

What you should observe about this lesson's program.

1. The definition of union Group looks similar to what would be used if we were to create a definition of a structure Group.
2. The definition of an enumeration simply lists identifiers. No data types are given.
3. The statement with typedef creates XXX as a synonym for struct Xxx by giving the data type for which a synonym is to be created followed by the synonym.
4. The variables ff and gg are of type struct Xxx.
5. The variable mm is of type union Group.
6. The variables hh and kk are enumerations.
7. The first printf statement in the program prints the numeric values of hh and kk.
8. The union member mm.dd is assigned a value of 12 and printed. Then, mm.ee is assigned a value of 28.4367 and both mm.dd and mm.ee are printed. At that point mm.dd no longer is printed to be 12.

Questions you should attempt to answer before reading the explanation.

1. What is the correlation between the enumeration identifier list and their numeric values?
2. Why is mm.dd no longer 12 after mm.ee is assigned to be 28.4367?

Source code

```
#include <stdio.h>

struct Xxx
{
    char aa;
    int bb;
};

union Group
{
    char cc;
    int dd;
    double ee;
};

enum Traffic_light
{
    red, yellow, green
};

typedef struct Xxx XXX;

void main(void)
{
    XXX ff, gg;
    union Group mm;
    enum Traffic_light hh, kk;
}
```

Definition of a union data type. Unlike a structure, a union uses the same memory cells for all its members.

Definition of an enumerated data type. In the list, red, yellow, and green are enumerators or members of the enumeration. Variables declared to be of type enum Traffic_light can have values of red, green, or yellow. (For this, red=0, yellow=1, green=2.)

This typedef allows us to use XXX instead of struct Xxx in declaring variables for the structure.

Declaring mm to be a variable of type union Group.

Declaring hh and kk to be variables of type enum Traffic_light.

```

ff.aa = 'z';
gg.bb = 5;

hh = red;
kk = green;
if (hh != kk) printf ("hh = %d, kk = %d\n", hh, kk);

mm.dd = 12;
printf ("mm.dd = %d\n", mm.dd);

mm.ee = 28.4367;
printf ("mm.ee = %lf, mm.dd = %d\n", mm.ee, mm.dd);
}

```

Annotations:

- Initializing the structure variables. (points to `ff.aa = 'z';` and `gg.bb = 5;`)
- Initializing the enumeration variables. (points to `hh = red;` and `kk = green;`)
- We can compare the values of enum variables. (points to `if (hh != kk)`)
- Initializing the union variable mm.dd. (points to `mm.dd = 12;`)
- Initializing mm.ee causes mm.dd to be overwritten. (points to `mm.ee = 28.4367;`)
- Printing mm.dd after mm.ee has overwritten it illustrates that mm.dd no longer is 12 even though we have not assigned anything new to mm.dd. (points to `printf ("mm.dd = %d\n", mm.dd);`)

Output

```

hh = 0, kk = 2
mm.dd = 12
mm.ee = 28.436700, mm.dd = 10695

```

Explanation

1. Does typedef create a new data type? No, it simply creates a new identifier for a data type already defined. For instance, in this lesson's program,

```
typedef struct Xxx XXX;
```

makes XXX equivalent to struct Xxx. For this statement to work properly, the struct Xxx data type must be defined before the typedef statement.

In general, the form for a typedef statement is

```
typedef data_type synonym_1, synonym_2, synonym_n;
```

where *data_type* can be any previously defined or standard data type, including int, float, double, or structure type and *synonym_1*, *synonym_2*, *synonym_n* represent a list of valid identifiers that can be used to represent *data_type*. Any number of synonyms can be used in the list of synonyms.

2. Why is typedef used? There are a number of reasons. Using a typedef that has a descriptive synonym can improve the readability and therefore the understandability of the code. Also, a typedef can make it easier to modify programs that are implementation dependent. For instance, suppose that you have developed a program that is dependent on the type int being 2 bytes long. Suppose further that you want the

program to work in an environment that has `int` as 4 bytes (instead of 2) but has `short` `int` as 2 bytes. Then, if you had the `typedef`

```
typedef int TWOBYTES;
```

in the original program, and `TWOBYTES` was used properly throughout the code, this one line could be modified to be

```
typedef short TWOBYTES;
```

to get the program to work in the new environment. Thus, `typedef` has made modification of the implementation-dependent code simple.

3. How would

```
typedef int GG[30];
```

be interpreted? The declaration in this `typedef` is for an array of `int` of size [30]. Therefore, using the declaration

```
GG yy, zz;
```

is equivalent to

```
int yy[30], zz[30];
```

4. Can we use `define` directives rather than `typedef`? In some cases, it is possible to use `define` directives in place of `typedef`; however, the mechanics are different because the preprocessor works with the `define` directive.

5. What is a union? A union is similar to a structure in that it defines a data type and members of that data type. However, a union differs from a structure in that the members of a union share the same memory cells. For instance, the union definition

```
union Group
{
  char cc;
  int dd;
  double ee;
};
```

defines the data type `union Group` to have three members: `cc`, `dd`, and `ee`. This definition allows us to make the declaration in the program

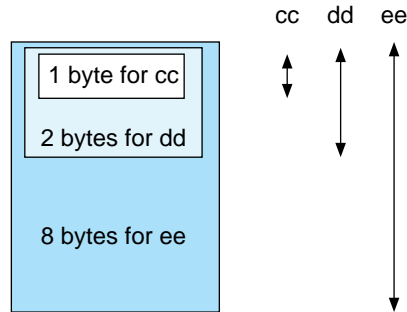
```
union Group mm;
```

that declares `mm` to have members, `cc`, `dd`, and `ee`. These members can be accessed using the notation `mm.cc`, `mm.dd`, `mm.ee`.

6. What is the impact of having the members share the same memory cells, as illustrated in Fig. 8.48? Only one member at a time can be used in the program.

FIG. 8.48

Memory cells shared for the union Group



In this lesson's program, we first assigned a value to `mm.dd` with the assignment statement

```
mm.dd=12;
```

However, a few statements later, we have the assignment statement

```
mm.ee = 28.4367;
```

Because `mm.dd` and `mm.ee` share the same memory cells, assigning a value to `mm.ee` causes `mm.dd` to be overwritten. So, the output from the `printf` statement

```
printf ("mm.ee = %lf, mm.dd = %d\n",mm.ee,mm.dd);
```

is

```
mm.ee = 28.4367, mm.dd = 17269
```

In this output, the value for `mm.ee` is correct, but the value for `mm.dd` is nonsense. The reason for this is that the value 28.4367 overwrites the original value of 12 for `mm.dd`. Because 28.4367 is written in the memory cells in the binary code for real numbers, when it is interpreted with the `%d` conversion specification, the result is meaningless.

7. Can we use the arrow (`->`) operator with a union? Yes; although we have not shown it in this lesson's program, the arrow operator can be used with union members in the same manner that we use it with structure members.

8. Why would we want to use a union? A union can be used in place of a structure in cases where only one member at a time is needed. Because the members of a union share memory cells, this can save memory.

If two union members have the same type, assigning a new value to one of them assigns that same value to the other. This may be a desirable feature for some programs.

Also, a union effectively divides a block of memory into byte-size regions. Having convenient access to just the first 2 bytes of an 8 byte region, for example, may be useful at times.

9. Are unions commonly used? Compared to structures, no. We include unions primarily to give a complete description of C's data structures.

10. What is an enumeration? An enumeration is a list of identifiers. Each identifier is automatically assigned a constant integer value. The value assigned depends on the order in which an identifier appears in the enumeration list. Unless otherwise specified, the first identifier in the list is assigned 0, the second is assigned 1, and so on through the list.

11. How do we define an enumeration? We define an enumeration with a tag and a list. For instance, the enumeration with the tag `Traffic_light`,

```
enum Traffic_light
    {
        red, yellow, green
    };
```

has the list `red, yellow, green`. This definition assigns the integer constants 0 to `red`, 1 to `yellow`, and 2 to `green`. Note that the items on the list are separated by commas.

The identifiers `red`, `yellow`, and `green` are members of the enumeration. The general form of an enumeration is

```
enum Enumeration_tag
    {
        identifier_0, identifier_1, identifier_2, identifier_n
    };
```

where `Enumeration_tag`, `identifier_0`, `identifier_1`, `identifier_2`, and `identifier_n` are any valid identifiers. Any number of identifiers can be listed. In this list, the identifiers are given the following integer values:

```
identifier_0=0
identifier_1=1
identifier_2=2
```

12. How do we use enumerations in a program? After giving the definition, we declare variables to be of the enumeration type, using

```
enum Enumeration_tag variable_1, variable_2, variable_n;
```

For instance, in this lesson's program,

```
enum Traffic_light hh, kk;
```

declares the variables `hh` and `kk` to be of the enumeration `Traffic_light`. Within the body of the program, we can use the assignments

```
hh = red;
kk = green;
if (hh != kk) printf ("hh = %d, kk = %d\n", hh, kk);
```

Because of the way that the enumeration was defined, `hh` is assigned the integer 0 and `kk` is assigned the integer 2 with these statements. We can compare enumerations and have done so with the `if` statement. The `printf` portion of this statement prints 0 for `hh` and 2 for `kk` as shown in the output.

13. Can we alter the integers assigned in an enumeration? Yes; for instance, if we had wanted to assign 0 to `red`, 20 to `yellow`, and 21 to `green`, we would have used the definition

```
enum Traffic_light
{
    red, yellow=20, green
};
```

Note that, in this definition, we specify that `yellow` is assigned 20. The value for `green` is automatically made to be 21 because it follows `yellow`. Because `red` is the first in the enumeration list and prior to `yellow`, it is assigned 0.

EXERCISES

1. True or false:
 - a. A typedef statement creates a synonym.
 - b. The members of a union occupy the same memory region.
 - c. An enumeration assigns double values to its variables.
 - d. The number of bytes occupied by a union is equal to the number of bytes needed for its largest member.
 - e. We commonly use the dot operator with an enumeration.
2. Given these definitions and declarations,

```
union Xxx
{
    int aa;
    double cc;
};
enum Count
{
    zero,
    one,
    two,
    three,
};
typedef union Xxx XXX;
typedef enum Count CT;

XXX kk, mm;
CT nn, pp;
```

find the errors, if any, in the following statements:

- a. `pp=four;`
- b. `nn.one = 15.3;`
- c. `kk.aa = mm.cc;`
- d. `CT.three = 3;`
- e. `mm.aa = nn1pp;`

Solutions

1. a (true), b (true), c (false), d (true), e (false)
2. a. `pp=three;`
- b. `nn=one;`
- c. No error, but only the integer portion of `mm.cc` will be assigned to `kk.aa`.
- d. `pp=three;`
- e. No error.

■ LESSON 8.19 FUNCTION-LIKE MACROS

Topics

- Using function-like macros
- Dangers of using function-like macros

Macros are not functions; however, sometimes you can use a macro instead of a very simple function. These macros are called *function-like macros*.

Recall that we used object-like macros earlier, when we defined constants in our programs. For instance, to define the identifier `PI` as `3.1415926` we used the following preprocessing directive:

```
#define PI 3.1415926
```

Recall also that the effect of this line was to direct the preprocessor (prior to compilation) to search the code for references to `PI` and replace them with `3.1415926`. On compilation, the identifier `PI` no longer was present in the code, having been replaced by the numerical value `3.1415926`. We also noted that the standard convention for naming macros is to use all capital letters.

The program for this lesson uses two function-like macros, `ADD1` and `ADD2`. Find their preprocessing directives. Use what you have learned about functions to deduce the way that the preprocessing directive works with a function-like macro.

What you should observe about this lesson's program.

1. Three lines of code beginning with `#define` are given at the very beginning of the program. Each of these lines has two parts, a part that is like a function call and part that is replacement text.
2. The replacement text for `ADD1` and `ADD2` differs only in the parentheses used.
3. The values printed for `w` and `x` are different although their assignment statements are very similar.

Questions you should attempt to answer before reading the explanation.

1. What is the replacement text for the code `ADD1(i,j,k)` ?
2. What is the replacement text for the code `COMPARE(w,x)`?

Source code

```
#define ADD1(a,b,c)
#define ADD2(a,b,c)
#define COMPARE(a,b)
```

```
    a+b+c
    ((a)+(b)+(c))
    if ((a) != (b))
```

Functionlike macros. Only one macro definition is allowed on each line. The preprocessor implements the macros by replacing code prior to compilation. It is not required, but by convention, all capital letters are used for the macro names.

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

```
void main(void)
{
```

```
    double i=3,j=4,k=5,m=6;
    double w,x,y,z;
```

For example, the definition of `ADD1` causes the text `ADD1(i,j,k)` to be replaced by `i+j+k`.

```
w = ADD1(i,j,k)/m;
x = ADD2(i,j,k)/m;
```

`ADD2(i,j,k)` is replaced by `((i)+(j)+(k))`.

```
y = ADD1(i,j,k+6)/m;
z = ADD2(i,j,k+6)/m;
```

`ADD1(i,j,k+6)` is replaced by `i+j+k+6`.

`ADD2(i,j,k)` is replaced by `((i)+(j)+(k+6))`.

```
printf (" w = %lf \n\r x = %lf \n\n", w,x);
```

```
COMPARE (w,x) printf ("w and x are not equal! \n\n");
```

`COMPARE(w,x)` is replaced by `if((w) != (x))`

```
}
```

Output

```
w = 7.833333
x = 2.000000

w and x are not equal!
```

Explanation

1. How does a function-like macro work? The form of the definition of a function-like macro is

```
#define macro_name(parameter_list) replacement_text_with_parameters
```

A function-like macro is used in a function body by writing the macro name with an expression list following it:

macro_name (expression_list)

The preprocessor searches the source code for the macro name. It then takes the expression list and parameter list from the macro definition to create replacement text. The following table lists the macro text, the replacement text created by the preprocessor, and the expression created with values:

Macro definition	Macro with parameters	Replacement text	Expression created in this lesson's program
ADD1(a,b,c) a+b+c	ADD1(i,j,k)	i+j+k	w= i+j+k/m = 3.+4.+5./6. = 7.8333
ADD2(a,b,c) ((a)+(b)+(c))	ADD2(i,j,k)	((i)+(j)+(k))	x=((i)+(j)+(k))/m =((3.)+(4.)+(5.))/6. = 2.00
ADD1(a,b,c) a+b+c	ADD1(i,j,k+6)	i+j+k+6	y=i+j+k+6/6 = 3.+4.+5.+6/6= 13.00
ADD2(a,b,c) ((a)+(b)+(c))	ADD2(i,j,k+6)	((i)+(j)+(k+6))	z=((i)+(j)+(k+6))/m=((3.)+(4.)+(5.+6))/6 = 3.00
COMPARE(a,b) if ((a)!= (b))	COMPARE(w,x)	if ((w)!= (x))	if ((w)!= (x)) printf ("w and x are not equal! \n\n");

2. Why are so many parentheses used in the definition of ADD2 for this lesson's code? Looking at the table, you can see that the parentheses make quite a big difference in the calculated results, because the replacement text goes directly into the location of the macro text. To be sure of using the macro correctly under all possible circumstances, it is necessary to use parentheses liberally, especially to enclose each of the macro's arguments. To be safe, we recommend that you embed many layers of parentheses within your function-like macros to get the result you want in any circumstance.

3. In a function-like macro definition, is any space allowed between the macro name and parameter list? No, it is important that you do not put a space between the macro name and the left parenthesis of the parameter list or within your macro name. If you do, the C compiler will treat your function-like macro as an object-like macro. Consider the following examples. Look at the macro name indicated and the replacement string:

```
#define ADD1 (a,b,c) a+b+c
```

Macro Replacement string

```
#define ADD1(a,b, c) a+b+c
```

Macro Replacement string

Note that spaces should not appear in a macro name but can occur in an argument list without causing a problem. If you have inadvertent spaces in your macro names, in most cases, the compiler will indicate errors prior to compilation. However, the compiler may not identify that the error is located in the macro. To be safe, we recommend that you not put spaces in your argument lists or your macro names.

4. Why use function-like macros rather than simple functions? The preprocessor handles function-like macros more efficiently than the compiler handles functions. So, your programs can run more efficiently with function-like macros than functions. Of course, only simple functions may be substituted with function-like macros.

5. How do we write function-like macros when they are too long to fit on one line? Put a backslash, \, at the end of each line to be continued. For example, a macro definition can be written

```
#define LONG_MACRO(a,b,c)    ((a)+(b)-(c)/(a))\
                             *((c)+(b)-(a))
```

6. What would be the effect of putting a macro into a quoted string? The macro will not be replaced with the replacement text. For instance, if we had put into this lesson's program the line

```
printf ("ADD1(i,j,k)");
```

the string ADD1(i,j,k) would be printed to the screen (not i+j+k).

EXERCISES

- True or false:
 - A macro name should not contain spaces.
 - Spaces are not allowed in a macro name, but they can appear in the replacement string.
 - ANSI C allows spaces between macro arguments. However, to reduce the possibility of error, we recommend that you do not use spaces.
 - Arguments in a macro argument list are type insensitive, meaning that the macro does not worry about the variable types of its arguments and allows you to use any type of data to replace the arguments.
 - If you use a macro 100 times in your program, your program instructions will need more space in memory because the preprocessor will replace the macro 100 times with the replacement string.
 - Theoretically, a program with a macro runs faster than a program with a function that is similar to the macro. The reason is that, when a program calls a function, the program control must shift back and forth between the function and the calling program.
- Find the error(s) in the following macro-related statements:
 - ```
#define ADD(a1b) a+b
```
  - ```
#define for(a,b)     a+b
```
 - ```
#define add(a,b) ((a)+(b))
printf("%d\n",add(3,4));
```

```

d. #define add(a,b) ((a)+(b))
 printf("%d\n",add(3.3,4));
e. #define add(a,b) ((a)+(b))
 printf("%d\n",add(3,4)+add(100,200));
f. #define add(a,b) ((a)+(b))
 printf("%d\n", add(add(add(1,2),3),4));

```

### Solutions

1. a (true), b (true), c (true), d (true), e (true), f (true)
2. a. Invalid macro argument separator, should be #define ADD(a,b) a+b, more parentheses should be used.
  - b. No error, but not recommended, because *for* is a keyword, more parentheses should be used.
  - c. No error.
  - d. Wrong format, result of 3.3+4 is 7.3, should use floating point format, such as %f, %g, or %e.
  - e. No error.
  - f. No error.

## LESSON 8.20 CONDITIONAL INCLUSION

### Topics

- Conditional preprocessor directives
- Techniques for debugging
- Techniques for producing portable code

*Conditional inclusion*, the ANSI C standard term for what often is called *conditional compilation*, refers to the use of preprocessor directives to include or exclude portions of the code during compilation. You may recall that all preprocessor directives begin with the symbol #. Recall also that no other C statement or portion of a C statement may be written on a line containing a preprocessor directive and that preprocessor directives are not terminated by a semicolon.

Look at the source code and find all of the preprocessor directives. You already are familiar with some of them, such as the ones beginning with #include and #define. The other directives, #ifdef, #if, #else, and #endif, work together in a manner similar to the if-else control structure we used many times. Look at the output. Can you deduce what code was included in the compilation and what was not?

### Source code

```

#define DEBUG_1
#define DEBUG_2 2
#define DEBUG_3 3
#include <stdio.h>

```

#define and #include  
preprocessor directives.

```

void main(void)
{
 int ii, jj, kk;
 double xx;

 ii=5;
 jj=20;
 kk=50;
 xx=15.3;

 #ifdef DEBUG_1
 printf("xx=%lf\n",xx);
 #endif

 #if DEBUG_2<1
 printf("ii=%d\n",ii);
 #else
 printf("jj=%d\n",jj);
 #endif

 #if DEBUG_3==1
 printf("ii=%d\n",ii);
 #elif DEBUG_2==1
 printf("jj=%d\n",jj);
 #elif defined DEBUG_1
 printf("kk=%d\n",kk);
 #endif

 #undef DEBUG_1
 #ifndef DEBUG_1
 printf("ii=%d\n",ii);
 #endif
}

```

Conditional inclusion.  
Because DEBUG\_1 is defined, the printf statement is included in the source code to be compiled.

Conditional inclusion.  
Because DEBUG\_2 is greater than 1, the printf statement in the False block is included in the source code to be compiled.

Conditional inclusion.  
This works similar to an if-else-if control structure. The preprocessor accepts the use of defined as a compile time operator.

Conditional inclusion.  
We can undefine a macro that has previously been defined. The directive, #ifndef reads "if not defined."

## Output

```

xx=15.300000
jj=20
kk=50
ii=5

```

## Explanation

*1. What are the meanings of the conditional inclusion preprocessor directives used in this lesson's program?* The following table lists the preprocessor directives and their meanings:

| Preprocessor directive | Meaning                                                                                                                                                                                                                                                                                                           |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>#ifdef</code>    | If the identifier following <code>#ifdef</code> is defined in a previous preprocessor directive, then this evaluates to True. Otherwise, it evaluates to False. If True, the statements in the True block are included. If False, the statements in the False block (if a False block is used) are included.      |
| <code>#if</code>       | If the constant expression following <code>#if</code> is True, then the statements in the True block are included. If the constant expression following <code>#if</code> is False, then the statements in the False block (if a False block is used) are included.                                                |
| <code>#else</code>     | The statements between <code>#else</code> and <code>#endif</code> form the False block.                                                                                                                                                                                                                           |
| <code>#endif</code>    | This marks the end of the conditional inclusion. One <code>#endif</code> must be used for each <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> .                                                                                                                                                 |
| <code>#ifndef</code>   | If the identifier following <code>#ifndef</code> is not defined in a previous preprocessor directive, then this evaluates to True. Otherwise, it evaluates to False. If True, the statements in the True block are included. If False, the statements in the False block (if a False block is used) are included. |
| <code>#elif</code>     | This works similar to “else if” in an if-else-if control structure.                                                                                                                                                                                                                                               |
| <code>#undef</code>    | This undefines a previously defined identifier.                                                                                                                                                                                                                                                                   |
| <code>defined</code>   | This is a compile time operator commonly used with <code>#elif</code> to make a conditional dependent on whether an identifier has been defined. It also can be used as <code>!defined</code> , meaning “not defined.”                                                                                            |

**2. What code was compiled in this lesson’s program?** Of the code shown, only the following statements were compiled:

```
void main(void)
{
 int ii, jj;
 double xx;
 ii=5;
 jj=20;
 xx=15.3;
 printf("xx=%lf\n",xx);
 printf("jj=%d\n",jj);
 printf("kk=%d\n",kk);
 printf("ii=%d\n",ii);
}
```

**3. How did this come about?** The three `#define` directives

```
#define DEBUG_1
#define DEBUG_2 2
#define DEBUG_3 3
```

cause the preprocessor to recognize that the identifier `DEBUG_1` is defined, that the identifier `DEBUG_2` is to be replaced with the constant 2, and that the identifier `DEBUG_3` is to be replaced with the constant 3. This causes the preprocessor to select the following code to compile:

|                                                                                                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                       |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>#ifdef DEBUG_1     printf("xx=%lf\n",xx); #endif</pre>                                                                                           | <div style="border: 1px solid black; padding: 5px; width: fit-content;">True block, included because DEBUG_1 previously has been defined.</div>                                                                                                                                                                                                                                                       |
| <pre>#if DEBUG_2&lt;1     printf("ii=%d\n",ii); #else     printf("jj=%d\n",jj); #endif</pre>                                                          | <div style="border: 1px solid black; padding: 5px; width: fit-content;">True block, not included because DEBUG_2 is replaced by 2, making 2&lt;1 evaluate to False.</div> <div style="border: 1px solid black; padding: 5px; width: fit-content;">False block, included because DEBUG_2&lt;1 evaluates to False.</div>                                                                                |
| <pre>#if DEBUG_3==1     printf("ii=%d\n",ii); #elif DEBUG_2==1     printf("jj=%d\n",jj); #elif defined DEBUG_1     printf("kk=%d\n",kk); #endif</pre> | <div style="border: 1px solid black; padding: 5px; width: fit-content;">Not included because DEBUG_3==1 evaluates to False.</div> <div style="border: 1px solid black; padding: 5px; width: fit-content;">Not included because DEBUG_2==1 evaluates to False.</div> <div style="border: 1px solid black; padding: 5px; width: fit-content;">Included because defined DEBUG_1 evaluates to True.</div> |
| <pre>#undef DEBUG_1 #ifdef DEBUG_1     printf("ii=%d\n",ii); #endif</pre>                                                                             | <div style="border: 1px solid black; padding: 5px; width: fit-content;">Included because DEBUG_1 is not defined due to #undef DEBUG_1.</div>                                                                                                                                                                                                                                                          |

**4. Can we nest conditional inclusions?** Yes, although we have not shown it. For instance, in this lesson's program, we could have used the following conditional inclusion:

```
#ifdef DEBUG_1
 #if DEBUG_2 ==2
 printf("ii=%d\n",ii);
 #else
 printf("jj=%d\n",jj);
 #endif
#else
 printf("kk=%d\n",kk);
#endif
```

Note the #endif line for both the inner and outer portions of the nest. This is required, as #endif always is needed to terminate a block of conditional inclusion.

**5. Why would we want to use conditional inclusion?** One use is to aid in debugging. In the process of debugging, it sometimes is helpful to eliminate portions of the

code to isolate the source of errors. We earlier illustrated the use of “commenting out” portions of code. However, because C does not permit nested comments, it often is not practical to “comment out” large sections of code because a comment may already exist within the large section. Using conditional inclusion, we need not be concerned about the presence of comments when excluding large sections of code.

Another common reason to use conditional inclusion is to easily manage the creation of several versions of a given program. For instance, you may have one version to run on one computer system and another to run on a different one. Instead of keeping the versions completely separate, we can use conditional inclusion to keep all the versions in one code. For instance, we can define

```
#define VERSION 1
```

or

```
#define VERSION 2
```

and use conditional inclusions of the sort

```
#if VERSION==1
...(statements for version 1)
#elif VERSION==2
...(statements for version 2)
#elif VERSION==3
...(statements for version 3)
#endif
```

#### EXERCISES

1. True or false:
  - a. Conditional inclusion is also called *conditional compilation*.
  - b. The preprocessor controls conditional inclusion.
  - c. It is not necessary to use #endif for all cases of using #if or #ifdef.
  - d. We can use relational expressions with conditional inclusion.
  - e. Conditional inclusion is convenient for maintaining several versions of a program.
2. Find the errors, if any, in these statements:
  - a. #if defined XX
  - b. #ifdef XX;
  - c. #else printf("kk=%d",kk);
  - d. #if def XX
  - e. #endif;

#### Solutions

1. a (true), b (true), c (false), d (true), e (true)
2. a. No error; however, #ifdef XX has the same effect.
  - b. #ifdef XX
  - c. #else  
printf("kk=%d",kk);
  - d. #ifdef XX
  - e. #endif

## LESSON 8.21 ARGUMENTS TO MAIN

### Topics

- Calling main using arguments
- Naming of arguments for main
- Meanings of arguments to main

We observed on several occasions in this book that the operating system calls the function main. However, to this point, we have not described how we can transfer information directly from the operating system to main. In this lesson, we illustrate how arguments can be passed to main when we execute the program.

Look in the source code to find

1. In the header for the function main, we do not have `void main(void)`, as we have had for most of the programs in this text. Instead, we have two arguments for main.
2. The first argument for main is an integer and the second argument is an array of pointers to strings.
3. After the source code, we show both the output and the program execution information (that is, what is given by the user when the program is executed). From this, `argc` and `argv[ ]` are given as input.
4. The first `printf` statement prints the value of the first argument for main. The output shows that its value is 4.
5. The addresses `argv[1]`, `argv[2]`, and `argv[3]` are used in the three `fopen` statements.
6. The names of the output files are the same as the information given in the Program Execution.

*A question you should attempt to answer before reading the explanation.*

1. The value of `argc` comes from the Program Execution. From this, what do you think the integer `argc` represents?

### Source code

```
#include <stdio.h>

void main(int argc, char *argv[])
{
 int i;
 FILE *outfile1, *outfile2, *outfile3;

 printf("argc=%d\n",argc);
```

Instead of void, we have two arguments for main, `argc` and `argv[ ]`. These are the only arguments allowed. They traditionally are called `argc` and `argv[ ]` but any valid identifiers are acceptable. The variable `argc` is an integer and `argv[ ]` is an array of pointers to strings.

Declaring pointers to three files. For this program, from the operating system, we pass the names of the three files.

`argc` automatically gives the number of strings represented by `argv[ ]`.

```

outfile1=fopen(argv[1],"w");
outfile2=fopen(argv[2],"w");
outfile3=fopen(argv [3],"w");

```

The strings represented by argv[ ] are used here as names of output files.

```

fprintf(outfile1,"Printout number 1\n");
fprintf(outfile2,"Printout number 2\n");
fprintf(outfile3,"Printout number 3\n");

```

Printing to the output files specified by the elements of argv[ ].

```

}

```

### Program execution

```

L8_21 file1.out file2.out file3.out

```

Program name.

This information is given on executing the program.

### Output

```

Screen.
argc=4

file1.out
Printout number 1

file2.out
Printout number 2

file3.out
Printout number 3

```

### Explanation

**1. What are the names, types, and meanings of main's arguments?** The function main has two arguments. Their names, types, and meanings for most implementations are

| Argument name<br>(name is traditional<br>but not required) | Type   | Meaning                                                                        | Example for this lesson's program                                                                                                                                                                       |
|------------------------------------------------------------|--------|--------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| argc                                                       | int    | Number of elements of the array argv[ ] (not including the null pointer)       | Here, argc = 4 because four command line arguments (strings) are given on program execution. The strings given for this execution are "L8_21", "file1.out", "file2.out", and "file3.out"                |
| argv[ ]                                                    | char * | Each element of this array is a pointer to a string given on program execution | For the execution of the program, given<br>argv[0]= address of "L8_21"<br>argv[1]= address of "file1.out"<br>argv[2]= address of "file2.out"<br>argv[3]= address of "file3.out"<br>argv[4]=null pointer |

## 2. *How has this program used main's arguments?* Instead of

```
void main(void)
```

for the header for the function main, we use

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

This indicates that a number of strings will be passed from the operating system to main at the time of program execution. ANSI C requires that the elements of `argv[ ]` point to each of the strings given on execution. The first element, `argv[0]`, points to the program name, which for this lesson is `L8_21`. The other elements of `argv[ ]` point to the other strings in the order given. C automatically makes these assignments. In addition, `argc` is assigned the value of the number of strings (including the program name).

For this lesson's execution,

```
argc=4
argv[0]= address of "L8_21"
argv[1]= address of "file1.out"
argv[2]= address of "file2.out"
argv[3]= address of "file3.out"
```

In the program, we use the last three of these strings to open three files with the statements

```
outfile1=fopen(argv[1],"w");
outfile2=fopen(argv[2],"w");
outfile3=fopen(argv [3],"w");
```

This effectively uses the strings passed to main as file names. We then print to these files with

```
fprintf(outfile1,"Printout number 1\n");
fprintf(outfile2,"Printout number 2\n");
fprintf(outfile3,"Printout number 3\n");
```

**3. *Can we use arguments to main for things other than file names?*** Yes, any information you want to transfer to a program in the form of a string at execution time can be passed to main. Such things as names, passwords, or identification type information sometimes are used as well.

**4. *Can we pass a numeric value using arguments to main?*** Yes, but it enters the program in the form of a string and must be converted to a numeric value using conversion functions such as `atoi()`, `atof()`, or other.

**5. *How many strings for argv[ ] can we use?*** It depends on your compiler, so check your compiler documentation. In many cases only a small number is permitted.

### EXERCISES

1. True or false:
  - a. There is no limit to the number of arguments to main that we can use.
  - b. The arguments to main must be called *argc* and *argv*.
  - c. The first argument to main always is an integer.

- d. The second argument to main always is a double.
- e. We cannot pass numeric information to main.

### **Solutions**

1. a (false), b (false), c (true), d (false), e (false)

## ■ LESSON 8.22 RANDOM FILE ACCESS

### Topics

- File positioning functions
- Determining the size of a file
- Macros used by the file positioning functions
- Using arrays for random access
- Types of constants

We saw in Lesson 8.14 that binary files store information in a manner similar to the way it is stored in memory. As a result, a binary file typically does not have the formatting that produces blank space or new lines with which a text file typically is filled. Because we directly write everything in a binary file that we create, it is relatively easy for us to move the file position indicator within a binary file to insert or extract specific information. We call moving the file position indicator within a file *positioning the file*, and C has a number of functions to help us do this, called *file positioning functions*. Also, C has a number of constant macros that are used with the file positioning functions.

In this lesson's program, we perform the following operations:

1. Copy the entire contents of an array into a binary file
2. Selectively read some of the array elements from the file
3. Sum the array elements
4. Determine the location of the file position indicator
5. Determine the size of the file

To do this, we use the file positioning functions `fseek` and `ftell` and constant macros `SEEK_SET` and `SEEK_END`. The function `fseek` positions the file so we can read an array element using `fread`, and `ftell` tells us the current location of the file position indicator in terms of the number of bytes that it is from the beginning of the file. The constant macros are located in `stdio.h`.

The function `fseek` uses the constant macro `SEEK_SET` (which is 0 for most C implementations) to establish the base point in the file from which to calculate where to move the file position indicator. A 0 means to start at the beginning of the file. The second argument to `fseek` specifies the number of bytes from the location indicated by the third argument to set the file position indicator.

### *What you should observe about this lesson's program.*

1. The first `fwrite` statement writes all of the array elements into a binary file.
2. In the second `for` loop in the program, we read just 9 of the 100 array elements.

3. Observe that *i* is used in the second argument in the call to `fseek` in the loop. Also, the second argument is cast to be type `long`.
4. The third argument in the call to `fseek` is the constant macro `SEEK_SET`.
5. After the `for` loop, we call the function `ftell`, which returns the position of the file.
6. The last call to `fseek` has `SEEK_END` as an argument.
7. The second argument in this call to `fseek` is `0L`, not just `0`.

### Source code

```

#include <stdio.h>
#include <stdlib.h>

void main(void)
{
 int i;
 long jj;
 double xx[100], yy, sum=0.0;
 FILE *outfile;

 outfile=fopen("L8_22.dat", "wb+");
 for(i=0; i<=100; i++) xx[i]=6*i;
 fwrite(xx, sizeof(double), 100, outfile);

 for(i=0; i<=40; i+=5)
 {
 fseek(outfile, (long)(i*sizeof(double)), SEEK_SET);
 fread(&yy, sizeof(double), 1, outfile);
 sum+=yy;
 }
 printf("sum=%lf\n", sum);

 jj=ftell(outfile);
 printf("The file position indicator is at %ld bytes.\n", jj);

 fseek(outfile, 0L, SEEK_END);
 jj=ftell(outfile);
 printf("The file length is %ld bytes.\n", jj);
}

```

**Initializing the array.**

In this program, we copy the contents of this array to a binary file.

Copying the entire array to a binary file.

Positioning the file so that we can read the *i*th element of the array as it is written on the disk file.

`SEEK_SET` indicates that the offset is measured from the beginning of the file.

From the beginning, we advance ( $i \times \text{sizeof}(\text{double})$ ) bytes. This argument must be of type `long int`.

We sum the values read.

We read one value and store it in the memory reserved for *yy*.

Because the file is positioned at the end, `ftell` returns the number of bytes in the file.

Moving the file position indicator to the end of the file (using `SEEK_END` with `0L` offset).

The function `ftell` returns the current location of the file position indicator in the file `outfile`. The return value is a long integer that represents the number of bytes from the beginning of the file.

## Output

```
sum=1080.000000
The file position indicator is at 328 bytes.
The file length is 800 bytes.
```

## Explanation

**1. What is meant by random file access?** There are two means by which files can be accessed, sequentially and randomly. Sequential access is what we have done throughout this text. Sequential access means that one item after another is either written to or read from a file. In sequential access, we cannot skip over an item to get to the subsequent item.

In random access, we can skip over items. For instance, if we want to read the last item in a file, with random access we can do so without reading all of the previous items. Also, if an entire file is empty and we want to write something at a location different from the beginning, we can do so with random access, whereas with sequential access this would not be possible. The functions `fseek` and `ftell` allow us to access a file in this manner.

You should be aware that the term *random access* is somewhat of a misnomer. We do not actually access the file in a random or haphazard manner. We very clearly specify how the file is to be accessed. A better name would be *nonsequential access*. However, in practice, the term *random access* is meant to indicate nonsequential access.

**2. What types of files usually are used for random access?** Binary files most commonly are used for random access, because binary files do not have formatting that makes it difficult to determine the exact byte offsets required by random access. However, it is possible to use the file positioning functions with text files. Because text files normally are not used for random access, we will cover only binary files.

**3. How many bytes are used to store constants?** When a constant (number) is written in a C source code, ANSI C requires it to be stored using a certain number of bytes. For instance, a number written with a decimal point or in scientific notation is required to be treated as a double (typically stored in 8 bytes of memory). A number written without a decimal point is required to be treated as the smallest type that can contain the value. For instance, with 2 byte integers, the constant 57 is required to be treated as a short int, whereas 587 is treated as an int and 53987 is a long int.

**4. Can we control the number of bytes used to store a constant?** To a certain extent, yes. We can modify the number of bytes used to store a constant as previously described above by using a suffix. The following table lists the available suffixes and their meanings. In this lesson's program, we use the constant `0L`, which means "zero represented as a long integer."

| Data type               | Suffix | Meaning                                                             | Examples                                                                                                   |
|-------------------------|--------|---------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| Integer type constants  | u or U | Stores the constant in the number of bytes used for an unsigned int | 54u<br>54U                                                                                                 |
|                         | l or L | Stores the constant in the number of bytes used for a long int      | 35l<br>35L<br>(The suffix <i>l</i> is not recommended because of its similarity to the number one)         |
| Floating type constants | f or F | Stores the constant in the number of bytes used for a float         | 48.6f<br>48.6F                                                                                             |
|                         | l or L | Stores the constant in the number of bytes used for a long double   | 987.35l<br>987.35L<br>(The suffix <i>l</i> is not recommended because of its similarity to the number one) |

**5. What are the meanings of the macros `SEEK_END`, `SEEK_SET`, and `SEEK_CUR`?** These macros are defined in the header file `stdio.h`. They are integer constant expressions meant to be used specifically for the third argument in a call to the function `fseek`. The meanings of these expressions follow:

| Macro                 | Meaning                          |
|-----------------------|----------------------------------|
| <code>SEEK_SET</code> | The beginning of the file        |
| <code>SEEK_END</code> | The end of the file              |
| <code>SEEK_CUR</code> | The current position of the file |

**6. How do we use `fseek` to position a file?** We need to pass three pieces of information to `fseek`: the file pointer, the number of bytes from the specified position (also called the *offset*), and the specified position. With this information, `fseek` moves the file position indicator to the location indicated by the second and third arguments. For instance, in the statement

```
fseek(outfile, 0L, SEEK_END);
```

the file to access is indicated by `outfile`, the number of bytes (or offset) from the specified position is 0 (0L), and the specified position is `SEEK_END` (meaning the end of the file). In general, the form for a call to `fseek` is

```
fseek(file_pointer, offset, specified_position)
```

where *file\_pointer* indicates the file with which to work, *offset* indicates the number of bytes from the specified position to set the file position indicator, and *specified\_position* is the position from which the offset is measured.

**7. What values are permitted for *offset*?** For *offset* we must use a long int. The value can be positive or negative; however, it must match the macro used for

*specified\_position*. For instance, if `SEEK_SET` (meaning the beginning of the file) is used for *specified\_position*, then a negative number cannot be used for *offset*. If `SEEK_END` is used for *specified\_position*, then a negative value for *offset* moves the file position indicator to a location before the current end of the file. If a positive value is used for *offset* with `SEEK_END` as the third argument, then the file position indicator moves past the current end of the file and expands the size of the file by an amount equal to *offset*. This is illustrated in Fig. 8.49.

### 8. What does the following statement do?

```
fseek(outfile, (long)(i*sizeof(double)), SEEK_SET);
```

This statement moves the file position indicator to a location  $i \times \text{sizeof}(\text{double})$  bytes from the beginning of the file (indicated by `SEEK_SET`). Because an array of `double` is stored in the file indicated by `outfile`, with this statement we are accessing the  $i$ th element of the array.

This illustrates a useful feature of using nonsequential access of files. When we store an array in a file, we can access the array elements using the file positioning functions in much the same way they can be accessed when the array simply is in memory. In other words, when an array is in memory, if we want the tenth element of the array, we need not read the first nine elements to access the tenth. Similarly, when an array is in a file, by using the file positioning functions, if we want the tenth element of the array, we need not read the first nine elements in order to access it.

This is useful because some arrays may be too large to store in memory and must be stored in a file. With the file positioning functions, we can treat these arrays in much the same way that we treat arrays in memory.

FIG. 8.49

Illustration of `SEEK_SET`, `SEEK_CUR`, `SEEK_END`, and the meanings of negative and positive offsets

