Using Libraries

WEEK
1

As you create more and more useful functions, it may seem to become harder and harder to get them all linked together. C offers a way of grouping all of your useful functions into libraries. Today you learn:

- ☐ How to work with multiple source files.
- ☐ How `#include` files operate.
- ☐ What libraries are.
- ☐ Why libraries are important.
- ☐ How to create a library.
- ☐ How to use the library you create.
- ☐ A note about libraries you are already using.

Review of Program Creation

At this point in your C learning experience, you may or may not have written programs using more than one C source file. A C source file is a file that contains programming instructions (code). As your programs get larger, you reach a point where it is best to use more than one source file.

Expert Tip: It is up to each person to determine when code should be broken into multiple source files. When the code gets beyond 250 lines, as a rule, you may want to begin breaking it into separate files. It is best to have only related functions in a file. For an application, you may choose to put the screen functions in one file, the database functions in a second file, and the rest of the functions in a third file.

Using one or more source files that you create, you can begin the process of creating an executable file. Typically, these end with the extension of .C. Header files, which are included in the source files, typically have an extension of .H. Using the compiler, header files are converted to object files (.OBJ). The objects are then linked with library files (.LIB) to create executable files (either .EXE or .COM). Figure 7.1 shows this process.

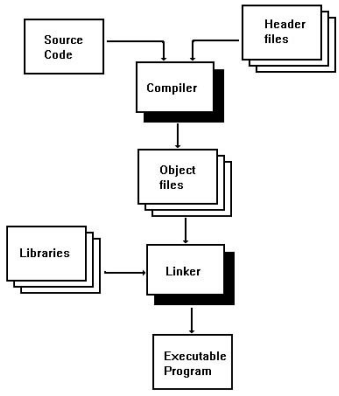

Figure 7.1. *The creation of an executable file.*

How To Work with Multiple Source Files

As you begin writing larger programs, or as you begin to write functions that may be reusable, you should start using multiple source files. There are several reasons for using separate source files. The first is size. Programs can get very large. Having multiple source files over 20K is not uncommon in "real-world" applications. In order to make maintaining files easier, most programmers break them into separate files that contain similar functions. For example, a programmer may set up three different source files for a large application. The programmer may put all of the screen functions in one file, all of the edit functions in another, and all the remaining code in a final source file. Figure 7.2 shows the creation of this executable file.

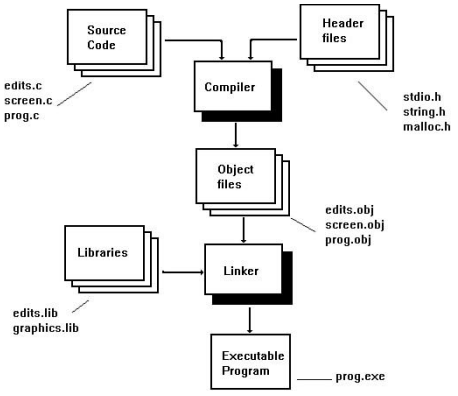

Figure 7.2. *The creation of an executable file with multiple source files.*

How To Include Header Files with *#include*

Header files are used in conjunction with the source files (.C). You have seen header files included in virtually every program in this book so far. These header files are generally included as follows:

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

In this example, `stdio.h` is the name of the header file. The `#include` is a preprocessor directive. This directive causes the following filename to be included in the current source file—hence the name. After the inclusion, the code in the header file becomes a part of the original source file. The following listings help to illustrate this point. These should all be in the same directory when you compile.

Type

Listing 7.1. Using include files.

```
1:   /* Program:  list0701.h
2:    * Author:   Bradley L. Jones
3:    * Purpose:  This is an include file with half a program.
4:    *============================================================*/
5:
6:   #include <stdio.h>
7:
8:   int main( void )
9:   {
10:
```

```
11:    printf("\nHELLO ");
12:
13:
```

Type

Listing 7.2. A second header file.

```
1:   /* Program:  list0702.h
2:    * Author:   Bradley L. Jones
3:    * Purpose:  This is another include file with the second
4:    *           half of a program.
5:    *=============================================================*/
6:
7:
8:    printf("WORLD!\n");
9:
10:   return;
11:  }
```

Type

Listing 7.3. The source file.

```
1:   /* Program:  list0703.c
2:    * Author:   Bradley L. Jones
3:    * Purpose:  This is a program to demonstrate include
4:    *           files.
5:    *=============================================================*/
6:
7:   #include "list0701.h"
8:   #include "list0702.h"
9:
10:
```

Note: To compile the previous listing, you need to only compile Listing 7.3 (LIST0703.C).

Output

```
HELLO WORLD!
```

Analysis

When LIST0703.C is compiled, the other two files are automatically included. This is obvious from the fact that the program runs! Listing 7.3 only contains the code for including the other two files. Listing 7.1 contains only the first half

of the code needed to produce the output. Line 11 of Listing 7.1 prints out the HELLO portion of the output. Listing 7.2 contains the code to print WORLD! Notice that Listing 7.3 does not appear to contain any relevant code at all. Lines 1 to 5 are simply comments and lines 7 and 8 are the includes.

The way the files in lines 7 and 8 of Listing 7.3 are included is slightly different than the way that you have been including files up until now. In addition, these lines are different than the include in line 6 of Listing 7.1. Instead of using <> around the file to be included, double quotes are used. Whether you use <>s or quotes makes a difference in what is included. Actually, the characters surrounding the included filename signal to the compiler where to look for the include file. Double quotes tell the compiler to look in the current directory for the include file first. The <> characters tell the compiler to start by looking in the include directory that was set up with the compiler.

Listing 7.4 is an equivalent listing to the previous three listings. This listing replaces the include statements in Listing 7.3 with the corresponding code listings. The precompiler would combine these listings into a listing similar to the following.

Listing 7.4. Partially precompiled version of Listing 7.3.

```
1:  /* Program:  list0703.c
2:   * Author:   Bradley L. Jones
3:   * Purpose:  This is a program to demonstrate include
4:   *           files.
5:   *=========================================================*/
6:
7:  /* Program:  list0701.h
8:   * Author:   Bradley L. Jones
9:   * Purpose:  This is an include file with half a program.
10:  *=========================================================*/
11:
12: #include <stdio.h>
13:
14: int main( void )
15: {
16:
17:   printf("\nHELLO ");
18:
19:
20:
21: /* Program:  list0702.h
22:  * Author:   Bradley L. Jones
23:  * Purpose:  This is another include file with the second
24:  *           half of a program.
25:  *=========================================================*/
26:
```

```
27:
28:     printf("WORLD!\n");
29:
30:     return;
31:  }
```

Output

```
HELLO WORLD!
```

Analysis

Lines 7 to 17 are an inserted copy of Listing 7.1. These lines of code replace the `#include` directive in Listing 7.3. Lines 21 to 31 are Listing 7.2. Again these lines have replaced the `#include` directive used in line 8 of Listing 7.3. This listing does not accurately reflect how the preprocessor would change the includes in line 12 also. This header file, stdio.h, would also be expanded out with the code in that file. This file is too long to add to this listing. In addition, each compiler comes with its own version of this file.

Note: The preceding listings were presented as a demonstration of how the `#include` directive works. The `#include` directories should not be used in the manner presented.

What Libraries Are

What is a library? A library is a set of functions that have been grouped together. A library allows all of the functions to be grouped into a single file. Any subset of the functions can be used by linking them with your programs.

Libraries enable you to share common functions. In fact, many—if not most—of the functions that you use in your programs are in libraries that were provided with your compiler. This includes functions such as `printf()`, `scanf()`, `strcmp()`, `malloc()`, and more. Most compilers come with a library reference manual that describes the usage of each function within the library. Notice that you don't need the source code for these functions. In fact, you don't really have to know how the internals of these functions work. What you do need to know is how to call them and what values they return. For instance, when using a function such as `puts()`, you need to know that a constant character pointer (a string) is passed to it. You also need to know that the function returns an integer. It is also beneficial to know what values the integer can

be and what each value represents. Whether `puts()` is written with a `for` loop or using system calls is irrelevant to its use. You can use the function without knowing its internals.

Review Tip: It is not required that you use a return value from a function. Some functions, such as `scanf()`, are used without regard to their return value.

You are not limited to the libraries the compiler comes with. You can create your own libraries of useful functions. Several of the later chapters in this book will have you create libraries of functions that will be useful in many of your applications. In addition, once you have created a library, you can give it to others to link with their programs. It is not uncommon to create several of your own libraries. In addition, most major programming shops have several of their own libraries.

In addition to creating your own libraries or getting them from your friends, you can also purchase libraries. The phrase "don't reinvent the wheel" is true for C programmers. If you are looking to get a quick jump on developing large scale applications, then the decision to purchase libraries should be considered. There are libraries available to do a number of tasks ranging from accessing or creating standard database formats to doing high-resolution graphics.

Note: Although "reinventing the wheel" is not necessarily good, it is important for learning. Many of the topics covered in this book could be avoided by purchasing libraries. However, by covering these topics, you will better understand how many of the functions work.

Working with Libraries

Virtually every C compiler comes with the capability to create libraries. This is invariably done with a library program. You will need to consult your computer's manuals to determine its library program.

If you are using Microsoft, the program used to manipulate libraries is LIB.

If you are using a Borland compiler, your library program is TLIB.

This book will use LIB from this point on. If you are using the Borland compiler, you will simply need to type TLIB in place of LIB. If you are using a compiler other than Microsoft's or Borland's, then you will need to consult your manuals for compatibility. Other compilers should have library functions that operate in a similar manner.

There are several tasks that may be done to manipulate libraries. Some of the functions you can perform on libraries include:

- ☐ Create a new library.
- ☐ List functions in a library.
- ☐ Add a function to a library.
- ☐ Update or replace a function in a library.
- ☐ Remove a function from a library.

The next few sections will cover each of these. In doing so, the following listings will be used. Enter these four listings and save them under their corresponding names.

Note: There is no output for each of the following listings. As you will see, these listings contain only individual functions. They are not complete "stand-alone" listings. They will be used in the following library manipulations.

Listing 7.5. A state edit.

```
1:   /* Program:  State.c
2:    * Author:   Bradley L. Jones
3:    * Purpose:  Validate a state abbreviation
4:    * Note:     Valid states are:
5:    *           AL, AK, AZ, CA, CO, CT, DE, FL, GA,
6:    *           HI, IA, ID, IL, IN, KS, KY, LA, MA,
7:    *           MD, ME, MI, MN, MO, MS, MT, NB, NC,
8:    *           ND, NE, NH, NV, NY, OH, OK, OR, PA,
9:    *           RI, SC, SD, TN, UT, VT, WS, WV, WY,
10:   * Return:   One of the following is returned:
11:   *           0 - Valid state
12:   *           1 - Invalid state
13:   *=================================================*/
14:
```

continues

Listing 7.5. continued

```
15:  #include <string.h>
16:  #include <stdio.h>
17:
18:  int is_valid_state( char *state )
19:  {
20:     char all_states[101] = {"ALAKAZCACOCTDEFLGA"
21:                             "HIIAIDILINKSKYLAMA"
22:                             "MDMEMIMNMOMSMTNBNC"
23:                             "NDNENHNVNYOHOKORPA"
24:                             "RISCSDTNUTVTWSWVWY" };
25:     int ctr;
26:
27:     for(ctr = 0; ctr < 100; ctr+=2)
28:     {
29:        if (strncmp(all_states+ctr, state, 2)==0)
30:        {
31:           return(0);    /* found state */
32:        }
33:     }
34:     return(1);
35  }
```

Analysis This function is an edit function. The comments in the first few lines of this function provide most of the details of what the function will do. As you see, the information in these comments completely documents the function's purpose and use. This function edits a string that is passed in to verify that it is a valid two-digit state abbreviation. A `for` loop starting in line 27 is used to move an offset through the `all_states` array. In line 29, each set of two characters of the `all_states` array is compared to the two characters passed to the function. If a match is found, the state is considered valid and the value of 0 is returned in line 31. If the state is not found, then the value of 1 is returned in line 34.

Like the state function there are several other edits that only allow for specific values. Listing 7.6 presents a function to verify the sex code.

Type

Listing 7.6. A sex code edit.

```
1:   /* Program:  Sex.c
2:    * Author:   Bradley L. Jones
3:    * Purpose:  Validate a sex code
4:    * Note:     Valid sex code. Valid values are:
5:    *           M or m - Male
6:    *           F or f - Female
7:    *           U or u - Other
8:    * Return:   One of the following is returned:
```

```
9:    *           0 - Valid code
10:   *           1 - Invalid code
11:   *=================================================*/
12:
13:  int is_valid_sex( char sex_code )
14:  {
15:     int rv = 0;
16:
17:     switch( sex_code )
18:     {
19:         case 'F':
20:         case 'f':
21:         case 'M':
22:         case 'm':
23:         case 'U':
24:         case 'u': rv = 0;
25:                   break;
26:         default:  rv = 1;
27:     }
28:     return(rv);
29:  }
```

Analysis This edit function verifies that the character received is a valid abbreviation for a sex code. The assumed valid values are M for male, F for female, and U for unknown. (This is the '90s!) This time a `switch` statement in line 17 is used to determine if the values are valid. If not, 1 is again returned. If the `sex_code` value is valid, then 0 is returned. You should notice that lines 20, 22, and 24 include the lowercase letters. This helps to make the edit more complete. Following in listing 7.7 is a more complex edit for a date.

Type **Listing 7.7. A date edit.**

```
1:   /* Program:  date.c
2:    * Author:   Bradley L. Jones
3:    * Purpose:  Pseudo-validate a date
4:    * Note:     This is not a very complete date edit.
5:    * Return:   One of the following is returned:
6:    *           +0  - Valid date
7:    *           -1  - Invalid day
8:    *           -2  - Invalid month
9:    *           -3  - Invalid year
10:   *=================================================*/
11:
12:  int is_valid_date( int month, int day, int year )
13:  {
14:     int rv = 0;
```

continues

Listing 7.7. continued

```
15:
16:     if( day < 1 || day > 31 )
17:     {
18:        rv = -1;
19:     }
20:     else
21:     {
22:        if( month < 1 || month > 12 )
23:        {
24:           rv = -2;
25:        }
26:        else
27:        {
28:           if( year < 1 || year > 2200 )
29:           {
30:              rv = -3;
31:           }
32:        }
33:     }
34:     return( rv );
35:  }
```

Analysis This is a third edit. As you can see, this is not as detailed as it could be. This edit validates a date. It receives three integers—`month`, `day`, and `year` in line 12. This edit does not fully edit the date. In line 16, it verifies that the day is 31 or less. In line 22, the edit checks to see if the month is from 1 to 12. In line 28, it checks to see that the year is a positive number less than 2200. Why 2200? Why not! It is just an arbitrary number that is higher than any date that would be used in any of my systems. You can use whatever cap you feel is adequate.

If the date values pass all of these checks, then 0 is returned to the calling function. If all the values don't pass, a negative number is returned signifying what was in error. Notice that three different negative numbers are being returned. A program calling the `is_valid_date()` function will be able to determine what was wrong. Listing 7.8 presents a function that uses the `is_valid_date()` function.

Type

Listing 7.8. A birthdate edit.

```
1:  /* Program:  bdate.c
2:   * Author:   Bradley L. Jones
3:   * Purpose:  Pseudo-validate a birthdate
4:   * Note:     This is not a very complete date edit.
5:   * Return:   One of the following is returned:
6:   *
7:   *            0  - Valid date
```

```
8:    *           -1  - Invalid day
9:    *           -2  - Invalid month
10:   *           -3  - Invalid year
11:   *           -4  - Invalid birthday (valid date > 1995 )
12:   *=====================================================*/
13:
14:  int is_valid_birthdate( int month, int day, int year )
15:  {
16:     int rv = 0;
17:
18:     rv = is_valid_date( month, day, year );
19:
20:     if( rv >= 0 )
21:     {
22:        if( year >= 1995 )
23:        {
24:           rv = -4;
25:        }
26:     }
27:     return( rv );
28:  }
```

Analysis This edit is a little more specific to applications. Like the date edit, it is not as complete as it could be. The reason for this simplicity is that this function is for use in describing library functions, not to emphasize editing values—although properly written edits are important.

The birthdate edit is a more specific version of the date edit. In fact, a part of the birthdate edit is to call the date edit (line 18). If the date edit in line 18 passes, then the birthdate edit verifies in line 22 that the date was before January 1, 1995. This is done by simply checking the year. If the year is equal to or greater than 1995, then the date is equal to or greater than January 1, 1995. A more appropriate check would be to verify that the date is before today's date. After all, you aren't born yet if your birthdate is tomorrow. Line 24 adds an additional error code to be returned if the date is valid but it is after January 1, 1995. If the date is invalid, the value returned from the date edit is returned. Notice that this edit requires the date edit to be complete.

How To Create a Library

Now that you have a set of edit functions, you will want to use them. The following program, presented in Listing 7.9 and Listing 7.10, uses the four edits that have been created. To create a program from multiple source files, you simply include them all when you compile. If you were using Borland's Turbo compiler, you could type:

```
TCC list0709.c state.c sex.c date.c bdate.c
```

The final outcome of this compilation would be an executable file called list0709.EXE.

Listing 7.9. Using the edit functions.

```
1:  /* Program:  list0709.c
2:   * Author:   Bradley L. Jones
3:   * Purpose:  This is a program to demonstrate the use of
4:   *           the edit functions.
5:   *=========================================================*/
6:
7:  #include <stdio.h>
8:  #include "edits.h"
9:
10: void main(void)
11: {
12:    int rv;
13:
14:    printf("\n\nUsing the edits:");
15:
16:    printf("\n\nUsing the state edit:");
17:    rv = is_valid_state("xx");
18:    printf("\n     State = xx, return value: %d", rv);
19:    rv = is_valid_state("IN");
20:    printf("\n     State = IN, return value: %d", rv);
21:
22:    printf("\n\nUsing the sex code edit:");
23:    rv = is_valid_sex('x');
24:    printf("\n     Sex code = x, return value: %d", rv);
25:    rv = is_valid_sex('F');
26:    printf("\n     Sex code = F, return value: %d", rv);
27:
28:    printf("\n\nUsing the date code edit:");
29:    rv = is_valid_date( 8, 11, 1812);
30:    printf("\n     Month:  8\n     Day:    11");
31:    printf("\n     Year:   1812, return value: %d", rv);
32:    rv = is_valid_date( 31, 11, 1812);
33:    printf("\n     Month:  31\n     Day:    11");
34:    printf("\n     Year:   1812, return value: %d", rv);
35:
36:    printf("\n\nUsing the birthdate code edit:");
37:    rv = is_valid_birthdate( 8, 11, 1999);
38:    printf("\n     Month:  8\n     Day:    11");
39:    printf("\n     Year:   1812, return value: %d", rv);
40:    rv = is_valid_date( 8, 11, 1812);
41:    printf("\n     Month:  31\n     Day:    11");
42:    printf("\n     Year:   1812, return value: %d", rv);
43: }
```

Type

Listing 7.10. A header file for the edit functions.

```
1:  /* edits.h
2:   * Prototypes for EDITS library
3:   *================================================*/
4:
5:  #if defined( __EDITS_H )
6:    /* this file has already been included */
7:  #else
8:  #define __EDITS_H
9:
10: int is_valid_state( char *state );
11: int is_valid_sex( char sex_code );
12: int is_valid_date( int month, int day, int year );
13: int is_valid_birthdate( int month, int day, int year );
14:
15: #endif
```

Output

```
Using the edits:

Using the state edit:
     State = xx, return value: 0
     State = IN, return value: 1

Using the sex code edit:

    Sex code = x, return value: 0
    Sex code = F, return value: 1

Using the date code edit:
     Month:  8
     Day:    11
     Year:   1812, return value: 0
     Month:  31
     Day:    11
     Year:   1812, return value: -2

Using the birthdate code edit:
     Month:  8
     Day:    11
     Year:   1812, return value: -4
     Month:  31
     Day:    11
     Year:   1812, return value: 0
```

Analysis

If additional listings are created that call these edit functions, you could include each of the appropriate edits during their compilations. This would be okay, but linking in a single library rather than a bunch of source files makes using the edits much easier.

Listing 7.9 should not offer any surprises. Line 8 includes a header file called edits.h. Notice that this is the same name as the library. This header file is presented in Listing 7.10. edits.h contains prototypes for each of the functions that are being included. It is good practice to create a file containing prototypes. Also notice the additional preprocessor directives in lines 5 to 8 and line 15 in Listing 7.10, edits.c. The `#if defined()` directive checks to see if the following value, `__EDITS_H`, has already been defined. If it has, then a comment is placed in the code stating that the file has already been included (line 6). If the value it has not been defined, then line 8 defines the value. If `__EDITS_H` has been defined, then the code from lines 8 to 14 is skipped. This logic helps to prevent a header file from being included more than once.

Expert Tip: Many programmers use the directives that are presented in Listing 7.10 in their own header files. This prevents the header files from being included more than once. Many programmers will create the defined name by adding two underscores to the name of the header file. Since periods cannot be part of a defined constant, the period is typically replaced with an additional underscore. This is how the define for Listing 7.10, `__EDITS_H`, was derived.

In the rest of the listing, each edit is called twice. In one case, an edit is called with valid data. In the second case, the edit is called with bad data.

Creating the library requires that each of the functions to be included be compiled into object files. Once you have the objects, you can create the library by using your compiler's library program. As stated earlier, Borland's library program is TLIB and Microsoft's is LIB. To create the library, you simply add each function to it.

DO	DON'T
DO understand the concepts behind libraries.	**DON'T** put different types of functions in the same library. A library should contain functions that have similarities. Examples of different libraries may be a graphics library, an edits library, a file management functions library, and a statistical library.
DO take advantage of the `#ifdef` preprocessor directive in your header files to prevent re-including the file.	

Adding New Functions to a Library

Adding functions to a library is done with the addition operator (+). This is true for both Borland and Microsoft compilers. To add a function, use the following format for the library command:

```
LIB libname +function_filename
```

`libname` is the name that you want the library to have. This name should be descriptive of the functions that are stored in the library. This brings up a good point. All of the functions stored in an individual library should be related in some fashion. For instance, all of the functions being used in the examples are edit functions. It would be appropriate to group these functions into a single library. A good name for the library might be EDITS.

Expert Tip: Only related functions should be stored in a library together.

`function_filename` is the name of the object file that is going to be added into the library. It is optional whether you include the .OBJ extension on the function filename. If you don't, the library program will assume that the extension is .OBJ. It is also assumed that the `libname` will have a .LIB extension. To create a library called EDITS that contains the `is_valid_state` function, you would enter the following:

```
LIB EDITS +state.obj
```

Compiler Warning: LIB is used in all the examples from this point on. Most compilers use a different program when working with libraries. As stated earlier, Borland uses TLIB. If using the Microsoft compiler, you will use LIB; however, you will need to add a semicolon to the end of each LIB statement.

To add additional functions, also use the addition operator. There is not a real difference between adding a function versus creating a library. If the library name already exists, you are adding functions. If it does not, you are creating a library. If you wanted to add the other three functions to the EDITS library, you would enter the following:

```
LIB EDITS +sex.obj +date.obj +bdate.obj
```

This adds the `is_valid_sex()`, `is_valid_date()`, and `is_valid_birthdate()` functions to the library. You should note that although the .OBJ extensions were included, they were not necessary.

Separate Source Files

You should notice that the functions are put into separate source files. By doing so, maintenance of future changes is made easier. Each function is isolated so that future enhancements only affect a single source file. Files are added at the object file level. A single object file is created from a source file. You should make it a practice of putting library functions into their own source files.

Expert Tip: You should make it a practice to put library functions into their own source files.

Using a Library

Once you have created a library, you can use it rather than each of the individual files that had been created. In fact, once the object files are stored in the library, you only need the original source file to make changes. You could give a library file to other users; they will be able to include any of its functions within their own programs. To use the library with Listing 7.9, you would enter the following at the command line:

```
CL list0709.c edits.lib
```

You should replace CL with the compile command that your compiler uses. Notice that the library file comes last. It should always be listed after all of the source files. This is equivalent to having included each of the files as shown previously.

Listing Information in a Library

Once you have created a library, you will probably want to know what functions are in it. Most library programs enable you to produce a listing that provides this information. To do this, you add the name of the list file to the command line as follows:

```
LIB libname ,listfile
```

In this case, `listfile` is the name of the file that the information on the library will be stored. A comma should be included on the command line before the list filename.

You could have included any operations in the command line after the library name and before the list filename. This includes the addition of functions as you have already seen, or any of the actions that follow.

Each compiler may produce a slightly different list file. If you use the Borland compiler, you would type the following to get a listing of the EDITS library stored in a file called INFO.LST:

```
TLIB EDITS ,INFO
```

If you are using the Borland compiler, you will notice that the .LST extension is the default extension that is added to the list file if an extension is not provided. Other compilers, such as the Microsoft compilers, may not add an extension to the list file automatically. As stated earlier, the .LIB extension is automatically added to a library file and therefore is not needed in the library name. If you typed the INFO.LST file, you would see the following:

```
Publics by module

BDATE      size = 46
     _is_valid_birthdate

DATE       size = 66
     _is_valid_date

SEX        size = 79
     _is_valid_sex

STATE      size = 203
     _is_valid_state
```

This provides you with information on both functions that are stored and the name of the file that the function is a part of. If you are using the Microsoft compiler, you would enter the following:

```
LIB EDITS ,LISTFILE.LST;
```

Printing the list file should provide output similar to the following:


```
_is_valid_birthdate..bdate          _is_valid_date....date
_is_valid_sex.....sex               _is_valid_state...state


state              Offset: 00000010H  Code and data size: cbH
  _is_valid_state

sex                Offset: 000002d0H  Code and data size: 4fH
  _is_valid_sex
```

```
date              Offset: 00000430H  Code and data size: 42H
  _is_valid_date

bdate             Offset: 00000560H  Code and data size: 2eH
  _is_valid_birthdate
```

While this is formatted a little different from the Borland output, the contents provide virtually the same information.

Removing Functions from a Library

In addition to adding functions to libraries, you can also remove functions that are no longer needed or that you don't want to distribute with a library. To remove a function, the subtraction operator (-) is used. The format is the same as adding functions.

```
LIB  libname -function_filename
```

As you can see, it is easy to remove a function. The following would remove the birthdate edit from the edit library:

```
LIB EDITS  -bdate.obj ,list
```

Notice that a list file called `list` is also being printed. Following is what would now be contained in the Borland and Microsoft list files after the subtraction:


```
_is_valid_date....date              _is_valid_sex.....sex
_is_valid_state...state

sex               Offset: 00000010H  Code and data size: 4fH
  _is_valid_sex

date              Offset: 00000160H  Code and data size: 42H
  _is_valid_date

state             Offset: 00000280H  Code and data size: afH
  _is_valid_state
```


```
Publics by module

DATE      size = 66
     _is_valid_date

SEX       size = 79
     _is_valid_sex

STATE     size = 175
     _is_valid_state
```

Updating Preexisting Functions in a Library

Updating a library is equivalent to deleting a function and then adding it again. The following could be used to update the `is_valid_state()` function:

```
LIB EDITS -state

LIB EDITS +state
```

Where LIB is your appropriate library program. You could also do this by combining the operations as such:

```
LIB EDITS -+state
```

This will remove the old state edit from the EDITS library and then add the new state function. This only works if the source filename is the same for both the old and the new function.

Extracting or Moving a Module from a Library

Not only can you put modules or object files into a library, but you can also copy them out of a library. To do this, you use the asterisk operator (*). The following pulls a copy of the STATE.OBJ file from the EDITS library where LIB is your appropriate library program:

```
LIB EDITS *state
```

Once you have keyed this, you will have made a copy of the state edit (within the STATE.OBJ file) that was in the EDITS library. This file will still be in the library also. The following will pull a copy of the STATE.OBJ file from the library and also remove it from the library:

```
LIB EDITS -*state
```

This will remove the state edit and create the object file.

What Libraries Are Available Already

As stated earlier, each compiler comes with its own set of libraries that have been created. In addition, most compilers come with Library Reference Manuals that detail each of the functions within the libraries. Generally, these standard libraries are automatically linked into your programs when you compile. If you use some of the more specific features of your compiler—such as graphics—then you may need to link in additional libraries.

A Final Example Using the Edits Library

Listing 7.11 is a listing that requires the EDITS.LIB file that you created earlier today. Notice that this is a completely different program from the one presented earlier. By including the library when compiling and linking, you can easily reuse the edit functions.

Type

Listing 7.11. Using the EDITS.LIB—again!

```
1:  /* Program:  list0711.c
2:   * Author:   Bradley L. Jones
3:   * Purpose:  This is a program to accept a valid birthday.
4:   * Note:     Requires EDITS.LIB to be linked
5:   *=========================================================*/
6:
7:  #include <stdio.h>
8:  #include "edits.h"
9:
10: void main(void)
11: {
12:    int rv;
13:    int month=0,
14:       day= 0,
15:       year=0;
16:
17:    printf("\n\nEnter your birthday (format: MM DD YY ):");
18:    scanf( "%d %d %d", &month, &day, &year );
19:
20:    rv = is_valid_birthdate(month, day, year);
21:
22:    while( rv < 0 )
23:    {
24:      printf("\n\nYou entered an invalid birthdate(%d-%d-%d)",
25:              month, day, year);
26:
27:      switch( rv )
28:      {
29:         case -1:  printf("\nError %d: BAD DAY", rv);
30:                   break;
31:         case -2:  printf("\nError %d: BAD MONTH", rv);
32:                   break;
33:         case -3:  printf("\nError %d: BAD YEAR", rv);
34:                   break;
35:         case -4:  printf("\nError %d: BAD BIRTHDATE", rv);
36:                   break;
37:         default:  printf("\nError %d: UNKNOWN ERROR", rv);
38:                   break;
```

```
39:        }
40:
41:        printf("\n\nRe-enter your birthday (format: MM DD YY):");
42:        scanf( "%d %d %d", &month, &day, &year );
43:
44:        rv = is_valid_birthdate(month, day, year);
45:      }
46:
47:      printf("\n\nYour birthdate is %d-%d-%d", month, day, year);
48:  }
```

Output

```
Enter your birthday (format: MM DD YY ):8 32 1965


You entered an invalid birthdate(8-32-1965)
Error -1: BAD DAY

Re-enter your birthday (format: MM DD YY):13 11 1965


You entered an invalid birthdate(13-11-1965)
Error -2: BAD MONTH

Re-enter your birthday (format: MM DD YY):8 11 -89


You entered an invalid birthdate(8-11—89)
Error -3: BAD YEAR

Re-enter your birthday (format: MM DD YY):8 11 1999


You entered an invalid birthdate(8-11-1999)
Error -4: BAD BIRTHDATE

Re-enter your birthday (format: MM DD YY):8 11 1965


Your birthdate is 8-11-1965
```

Analysis

This program enables you to enter the birthdate of a person. The user is required to enter the information until it is correct. If invalid information is entered, then the user is required to re-enter the entire date. As you will quickly see, the `is_valid_birthdate()` function or more appropriately, the `is_valid_date()` function that the birthday calls is not entirely accurate. It will consider dates such as February 30th valid. One of today's exercises asks you to rewrite this function so that it is more accurate.

Summary

Today, you were presented with a quick review of using multiple source files. This was followed with an explanation of using include files with the `#include` directive. The difference between double quotes and the more familiar <> signs was explained. The day then progressed into a description of libraries—what they are and why they are important. The usage of libraries along with how to add functions to them, remove functions from them, or update functions within them was explained. In addition, information on listing out information about a library was covered.

Q&A

Q Do all the functions in a library link into a program, thus increasing the programs size with unused functions?

A No! Only the functions that are called are linked.

Q Why can't I put all the functions in a library in the same source file?

A By isolating the functions into separate source files, you make the code easier to maintain. Since you add, update, and remove functions at the source code level, you isolate functions and changes by having separate source files. If you make a change to a function, you don't want to risk changing other functions.

Q What happens if you add a function to a library that is already there?

A You should be careful not to re-add a function. Some libraries will accept a function more than one time. This corrupts your library since you will not be sure which function is being used.

Q Can more than one library be used at a time?

A Yes! You will invariably use the default libraries that come with your compiler. When you create or add your own, you are most often using these libraries in addition to the default libraries. Some compilers have limitations to the number of libraries you can link in; however, these limits are seldom reached. For example, the Microsoft limit is 32 libraries.

Q What other functions can be performed on libraries?

A Only the basics of library manipulation were presented today. Most library programs will do additional functions; however, the way these more advanced functions are performed is not as standard. You should consult your compiler's manuals for more information. Such functions may include combining libraries or expanding some of the compiler's default libraries.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is a library?
2. Where do libraries come from?
3. How do you add a function to a library?
4. How do you remove a function from a library?
5. How do you update a function in a library?
6. How do you list the information in a library?
7. How do you copy an object file from a library?
8. How do you extract or move an object out of a library into an object file?
9. What is the difference between including a file with quotes and with <>s?
10. Can source files be included in libraries?

Exercises

1. How would you create a library called STATES.LIB containing the modules RI.OBJ, IL.OBJ, IN.OBJ?
2. How would you get a listing from the STATES library?
3. How would you create the STATES library from Exercise 1 and get a listing at the same time?

4. How would you add KY.OBJ and FL.OBJ to the STATES library?
5. How would you remove the KY module from the STATES library?
6. How would you replace the FL module with a new one in the STATES library?
7. How would you get a copy of the IL object from the STATES library?
8. Create a function to verify that a string contains all uppercase alpha characters (A to Z). If the string does contain all uppercase characters, return 1; otherwise, return 0.
9. Add the previous function to your edits library.
10. **ON YOUR OWN:** Rewrite the date edit to be more accurate. It should take into consideration the number of days that a particular month has. You may also want to include leap years!
11. **ON YOUR OWN:** Update your edits library with your new birthdate function.
12. **BUG BUSTER:** What is wrong with the following?

```
CL source1.c library.lib source2.obj
```

13. **ON YOUR OWN:** Create a listing file for some of the libraries that come with your compiler. These libraries are typically stored in a subdirectory called LIBRARY or LIB. You should notice many familiar functions.