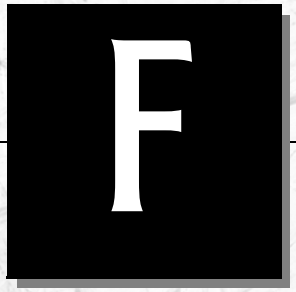

C Legacy Code Topics

Objectives

In this appendix you'll:

- Redirect keyboard input to come from a file and redirect screen output to a file.
- Write functions that use variable-length argument lists.
- Process command-line arguments.
- Process unexpected events within a program.
- Allocate memory dynamically for arrays, using C-style dynamic memory allocation.
- Resize memory dynamically allocated using C-style dynamic memory allocation.

Outline

F.1 Introduction

This appendix presents several additional C legacy code topics. Many of the capabilities discussed here are specific to particular operating systems, especially UNIX/LINUX/Mac OS X and/or Windows. Much of the material is for the benefit of C++ programmers who will need to work with older C legacy code.

F.2 Redirecting Input/Output on UNIX/Linux/Mac OS X and Windows Systems

Normally, the input to a program is from the keyboard (standard input), and the output from a program is displayed on the screen (standard output). On most computer systems—UNIX, LINUX, Mac OS X and Windows systems in particular—it is possible to **redirect** inputs to come from a file, and redirect outputs to be placed in a file. Both forms of redirection can be accomplished without using the file-processing capabilities of the standard library.

There are several ways to redirect input and output from the UNIX command line. Consider the executable file `sum` that inputs integers one at a time, keeps a running total of the values until the end-of-file indicator is set, then prints the result. Normally the user inputs integers from the keyboard and enters the end-of-file key combination to indicate that no further values will be input. With input redirection, the input can be stored in a file. For example, if the data are stored in file `input`, the command line

```
$ sum < input
```

causes program `sum` to be executed; the **redirect input symbol** (**<**) indicates that the data in file `input` (instead of the keyboard) is to be used as input by the program. Redirecting input in a Windows **Command Prompt** is performed identically.

Note that `$` represents the UNIX command-line prompt. (UNIX prompts vary from system to system and between shells on a single system.) Redirection is an operating-system function, not another C++ feature.

The second method of redirecting input is **piping**. A **pipe** (**|**) causes the output of one program to be redirected as the input to another program. Suppose program `random`

outputs a series of random integers; the output of `random` can be "piped" directly to program `sum` using the UNIX command line

```
$ random | sum
```

This causes the sum of the integers produced by `random` to be calculated. Piping can be performed in UNIX, LINUX, Mac OS X and Windows.

Program output can be redirected to a file by using the **redirect output symbol (>)**. (The same symbol is used for UNIX, LINUX, Mac OS X and Windows.) For example, to redirect the output of program `random` to a new file called `out`, use

```
$ random > out
```

Finally, program output can be appended to the end of an existing file by using the **append output symbol (>>)**. (The same symbol is used for UNIX, LINUX, Mac OS X and Windows.) For example, to append the output from program `random` to file `out` created in the preceding command line, use the command line

```
$ random >> out
```

F.3 Variable-Length Argument Lists

It is possible to create functions that receive an unspecified number of arguments. An ellipsis (`...`) in a function's prototype indicates that the function receives a variable number of arguments of any type.[1] Note that the ellipsis must always be placed at the end of the parameter list, and there must be at least one argument before the ellipsis. The macros and definitions of the **variable arguments header `<cstdarg>`** (Fig. F.1) provide the capabilities necessary to build functions with variable-length argument lists.

Identifier	Description
`va_list`	A type suitable for holding information needed by macros `va_start`, `va_arg` and `va_end`. To access the arguments in a variable-length argument list, an object of type `va_list` must be declared.
`va_start`	A macro that is invoked before the arguments of a variable-length argument list can be accessed. The macro initializes the object declared with `va_list` for use by the `va_arg` and `va_end` macros.
`va_arg`	A macro that expands to an expression of the value and type of the next argument in the variable-length argument list. Each invocation of `va_arg` modifies the object declared with `va_list` so that the object points to the next argument in the list.
`va_end`	A macro that performs termination housekeeping in a function whose variable-length argument list was referred to by the `va_start` macro.

Fig. F.1 | The type and the macros defined in header `<cstdarg>`.

1. In C++, programmers use function overloading to accomplish much of what C programmers accomplish with variable-length argument lists.

Figure F.2 demonstrates function `average` that receives a variable number of arguments. The first argument of `average` is always the number of values to be averaged, and the remainder of the arguments must all be of type `double`.

Function `average` uses all the definitions and macros of header `<cstdarg>`. Object `list`, of type **`va_list`**, is used by macros **`va_start`**, **`va_arg`** and **`va_end`** to process the vari-

```
1  // Fig. F.2: figF_02.cpp
2  // Using variable-length argument lists.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdarg>
6  using namespace std;
7
8  double average( int, ... );
9
10 int main()
11 {
12    double double1 = 37.5;
13    double double2 = 22.5;
14    double double3 = 1.7;
15    double double4 = 10.2;
16
17    cout << fixed << setprecision( 1 ) << "double1 = "
18         << double1 << "\ndouble2 = " << double2 << "\ndouble3 = "
19         << double3 << "\ndouble4 = " << double4 << endl
20         << setprecision( 3 )
21         << "\nThe average of double1 and double2 is "
22         << average( 2, double1, double2 )
23         << "\nThe average of double1, double2, and double3 is "
24         << average( 3, double1, double2, double3 )
25         << "\nThe average of double1, double2, double3"
26         << " and double4 is "
27         << average( 4, double1, double2, double3, double4 )
28         << endl;
29 } // end main
30
31 // calculate average
32 double average( int count, ... )
33 {
34    double total = 0;
35    va_list list;  // for storing information needed by va_start
36
37    va_start( list, count );
38
39    // process variable-length argument list
40    for ( int i = 1; i <= count; i++ )
41       total += va_arg( list, double );
42
43    va_end( list ); // end the va_start
44    return total / count;
45 } // end function average
```

Fig. F.2 | Using variable-length argument lists. (Part 1 of 2.)

```
double1 = 37.5
double2 = 22.5
double3 = 1.7
double4 = 10.2

The average of double1 and double2 is 30.000
The average of double1, double2, and double3 is 20.567
The average of double1, double2, double3 and double4 is 17.975
```

Fig. F.2 | Using variable-length argument lists. (Part 2 of 2.)

able-length argument list of function `average`. The function invokes `va_start` to initialize object `list` for use in `va_arg` and `va_end`. The macro receives two arguments—object `list` and the identifier of the rightmost argument in the argument list before the ellipsis—`count` in this case (`va_start` uses `count` here to determine where the variable-length argument list begins).

Next, function `average` repeatedly adds the arguments in the variable-length argument list to the `total`. The value to be added to `total` is retrieved from the argument list by invoking macro `va_arg`. Macro `va_arg` receives two arguments—object `list` and the type of the value expected in the argument list (`double` in this case)—and returns the value of the argument. Function `average` invokes macro `va_end` with object `list` as an argument before returning. Finally, the average is calculated and returned to `main`. Note that we used only `double` arguments for the variable-length portion of the argument list.

Variable-length argument lists promote variables of type `float` to type `double`. These argument lists also promote integral variables that are smaller than `int` to type `int` (variables of type `int`, `unsigned`, `long` and `unsigned long` are left alone).

Software Engineering Observation F.1

Variable-length argument lists can be used only with fundamental-type arguments and with `struct`*-type arguments that do not contain C++ specific features such as* `virtual` *functions, constructors, destructors, references,* `const` *data members and* `virtual` *base classes.*

Common Programming Error F.1

Placing an ellipsis in the middle of a function parameter list is a syntax error. An ellipsis may be placed only at the end of the parameter list.

F.4 Using Command-Line Arguments

On many systems it is possible to pass arguments to `main` from a command line by including parameters `int argc` and `char *argv[]` in the parameter list of `main`. Parameter **`argc`** receives the number of command-line arguments. Parameter **`argv`** is an array of `char *`'s pointing to strings in which the actual command-line arguments are stored. Common uses of command-line arguments include printing the arguments, passing options to a program and passing filenames to a program.

Figure F.3 copies a file into another file one character at a time. The executable file for the program is called `copyFile` (i.e., the executable name for the file). A typical command line for the `copyFile` program on a UNIX system is

```
$ copyFile input output
```

```
1  // Fig. F.3: figF_03.cpp
2  // Using command-line arguments
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  int main( int argc, char *argv[] )l
8  {
9     // check number of command-line arguments
10    if ( argc != 3 )
11       cout << "Usage: copyFile infile_name outfile_name" << endl;
12    else
13    {
14       ifstream inFile( argv[ 1 ], ios::in );
15
16       // input file could not be opened
17       if ( !inFile )
18       {
19          cout << argv[ 1 ] << " could not be opened" << endl;
20          return -1;
21       }  // end if
22
23       ofstream outFile( argv[ 2 ], ios::out );
24
25       // output file could not be opened
26       if ( !outFile )
27       {
28          cout << argv[ 2 ] << " could not be opened" << endl;
29          inFile.close();
30          return -2;
31       } // end if
32
33       char c = inFile.get(); // read first character
34
35       while ( inFile )
36       {
37          outFile.put( c );   // output character
38          c = inFile.get();   // read next character
39       }  // end while
40    }  // end else
41 }  // end main
```

Fig. F.3 | Using command-line arguments.

This command line indicates that file `input` is to be copied to file `output`. When the program executes, if `argc` is not `3` (`copyFile` counts as one of the arguments), the program prints an error message (line 11). Otherwise, array `argv` contains the strings `"copyFile"`, `"input"` and `"output"`. The second and third arguments on the command line are used as file names by the program. The files are opened by creating `ifstream` object `inFile` and `ofstream` object `outFile` (lines 14 and 23). If both files are opened successfully, characters are read from file `input` with member function `get` and written to file `output` with member function `put` until the end-of-file indicator for file `input` is set (lines 35–39). Then the program terminates. The result is an exact copy of file `input`. Note that not all computer

systems support command-line arguments as easily as UNIX, LINUX, Mac OS X and Windows. Some VMS and older Macintosh systems, for example, require special settings for processing command-line arguments. See the manuals for your system for more information on command-line arguments.

F.5 Notes on Compiling Multiple-Source-File Programs

As stated earlier in the text, it is normal to build programs that consist of multiple source files. There are several considerations when creating programs in multiple files. For example, the definition of a function must be entirely contained in one file—it cannot span two or more files.

In Chapter 6, we introduced the concepts of storage duration and scope. We learned that variables declared outside any function definition are of static storage duration by default and are referred to as global variables. Global variables are accessible to any function defined in the same file after the variable is declared. Global variables also are accessible to functions in other files; however, the global variables must be declared in each file in which they are used. For example, if we define global integer variable `flag` in one file, and refer to it in a second file, the second file must contain the declaration

```
extern int flag;
```

prior to the variable's use in that file. In the preceding declaration, the storage class-specifier `extern` indicates to the compiler that variable `flag` is defined either later in the same file or in a different file. The compiler informs the linker that unresolved references to variable `flag` appear in the file. (The compiler does not know where `flag` is defined, so it lets the linker attempt to find `flag`.) If the linker cannot locate a definition of `flag`, a linker error is reported. If a proper global definition is located, the linker resolves the references by indicating where `flag` is located.

Performance Tip F.1

Global variables increase performance because they can be accessed directly by any function—the overhead of passing data to functions is eliminated.

Software Engineering Observation F.2

Global variables should be avoided unless application performance is critical or the variable represents a shared global resource such as `cin`, because they violate the principle of least privilege, and they make software difficult to maintain.

Just as `extern` declarations can be used to declare global variables to other program files, function prototypes can be used to declare functions in other program files. (The `extern` specifier is not required in prototypes.) This is accomplished by including the function prototype in each file in which the function is invoked, then compiling each source file and linking the resulting object code files together. Function prototypes indicate to the compiler that the specified function is defined either later in the same file or in a different file. The compiler does not attempt to resolve references to such a function—that task is left to the linker. If the linker cannot locate a function definition, an error is generated.

As an example of using function prototypes to extend the scope of a function, consider any program containing the preprocessor directive `#include <cstring>`. This directive

includes in a file the function prototypes for functions such as `strcmp` and `strcat`. Other functions in the file can use `strcmp` and `strcat` to accomplish their tasks. The `strcmp` and `strcat` functions are defined for us separately. We do not need to know where they are defined. We are simply reusing the code in our programs. The linker resolves our references to these functions. This process enables us to use the functions in the standard library.

Software Engineering Observation F.3

Creating programs in multiple source files facilitates software reusability and good software engineering. Functions may be common to many applications. In such instances, those functions should be stored in their own source files, and each source file should have a corresponding header file containing function prototypes. This enables programmers of different applications to reuse the same code by including the proper header file and compiling their application with the corresponding source file.

Portability Tip F.1

Some systems do not support global variable names or function names of more than six characters. This should be considered when writing programs that will be ported to multiple platforms.

It is possible to restrict the scope of a global variable or function to the file in which it is defined. The storage-class specifier `static`, when applied to a global namespace scope variable or a function, prevents it from being used by any function that is not defined in the same file. This is referred to as **internal linkage**. Global variables (except those that are `const`) and functions that are not preceded by `static` in their definitions have **external linkage**—they can be accessed in other files if those files contain proper declarations and/or function prototypes.

The global variable declaration

```
static double pi = 3.14159;
```

creates variable `pi` of type `double`, initializes it to `3.14159` and indicates that `pi` is known only to functions in the file in which it is defined.

The `static` specifier is commonly used with utility functions that are called only by functions in a particular file. If a function is not required outside a particular file, the principle of least privilege should be enforced by using `static`. If a function is defined before it is used in a file, `static` should be applied to the function definition. Otherwise, `static` should be applied to the function prototype. Identifiers defined in the unnamed `namespace` also have internal linkage. The C++ standard recommends using the unnamed `namespace` rather than `static`.

When building large programs from multiple source files, compiling the program becomes tedious if making small changes to one file means that the entire program must be recompiled. Many systems provide special utilities that recompile only source files dependent on the modified program file. On UNIX systems, the utility is called `make`. Utility `make` reads a file called `Makefile` that contains instructions for compiling and linking the program. Systems such as Borland C++ and Microsoft Visual C++ for PCs provide `make` utilities and "projects." For more information on `make` utilities, see the manual for your particular system.

F.6 Program Termination with `exit` and `atexit`

The general utilities library (<cstdlib>) provides methods of terminating program execution other than a conventional return from `main`. Function `exit` forces a program to terminate as if it executed normally. The function often is used to terminate a program when an error is detected or if a file to be processed by the program cannot be opened.

Function `atexit` **registers** a function in the program to be called when the program terminates by reaching the end of `main` or when `exit` is invoked. Function **`atexit`** takes a pointer to a function (i.e., the function name) as an argument. Functions called at program termination cannot have arguments and cannot return a value.

Function `exit` takes one argument. The argument is normally the symbolic constant **`EXIT_SUCCESS`** or **`EXIT_FAILURE`**. If `exit` is called with `EXIT_SUCCESS`, the implementation-defined value for successful termination is returned to the calling environment. If `exit` is called with `EXIT_FAILURE`, the implementation-defined value for unsuccessful termination is returned. When function `exit` is invoked, any functions previously registered with `atexit` are invoked in the reverse order of their registration, all streams associated with the program are flushed and closed, and control returns to the host environment. Figure F.4 tests functions `exit` and `atexit`. The program prompts the user to determine whether the program should be terminated with `exit` or by reaching the end of `main`. Note that function `print` is executed at program termination in each case.

```
1  // Fig. F.4: figF_04.cpp
2  // Using the exit and atexit functions
3  #include <iostream>
4  #include <cstdlib>
5  using namespace std;
6
7  void print();
8
9  int main()
10 {
11    atexit( print ); // register function print
12
13    cout << "Enter 1 to terminate program with function exit"
14         << "\nEnter 2 to terminate program normally\n";
15
16    int answer;
17    cin >> answer;
18
19    // exit if answer is 1
20    if ( answer == 1 )
21    {
22       cout << "\nTerminating program with function exit\n";
23       exit( EXIT_SUCCESS );
24    }  // end if
25
26    cout << "\nTerminating program by reaching the end of main"
27         << endl;
28 }  // end main
```

Fig. F.4 | Using the `exit` and `atexit` functions. (Part 1 of 2.)

```
29
30 // display message before termination
31 void print()
32 {
33    cout << "Executing function print at program termination\n"
34         << "Program terminated" << endl;
35 }  // end function print
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
2

Terminating program by reaching the end of main
Executing function print at program termination
Program terminated
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
1

Terminating program with function exit
Executing function print at program termination
Program terminated
```

Fig. F.4 | Using the `exit` and `atexit` functions. (Part 2 of 2.)

Terminating a program with function `exit` runs the destructors for only the static and global objects in the program. Terminating with function `abort` ends the program without running any destructors.

F.7 Type Qualifier `volatile`

The **`volatile`** type qualifier is applied to a definition of a variable that may be altered from outside the program (i.e., the variable is not completely under the control of the program). Thus, the compiler cannot perform optimizations (such as speeding program execution or reducing memory consumption, for example) that depend on "knowing that a variable's behavior is influenced only by program activities the compiler can observe."

F.8 Suffixes for Integer and Floating-Point Constants

C++ provides integer and floating-point suffixes for specifying the types of integer and floating-point constants. The integer suffixes are: **`u`** or **`U`** for an **`unsigned`** integer, **`l`** or **`L`** for a **`long`** integer, and **`ul`** or **`UL`** for an **`unsigned long`** integer. The following constants are of type `unsigned`, `long` and `unsigned long`, respectively:

```
174u
8358L
28373ul
```

If an integer constant is not suffixed, its type is `int`; if the constant cannot be stored in an `int`, it is stored in a `long`.

The floating-point suffixes are `f` or `F` for a `float` and `l` or `L` for a `long double`. The following constants are of type `long double` and `float`, respectively:

```
3.14159L
1.28f
```

A floating-point constant that is not suffixed is of type `double`. A constant with an improper suffix results in either a compiler warning or an error.

F.9 Signal Handling

An unexpected event, or **signal**, can terminate a program prematurely. Such events include **interrupts** (pressing <*Ctrl*> *C* on a UNIX, LINUX, Mac OS X or Windows system), **illegal instructions**, **segmentation violations**, **termination orders from the operating system** and **floating-point exceptions** (division by zero or multiplying large floating-point values). The **signal-handling library** provides function `signal` to **trap unexpected events**. Function `signal` receives two arguments—an integer signal number and a pointer to a signal-handling function. Signals can be generated by function `raise`, which takes an integer signal number as an argument. Figure F.5 summarizes the standard signals defined in header file `<csignal>`. The next example demonstrates functions `signal` and `raise`.

Figure F.6 traps an interactive signal (`SIGINT`) with function `signal`. The program calls `signal` with `SIGINT` and a pointer to function `signalHandler`. (Recall that a function's name is a pointer to that function.) When a signal of type `SIGINT` occurs, function `signalHandler` is called, a message is printed and the user is given the option to continue normal program execution. If the user wishes to continue execution, the signal handler is reinitialized by calling `signal` again (some systems require the signal handler to be reinitialized), and control returns to the point in the program at which the signal was detected. In this program, function `raise` is used to simulate an interactive signal. A random number between `1` and `50` is chosen. If the number is `25`, then `raise` is called to generate the signal. Normally, interactive signals are initiated outside the program. For example, pressing <*Ctrl*> *C* during program execution on a UNIX, LINUX, Mac OS X or Windows system generates an interactive signal that terminates program execution. Signal handling can be used to trap the interactive signal and prevent the program from terminating.

Signal	Explanation
`SIGABRT`	Abnormal termination of the program (such as a call to `abort`).
`SIGFPE`	An erroneous arithmetic operation, such as a divide by zero or an operation resulting in overflow.
`SIGILL`	Detection of an illegal instruction.
`SIGINT`	Receipt of an interactive attention signal.
`SIGSEGV`	An invalid access to storage.
`SIGTERM`	A termination request sent to the program.

Fig. F.5 | Signals defined in header `<csignal>`.

```
// Fig. F.6: figF_06.cpp
// Using signal handling
#include <iostream>
#include <iomanip>
#include <csignal>
#include <cstdlib>
#include <ctime>
using namespace std;

void signalHandler( int );

int main()
{
   signal( SIGINT, signalHandler );
   srand( time( 0 ) );

   // create and output random numbers
   for ( int i = 1; i <= 100; i++ )
   {
      int x = 1 + rand() % 50;

      if ( x == 25 )
         raise( SIGINT ); // raise SIGINT when x is 25

      cout << setw( 4 ) << i;

      if ( i % 10 == 0 )
         cout << endl; // output endl when i is a multiple of 10
   } // end for
} // end main

// handles signal
void signalHandler( int signalValue )
{
   cout << "\nInterrupt signal (" << signalValue
        << ") received.\n"
        << "Do you wish to continue (1 = yes or 2 = no)? ";

   int response;

   cin >> response;

   // check for invalid responses
   while ( response != 1 && response != 2 )
   {
      cout << "(1 = yes or 2 = no)? ";
      cin >> response;
   }  // end while

   // determine if it is time to exit
   if ( response != 1 )
      exit( EXIT_SUCCESS );

```

Fig. F.6 | Using signal handling. (Part 1 of 2.)

```
54      // call signal and pass it SIGINT and address of signalHandler
55      signal( SIGINT, signalHandler );
56   }  // end function signalHandler
```

```
   1   2   3   4   5   6   7   8   9  10
  11  12  13  14  15  16  17  18  19  20
  21  22  23  24  25  26  27  28  29  30
  31  32  33  34  35  36  37  38  39  40
  41  42  43  44  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60
  61  62  63  64  65  66  67  68  69  70
  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 1
 100
```

```
   1   2   3   4
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 2
```

Fig. F.6 | Using signal handling. (Part 2 of 2.)

F.10 Dynamic Memory Allocation with calloc and realloc

In Chapter 10, we discussed C++-style dynamic memory allocation with `new` and `delete`. C++ programmers should use `new` and `delete`, rather than C's functions **`malloc`** and **`free`** (header `<cstdlib>`). However, most C++ programmers will find themselves reading a great deal of C legacy code, and therefore we include this additional discussion on C-style dynamic memory allocation.

The general utilities library (`<cstdlib>`) provides two other functions for dynamic memory allocation—`calloc` and `realloc`. These functions can be used to create and modify **dynamic arrays**. As shown in Chapter 8, a pointer to an array can be subscripted like an array. Thus, a pointer to a contiguous portion of memory created by `calloc` can be manipulated as an array. Function **`calloc`** dynamically allocates memory for an array and initializes the memory to zeros. The prototype for `calloc` is

```
void *calloc( size_t nmemb, size_t size );
```

It receives two arguments—the number of elements (`nmemb`) and the size of each element (`size`)—and initializes the elements of the array to zero. The function returns a pointer to the allocated memory or a null pointer (`0`) if the memory is not allocated.

Function **`realloc`** changes the size of an object allocated by a previous call to `malloc`, `calloc` or `realloc`. The original object's contents are not modified, provided that the memory allocated is larger than the amount allocated previously. Otherwise, the contents are unchanged up to the size of the new object. The prototype for `realloc` is

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

Function `realloc` takes two arguments—a pointer to the original object (`ptr`) and the new size of the object (`size`). If `ptr` is `0`, `realloc` works identically to `malloc`. If `size` is `0` and `ptr` is not `0`, the memory for the object is freed. Otherwise, if `ptr` is not `0` and size is greater than zero, `realloc` tries to allocate a new block of memory. If the new space cannot be allocated, the object pointed to by `ptr` is unchanged. Function `realloc` returns either a pointer to the reallocated memory or a null pointer.

Common Programming Error F.2

Runtime errors may occur if you use the `delete` operator on a pointer resulting from `malloc`, `calloc` or `realloc`, or if you use `realloc` or `free` on a pointer resulting from the `new` operator.

F.11 Unconditional Branch: goto

Throughout the text we've stressed the importance of using structured programming techniques to build reliable software that is easy to debug, maintain and modify. In some cases, performance is more important than strict adherence to structured-programming techniques. In these cases, some unstructured programming techniques may be used. For example, we can use `break` to terminate execution of a repetition structure before the loop-continuation condition becomes false. This saves unnecessary repetitions of the loop if the task is completed before loop termination.

Another instance of unstructured programming is the **`goto` statement**—an unconditional branch. The result of the `goto` statement is a change in the flow of control of the program to the first statement after the **label** specified in the `goto` statement. A label is an identifier followed by a colon. A label must appear in the same function as the `goto` statement that refers to it. Figure F.7 uses `goto` statements to loop 10 times and print the counter value each time. After initializing `count` to `1`, the program tests `count` to determine whether it is greater than `10`. (The label `start` is skipped, because labels do not perform any action.) If so, control is transferred from the `goto` to the first statement after the label `end`. Otherwise, `count` is printed and incremented, and control is transferred from the `goto` to the first statement after the label `start`.

```
1  // Fig. F.7: figF_07.cpp
2  // Using goto.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     int count = 1;
10
11    start: // label
12       // goto end when count exceeds 10
13       if ( count > 10 )
14          goto end;
15
```

Fig. F.7 | Using `goto`. (Part 1 of 2.)

```
16          cout << setw( 2 ) << left << count;
17          ++count;
18
19          // goto start on line 17
20          goto start;
21
22       end: // label
23          cout << endl;
24    }  // end main
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. F.7 | Using `goto`. (Part 2 of 2.)

In Chapter 4, we stated that only three control structures are required to write any program—sequence, selection and repetition. When the rules of structured programming are followed, it is possible to create deeply nested control structures from which it is difficult to escape efficiently. Some programmers use `goto` statements in such situations as a quick exit from a deeply nested structure. This eliminates the need to test multiple conditions to escape from a control structure.

Performance Tip F.2

The `goto` statement can be used to exit deeply nested control structures efficiently but can make code more difficult to read and maintain. Its use is strongly discouraged.

Error-Prevention Tip F.1

The `goto` statement should be used only in performance-oriented applications. The `goto` statement is unstructured and can lead to programs that are more difficult to debug, maintain, modify and understand.

F.12 Unions

A **union** (defined with keyword **`union`**) is a region of memory that, over time, can contain objects of a variety of types. However, at any moment, a `union` can contain a maximum of one object, because the members of a `union` share the same storage space. It is your responsibility to ensure that the data in a `union` is referenced with a member name of the proper data type.

Common Programming Error F.3

The result of referencing a `union` member other than the last one stored is undefined. It treats the stored data as a different type.

Portability Tip F.2

If data are stored in a `union` as one type and referenced as another type, the results are implementation dependent.

At different times during a program's execution, some objects might not be relevant, while one other object is—so a `union` shares the space instead of wasting storage on objects

that are not being used. The number of bytes used to store a union will be at least enough to hold the largest member.

Performance Tip F.3

Using unions conserves storage.

Portability Tip F.3

The amount of storage required to store a union is implementation dependent.

A union is declared in the same format as a struct or a class. For example,

```
union Number
{
   int x;
   double y;
};
```

indicates that Number is a union type with members int x and double y. The union definition must precede all functions in which it will be used.

Software Engineering Observation F.4

Like a struct or a class declaration, a union declaration creates a new type. Placing a union or struct declaration outside any function does not create a global variable.

The only valid built-in operations that can be performed on a union are assigning a union to another union of the same type, taking the address (&) of a union and accessing union members using the structure member operator (.) and the structure pointer operator (->). unions cannot be compared.

Common Programming Error F.4

Comparing unions is a compilation error, because the compiler does not know which member of each is active and hence which member of one to compare to which member of the other.

A union is similar to a class in that it can have a constructor to initialize any of its members. A union that has no constructor can be initialized with another union of the same type, with an expression of the type of the first member of the union or with an initializer (enclosed in braces) of the type of the first member of the union. unions can have other member functions, such as destructors, but a union's member functions cannot be declared virtual. The members of a union are public by default.

Common Programming Error F.5

Initializing a union in a declaration with a value or an expression whose type is different from the type of the union's first member is a compilation error.

A union cannot be used as a base class in inheritance (i.e., classes cannot be derived from unions). unions can have objects as members only if these objects do not have a constructor, a destructor or an overloaded assignment operator. None of a union's data members can be declared static.

Figure F.8 uses the variable `value` of type `union Number` to display the value stored in the `union` as both an `int` and a `double`. The program output is implementation dependent. The program output shows that the internal representation of a `double` value can be quite different from the representation of an `int`.

```
1  // Fig. F.8: figF_08.cpp
2  // An example of a union.
3  #include <iostream>
4  using namespace std;
5
6  // define union Number
7  union Number
8  {
9     int integer1;
10    double double1;
11 };  // end union Number
12
13 int main()
14 {
15    Number value; // union variable
16
17    value.integer1 = 100; // assign 100 to member integer1
18
19    cout << "Put a value in the integer member\n"
20         << "and print both members.\nint:   "
21         << value.integer1 << "\ndouble: " << value.double1
22         << endl;
23
24    value.double1 = 100.0;  // assign 100.0 to member double1
25
26    cout << "Put a value in the floating member\n"
27         << "and print both members.\nint:   "
28         << value.integer1 << "\ndouble: " << value.double1
29         << endl;
30 }  // end main
```

```
Put a value in the integer member
and print both members.
int:   100
double: -9.25596e+061
Put a value in the floating member
and print both members.
int:   0
double: 100
```

Fig. F.8 | Printing the value of a union in both member data types.

An **anonymous** `union` is a `union` without a type name that does not attempt to define objects or pointers before its terminating semicolon. Such a `union` does not create a type but does create an unnamed object. An anonymous `union`'s members may be accessed directly in the scope in which the anonymous `union` is declared just as are any other local variables—there is no need to use the dot (`.`) or arrow (`->`) operators.

Anonymous unions have some restrictions. Anonymous unions can contain only data members. All members of an anonymous union must be public. And an anonymous union declared globally (i.e., at global namespace scope) must be explicitly declared static. Figure F.9 illustrates the use of an anonymous union.

```
// Fig. F.9: figF_09.cpp
// Using an anonymous union.
#include <iostream>
using namespace std;

int main()
{
   // declare an anonymous union
   // members integer1, double1 and charPtr share the same space
   union
   {
      int integer1;
      double double1;
      char *charPtr;
   };  // end anonymous union

   // declare local variables
   int integer2 = 1;
   double double2 = 3.3;
   char *char2Ptr = "Anonymous";

   // assign value to each union member
   // successively and print each
   cout << integer2 << ' ';
   integer1 = 2;
   cout << integer1 << endl;

   cout << double2 << ' ';
   double1 = 4.4;
   cout << double1 << endl;

   cout << char2Ptr << ' ';
   charPtr = "union";
   cout << charPtr << endl;
}  // end main
```

```
1 2
3.3 4.4
Anonymous union
```

Fig. F.9 | Using an anonymous union.

F.13 Linkage Specifications

It is possible from a C++ program to call functions written and compiled with a C compiler. As stated in Section 6.17, C++ specially encodes function names for type-safe linkage. C, however, does not encode its function names. Thus, a function compiled in C will

not be recognized when an attempt is made to link C code with C++ code, because the C++ code expects a specially encoded function name. C++ enables you to provide **linkage specifications** to inform the compiler that a function was compiled on a C compiler and to prevent the name of the function from being encoded by the C++ compiler. Linkage specifications are useful when large libraries of specialized functions have been developed, and the user either does not have access to the source code for recompilation into C++ or does not have time to convert the library functions from C to C++.

To inform the compiler that one or several functions have been compiled in C, write the function prototypes as follows:

```
extern "C" function prototype    // single function
extern "C"   // multiple functions
{
   function prototypes
}
```

These declarations inform the compiler that the specified functions are not compiled in C++, so name encoding should not be performed on the functions listed in the linkage specification. These functions can then be linked properly with the program. C++ environments normally include the standard C libraries and do not require you to use linkage specifications for those functions.

F.14 Wrap-Up

This appendix introduced a number of C legacy-code topics. We discussed redirecting keyboard input to come from a file and redirecting screen output to a file. We also introduced variable-length argument lists, command-line arguments and processing of unexpected events. You also learned about allocating and resizing memory dynamically.