

If `c` is an uppercase letter, the corresponding lowercase letter is returned; otherwise, `c` is returned.

- `int toupper(int c);`

If `c` is a lowercase letter, the corresponding uppercase letter is returned; otherwise, `c` is returned.

The next three macros often occur on ASCII machines. The first two are related to, but not the same as, `tolower()` and `toupper()`.

```
#define   _tolower(c)   ((c) + 'a' - 'A')
#define   _toupper(c)   ((c) + 'A' - 'a')
#define   toascii(c)    ((c) & 0x7f)
```

The hexadecimal constant `0x7f` is a mask for the low-order 7 bits.

A.3 Errors: <errno.h>

The identifier `errno` is defined here, along with several macros that are used to report error conditions.

```
extern int   errno;
```

Typically, there are lots of macros in *errno.h*. Which macros occur is system-dependent, but all names must begin with E. Various library functions use these macros for error reporting.

Two macros are common to all systems. These are used by the mathematical functions in the library:

```
#define   EDOM     33       /* domain error */
#define   ERANGE   34       /* range error */
```

Values other than 33 and 34 could be used here, but these values are typical.

The domain of a mathematical function is the set of argument values for which it is defined. For example, the domain of the square root function is the set of all nonnegative numbers. A *domain error* occurs when a mathematical function is called with an argument not in its domain. When this happens, the system assigns the value EDOM to `errno`. The programmer can use `perror()` and `strerror()` to print a message associated with the value stored in `errno`.

A *range error* occurs when the value to be returned by the function is defined mathematically but cannot be represented in a `double`. When this happens, the system assigns the value ERANGE to `errno`.

A.4 Floating Limits: <float.h>

Macros that define various floating characteristics and limits are defined here. There are many of them. Some examples are

```
#define   DBL_MAX         1.7976931348623157e+308
#define   FLT_MAX         3.40282347e+38F
#define   LDBL_MAX        1.7976931348623157e+308

#define   DBL_MIN         2.2250738585072014e-308
#define   FLT_MIN         1.17549435e-38F
#define   LDBL_MIN        2.2250738585072014e-308

#define   DBL_EPSILON     2.2204460492503131e-16
#define   FLT_EPSILON     1.19209290e-07F
#define   LDBL_EPSILON    2.2204460492503131e-16
```

The constants are system-dependent. We are assuming that a `long double` is implemented as a `double`. Not all systems do this. Some provide more precision and range; see *float.h* on your system.

A.5 Integral Limits: <limits.h>

Macros that define various integral characteristics and limits are defined here. There are many of them. Some examples are

```
#define   CHAR_BIT    8       /* number of bits in a byte */
#define   CHAR_MAX    127
#define   CHAR_MIN    (-128)
#define   SHRT_MAX    32767
#define   SHRT_MIN    (-32768)
#define   INT_MAX     2147483647
#define   INT_MIN     (-2147483648)
```

The constants are system-dependent.

A.6 Localization: <locale.h>

This header contains programming constructs that can be used to set or access properties suitable for the current locale. The following structure type is defined:

```
struct lconv {
   char   *decimal_point;
   char   *thousands_sep;
   char   *currency_symbol;
   .....
};
```

The members allow for local variations, such as using a comma instead of a period for a decimal point. At least six symbolic constants are defined.

```
#define LC_ALL        1 /* all categories */
#define LC_COLLATE    2 /*strcoll() and strxfrm */
#define LC_CTYPE      3 /*character handling functions */
#define LC_MONETARY   4 /* monetary info in localeconv() */
#define LC_NUMERIC    5 /*decimal point in lib fcts */
#define LC_TIME       6 /*strftime()*/
```

The values of the symbolic constants are system-dependent. Other macros beginning with LC_ can be specified. These macros can be used as the first argument to the setlocale() function.

- `char *setlocale(int category, const char *locale);`

The first argument is typically one of the above symbolic constants. The second argument is "C", "", or some other string. The function returns a pointer to a string of static duration, supplied by the system, that describes the new locale, if it is available; otherwise, the NULL pointer is returned. At program startup, the system behaves as if

```
setlocale(LC_ALL, "C");
```

has been executed. This specifies a minimal environment for C translation. The statement

```
setlocale(LC_ALL, "");
```

specifies the native environment, which is system-dependent. Using a macro other than LC_ALL affects only part of the locale. For example, LC_MONETARY affects only that part of the locale dealing with monetary information.

- `struct lconv *localeconv(void);`

A pointer to a structure provided by the system is returned. It is of static duration and contains numeric information about the current locale. Further calls to the function setlocale() can change the values stored in the structure.

A.7 Mathematics: <math.h>

This header file contains prototypes for the mathematical functions in the library. It also contains one macro definition:

```
#define   HUGE_VAL   1.7976931348623157e+308
```

The value of the macro is system-dependent.

The domain of a mathematical function is the set of argument values for which it is defined. A *domain error* occurs when a mathematical function is called with an argument not in its domain. When this happens, the function returns a system-dependent value, and the system assigns the value EDOM to errno.

A *range error* occurs when the value to be returned by the function is defined mathematically but cannot be represented in a double. If the value is too large in magnitude (overflow), then either HUGE_VAL or -HUGE_VAL is returned. If the value is too small in magnitude (underflow), zero is returned. On overflow, the value of the macro ERANGE is stored in errno. What happens on underflow is system-dependent. Some systems store ERANGE in errno; others do not.

- ```
  double cos(double x);
  double sin(double x);
  double tan(double x);
  ```

These are the cosine, sine, and tangent functions, respectively.

- ```
  double acos(double x);              /* arccosine of x */
  double asin(double x);              /* arcsine of x */
  double atan(double x);              /* arctangent of x */
  double atan2(double y, double x);   /* arctangent of y/x */
  ```

These are inverse trigonometric functions. The angle *theta* returned by each of them is in radians. The range of the acos() function is [0, π]. The range of the asin() and atan() functions is [-π/2, π/2]. The range of the atan2() function is [-π, π]. Its principal use is to assist in changing rectangular coordinates into polar coordinates. For the functions acos() and asin(), a domain error occurs if the argument is not in the range [-1, 1]. For the function atan2(), a domain error occurs if both arguments are zero and y/x cannot be represented.

- ```
  double cosh(double x);
  double sinh(double x);
  double tanh(double x);
  ```

These are the hyperbolic cosine, hyperbolic sine, and hyperbolic tangent functions, respectively.

- ```
  double exp(double x);
  double log(double x);
  double log10(double x);
  ```

The exp() function returns e^x. The log() function returns the natural logarithm (base *e*) of x. The log10() function returns the base 10 logarithm of x. For both log functions, a domain error occurs if x is negative. A range error occurs if x is zero and the logarithm of zero cannot be represented. (Some systems can represent infinity.)

■ `double ceil(double x);`
`double floor(double x);`

The ceiling function returns the smallest integer not less than x. The floor function returns the largest integer not greater than x.

■ `double fabs(double x);  /* floating absolute value */`

Returns the absolute value of x. *Caution:* The related function `abs()` is designed for integer values, not floating values. Do not confuse `abs()` with `fabs()`.

■ `double fmod(double x, double y);  /* floating modulus */`

Returns the value x (mod y). More explicitly, if y is nonzero, the value `x - i * y` is returned, where `i` is an integer such that the result is zero, or has the same sign as x and magnitude less than the magnitude of y. If y is zero, what is returned is system-dependent, but zero is typical. In this case a domain error occurs on some systems.

■ `double pow(double x, double y);  /* power function */`

Returns x raised to the y power. A domain error occurs if x is negative and y is not an integer.

■ `double sqrt(double x);  /* square root */`

Returns the square root of x, provided x is nonnegative. A domain error occurs if x is negative.

■ `double frexp(double value, int *exp_ptr); /* free exponent */`

This is a primitive used by other functions in the library. It splits `value` into mantissa and exponent. The statement

```
x = frexp(value, &exp);
```

causes the relationship

$$value = x * 2^{exp}$$

to hold, where the magnitude of x is in the interval [1/2, 1] or x is zero.

■ `double ldexp(double x, int exp);  /* load exponent */`

The value $x * 2^{exp}$ is returned.

■ `double modf(double value, double *i_ptr);`

Breaks `value` into integer and fractional parts. The function call `modf(value, &i)` returns the value *f*, and indirectly the value `i`, so that

```
value = i + f
```

A.8 Nonlocal Jumps: <setjmp.h>

This header provides one type definition and two prototypes. These declarations allow the programmer to make nonlocal jumps. A nonlocal jump is like a `goto`, but with the flow of control leaving the function in which it occurs. The type definition is system-dependent. The following is an example:

```
typedef   long   jmp_buf[16];
```

An array of type `jmp_buf` is used to hold system information that will be used to restore the calling environment.

■ `int setjmp(jmp_buf env);`

Saves the current calling environment in the array `env` for later use by `longjmp()` and returns zero. Although on many systems this is implemented as a function, in ANSI C it is supposed to be implemented as a macro.

■ `void longjmp(jmp_buf env, int value);`

The function call `longjmp(env, value)` restores the environment saved by the most recent invocation of `setjmp(env)`. If `setjmp(env)` was not invoked, or if the function in which it was invoked is no longer active, the behavior is undefined. A successful call causes program control to jump to the place following the previous call to `setjmp(env)`. If `value` is nonzero, the effect is as if `setjmp(env)` were called again with `value` being returned. If `value` is zero, the effect is as if `setjmp(env)` were called again with one being returned.

A.9 Signal Handling: <signal.h>

This header contains constructs used by the programmer to handle exceptional conditions, or signals. The following macros are defined in this header:

```
#define   SIGINT    2      /* interrupt */
#define   SIGILL    4      /* illegal instruction */
#define   SIGFPE    8      /* floating-point exception */
#define   SIGSEGV   11     /* segment violation */
#define   SIGTERM   15     /* asynchronous termination */
#define   SIGABRT   22     /* abort */
```

The constants are system-dependent, but these are commonly used. Other signals are usually supported; see the file *signal.h* on your system.

The macros in the following set may be used as the second argument of the function `signal()`:

```
#define   SIG_DFL   ((void (*)(int)) 0)      /* default */
#define   SIG_ERR   ((void (*)(int)) -1)     /* error */
#define   SIG_IGN   ((void (*)(int)) 1)      /* ignore */
```

A system may supply other such macros. The names must begin with `SIG_` followed by a capital letter.

- `void (*signal(int sig, void (*func)(int)))(int);`

The function call `signal(sig, func)` associates the signal `sig` with the signal handler `func()`. If the call is successful, the pointer value `func` of the previous call with first argument `sig` is returned, or `NULL` is returned if there was no previous call. If the call is unsuccessful, the pointer value `SIG_ERR` is returned.

The function call `signal(sig, func)` instructs the system to invoke `func(sig)` when the signal `sig` is raised. If the second argument to `signal()` is `SIG_DFL`, default action occurs; if it is `SIG_IGN`, the signal is ignored. When program control returns from `func()`, it returns to the place where `sig` was raised.

- `int raise(int sig);`

Causes the signal `sig` to be raised. If the call is successful, zero is returned; otherwise, a nonzero value is returned. This function can be for testing purposes.

A.10 Variable Arguments: <stdarg.h>

This header file provides the programmer with a portable means of writing functions such as `printf()` that have a variable number of arguments. The header file contains one `typedef` and three macros. How these are implemented is system-dependent, but here is one way it can be done:

```
typedef   char *   va_list;

#define   va_start(ap, v)  \
             ((void) (ap = (va_list) &v + sizeof(v)))
#define   va_arg(ap, type)    (*((type *)(ap))++)
#define   va_end(ap)          ((void) (ap = 0))
```

In the macro `va_start()`, the variable `v` is the last argument that is declared in the header to your variable argument function definition. This variable cannot be of storage class `register`, and it cannot be an array type or a type such as `char` that is widened by automatic conversions. The macro `va_start()` initializes the argument pointer `ap`. The macro `va_arg()` accesses the next argument in the list. The macro `va_end()` performs any cleanup that may be required before function exit. The following program illustrates the use of these constructs:

```
#include   <stdio.h>
#include   <stdarg.h>

int   va_sum(int cnt, ...);

int main(void)
{
   int      a = 1, b = 2, c = 3;

   printf("First call:  sum = %d\n", va_sum(2, a, b));
   printf("Second call: sum = %d\n", va_sum(3, a, b, c));
   return 0;
}
```

```
int va_sum(int cnt, ...)            /* sum the arguments */
{
   int       i, sum = 0;
   va_list   ap;

   va_start(ap, cnt);               /* startup */
   for (i = 0; i < cnt; ++i)
      sum += va_arg(ap, int);       /* get next argument */
   va_end(ap);                      /* cleanup */
   return sum;
}
```

A.11 Common Definitions: <stddef.h>

This header file contains some type definitions and macros that are commonly used in other places. How they are implemented is system-dependent, but here is one way of doing it:

```
typedef   char       wchar_t;
typedef   int        ptrdiff_t;
typedef   unsigned   size_t;

#define   NULL       ((void *) 0)
#define   offsetof(s_type, m)  \
                  ((size_t) &(((s_type *) 0) -> m))
```

Here, we defined the wide character type `wchar_t` as a plain `char`. A system can define it to be any integral type. It must be able to hold the largest extended character set of all the locales that are supported. The type `ptrdiff_t` is the type obtained when two pointers are subtracted. The type `size_t` is the type obtained with use of the `sizeof` operator. A macro call of the form `offsetof(`*s_type*`, m)` computes the offset in bytes of the member `m` from the beginning of the structure *s_type*. The following program illustrates its use:


```
#include   <stdio.h>
#include   <stddef.h>

typedef struct {
   double   a, b, c;
} data;

int main(void)
{
   printf("%d %d\n",
          offsetof(data, a), offsetof(data, b));
   return 0;
}
```

On most systems, this program causes 0 and 8 to be printed.

A.12 Input/Output: <stdio.h>

This header file contains macros, type definitions, and prototypes of functions used by the programmer to access files. Here are some example macros and type definitions:

```
#define   BUFSIZ          1024    /*buf size for all I/O*/
#define   EOF             (-1)    /*returned on EOF*/
#define   FILENAME_MAX    255     /*max filename chars */
#define   FOPEN_MAX       20      /*max open files*/
#define   L_tmpnam        16      /*size tmp filename*/
#define   NULL            0       /*null pointer value */
#define   TMP_MAX         65535   /*max unique filenames*/

typedef   long       pos_t;       /*used with fsetpos() */
typedef   unsigned   size_t;      /*type from sizeof op*/
typedef   char *     va_list;     /*used with vfprintf()*/
```

The structure type `FILE` has members that describe the current state of a file. The name and number of its members are system-dependent. Here is an example:

```
typedef struct {
    int             cnt;     /* size of unused part of buf */
    unsigned char *b_ptr;    /* next buffer loc to access */
    unsigned char *base;     /* start of buffer */
    int             bufsize; /* buffer size */
    short           flag;    /* info stored bitwise */
    char            fd;      /* file descriptor */
} FILE;

extern FILE   _iob[];
```

An object of type FILE should be capable of recording all the information needed to control a stream, including a file position indicator, a pointer to its associated buffer, an *error indicator* that records whether a read/write error has occurred, and an *end-of-file indicator* that records whether the end-of-file mark has been reached. How this is implemented is system-dependent. For example, the error indicator and the end-of-file indicator might be encoded bitwise in the structure member flag.

Typically, the type fpos_t is given by:

```
typedef   long   fpos_t;
```

An object of this type is supposed to be capable of recording all the information needed to uniquely specify every position in a file.

Macros are used to define stdin, stdout, and stderr. Although we think of them as files, they are actually pointers.

```
#define   stdin    (&_iob[0])
#define   stdout   (&_iob[1])
#define   stderr   (&_iob[2])
```

Unlike other files, stdin, stdout, and stderr do not have to be opened explicitly by the programmer.

A few macros are intended for use with functions:

```
#define   _IOFBF     0     /* setvbuf(): full buffering */
#define   _IOLBF     0x80  /* setvbuf(): line buffering */
#define   _IONBF     0x04  /* setvbuf(): no buffering */
#define   SEEK_SET   0     /* fseek(): file beginning */
#define   SEEK_CUR   1     /* fseek(): current file pos */
#define   SEEK_END   2     /* fseek(): EOF */
```

When a file is opened, the operating system associates it with a *stream* and keeps information about the stream in an object of type FILE. A pointer to FILE can be thought of as being associated with the file or the stream or both.

Opening, Closing, and Conditioning a File

■ `FILE *fopen(const char *filename, const char *mode);`

Performs the necessary housekeeping to open a buffered file. A successful call creates a stream and returns a pointer to FILE that is associated with the stream. If filename cannot be accessed, NULL is returned. The basic file modes are "r", "w", and "a", corresponding to read, write, and append, respectively. The file position indicator is set at the beginning of the file if the file mode is "r" or "w", and it is set at the end of the file if the file mode is "a". If the file mode is "w" or "a" and the file does not exist, it is created. An update mode (both reading and writing) is indicated with a +. A binary file is indicated with a b. For example, the mode "r+" is used to open a text file for both reading and writing. The mode "rb" is used to open a binary file for reading. The mode "rb+" or "r+b" is used to open a binary file for reading and writing. Similar conventions apply to "w" and "a." (See Section 11.9, "File Access Permissions," on page 517.) In update mode, input may not be directly followed by output unless the end-of-file mark has been reached or an intervening call to one of the file positioning functions fseek(), fsetpos(), or rewind() has occurred. In a similar fashion, output may not be directly followed by input unless an intervening call to fflush() or to one of the file positioning functions fseek(), fsetpos(), or rewind() has occurred.

■ `int fclose(FILE *fp);`

Performs the necessary housekeeping to empty buffers and break all connections to the file associated with fp. If the file is successfully closed, zero is returned. If an error occurs or the file was already closed, EOF is returned. Open files are a limited resource. At most, FOPEN_MAX files can be open simultaneously. System efficiency is improved by keeping only needed files open.

■ `int fflush(FILE *fp);`

Any buffered data is delivered. If the call is successful, zero is returned; otherwise, EOF is returned.

■ `FILE *freopen(const char *filename, const char *mode, FILE *fp);`

Closes the file associated with fp, opens filename as specified by mode, and associates fp with the new file. If the function call is successful, fp is returned; otherwise, NULL is returned. This function is useful for changing the file associated with stdin, stdout, or stderr.

■ `void setbuf(FILE *fp, char *buf);`

If `fp` is not `NULL`, the function call `setbuf(fp, buf)` is equivalent to

```
setvbuf(fp, buf, _IOFBF, BUFSIZ)
```

except that nothing is returned. If `fp` is `NULL`, the mode is `_IONBF`.

■ `int setvbuf(FILE *fp, char *buf, int mode, size_t n);`

Determines how the file associated with `fp` is to be buffered. The function must be invoked after the file has been opened but before it is accessed. The modes `_IOFBF`, `_IOLBF`, and `_IONBF` cause the file to be fully buffered, line buffered, and unbuffered, respectively. If `buf` is not `NULL`, the array of size `n` pointed to by `buf` is used as a buffer. If `buf` is `NULL`, the system provides the buffer. A successful call returns zero. *Caution:* If an array of storage class automatic is used as a buffer, the file should be closed before the function is exited.

■ `FILE *tmpfile(void);`

Opens a temporary file with mode `"wb+"` and returns a pointer associated with the file. If the request cannot be honored, `NULL` is returned. The system removes the file after it is closed, or on program exit.

■ `char *tmpnam(char *s);`

Creates a unique temporary name that is typically used as a file name. If `s` is not `NULL`, the name is stored in `s`, which must be of size `L_tmpnam` or larger. If `s` is `NULL`, the system provides an array of static duration to store the name. Further calls to `tmpnam()` can overwrite this space. In all cases, the base address of the array in which the name is stored is returned. Repeated calls to `tmpnam()` will generate at least `TMP_MAX` unique names.

Accessing the File Position Indicator

Functions in this section are used by the programmer to access a file randomly. The traditional functions for this purpose are `fseek()`, `ftell()`, and `rewind()`. ANSI C has added `fgetpos()` and `fsetpos()`. An implementation can design these functions to access files that are too large to be handled by the traditional functions. However, early versions of many ANSI C compilers have not taken advantage of this opportunity.

■ `int fseek(FILE *fp, long offset, int place);`

Sets the file position indicator for the next input or output operation. The position is `offset` bytes from `place`. The value of `place` can be `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, which correspond to the beginning of the file, the current position in the file, or the end of the file, respectively. If the function call is successful, the end-of-file indicator is cleared and zero is returned.

■ `long ftell(FILE *fp);`

Returns the current value of the file position indicator for the file associated with `fp`. On a binary file, this value is a count of the number of bytes from the beginning of the file. For text files on some systems, this value is a "magic cookie." In any case, by saving the value returned, `fseek()` can be used to reset the file position indicator. An unsuccessful call returns -1 and stores a system-dependent value in `errno`.

■ `void rewind(FILE *fp);`

Sets the file position indicator to the beginning of the file and clears the end-of-file and error indicators. The function call `rewind(fp)` is equivalent to

```
(void) fseek(fp, 0L, SEEK_SET)
```

except that `fseek()` clears only the end-of-file indicator.

■ `int fgetpos(FILE *fp, fpos_t *pos);`

Gets the current value of the file position indicator for the file associated with `fp` and stores it in the object pointed to by `pos`. The stored value can be used later by `fsetpos()` to reset the file position indicator. A successful call returns zero; otherwise, a system-dependent value is stored in `errno`, and a nonzero value is returned.

■ `int fsetpos(FILE *fp, const fpos_t *pos);`

Sets the file position indicator to the value pointed to by `pos`. A successful call clears the end-of-file indicator and returns zero; otherwise, a system-dependent value is written to `errno`, and a nonzero value is returned.

Error Handling

- `void clearerr(FILE *fp);`

Clears the error and end-of-file indicators for the file associated with `fp`.

- `int feof(FILE *fp);`

Returns a nonzero value if the end-of-file indicator has been set for the file associated with `fp`.

- `int ferror(FILE *fp);`

Returns a nonzero value if the error indicator has been set for the file associated with `fp`.

- `void perror(const char *s);`

Prints an error message associated with `errno` on `stderr`. First, the string `s` is printed, followed by a colon and a space. Then the associated error message is printed, followed by a newline. (The function call `strerror(errno)` prints only the associated error message.)

Character Input/Output

- `int getc(FILE *fp);`

Equivalent to `fgetc()`, except that it is implemented as a macro. Since `fp` may be evaluated more than once in the macro definition, a call with an argument that has side-effects, such as `fgetc(*p++)`, may not work correctly.

- `int getchar(void);`

The call `getchar()` is equivalent to `getc(stdin)`.

- `char *gets(char *s);`

Reads characters from `stdin` and stores them in the array pointed to by `s` until a newline is read or the end-of-file is reached, whichever occurs first. At this point, any newline is discarded and a null character is written. (In contrast, `fgets()` preserves the newline.) If any characters are written, `s` is returned; otherwise, `NULL` is returned.

- `int fgetc(FILE *fp);`

Gets the next character from the file associated with `fp` and returns the value of the character read. If the end-of-file is encountered, the end-of-file indicator is set, and `EOF` is returned. If an error occurs, the error indicator is set, and `EOF` is returned.

- `char *fgets(char *line, int n, FILE *fp);`

Reads at most `n - 1` characters from the file associated with `fp` into the array pointed to by `line`. As soon as a newline is read into the array or an end-of-file is encountered, no additional characters are read from the file. A null character is written into the array to end the process. If an end-of-file is encountered right at the start, the contents of `line` are undisturbed, and `NULL` is returned; otherwise, `line` is returned.

- `int fputc(int c, FILE *fp);`

Converts the argument `c` to an `unsigned char` and writes it in the file associated with `fp`. If the call `fputc(c)` is successful, it returns

```
(int) (unsigned char) c
```

Otherwise, it sets the error indicator and returns `EOF`.

- `int fputs(const char *s, FILE *fp);`

Copies the null-terminated string `s` into the file associated with `fp`, except for the terminating null character itself. (The related function `puts()` appends a newline.) A successful call returns a nonnegative value; otherwise, `EOF` is returned.

- `int putc(int c, FILE *fp);`

The `putc()` function is equivalent to `fputc()`, except that it is implemented as a macro. Since `fp` may be evaluated more than once in the macro definition, a call with an argument that has side-effects, such as `putc(*p++)`, may not work correctly.

- `int putchar(int c);`

The call `putchar(c)` is equivalent to `putc(c, stdout)`.

■ `int puts(const char *s);`

Copies the null-terminated string `s` to the standard output file, except the terminating null character itself. Then a newline is written. (The related function `fputs()` does not append a newline.) A successful call returns a nonnegative value; otherwise, `EOF` is returned.

■ `int ungetc(int c, FILE *fp);`

Pushes the value `(unsigned char) c` back onto the stream associated with `fp`, provided the value of `c` is not `EOF`. At least one character can be pushed back. (Most systems allow more.) Pushed back characters will be read from the stream in the reverse order in which they were pushed back. Once they have been read, they are forgotten; they are not placed permanently in the file. *Caution:* An intervening call to one of the file positioning functions `fseek()`, `fsetpos()`, or `rewind()` causes any pushed-back characters to be lost. Also, until the pushed-back characters have been read, `ftell()` may be unreliable.

Formatted Input/Output

■ `int fprintf(FILE *fp, const char *cntrl_string, ...);`

Writes formatted text into the file associated with `fp` and returns the number of characters written. If an error occurs, the error indicator is set, and a negative value is returned. Conversion specifications, or formats, can occur in `cntrl_string`. They begin with a `%` and end with a conversion character. The formats determine how the other arguments are printed. (See Section 11.1, "The Output Function `printf()`," on page 493.)

■ `int printf(const char *cntrl_string, ...);`

A function call of the form `printf(cntrl_string, other_arguments)` is equivalent to

`fprintf(stdout, cntrl_string, other_arguments )`

■ `int sprintf(char *s, const char *cntrl_string, ...);`

This is the string version of `printf()`. Instead of writing to `stdout`, it writes to the string pointed to by `s`.

■ `int vfprintf(FILE *fp, const char *cntrl_string, va_list ap);`
`int vprintf(const char *cntrl_string, va_list ap);`
`int vsprintf(char s*, const char *cntrl_string, va_list ap);`

These functions correspond to `fprintf()`, `printf()`, and `sprintf()`, respectively. Instead of a variable length argument list, they have a pointer to an array of arguments as defined in *stdarg.h*.

■ `int fscanf(FILE *fp, const char *cntrl_string, ...);`

Reads text from the file stream associated with `fp` and processes it according to the directives in the control string. There are three kinds of *directives*: ordinary characters, white space, and conversion specifications. Ordinary characters are matched, and white space is matched with optional white space. A conversion specification begins with a `%` and ends with a conversion character; it causes characters to be read from the input stream, a corresponding value to be computed, and the value to be placed in memory at an address specified by one of the other arguments. If the function is invoked and the input stream is empty, *EOF* is returned; otherwise, the number of successful conversions is returned. (See Section 11.2, "The Input Function `scanf()`," on page 499.)

■ `int scanf(const char *cntrl_string, ...);`

A function call of the form `scanf(cntrl_string, other_arguments )` is equivalent to

`fscanf(stdin, cntrl_string, other_arguments )`

■ `int sscanf(const char *s, const char *cntrl_string, ...);`

This is the string version of `scanf()`. Instead of reading from `stdin`, it reads from the string pointed to by `s`. Reading from a string is unlike reading from a file; if we use `sscanf()` to read from `s` again, then the input starts at the beginning of the string, not where we left off before.

Direct Input/Output

The functions `fread()` and `fwrite()` are used to read and write binary files, respectively. No conversions are performed. In certain applications, the use of these functions can save considerable time.

- `size_t fread(void *a_ptr, size_t el_size, size_t n, FILE *fp);`

Reads at most `n * el_size` bytes (characters) from the file associated with `fp` into the array pointed to by `a_ptr`. The number of array elements successfully written is returned. If an end-of-file is encountered, the end-of-file indicator is set and a short count is returned. If `el_size` or `n` is zero, the input stream is not read, and zero is returned.

- `size_t fwrite(const void *a_ptr, size_t el_size, size_t n, FILE *fp);`

Reads `n * el_size` bytes (characters) from the array pointed to by `a_ptr` and writes them to the file associated with `fp`. The number of array elements successfully written is returned. If an error occurs, a short count is returned. If `el_size` or `n` is zero, the array is not accessed, and zero is returned.

Removing or Renaming a File

- `int remove(const char *filename);`

Removes the file with the name `filename` from the file system. If the call is successful, zero is returned; otherwise, -1 is returned. (This is the `unlink()` function in traditional C.)

- `int rename(const char *from, const char *to);`

Changes the name of a file. The old name is in the string pointed to by `from`. The new name is in the string pointed to by `to`. If a file with the new name already exists, what happens is system-dependent; typically, in UNIX the file is overwritten. On most systems, the old and new names can be either files or directories. If one of the arguments is a directory name, the other one must be too. Zero is returned if the call is successful; otherwise, -1 is returned, and a system-dependent value is written to `errno`.

A.13 General Utilities: <stdlib.h>

This header file contains prototypes of functions for general use, along with related macros and type definitions. Here are some examples of the macros and type definitions:

```
#include <stddef.h>           /* for size_t and wchar_t */

#define   EXIT_SUCCESS   0    /* for use with exit() */
#define   EXIT_FAILURE   1    /* for use with exit() */
#define   NULL           0    /* null pointer value */
#define   RAND_MAX       32767   /* 2^15 - 1 */

typedef struct {
   int   quot;      /* quotient */
   int   rem;       /* remainder */
} div_t;

typedef struct {
   long quot;       /* quotient */
   long rem;        /* remainder */
} ldiv_t;
```

Dynamic Allocation of Memory

- `void *calloc(size_t n, size_t el_size);`

Allocates contiguous space in memory for an array of `n` elements, with each element requiring `el_size` bytes. The space is initialized with all bits set to zero. A successful call returns the base address of the allocated space; otherwise, `NULL` is returned.

- `void *malloc(size_t size);`

Allocates a block of space in memory consisting of `size` bytes. The space is not initialized. A successful call returns the base address of the allocated space; otherwise, `NULL` is returned.

- `void *realloc(void *ptr, size_t size);`

Changes the size of the block pointed to by `ptr` to `size` bytes. The contents of the space will be unchanged up to the lesser of the old and new sizes. Any new space is not

initialized. The function attempts to keep the base address of the block the same; if this is not possible, it allocates a new block of memory, copying the relevant portion of the old block and deallocating it. If `ptr` is `NULL`, the effect is the same as calling `malloc()`. If `ptr` is not `NULL`, it must be the base address of space previously allocated by a call to `calloc()`, `malloc()`, or `realloc()` that has not yet been deallocated by a call to `free()` or `realloc()`. A successful call returns the base address of the resized (or new) space; otherwise, `NULL` is returned.

- `void free(void *ptr);`

Causes the space in memory pointed to by `ptr` to be deallocated. If `ptr` is `NULL`, the function has no effect. If `ptr` is not `NULL`, it must be the base address of space previously allocated by a call to `calloc()`, `malloc()`, or `realloc()` that has not yet been deallocated by a call to `free()` or `realloc()`; otherwise, the call is in error. The effect of the error is system-dependent.

Searching and Sorting

-
```
void *bsearch(const void *key_ptr, const void *a_ptr,
              size_t n_els, size_t el_size,
              int compare(const void *, const void *));
```

Searches the sorted array pointed to by `a_ptr` for an element that matches the object pointed to by `key_ptr`. If a match is found, the address of the element is returned; otherwise, `NULL` is returned. The number of elements in the array is `n_els`, and each element is stored in memory in `el_size` bytes. The elements of the array must be in ascending sorted order with respect to the comparison function `compare()`. The comparison function takes two arguments, each one being an address of an element of the array. The comparison function returns an `int` that is less than, equal to, or greater than zero, depending on whether the element pointed to by its first argument is considered to be less than, equal to, or greater than the element pointed to by its second argument. (The function `bsearch()` uses a binary search algorithm.)

-
```
void qsort(void *a_ptr, size_t n_els, size_t el_size,
           int compare(const void *, const void *));
```

Sorts the array pointed to by `a_ptr` in ascending order with respect to the comparison function `compare()`. The number of elements in the array is `n_els`, and each element is stored in memory in `el_size` bytes. The comparison function takes two arguments, each one being an address of an element of the array. The comparison function returns an `int` that is less than, equal to, or greater than zero, depending on whether the element pointed to by its first argument is considered to be less than, equal to, or greater than the element pointed to by its second argument. (By tradition, the function `qsort()` implements a "quicker-sort" algorithm.)

Pseudo Random-Number Generator

- `int rand(void);`

Each call generates an integer and returns it. Repeated calls generate what appears to be a randomly distributed sequence of integers in the interval [0, `RAND_MAX`].

- `void srand(unsigned seed);`

Seeds the random-number generator, causing the sequence generated by repeated calls to `rand()` to start in a different place each time. On program startup, the random-number generator acts as if `srand(1)` had been called. The statement

```
srand(time(NULL));
```

can be used to seed the random-number generator with a different value each time the program is invoked.

Communicating with the Environment

- `char *getenv(const char *name);`

Searches a list of environment variables provided by the operating system. If `name` is one of the variables in the list, the base address of its corresponding string value is returned; otherwise, `NULL` is returned. (See Section 11.12, "Environment Variables," on page 521.)

- `int system(const char *s);`

Passes the string `s` as a command to be executed by the command interpreter (the shell) provided by the operating system. If `s` is not `NULL` and a connection to the operating system exits, the function returns the exit status returned by the command. If `s` is `NULL`, the function returns a nonzero value if the command interpreter is available via this mechanism; otherwise, it returns zero.

Integer Arithmetic

- ```
  int  abs(int i);
  long labs(long i);
  ```

Both functions return the absolute value of `i`.

- ```
  div_t  div(int numer, int denom);
  ldiv_t ldiv(long numer, long denom);
  ```

Both functions divide `numer` by `denom` and return a structure that has the quotient and remainder as members. The following is an example:

```
div_t   d;

d = div(17, 5);
printf("quotient = %d, remainder = %d\n", d.quot, d.rem);
```

When executed, this code prints the line

```
quotient = 3, remainder = 2
```

String Conversion

Members of the two families `ato...()` and `strto...()` are used to convert a string to a value. The conversion is conceptual; it interprets the characters in the string, but the string itself does not change. The string can begin with optional white space. The conversion stops with the first inappropriate character. For example, both of the function calls

```
strtod("123x456", NULL)     and     strtod("\n 123 456", NULL)
```

return the `double` value 123.0. The `strto...()` family provides more control over the conversion process and provides for error checking.

- ```
  double atof(const char *s); /* ascii to floating number */
  ```

Converts the string `s` to a `double` and returns it. Except for error behavior, the function call

```
atof(s)       is equivalent to       strtod(s, NULL)
```

If no conversion takes place, the function returns zero.


- ```
  int atoi(const char *s);     /* ascii to integer */
  ```

Converts the string `s` to an `int` and returns it. Except for error behavior, the function call

```
atoi(s)       is equivalent to       (int) strtol(s, NULL, 10)
```

If no conversion takes place, the function returns zero.

- ```
  long atol(const char *s);    /* ascii to long */
  ```

Converts the string `s` to a `long` and returns it. Except for error behavior, the function call

```
atol(s)       is equivalent to       strtol(s, NULL, 10)
```

If no conversion takes place, the function returns zero.

- ```
  double strtod(const char *s, char **end_ptr);
  ```

Converts the string `s` to a `double` and returns it. If no conversion takes place, zero is returned. If `end_ptr` is not `NULL` and conversion takes place, the address of the character that stops the conversion process is stored in the object pointed to by `end_ptr`. If `end_ptr` is not `NULL` and no conversion takes place, the value `s` is stored in the object pointed to by `end_ptr`. On overflow, either `HUGE_VAL` or `-HUGE_VAL` is returned, and `ERANGE` is stored in `errno`. On underflow, zero is returned, and `ERANGE` is stored in `errno`.

- ```
  long strtol(const char *s, char **end_ptr, int base);
  ```

Converts the string `s` to a `long` and returns it. If `base` has a value from 2 to 36, the digits and letters in `s` are interpreted in that base. In base 36, the letters `a` through `z` and `A` through `Z` are interpreted as 10 through 35, respectively. With a smaller base, only those digits and letters with corresponding values less than the base are interpreted. If `end_ptr` is not `NULL` and conversion takes place, the address of the character that stops the conversion process is stored in the object pointed to by `end_ptr`. For example,

```
char   *p;
long   value;

value = strtol("12345", &p, 3);
printf("value = %ld, end string = \"%s\"\n", value, p);
```
```

When executed, this code prints the line

```
value = 5, end string = "345"
```

Since the base is 3, the character 3 in the string "12345" stops the conversion process. Only the first two characters in the string are converted. In base 3, the characters 12 get converted to decimal value 5. In a similar fashion, the code

```
value = strtol("abcde", &p, 12);
printf("value = %ld, end string = \"%s\"\n", value, p);
```

prints the line

```
value = 131, end string = "cde"
```

Since the base is 12, the character c in the string "abcde" stops the conversion process. Only the first two characters in the string are converted. In base 12, the characters ab get converted to decimal value 131.

If base is zero, s is interpreted as either a hexadecimal, octal, or decimal integer, depending on the leading nonwhite characters in s. With an optional sign and 0x or 0X, the string is interpreted as a hexadecimal integer (base 16). With an optional sign and 0, but not 0x or 0X, the string is interpreted as an octal integer (base 8). Otherwise, it is interpreted as a decimal integer.

If no conversion takes place, zero is returned. If end_ptr is not NULL and no conversion takes place, the value s is stored in the object pointed to by end_ptr. On overflow, either LONG_MAX or -LONG_MAX is returned, and ERANGE is stored in errno.

- `unsigned long strtoul(const char *s, char **end_ptr, int base);`

The strtoul() function is similar to strtol(), but returns an unsigned long. On overflow, either ULONG_MAX or -ULONG_MAX is returned.

Multibyte Character Functions

Multibyte characters are used to represent members of an extended character set. How the members of an extended character set are defined is locale-dependent.

- `int mblen(const char *s, size_t n);`

If s is NULL, the function returns a nonzero or zero value, depending on whether multibyte characters do or do not have a state-dependent encoding. If s is not NULL, the function examines at most n characters in s and returns the number of bytes that comprise the next multibyte character. If s points to the null character, zero is returned. If s does not point to a multibyte character, the value -1 is returned.

- `int mbtowc(wchar_t *p, const char *s, size_t n);`

Acts the same as mblen(), but with the following additional capability: If p is not NULL, the function converts the next multibyte character in s to its corresponding wide character type and stores it in the object pointed to by p.

- `int wctomb(char *s, wchar_t wc);`

If s is NULL, the function returns a nonzero or zero value, depending on whether multibyte characters do or do not have a state-dependent encoding. If s is not NULL and wc is a wide character corresponding to a multibyte character, the function stores the multibyte character in s and returns the number of bytes required to represent it. If s is not NULL and wc does not correspond to a multibyte character, the value -1 is returned.

Multibyte String Functions

- `size_t mbstowcs(wchar_t *wcs, const char *mbs, size_t n);`

Reads the multibyte string pointed to by mbs and writes the corresponding wide character string into wcs. At most, n wide characters are written, followed by a wide null character. If the conversion is successful, the number of wide characters written is returned, not counting the final wide null character; otherwise, -1 is returned.

- `int wcstombs(char *mbs, const wchar_t *wcs, size_t n);`

Reads the wide character string pointed to by wcs and writes the corresponding multibyte string into mbs. The conversion process stops after n wide characters have been written or a null character is written, whichever comes first. If the conversion is successful, the number of characters written is returned, not counting the null character (if any); otherwise, -1 is returned.

Leaving the Program

- `void abort(void);`

Causes abnormal program termination, unless a signal handler catches `SIGABRT` and does not return. It depends on the implementation whether any open files are properly closed and any temporary files are removed.

- `int atexit(void (*func)(void));`

Registers the function pointed to by `func` for execution upon normal program exit. A successful call returns zero; otherwise, a nonzero value is returned. At least 32 such functions can be registered. Execution of registered functions occurs in the reverse order of registration. Only global variables are available to these functions.

- `void exit(int status);`

Causes normal program termination. The functions registered by `atexit()` are invoked in the reverse order in which they were registered, buffered streams are flushed, files are closed, and temporary files that were created by `tmpfile()` are removed. The value `status`, along with control, is returned to the host environment. If the value of `status` is zero or `EXIT_SUCCESS`, the host environment assumes that the program executed successfully; if the value is `EXIT_FAILURE`, it assumes that the program did not execute successfully. The host environment may recognize other values for `status`.

A.14 Memory and String Handling: <string.h>

This header file contains prototypes of functions in two families. The memory functions `mem...()` are used to manipulate blocks of memory of a specified size. These blocks can be thought of as arrays of bytes (characters). They are like strings, except that they are not null-terminated. The functions `str...()` are used to manipulate null-terminated strings. Typically, the following line is at the top of the header file:

```
#include <stddef.h>          /* for NULL and size_t */
```


Memory-Handling Functions

- `void *memchr(const void *p, int c, size_t n);`

Starting in memory at the address `p`, a search is made for the first unsigned character (byte) that matches the value `(unsigned char)c`. At most, `n` bytes are searched. If successful, a pointer to the character is returned; otherwise, `NULL` is returned.

- `int memcmp(const void *p, const void *q, size_t n);`

Compares two blocks in memory of size `n`. The bytes are treated as unsigned characters. The function returns a value that is less than, equal to, or greater than zero, depending on whether the block pointed to by `p` is lexicographically less than, equal to, or greater than the block pointed to by `q`.

- `void *memcpy(void *to, void *from, size_t n);`

Copies the block of `n` bytes pointed to by `from` to the block pointed to by `to`. The value `to` is returned. If the blocks overlap, the behavior is undefined.

- `void *memmove(void *to, void *from, size_t n);`

Copies the block of `n` bytes pointed to by `from` to the block pointed to by `to`. The value `to` is returned. If the blocks overlap, each byte in the block pointed to by `from` is accessed before a new value is written in that byte. Thus, a correct copy is made, even when the blocks overlap.

- `void *memset(void *p, int c, size_t n);`

Sets each byte in the block of size `n` pointed to by `p` to value `(unsigned char)c`. The value `p` is returned.

String-Handling Functions

- `char *strcat(char *s1, const char *s2);`

Concatenates the strings `s1` and `s2`. That is, a copy of `s2` is appended to the end of `s1`. The programmer must ensure that `s1` points to enough space to hold the result. The string `s1` is returned.

■ `char *strchr(const char *s, int c);`

Searches for the first character in `s` that matches the value `(char) c`. If the character is found, its address is returned; otherwise, `NULL` is returned. The call `strchr(s, '\0')` returns a pointer to the terminating null character in `s`.

■ `int strcmp(const char *s1, const char *s2);`

Compares the two strings `s1` and `s2` lexicographically. The elements of the strings are treated as unsigned characters. The function returns a value that is less than, equal to, or greater than zero, depending on whether `s1` is lexicographically less than, equal to, or greater than `s2`.

■ `int strcoll(const char *s1, const char *s2);`

Compares the two strings `s1` and `s2` using a comparison rule that depends on the current locale. The function returns a value that is less than, equal to, or greater than zero, depending on whether `s1` is considered less than, equal to, or greater than `s2`.

■ `char *strcpy(char *s1, const char *s2);`

Copies the string `s2` into the string `s1`, including the terminating null character. Whatever exists in `s1` is overwritten. The programmer must ensure that `s1` points to enough space to hold the result. The value `s1` is returned.

■ `size_t strcspn(const char *s1, const char *s2);`

Computes the length of the maximal initial substring in `s1` consisting entirely of characters *not* in `s2`. For example, the function call

```
strcspn("April is the cruelest month", "abc")
```

returns the value 13, because `"April is the "` is the maximal initial substring of the first argument having no characters in common with `"abc"`. (The character `c` in the name `strcspn` stands for "complement," and the letters `spn` stand for "span.")

■ `char *strerror(int error_number);`

Returns a pointer to an error string provided by the system. The contents of the string must not be changed by the program. If an error causes the system to write a value in `errno`, the programmer can invoke `strerror(errno)` to print the associated error message. (The related function `perror()` can also be used to print the error message.)

■ `size_t strlen(const char *s);`

Returns the length of the string `s`. The length is the number of characters in the string, not counting the terminating null character.

■ `char *strncat(char *s1, const char *s2, size_t n);`

At most, `n` characters in `s2`, not counting the null character, are appended to `s1`. Then a null character is written in `s1`. The programmer must ensure that `s1` points to enough space to hold the result. The string `s1` is returned.

■ `int strncmp(const char *s1, const char *s2, size_t n);`

Compares at most `n` characters lexicographically in each of the two strings `s1` and `s2`. The comparison stops with the *n*th character or a terminating null character, whichever comes first. The elements of the strings are treated as unsigned characters. The function returns a value that is less than, equal to, or greater than zero, depending on whether the compared portion of `s1` is lexicographically less than, equal to, or greater than the compared portion of `s2`.

■ `char *strncpy(char *s1, const char *s2, size_t n);`

Precisely `n` characters are written into `s1`, overwriting whatever is there. The characters are taken from `s2` until `n` of them have been copied or a null character has been copied, whichever comes first. Any remaining characters in `s1` are assigned the value `'\0'`. If the length of `s2` is `n` or larger, `s1` will not be null-terminated. The programmer must ensure that `s1` points to enough space to hold the result. The value `s1` is returned.

■ `char *strpbrk(const char *s1, const char *s2);`

Searches for the first character in `s1` that matches any one of the characters in `s2`. If the search is successful, the address of the character found in `s1` is returned; otherwise, `NULL` is returned. For example, the function call

```
strpbrk("April is the cruelest month", "abc")
```

returns the address of `c` in `cruelest`. (The letters `pbrk` in the name `strpbrk` stand for "pointer to break.")

■ `char *strrchr(const char *s, int c);`

Searches from the right for the first character in `s` that matches the value `(char) c`. If the character is found, its address is returned; otherwise, `NULL` is returned. The call `strchr(s, '\0')` returns a pointer to the terminating null character in `s`.

- `size_t strspn(const char *s1, const char *s2);`

Computes the length of the maximal initial substring in `s1` consisting entirely of characters in `s2`. For example, the function call

```
strspn("April is the cruelest month", "A is for apple")
```

returns the value 9, because all the characters in the first argument preceding the `t` in `the` occur in the second argument, but the letter `t` does not. (The letters `spn` in the name `strspn` stand for "span.")

- `char *strstr(const char *s1, const char *s2);`

Searches in `s1` for the first occurrence of the substring `s2`. If the search is successful, a pointer to the base address of the substring in `s1` is returned; otherwise, `NULL` is returned.

- `char *strtok(char *s1, const char *s2);`

Searches for tokens in `s1`, using the characters in `s2` as token separators. If `s1` contains one or more tokens, the first token in `s1` is found, the character immediately following the token is overwritten with a null character, the remainder of `s1` is stored elsewhere by the system, and the address of the first character in the token is returned. Subsequent calls with `s1` equal to `NULL` return the base address of a string supplied by the system that contains the next token. If no additional tokens are available, `NULL` is returned. The initial call `strtok(s1, s2)` returns `NULL` if `s1` contains no tokens. The following is an example:

```
char   s1[] = " this is,an   example ; ";
char   s2[] = ",; ";
char   *p;

printf("\"%s\"", strtok(s1, s2));
while ((p = strtok(NULL, s2)) != NULL)
   printf(" \"%s\"", p);
putchar('\n');
```

When executed, this code prints the line

```
"this" "is" "an" "example"
```


- `size_t strxfrm(char *s1, const char *s2, size_t n);`

Transforms the string `s2` and places the result in `s1`, overwriting whatever is there. At most, `n` characters, including a terminating null character, are written in `s1`. The length of `s1` is returned. The transformation is such that when two transformed strings are used as arguments to `strcmp()`, the value returned is less than, equal to, or greater than zero, depending on whether `strcoll()` applied to the untransformed strings returns a value less than, equal to, or greater than zero. (The letters `xfrm` in the name `strxfrm` stand for "transform.")

A.15 Date and Time: <time.h>

This header file contains prototypes of functions that deal with date, time, and the internal clock. Here are examples of some macros and type definitions:

```
#include <stddef.h>              /* for NULL and size_t */

#define   CLOCKS_PER_SEC   60    /* machine-dependent */

typedef   long   clock_t;
typedef   long   time_t;
```

Objects of type `struct tm` are used to store the date and time.

```
struct tm {
   int   tm_sec;   /* seconds after the minute: [0, 60] */
   int   tm_min;   /* minutes after the hour: [0, 59]   */
   int   tm_hour;  /* hours since midnight: [0, 23]     */
   int   tm_mday;  /* day of the month: [1, 31]         */
   int   tm_mon;   /* months since January: [0, 11]     */
   int   tm_year;  /* years since 1900                  */
   int   tm_wday;  /* days since Sunday: [0, 6]         */
   int   tm_yday;  /* days since 1 January: [0, 365]    */
   int   tm_isdst; /* Daylight Savings Time flag        */
};
```

Note that the range of values for `tm_sec` has to accommodate a "leap second," which occurs only sporadically. The flag `tm_isdst` is positive if Daylight Savings Time is in effect, zero if it is not, and negative if the information is not available.

Accessing the Clock

On most systems, the `clock()` function provides access to the underlying machine clock. The rate at which the clock runs is machine-dependent.

- `clock_t clock(void);`

Returns an approximation to the number of CPU "clock ticks" used by the program up to the point of invocation. To convert it to seconds, the value returned can be divided by `CLOCKS_PER_SEC`. If the CPU clock is not available, the value -1 is returned. (See Section 11.16, "How to Time C Code," on page 528.)

Accessing the Time

In ANSI C, time comes in two principal versions: a "calendar time" expressed as an integer, which on most systems represents the number of seconds that have elapsed since 1 January 1970, and a "broken-down time" expressed as a structure of type `struct tm`. The calendar time is encoded with respect to Universal Time Coordinated (UTC). The programmer can use library functions to convert one version of time to the other. Also, functions are available to print the time as a string.

- `time_t time(time_t *tp);`

Returns the current calendar time, expressed as the number of seconds that have elapsed since 1 January 1970 (UTC). Other units and other starting dates are possible, but these are the ones typically used. If `tp` is not `NULL`, the value also is stored in the object pointed to by `tp`. Consider the following code:

```
time_t   now;

now = time(NULL);
printf("\n%s%ld\n%s%s%s%s\n",
  "                        now = ", now,
  "              ctime(&now) = ", ctime(&now),
  "asctime(localtime(&now)) = ",asctime(localtime(&now)));
```

When executed on our system, this code printed these lines:

```
                        now = 685136007
              ctime(&now) = Tue Sep 17 12:33:27 1991
asctime(localtime(&now)) = Tue Sep 17 12:33:27 1991
```


- `char *asctime(const struct tm *tp);`

Converts the broken-down time pointed to by `tp` to a string provided by the system. The function returns the base address of the string. Later calls to `asctime()` and `ctime()` overwrite the string.

- `char *ctime(const time_t *t_ptr);`

Converts the calendar time pointed to by `t_ptr` to a string provided by the system. The function returns the base address of the string. Later calls to `asctime()` and `ctime()` overwrite the string. The two function calls

`ctime(&now)` and `asctime(localtime(&now))`

are equivalent.

- `double difftime(time_t t0, time_t t1);`

Computes the difference `t1 - t0` and, if necessary, converts this value to the number of seconds that have elapsed between the calendar times `t0` and `t1`. The value is returned as a `double`.

- `struct tm *gmtime(const time_t *t_ptr);`

Converts the calendar time pointed to by `t_ptr` to a broken-down time and stores it in an object of type `struct tm` that is provided by the system. The address of the structure is returned. The function computes the broken-down time with respect to Universal Time Coordinated (UTC). This used to be called Greenwich Mean Time (GMT); hence, the name of the function. Later calls to `gmtime()` and `localtime()` overwrite the structure.

- `struct tm *localtime(const time_t *t_ptr);`

Converts the calendar time pointed to by `t_ptr` to a broken-down local time and stores it in an object of type `struct tm` that is provided by the system. The address of the structure is returned. Later calls to `gmtime()` and `localtime()` overwrite the structure.

- `time_t mktime(struct tm *tp);`

Converts the broken-down local time in the structure pointed to by `tp` to the corresponding calendar time. If the call is successful, the calendar time is returned; otherwise, -1 is returned. For the purpose of the computation, the `tm_wday` and `tm_yday` members of the structure are disregarded. Before the computation, other members can

have values outside their usual range. After the computation, the members of the structure may be overwritten with an equivalent set of values in which each member lies within its normal range. The values for `tm_wday` and `tm_yday` are computed from those for the other members. For example, the following code can be used to find the date 1,000 days from now:

```
struct tm   *tp;
time_t      now, later;

now = time(NULL);
tp = localtime(&now);
tp -> tm_mday += 1000;
later = mktime(tp);
printf("\n1000 days from now: %s\n", ctime(&later));
```

■ `size_t strftime(char *s, size_t n,`
`const char *cntrl_str, const struct tm *tp);`

Writes characters into the string pointed to by `s` under the direction of the control string pointed to by `cntrl_str`. At most, `n` characters are written, including the null character. If more than `n` characters are required, the function returns zero and the contents of `s` are indeterminate; otherwise, the length of `s` is returned. The control string consists of ordinary characters and conversion specifications, or formats, that determine how values from the broken-down time in the structure pointed to by `tp` are to be written. Each conversion specification consists of a % followed by a conversion character.

Using strftime()

Conversion specification	What is printed	Example
%a	abbreviated weekday name	Fri
%A	full weekday name	Friday
%b	abbreviated month name	Sep
%B	full month name	September
%c	date and time	Sep 01 02:17:23 1993
%d	day of the month	01
%H	hour of the 24-hour day	02
%h	hour of the 12-hour day	02
%j	day of the year	243
%m	month of the year	9
%M	minutes after the hour	17
%p	AM or PM	AM
%s	seconds after the hour	23
%U	week of the year (Sun-Sat)	34
%w	day of the week (0-6)	5
%x	date	Sep 01 1993
%X	time	02:17:23
%y	year of the century	93
%Y	year	1993
%Z	time zone	PDT
%%	percent character	%

Consider the following code:

```
char     s[100];
time_t   now;

now = time(NULL);
strftime(s, 100, "%H:%M:%S on %A, %d %B %Y",
        localtime(&now));
printf("%s\n\n", s);
```

When we executed a program containing these lines, the following line was printed:

```
13:01:15 on Tuesday, 17 September 1991
```

A.16 Miscellaneous

In addition to the functions specified by ANSI C, the system may provide other functions in the library. In this section, we describe the non-ANSI C functions that are widely available. Some functions, such as `execl()`, are common to most systems. Other functions, such as `fork()` or `spawnl()`, are generally available in one operating system but not another. The name of the associated header file is system-dependent.

File Access

- `int access(const char *path, int amode);`

Checks the file with the name path for accessibility according to the bit pattern contained in the access mode amode. The function prototype is in *unistd.h* on UNIX systems and in *io.h* on MS-DOS systems. The following symbolic constants are defined in the header file:

`F_OK`	Check for existence.
`R_OK`	Test for read permission.
`W_OK`	Test for write permission.
`X_OK`	Test for execute or search permission.

Typically, the desired access mode is constructed by an OR of these symbolic constants. For example, the function call

```
access(path, R_OK | W-OK)
```

could be used to check whether the file permits both read and write access to the file. The function returns `0` if the requested access is permitted; otherwise, `-1` is returned and `errno` is set to indicate the error.

Using File Descriptors

- `int open(const char *filename, int flag, ...);`

Opens the named file for reading and/or writing as specified by the information stored bitwise in `flag`. If a file is being created, a third argument of type `unsigned` is needed; it sets the file permissions for the new file. If the call is successful, a nonnegative integer called the *file descriptor* is returned; otherwise, `errno` is set and `-1` is returned. Values that can be used for `flag` are given in the header file that contains the prototype for `open()`. These values are system-dependent.

- `int close(int fd);`

Closes the file associated with the file descriptor `fd`. If the call is successful, zero is returned; otherwise, `errno` is set and `-1` is returned.

- `int read(int fd, char *buf, int n);`

Reads at most `n` bytes from the file associated with the file descriptor `fd` into the object pointed to by `buf`. If the call is successful, the number of bytes written in `buf` is returned; otherwise, `errno` is set and `-1` is returned. A short count is returned if the end-of-file is encountered.

- `int write(int fd, const char *buf, int n);`

Writes at most `n` bytes from the object pointed to by `buf` into the file associated with the file descriptor `fd`. If the call is successful, the number of bytes written in the file is returned; otherwise, `errno` is set and `-1` is returned. A short count can indicate that the disk is full.

Creating a Concurrent Process

- `int fork(void);`

Copies the current process and begins executing it concurrently. The child process has its own process identification number. When `fork()` is called, it returns zero to the child and the child's process ID to the parent. If the call fails, `errno` is set, and `-1` is returned. This function is not available in MS-DOS.

- `int vfork(void);`

Spawns a new process in a virtual memory efficient way. The child process has its own process identification number. The address space of the parent process is not fully copied, which is very inefficient in a paged environment. The child borrows the parent's memory and thread of control until a call to `exec...()` occurs or the child exits. The parent process is suspended while the child is using its resources. When `vfork()` is called, it returns zero to the child and the child's process ID to the parent. If the call fails, `errno` is set, and -1 is returned. This function is not available in MS-DOS.

Overlaying a Process

In this section, we describe the two families `exec...()` and `spawn...()`. The first is generally available on both MS-DOS and UNIX systems, the second only on MS-DOS systems. On UNIX systems, `fork()` can be used with `exec...()` to achieve the effect of `spawn...()`.

- ```
  int execl(char *name, char *arg0, ..., char *argN);
  int execle(char *name, char *arg0, ..., char *argN,
             char **envp);
  int execlp(char *name, char *arg0, ..., char *argN);
  int execlpe(char *name, char *arg0, ..., char *argN,
              char **envp);
  int execv(char *name, char **argv);
  int execve(char *name, char **argv, char **envp);
  int execvp(char *name, char **argv);
  int execvpe(char *name, char **argv, char **envp);
  ```

These functions overlay the current process with the named program. There is no return to the parent process. By default, the child process inherits the environment of the parent. Members of the family with names that begin with `execl` require a list of arguments that are taken as the command line arguments for the child process. The last argument in the list must be the `NULL` pointer. Members of the family with names that begin with `execv` use the array `argv` to supply command line arguments to the child process. The last element of `argv` must have the value `NULL`. Members of the family with names ending in `e` use the array `envp` to supply environment variables to the child process. The last element of `envp` must have the value `NULL`. Members of the family with `p` in their name use the path variable specified in the environment to determine which directories to search for the program.

- ```
  int spawnl(int mode, char *name, char *arg0, ...,
             char *argN);
  .....
  ```


This family of functions corresponds to the `exec...()` family, except that each member has an initial integer argument. The values for `mode` are 0, 1, and 2. The value 0 causes the parent process to wait for the child process to finish before continuing. With value 1, the parent and child processes should execute concurrently, except that this has not been implemented yet. The use of this value will cause an error. The value 2 causes the child process to overlay the parent process.

Interprocess Communication

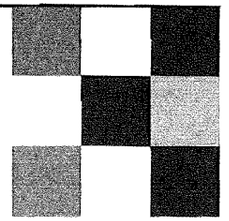
- `int pipe(int pd[2]);`

Creates an input/output mechanism called a *pipe*, and puts the associated file descriptors (pipe descriptors) in the array `pd`. If the call is successful, zero is returned; otherwise, `errno` is set and -1 is returned. After a pipe has been created, the system assumes that two or more cooperating processes created by subsequent calls to `fork()` will use `read()` and `write()` to pass data through the pipe. One descriptor, `pd[0]`, is read from; the other, `pd[1]`, is written to. The pipe capacity is system-dependent, but is at least 4,096 bytes. If a write fills the pipe, it blocks until data is read out of it. As with other file descriptors, `close()` can be used to explicitly close `pd[0]` and `pd[1]`. This function is not available in MS-DOS.

Suspending Program Execution

- `void sleep(unsigned seconds);`

Suspends the current process from execution for the number of seconds requested. The time is only approximate.



Appendix B

Language Syntax

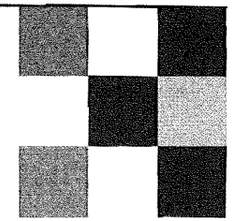
In this appendix, we give an extended BNF syntax for the ANSI version of the C language. (See Section 2.2, "Syntax Rules," on page 73.) This syntax, although intended for the human reader, is concisely written. The C language is inherently context-sensitive; restrictions and special cases are left to the main text. The conceptual output of the preprocessor is called a *translation unit*. The syntax of the C language pertains to translation units. The syntax for preprocessing directives is independent of the rest of the C language. We present it at the end of this appendix.

B.1 Program

program ::= { *file* }$_{1+}$

file ::= *decls_and_fct_definitions*

decls_and_fct_definitions ::= { *declaration* }$_{1+}$ *decls_and_fct_definitions*$_{\text{opt}}$
| { *function_definition* }$_{1+}$*decls_and_fct_definitions*$_{\text{opt}}$



Appendix B

Language Syntax

In this appendix, we give an extended BNF syntax for the ANSI version of the C language. (See Section 2.2, "Syntax Rules," on page 73.) This syntax, although intended for the human reader, is concisely written. The C language is inherently context-sensitive; restrictions and special cases are left to the main text. The conceptual output of the preprocessor is called a *translation unit*. The syntax of the C language pertains to translation units. The syntax for preprocessing directives is independent of the rest of the C language. We present it at the end of this appendix.

B.1 Program

program ::= { *file* }$_{1+}$

file ::= *decls_and_fct_definitions*

decls_and_fct_definitions ::= { *declaration* }$_{1+}$ *decls_and_fct_definitions*$_{\text{opt}}$
| { *function_definition* }$_{1+}$*decls_and_fct_definitions*$_{\text{opt}}$

B.2 Function Definition

function_definition ::= { `extern` | `static` }$_{opt}$ *type_specifier*
function_name (*parameter_declaration_list*$_{opt}$)
compound_statement

function_name ::= *identifier*

parameter_declaration_list ::= *parameter_declaration* { , *parameter_declaration* }$_{0+}$

B.3 Declaration

declaration ::= *declaration_specifiers* *init_declarator_list*$_{opt}$

declaration_specifiers ::= *storage_class_specifier_or_typedef* *declaration_specifiers*$_{opt}$
| *type_specifier* *declaration_specifiers*$_{opt}$
| *type_qualifier* *declaration_specifiers*$_{opt}$

storage_class_specifier_or_typedef ::= `auto` | `extern` | `register` | `static`
| `typedef`

type_specifier ::= `char` | `double` | `float` | `int` | `long` | `short` | `signed`
| `unsigned`
| `void` | *enum_specifier* | *struct_or_union_specifier*
| *typedef_name*

enum_specifier ::= `enum` *tag*$_{opt}$ { *enumerator_list* } | `enum` *tag*

tag ::= *identifier*

enumerator_list ::= *enumerator* { , *enumerator* }$_{opt}$

enumerator ::= *enumeration_constant* { = *const_integral_expr* }$_{opt}$

enumeration_constant ::= *identifier*

struct_or_union_specifier ::= *struct_or_union* *tag*$_{opt}$ { *struct_declaration_list* }
| *struct_or_union* *tag*

struct_or_union ::= `struct` | `union`

struct_declaration_list ::= { *struct_declaration* }$_{1+}$

struct_declaration ::= *type_specifier_qualifier_list* *struct_declarator_list* ;

type_specifier_qualifier_list ::= *type_specifier* *type_specifier_qualifier_list*$_{opt}$
| *type_qualifier* *type_specifier_qualifier_list*$_{opt}$

struct_declarator_list ::= *struct_declarator* { , *struct_declarator* }$_{0+}$

struct_declarator ::= *declarator* | *declarator*$_{opt}$: *const_integral_expr*

type_qualifier ::= `const` | `volatile`

declarator ::= *pointer*$_{opt}$ *direct_declarator*

pointer ::= { * | *type_qualifier_list*$_{opt}$ }$_{1+}$

type_qualifier_list ::= { *type_qualifier* }$_{1+}$

direct_declarator ::= *identifier* | (*declarator*)
| *direct_declarator* [*const_integral_expr*$_{opt}$]
| *direct_declarator* (*parameter_type_list*)
| *direct_declarator* (identifier_list$_{opt}$)

parameter_type_list ::= *parameter_list* | *parameter_list* , ...

parameter_list ::= *parameter_declaration* { , *parameter_declaration* }$_{0+}$

parameter_declaration ::= *declaration_specifiers* *declarator*
| *declaration_specifiers* *abstract_declarator*$_{opt}$

abstract_declarator ::= *pointer* | pointer$_{opt}$ *direct_abstract_declarator*

direct_abstract_declarator ::= (*abstract_declarator*)
| *direct_abstract_declarator*$_{opt}$ [*const_integral_expr*$_{opt}$]
| *direct_abstract_declarator*$_{opt}$ (*parameter_type_list*$_{opt}$)

identifier_list ::= *identifier* { , *identifier* }$_{0+}$

typedef_name ::= *identifier*

init_declarator_list ::= *init_declarator* { , *init_declarator* }$_{opt}$

init_declarator ::= *declarator* | *declarator* = *initializer*

initializer ::= *assignment_expression* | { *initializer_list* } | { *initializer_list* , }

initializer_list ::= *initializer* { , *initializer* }$_{0+}$

B.4 Statement

statement ::= *compound_statement* | *expression_statement* | *iteration_statement*
| *jump_statement* | *labeled_statement* | *selection_statement*

compound_statement ::= { *declaration_list*$_{opt}$ *statement_list*$_{opt}$ }

declaration_list ::= { *declaration* }$_{1+}$

statement_list ::= { *statement* }$_{1+}$

expression_statement ::= *expression*$_{opt}$;

jump_statement ::= `break ;` | `continue ;` | `goto` *identifier* ;
| `return` *expression*$_{opt}$;

labeled_statement ::= *identifier* : *statement*
| `case` *const_integral_expr* : *statement*
| `default` : *statement*

selection_statement ::= `if (` *expression* `)` *statement*
| `if (` *expression* `)` *statement* `else` *statement*
| *switch_statement*

switch_statement ::= `switch (` *integral_expression* `)`
{ *case_statement* | `default` : *statement* | *switch_block* }$_1$

case_statement ::= { `case` *const_integral_expr* : }$_{1+}$ *statement*

switch_block ::= { { *declaration_list* }$_{opt}$ *case_default_group* }

case_default_group ::= { *case_group* }$_{1+}$
| { *case_group* }$_{0+}$ *default_group* { *case_group* }$_{0+}$

case_group ::= { `case` *const_integral_expr* : }$_{1+}$ { *statement* }$_{1+}$

default_group ::= `default` : {*statement*}$_{1+}$

B.5 Expression

expression ::= *constant* | *string_literal* | (*expression*) | *lvalue*
| *assignment_expression* | *expression* , *expression* | + *expression*
| - *expression* | *function_expression* | *relational_expression*
| *equality_expression* | *logical_expression*
| *expression arithmetic_op expression* | *bitwise_expression*
| *expression* ? *expression* : *expression* | `sizeof` *expression*
| `sizeof (` *type_name*) | (*type_name*) *expression*

lvalue ::= & *lvalue* | ++ *lvalue* | *lvalue* ++ | -- *lvalue* | *lvalue* --
| *identifier* | * *expression* | *lvalue* [*expression*] | (*lvalue*)
| *lvalue* . *identifier* | *lvalue* -> *identifier*

assignment_expression ::= *lvalue assignment_op expression*

assignment_op ::= = | += | -= | *= | /= | %= | &= | ^= | |= | >>= | <<=

arithmetic_op ::= + | - | * | / | %

relational_expression ::= *expression* < *expression* | *expression* > *expression*
| *expression* <= *expression* | *expression* >= *expression*

equality_expression ::= *expression* == *expression* | *expression* != *expression*

logical_expression ::= ! *expression* | *expression* || *expression*
| *expression* && *expression*

bitwise_expression ::= ~ *expression* | ^ *expression*
| *expression* & *expression* | *expression* | *expression*
| *expression* << *expression* | *expression* >> *expression*

function_expression ::= *function_name*(*argument_list*$_{opt}$)
| (* *pointer*) (*argument_list*$_{opt}$)

argument_list ::= *expression* { , *expression* }$_{0+}$

type_name ::= *type_specifier* *declarator*$_{opt}$

B.6 Constant

constant ::= *character_constant* | *enumeration_constant* | *floating_constant*
| *integer_constant*

character_constant ::= ' *c* ' | L' *c* '

c ::= any character from the source character set except ' or \ or *newline*
| *escape_sequence*

escape_sequence ::= \' | \" | \? | \\ | \a | \b | \f | \n | \r | \t | \v
| \ *octal_digit* *octal_digit*$_{opt}$ *octal_digit*$_{opt}$
| \x *hexadecimal_digit* { *hexadecimal_digit* }$_{0+}$

enumeration_constant ::= *identifier*

string_literal ::= "{*character_constant*}$_{0+}$" | L *string_literal*

floating_constant ::= *fractional_constant* *exponential_part*$_{opt}$ *floating_suffix*$_{opt}$
digit_sequence *exponential_part* *floating_suffix*$_{opt}$

fractional_constant ::= *digit_sequence*$_{opt}$. *digit_sequence* | *digit_sequence* .

digit_sequence ::= { *digit* }$_{1+}$

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

exponential_part ::= { e | E }$_1$ { + | - }$_{opt}$ *digit_sequence*

floating_suffix ::= f | F | l | L

integer_constant ::= *decimal_constant* *integer_suffix*$_{opt}$
| *octal_constant* *integer_suffix*$_{opt}$
| *hexadecimal_constant* *integer_suffix*$_{opt}$

decimal_constant ::= 0 | *nonzero_digit* *digit_sequence*

nonzero_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

octal_constant ::= 0 { *octal_digit* }$_{0+}$

octal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

hexadecimal_constant ::= { 0x | 0X }$_1$ { *hexadecimal_digit* }$_{1+}$

hexadecimal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
a | b | c | d | e | f | A | B | C | D | E | F

integer_suffix ::= *unsigned_suffix* *long_suffix*$_{opt}$ | *long_suffix* *unsigned_suffix*$_{opt}$

unsigned_suffix ::= u | U

long_suffix ::= l | L

B.7 String Literal

string_literal ::= " *s_char_sequence* " | L" *s_char_sequence* "

s_char_sequence ::= { *sc* }$_{1+}$

sc ::= any character from the source character set except " or \ or *newline*
| *escape_sequence*

B.8 Preprocessor

preprocessing_directive ::= *control_line newline* | *if_section* | *pp_token newline*

control_line ::= `#` `include` { < *identifier* > | " *identifier* " }
| `#` `undef` *identifier* | `#` `line` *pp_token* | `#` `error` *pp_token*
| `#` `pragma` *pp_token*
| `#` `define` *identifier* { (*identifier_list*) }$_{opt}$ { *pp_token* }$_{0+}$

pp_token ::= *identifier* | *constant* | *string_literal* | *operator* | *punctuator*
| *pp_token* ## *pp_token* | # *identifier*

if_section ::= *if_group* { *elif_group* }$_{0+}$ { *else_group* }$_{opt}$ *end_if_line*

if_group ::= `#` `if` *const_integral_expr newline preprocessing_directive*$_{opt}$
| `#` `ifdef` *identifier newline* { *preprocessing_directive* }$_{opt}$
| `#` `ifndef` *identifier newline* { *preprocessing_directive* }$_{opt}$

elif_group ::= `#` `elif` *constant_expression newline* { *preprocessing_directive* }$_{opt}$

else_group ::= `#` `else` *newline* { *preprocessing_directive* }$_{opt}$

end_if_line ::= `#` `endif` *newline*

newline ::= the newline character

Appendix C

ANSI C Compared to Traditional C

In this appendix, we list the major differences between ANSI C and traditional C. Where appropriate, we have included examples. The list is not complete; only the major changes are noted.

C.1 Types

- The keyword `signed` has been added to the language.
- Three types of characters are specified: plain `char`, `signed char`, and `unsigned char`. An implementation may represent a plain `char` as either a `signed char` or an `unsigned char`.
- The keyword `signed` can be used in declarations of any of the signed integral types and in casts. Except with `char`, its use is always optional.
- In traditional C, the type `long float` is equivalent to `double`. Since `long float` was rarely used, it has been removed from ANSI C.
- The type `long double` has been added to ANSI C. Constants of this type are specified with the suffix `L`. A `long double` may provide more precision and range than a `double`, but it is not required to do so.
- The keyword `void` is used to indicate that a function takes no arguments or t returns no value.

- The type `void *` is used for generic pointers. For example, the function prototype for `malloc()` is given by

```
void *malloc(size_t size);
```

  A generic pointer can be assigned a pointer value of any type, and a variable of any pointer type can be assigned a generic pointer value. Casts are not needed. In contrast, the generic pointer type in traditional C is `char *`. Here, casts are necessary.

- Enumeration types are supported. An example is

```
enum day {sun, mon, tue, wed, thu, fri, sat};
```

  The enumerators in this example are `sun`, `mon`, . . . , `sat`. Enumerators are constants of type `int`. Thus, they can be used in `case` labels in `switch` statements.

C.2 Constants

- String constants separated by white space are concatenated. Thus,

```
"abc"
"def" "ghi"      is equivalent to      "abcdefghi"
```

- String constants are not modifiable. (Not all compilers enforce this.)

- The type of a numeric constant can be specified by letter suffixes. Some examples are

```
123L        /* long           */
123U        /* unsigned       */
123UL       /* unsigned long  */
1.23F       /* float          */
1.23L       /* long double    */
```

  Suffixes may be lower- or uppercase. A numeric constant without a suffix is a type big enough to contain the value.

- The digits 8 and 9 are no longer considered octal digits. They may not be used in an octal constant.

- Hexadecimal escape sequences beginning with `\x` have been introduced. As with octal escape sequences beginning with `\0`, they are used in character and string constants.

C.3 Declarations

- The type qualifier `const` has been added. It means that variables so declared are not modifiable. (Compilers do not always enforce this.)

- The type qualifier `volatile` has been added. It means that variables so declared are modifiable by an agent external to the program. For example, some systems put the declaration

```
extern volatile int   errno;
```

  in the header file *errno.h*.

C.4 Initializations

- In ANSI C, automatic aggregates such as arrays and structures can be initialized. In traditional C, they must be external or of storage class `static`.

- Unions can be initialized. An initialization refers to the union's first member.

- Character arrays of size *n* can be initialized using a string constant of exactly *n* characters. An example is

```
char   today[3] = "Fri";
```

  The end-of-string sentinel `\0` in `"Fri"` is not copied into `today`.

C.5 Expressions

- For reasons of symmetry, a unary plus operator has been added to the language.
- In traditional C, expressions involving one of the commutative binary operators such as + or * can be reordered at the convenience of the compiler, even though they have been parenthesized in the program. For example, in the statement

  ```
  x = (a + b) + c;
  ```

  the variables can be summed by the compiler in some unspecified order. In ANSI C, this is not true. The parentheses must be honored.
- A pointer to a function can be dereferenced either explicitly or implicitly. If, for example, `f` is a pointer to a function that takes three arguments, then the expression

  `f(a, b, c)` is equivalent to `(*f)(a, b, c)`
- The `sizeof` operator yields a value of type `size_t`. The type definition for `size_t` is given in *stddef.h*.
- A pointer of type `void *` cannot be dereferenced without first casting it to an appropriate type. However, it can be used in logical expressions, where it is compared to another pointer.

C.6 Functions

- ANSI C provides a new function definition syntax. A parameter declaration list occurs in the parentheses following the function name. An example is

  ```
  int f(int a, float b)
  {
     .....
  ```


  In contrast, the traditional C style is

  ```
  int f(a, b)
  int     a;
  float   b;
  {
     .....
  ```

- ANSI C provides the function prototype, which is a new style of function declaration. A parameter type list occurs in the parentheses following the function name. Identifiers are optional. For example,

  `int f(int, float);` and `int f(int a, float b);`

  are equivalent function prototypes. In contrast, the traditional C style is

  ```
  int f();
  ```

  If a function takes no arguments, then `void` is used as the parameter type in the function prototype. If a function takes a variable number of arguments, then the ellipsis is used as the rightmost parameter in the function prototype.
- Redeclaring a parameter identifier in the outer block of a function definition is illegal. The following code illustrates the error:

  ```
  void f(int a, int b, int c)
  {
     int   a;     /* error: a cannot be redefined here */
     .....
  ```

  Although this is legal in traditional C, it is almost always a programming error. Indeed, it can be a difficult bug to find.
- Structures and unions can be passed as arguments to functions, and they can be returned from functions. The passing mechanism is call-by-value, which means that a local copy is made.

C.7 Conversions

- An expression of type `float` is not automatically converted to a `double`.
- When arguments to functions are evaluated, the resulting value is converted to the type specified by the function prototype, provided the conversion is compatible. Otherwise, a syntax error occurs.
- Arithmetic conversions are more carefully specified. (See Section 3.11, "Conversions and Casts," on page 131.) In ANSI C, the basic philosophy for conversions is to preserve values, if possible. Because of this, the rules require some conversions on a machine with 2-byte words to be different from those on a machine with 4-byte words.
- The resulting type of a shift operation is not dependent on the right operand. In ANSI C, the integral promotions are performed on each operand, and the type of the result is that of the promoted left operand.

C.8 Array Pointers

- Many traditional C compilers do not allow the operand of the address operator & to be an array. In ANSI C, since this is legal, pointers to multidimensional arrays can be used. Here is an example:

```
int   a[2][3] = {2, 3, 5, 7, 11, 13};
int   (*p)[][3];              /* the first dimension
                                 need not be specified */

p = &a;
printf("%d\n", (*p)[1][2]);   /* 13 is printed */
```


C.9 Structures and Unions

- Structures and unions can be used in assignments. If `s1` and `s2` are two structure variables of the same type, the expression `s1 = s2` is valid. Values of members in `s2` are copied into corresponding members of `s1`.
- Structures and unions can be passed as arguments to functions, and they can be returned from functions. All arguments to functions, including structures and unions, are passed call-by-value.
- If `m` is a member of a structure or union and the function call `f()` returns a structure or union of the same type, then the expression `f().m` is valid.
- Structures and unions can be used with the comma operator and in conditional expressions. Some examples are

```
int        a, b;
struct s   s1, s2, s3;

.....
(a, s1)          /* comma expression having structure type */
a < b ? s1 : s2  /* conditional expression struct type */
```

- If *expr* is a structure or union expression and `m` is a member, then an expression of the form *expr*`.m` is valid. However, *expr*`.m` can be assigned a value only if *expr* can. Even though expressions such as

```
(s1 = s2).m    (a, s1).m    (a < b ? s1 : s2).m    f().m
```

are valid, they cannot occur on the left side of an assignment operator.

C.10 Preprocessor

- Preprocessing directives do not have to begin in column 1.
- The following predefined macros have been added:

  ```
  __DATE__    __FILE__    __LINE__    __STDC__    __TIME__
  ```

  They may not be redefined or undefined. (See Section 8.9, "The Predefined Macros," on page 387.)
- A macro may not be redefined without first undefining it. Multiple definitions are allowed, provided they are the same.
- The preprocessor operators # and ## have been added. The unary operator # causes the "stringization" of a formal parameter in a macro definition. The binary operator ## merges tokens. (See Section 8.10, "The Operators # and ##," on page 387.)
- The preprocessor operator `defined` has been added. (See Section 8.8, "Conditional Compilation," on page 384.)
- The preprocessing directives `#elif`, `#error`, and `#pragma` have been added. (See Section 8.8, "Conditional Compilation," on page 384, and Section 8.12, "The Use of `#error` and `#pragma`," on page 389.)
- In traditional C, `toupper()` and `tolower()` are defined as macros in *ctype.h*:

  ```
  #define   toupper(c)   ((c)-'a'+'A')
  #define   tolower(c)   ((c)-'A'+'a')
  ```

  The macro call `toupper(c)` will work properly only when `c` has the value of a lowercase letter. Similarly, the macro call `tolower(c)` will work properly only when `c` has the value of an uppercase letter. In ANSI C, `toupper()` and `tolower()` are implemented either as functions or as macros, but their behavior is different. If `c` has the value of a lowercase letter, then `toupper(c)` returns the value of the corresponding uppercase letter. If `c` does not have the value of a lowercase letter, then the value `c` is returned. Similar remarks hold with respect to `tolower()`.
- In ANSI C, every macro is also available as a function. Suppose *stdio.h* has been included. Then `putchar(c)` is a macro call, but `(putchar)(c)` is a function call.

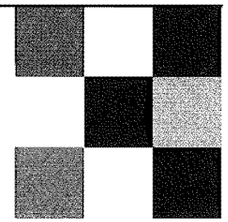
C.11 Header Files

- ANSI C has added new header files. The header file *stdlib.h* contains function prototypes for many of the functions in the standard library.
- The header files *float.h* and *limits.h* contain macro definitions describing implementation characteristics. ANSI C requires that certain minimum values and ranges be supported for each arithmetic type.

C.12 Miscellaneous

- In traditional C, the operators += and =+ are synonymous, although the use of =+ is considered old-fashioned. In ANSI C, the use of =+, =*, and so on is not allowed.
- An ANSI C compiler treats an assignment operator such as += as a single token. In traditional C, since it is treated as two tokens, white space can occur between the + and =. Thus, the expression `a + = 2` is legal in traditional C but illegal in ANSI C.
- Each of the following has a distinct name space: label identifiers, variable identifiers, tag names, and member names for each structure and union. All tags for `enum`, `struct`, and `union` comprise a single name space.
- Two identifiers are considered distinct if they differ within the first *n* characters, where *n* must be at least 31.
- The expression controlling a `switch` statement can be any integral type. Floating types are not allowed. The constant integral expression in a `case` label can be any integral type, including an enumerator.
- Pointers and `int`s are not interchangeable. Only the integer 0 can be assigned to a pointer without a cast.
- Pointer expressions may point to one element beyond an allocated array.

- External declarations and linkage rules are more carefully defined.
- Many changes have been made to the standard library and its associated header files.



Appendix D

ASCII Character Codes

American Standard Code for Information Interchange										
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	np	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	‘	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

How to Read the Table

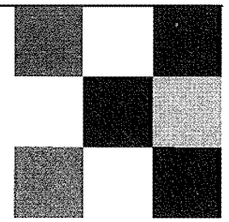
- Observe that the character A is in row six, column five. This means that the character A has value 65.

Some Observations

- Character codes 0 through 31 and 127 are nonprinting.
- Character code 32 prints a single space.
- Character codes for digits 0 through 9 are contiguous, letters A through Z are contiguous, and letters a through z are contiguous.
- The difference between a capital letter and the corresponding lowercase letter is 32.

The meaning of some of the abbreviations			
bel	audible bell	ht	horizontal tab
bs	backspace	nl	newline
cr	carriage return	nul	null
esc	escape	vt	vertical tab

Note: On most UNIX systems, the command *man ascii* causes the ASCII table to be printed on the screen in decimal, octal, and hexadecimal.

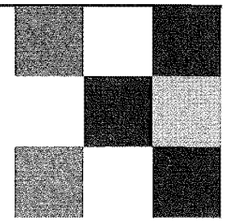


Appendix E

Operator Precedence and Associativity

The table below shows precedence and associativity for all the C++ operators. In case of doubt, parenthesize.

Operators	Associativity
`()` `[]` `->` `.` `++` *(postfix)* `--` *(postfix)*	left to right
`++` *(prefix)* `--` *(prefix)* `!` `~` `sizeof(`*type*`)` `+` *(unary)* `-` *(unary)* `&` *(address)* `*` *(indirection)*	right to left
`*` `/` `%`	left to right
`+` `-`	left to right
`<<` `>>`	left to right
`<` `<=` `>` `>=`	left to right
`==` `!=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`=` `+=` `-=` `*=` `/=` `*=` `>>=` `<<=` `&=` `^=` `|=`	right to left
`,` *(comma operator)*	left to right



Index

Symbols

D

E

F

G

H

I

J

K

P

Q

R

S

T

U

V

W

X

Y

Z