CHAPTER V

Working with the Preprocessor

This chapter focuses on questions pertaining to the preprocessor. The preprocessor is the program that is run before your program gets passed on to the compiler. You might never have seen this program before, because it is usually run "behind the scenes" and is hidden from the programmer. Nevertheless, its function is important.

The preprocessor is used to modify your program according to the *preprocessor directives* in your source code. Preprocessor directives (such as `#define`) give the preprocessor specific instructions on how to modify your source code. The preprocessor reads in all of your include files and the source code you are compiling and creates a preprocessed version of your source code. This preprocessed version has all of its macros and constant symbols replaced by their corresponding code and value assignments. If your source code contains any conditional preprocessor directives (such as `#if`), the preprocessor evaluates the condition and modifies your source code accordingly.

The preprocessor contains many features that are powerful to use, such as creating macros, performing conditional compilation, inserting predefined environment variables into your code, and turning compiler features on and off. For the professional programmer, in-depth knowledge of the features of the preprocessor can be one of the keys to creating fast, efficient programs.

As you read through the frequently asked questions in this chapter, keep in mind the techniques presented (as well as some of the common traps) so that you can tap into the full power behind the preprocessor and use its features effectively in your development cycle.

V.1: What is a macro, and how do you use it?

Answer:

A macro is a preprocessor directive that provides a mechanism for token replacement in your source code. Macros are created by using the `#define` statement. Here is an example of a macro:

```
#define VERSION_STAMP "1.02"
```

The macro being defined in this example is commonly referred to as a *symbol.* The symbol `VERSION_STAMP` is simply a physical representation of the string `"1.02"`. When the *preprocessor* is invoked (see FAQ V.2), every occurrence of the `VERSION_STAMP` symbol is replaced with the literal string `"1.02"`.

Here is another example of a macro:

```
#define CUBE(x) ((x) * (x) * (x))
```

The macro being defined here is named `CUBE`, and it takes one argument, `x`. The rest of the code on the line represents the body of the `CUBE` macro. Thus, the simplistic macro `CUBE(x)` will represent the more complex expression `((x) * (x) * (x))`. When the preprocessor is invoked, every instance of the macro `CUBE(x)` in your program is replaced with the code `((x) * (x) * (x))`.

Macros can save you many keystrokes when you are coding your program. They can also make your program much more readable and reliable, because you enter a macro in one place and use it in potentially several places. There is no overhead associated with macros, because the code that the macro represents is expanded in-place, and no jump in your program is invoked. Additionally, the arguments are not type-sensitive, so you don't have to worry about what data type you are passing to the macro.

Note that there must be *no white space* between your macro name and the parentheses containing the argument definition. Also, you should enclose the body of the macro in parentheses to avoid possible ambiguity regarding the translation of the macro. For instance, the following example shows the `CUBE` macro defined incorrectly:

```
#define CUBE (x) x * x * x
```

You also should be careful with what is passed to a macro. For instance, a very common mistake is to pass an incremented variable to a macro, as in the following example:

```
#include <stdio.h>

#define CUBE(x) (x*x*x)

void main(void);

void main(void)
{

     int x, y;

     x = 5;
     y = CUBE(++x);

     printf("y is %d\n", y);

}
```

What will y be equal to? You might be surprised to find out that y is *not* equal to 125 (the cubed value of 5) and *not* equal to 336 (6 * 7 * 8), but rather is 512. This is because the variable x is incremented while being passed as a parameter to the macro. Thus, the expanded CUBE macro in the preceding example actually appears as follows:

```
y = ((++x) * (++x) * (++x));
```

Each time x is referenced, it is incremented, so you wind up with a very different result from what you had intended. Because x is referenced three times and you are using a prefix increment operator, x is actually 8 when the code is expanded. Thus, you wind up with the cubed value of 8 rather than 5. This common mistake is one you should take note of because tracking down such bugs in your software can be a very frustrating experience. I personally have seen this mistake made by people with many years of C programming under their belts. I recommend that you type the example program and see for yourself how surprising the resulting value (512) is.

Macros can also utilize special operators such as the *stringizing* operator (#) and the *concatenation* operator (##). The stringizing operator can be used to convert macro parameters to quoted strings, as in the following example:

```
#define DEBUG_VALUE(v) printf(#v " is equal to %d.\n", v)
```

In your program, you can check the value of a variable by invoking the DEBUG_VALUE macro:

```
...

int x = 20;

DEBUG_VALUE(x);

...
```

The preceding code prints "x is equal to 20." on-screen. This example shows that the stringizing operator used with macros can be a very handy debugging tool.

The concatenation operator (##) is used to concatenate (combine) two separate strings into one single string. See FAQ V.16 for a detailed explanation of how to use the concatenation operator.

Cross Reference:

V.10: Is it better to use a macro or a function?

V.16: What is the concatenation operator?

V.17: How can type-insensitive macros be created?

V.18: What are the standard predefined macros?

V.31: How do you override a defined macro?

V.2: What will the preprocessor do for a program?

Answer:

The C preprocessor is used to modify your program according to the *preprocessor directives* in your source code. A preprocessor directive is a statement (such as `#define`) that gives the preprocessor specific instructions on how to modify your source code. The preprocessor is invoked as the first part of your compiler program's compilation step. It is usually hidden from the programmer because it is run automatically by the compiler.

The preprocessor reads in all of your include files and the source code you are compiling and creates a preprocessed version of your source code. This preprocessed version has all of its macros and constant symbols replaced by their corresponding code and value assignments. If your source code contains any conditional preprocessor directives (such as `#if`), the preprocessor evaluates the condition and modifies your source code accordingly.

Here is an example of a program that uses the preprocessor extensively:

```
#include <stdio.h>

#define TRUE                    1
#define FALSE                   (!TRUE)

#define GREATER(a,b)      ((a) > (b) ? (TRUE) : (FALSE))

#define PIG_LATIN         FALSE

void main(void);

void main(void)
{

     int x, y;

#if PIG_LATIN
     printf("Easeplay enternay ethay aluevay orfay xnay: ");
     scanf("%d", &x);
     printf("Easeplay enternay ethay aluevay orfay ynay: ");
     scanf("%d", &y);
#else
     printf("Please enter the value for x: ");
     scanf("%d", &x);
     printf("Please enter the value for y: ");
     scanf("%d", &y);
#endif

     if (GREATER(x,y) == TRUE)
     {

#if PIG_LATIN
          printf("xnay islay eatergray anthay ynay!\n");
#else
          printf("x is greater than y!\n");
#endif

     }
     else
     {
```

```
#if PIG_LATIN
          printf("xnay islay otnay eatergray anthay ynay!\n");
#else
          printf("x is not greater than y!\n");
#endif

     }

}
```

This program uses preprocessor directives to define symbolic constants (such as TRUE, FALSE, and PIG_LATIN), a macro (such as GREATER(a,b)), and conditional compilation (by using the #if statement). When the preprocessor is invoked on this source code, it reads in the stdio.h file and interprets its preprocessor directives, then it replaces all symbolic constants and macros in your program with the corresponding values and code. Next, it evaluates whether PIG_LATIN is set to TRUE and includes either the pig latin text or the plain English text.

If PIG_LATIN is set to FALSE, as in the preceding example, a preprocessed version of the source code would look like this:

```
/* Here is where all the include files
   would be expanded. */

void main(void)
{

     int x, y;

     printf("Please enter the value for x: ");
     scanf("%d", &x);
     printf("Please enter the value for y: ");
     scanf("%d", &y);

     if (((x) > (y) ? (1) : (!1)) == 1)
     {

          printf("x is greater than y!\n");

     }
     else
     {

          printf("x is not greater than y!\n");

     }

}
```

This preprocessed version of the source code can then be passed on to the compiler. If you want to see a preprocessed version of a program, most compilers have a command-line option or a standalone preprocessor program to invoke only the preprocessor and save the preprocessed version of your source code to a file. This capability can sometimes be handy in debugging strange errors with macros and other preprocessor directives, because it shows your source code *after* it has been run through the preprocessor.

Cross Reference:

V.3: How can you avoid including a header more than once?

V.4: Can a file other than a .h file be included with `#include`?

V.12: What is the difference between `#include <file>` and `#include "file"`?

V.22: What is a pragma?

V.23: What is `#line` used for?

V.3: How can you avoid including a header more than once?

Answer:

One easy technique to avoid multiple inclusions of the same header is to use the `#ifndef` and `#define` preprocessor directives. When you create a header for your program, you can `#define` a symbolic name that is unique to that header. You can use the conditional preprocessor directive named `#ifndef` to check whether that symbolic name has already been assigned. If it is assigned, you should not include the header, because it has already been preprocessed. If it is not defined, you should define it to avoid any further inclusions of the header. The following header illustrates this technique:

```
#ifndef _FILENAME_H
#define _FILENAME_H

#define VER_NUM       "1.00.00"
#define REL_DATE      "08/01/94"

#if __WINDOWS__
#define OS_VER            "WINDOWS"
#else
#define OS_VER            "DOS"
#endif

#endif
```

When the preprocessor encounters this header, it first checks to see whether `_FILENAME_H` has been defined. If it hasn't been defined, the header has not been included yet, and the `_FILENAME_H` symbolic name is defined. Then, the rest of the header is parsed until the last `#endif` is encountered, signaling the end of the conditional `#ifndef _FILENAME_H` statement. Substitute the actual name of the header file for "`FILENAME`" in the preceding example to make it applicable for your programs.

Cross Reference:

V.4: Can a file other than a .h file be included with `#include`?

V.12: What is the difference between `#include <file>` and `#include "file"`?

V.14: Can include files be nested?

V.15: How many levels deep can include files be nested?

V.4: Can a file other than a .h file be included with *#include*?

Answer:

The preprocessor will include whatever file you specify in your `#include` statement. Therefore, if you have the line

```
#include <macros.inc>
```

in your program, the file macros.inc will be included in your precompiled program.

It is, however, unusual programming practice to put any file that does not have a .h or .hpp extension in an `#include` statement. You should always put a .h extension on any of your C files you are going to include. This method makes it easier for you and others to identify which files are being used for preprocessing purposes. For instance, someone modifying or debugging your program might not know to look at the macros.inc file for macro definitions. That person might try in vain by searching all files with .h extensions and come up empty. If your file had been named macros.h, the search would have included the macros.h file, and the searcher would have been able to see what macros you defined in it.

Cross Reference:

V.3: How can you avoid including a header more than once?

V.12: What is the difference between `#include <file>` and `#include "file"`?

V.14: Can include files be nested?

V.15: How many levels deep can include files be nested?

V.5: What is the benefit of using *#define* to declare a constant?

Answer:

Using the `#define` method of declaring a constant enables you to declare a constant in one place and use it throughout your program. This helps make your programs more maintainable, because you need to maintain only the `#define` statement and not several instances of individual constants throughout your program. For instance, if your program used the value of pi (approximately 3.14159) several times, you might want to declare a constant for pi as follows:

```
#define PI 3.14159
```

This way, if you wanted to expand the precision of pi for more accuracy, you could change it in one place rather than several places. Usually, it is best to put `#define` statements in an include file so that several modules can use the same constant value.

Using the `#define` method of declaring a constant is probably the most familiar way of declaring constants to traditional C programmers. Besides being the most common method of declaring constants, it also takes up the least memory. Constants defined in this manner are simply placed directly into your source code, with no variable space allocated in memory. Unfortunately, this is one reason why most debuggers cannot inspect constants created using the `#define` method.

Constants defined with the `#define` method can also be overridden using the `#undef` preprocessor directive. This means that if a symbol such as `NULL` is not defined the way you would like to see it defined, you can remove the previous definition of `NULL` and instantiate your own custom definition. See FAQ V.31 for a more detailed explanation of how this can be done.

Cross Reference:

V.6: What is the benefit of using `enum` to declare a constant?

V.7: What is the benefit of using an enum rather than a #define constant?

V.31: How do you override a defined macro?

V.6: What is the benefit of using *enum* to declare a constant?

Answer:

Using the `enum` keyword to define a constant can have several benefits. First, constants declared with `enum` are automatically generated by the compiler, thereby relieving the programmer of manually assigning unique values to each constant. Also, constants declared with `enum` tend to be more readable to the programmer, because there is usually an enumerated type identifier associated with the constant's definition.

Additionally, enumerated constants can usually be inspected during a debugging session. This can be an enormous benefit, especially when the alternative is having to manually look up the constant's value in a header file. Unfortunately, using the `enum` method of declaring constants takes up slightly more memory space than using the `#define` method of declaring constants, because a memory location must be set up to store the constant.

Here is an example of an enumerated constant used for tracking errors in your program:

```
enum Error_Code
{
     OUT_OF_MEMORY,
     INSUFFICIENT_DISK_SPACE,
     LOGIC_ERROR,
     FILE_NOT_FOUND
};
```

See FAQ V.7 for a more detailed look at the benefits of using the `enum` method compared with using the `#define` method of declaring constants.

Cross Reference:

V.5: What is the benefit of using `#define` to declare a constant?

V.7: What is the benefit of using an `enum` rather than a `#define` constant?

V.7: What is the benefit of using an *enum* rather than a *#define* constant?

Answer:

The use of an *enumeration constant* (`enum`) has many advantages over using the traditional *symbolic constant* style of `#define`. These advantages include a lower maintenance requirement, improved program readability, and better debugging capability. The first advantage is that enumerated constants are generated automatically by the compiler. Conversely, symbolic constants must be manually assigned values by the programmer. For instance, if you had an enumerated constant type for error codes that could occur in your program, your `enum` definition could look something like this:

```
enum Error_Code
{
     OUT_OF_MEMORY,
     INSUFFICIENT_DISK_SPACE,
     LOGIC_ERROR,
     FILE_NOT_FOUND
};
```

In the preceding example, `OUT_OF_MEMORY` is automatically assigned the value of 0 (zero) by the compiler because it appears first in the definition. The compiler then continues to automatically assign numbers to the enumerated constants, making `INSUFFICIENT_DISK_SPACE` equal to 1, `LOGIC_ERROR` equal to 2, and so on. If you were to approach the same example by using symbolic constants, your code would look something like this:

```
#define OUT_OF_MEMORY                0
#define INSUFFICIENT_DISK_SPACE      1
#define LOGIC_ERROR                  2
#define FILE_NOT_FOUND               3
```

Each of the two methods arrives at the same result: four constants assigned numeric values to represent error codes. Consider the maintenance required, however, if you were to add two constants to represent the error codes `DRIVE_NOT_READY` and `CORRUPT_FILE`. Using the enumeration constant method, you simply would put these two constants anywhere in the `enum` definition. The compiler would generate two *unique* values for these constants. Using the symbolic constant method, you would have to *manually* assign two new numbers to these constants. Additionally, you would want to ensure that the numbers you assign to these constants are unique. Because you don't have to worry about the actual values, defining your constants using the enumerated method is easier than using the symbolic constant method. The enumerated method also helps prevent accidentally reusing the same number for different constants.

Another advantage of using the enumeration constant method is that your programs are more readable and thus can be understood better by others who might have to update your program later. For instance, consider the following piece of code:

```
void copy_file(char* source_file_name, char* dest_file_name)
{
     ...
     Error_Code err;
     ...
```

```
     if (drive_ready() != TRUE)
          err = DRIVE_NOT_READY;
     ...
}
```

Looking at this example, you can derive from the definition of the variable `err` that `err` should be assigned only numbers of the enumerated type `Error_Code`. Hence, if another programmer were to modify or add functionality to this program, the programmer would know from the definition of `Error_Code` what constants are valid for assigning to `err`.

Conversely, if the same example were to be applied using the symbolic constant method, the code would look like this:

```
void copy_file(char* source_file, char* dest_file)
{
     ...
     int err;
     ...
     if (drive_ready() != TRUE)
          err = DRIVE_NOT_READY;
     ...
}
```

Looking at the preceding example, a programmer modifying or adding functionality to the `copy_file()` function would not immediately know what values are valid for assigning to the `err` variable. The programmer would need to search for the `#define DRIVE_NOT_READY` statement and hope that all relevant constants are defined in the same header file. This could make maintenance more difficult than it needs to be and make your programs harder to understand.

NOTE

Simply defining your variable to be of an enumerated type does not ensure that only valid values of that enumerated type will be assigned to that variable. In the preceding example, the compiler will not require that only values found in the enumerated type `Error_Code` be assigned to `err`; it is up to the programmer to ensure that only valid values found in the `Error_Code` type definition are used.

A third advantage to using enumeration constants is that some symbolic debuggers can print the value of an enumeration constant. Conversely, most symbolic debuggers cannot print the value of a symbolic constant. This can be an enormous help in debugging your program, because if your program is stopped at a line that uses an `enum`, you can simply inspect that constant and instantly know its value. On the other hand, because most debuggers cannot print `#define` values, you would most likely have to search for that value by manually looking it up in a header file.

Cross Reference:

V.5: What is the benefit of using `#define` to declare a constant?

V.6: What is the benefit of using `enum` to declare a constant?

V.8: How are portions of a program disabled in demo versions?

Answer:

If you are distributing a demo version of your program, the preprocessor can be used to enable or disable portions of your program. The following portion of code shows how this task is accomplished, using the preprocessor directives `#if` and `#endif`:

```
int save_document(char* doc_name)
{

#if DEMO_VERSION
     printf("Sorry! You can't save documents using the DEMO version of
     ➥this program!\n");
     return(0);
#endif

     ...

}
```

When you are compiling the demo version of your program, insert the line `#define DEMO_VERSION` and the preprocessor will include the conditional code that you specified in the `save_document()` function. This action prevents the users of your demo program from saving their documents.

As a better alternative, you could define `DEMO_VERSION` in your compiler options when compiling and avoid having to change the source code for the program.

This technique can be applied to many different situations. For instance, you might be writing a program that will support several operating systems or operating environments. You can create macros such as `WINDOWS_VER`, `UNIX_VER`, and `DOS_VER` that direct the preprocessor as to what code to include in your program depending on what operating system you are compiling for.

Cross Reference:

V.32: How can you check to see whether a symbol is defined?

V.9: When should you use a macro rather than a function?

Answer:

See the answer to FAQ V.10.

Cross Reference:

V.1: What is a macro, and how do you use it?

V.10: Is it better to use a macro or a function?

V.17: How can type-insensitive macros be created?

V.10: Is it better to use a macro or a function?

Answer:

The answer depends on the situation you are writing code for. Macros have the distinct advantage of being more efficient (and faster) than functions, because their corresponding code is inserted directly into your source code at the point where the macro is called. There is no overhead involved in using a macro like there is in placing a call to a function. However, macros are generally small and cannot handle large, complex coding constructs. A function is more suited for this type of situation. Additionally, macros are expanded inline, which means that the code is replicated for each occurrence of a macro. Your code therefore could be somewhat larger when you use macros than if you were to use functions.

Thus, the choice between using a macro and using a function is one of deciding between the tradeoff of faster program speed versus smaller program size. Generally, you should use macros to replace small, repeatable code sections, and you should use functions for larger coding tasks that might require several lines of code.

Cross Reference:

V.1: What is a macro, and how do you use it?

V.17: How can type-insensitive macros be created?

V.11: What is the best way to comment out a section of code that contains comments?

Answer:

Most C compilers offer two ways of putting comments in your program. The first method is to use the `/*` and `*/` symbols to denote the beginning and end of a comment. Everything from the `/*` symbol to the `*/` symbol is considered a comment and is omitted from the compiled version of the program. This method is best for commenting out sections of code that contain many comments. For instance, you can comment out a paragraph containing comments like this:

```
/*
This portion of the program contains
a comment that is several lines long
and is not included in the compiled
version of the program.
*/
```

The other way to put comments in your program is to use the `//` symbol. Everything from the `//` symbol to the end of the current line is omitted from the compiled version of the program. This method is best for one-line comments, because the `//` symbol must be replicated for each line that you want to add a comment to. The preceding example, which contains four lines of comments, would not be a good candidate for this method of commenting, as demonstrated here:

```
// This portion of the program contains
// a comment that is several lines long
// and is not included in the compiled
```

```
// version of the program.
```

You should consider using the `/*` and `*/` method of commenting rather than the `//` method, because the `//` method of commenting is not ANSI compatible. Many older compilers might not support the `//` comments.

Cross Reference:

V.8: How are portions of a program disabled in demo versions?

V.12: What is the difference between *#include <file>* and *#include "file"*?

Answer:

When writing your C program, you can include files in two ways. The first way is to surround the file you want to include with the angled brackets < and >. This method of inclusion tells the preprocessor to look for the file in the predefined default location. This predefined default location is often an `INCLUDE` environment variable that denotes the path to your include files. For instance, given the `INCLUDE` variable

```
INCLUDE=C:\COMPILER\INCLUDE;S:\SOURCE\HEADERS;
```

using the `#include <file>` version of file inclusion, the compiler first checks the C:\COMPILER\INCLUDE directory for the specified file. If the file is not found there, the compiler then checks the S:\SOURCE\HEADERS directory. If the file is still not found, the preprocessor checks the current directory.

The second way to include files is to surround the file you want to include with double quotation marks. This method of inclusion tells the preprocessor to look for the file in the current directory first, then look for it in the predefined locations you have set up. Using the `#include "file"` version of file inclusion and applying it to the preceding example, the preprocessor first checks the current directory for the specified file. If the file is not found in the current directory, the C:\COMPILER\INCLUDE directory is searched. If the file is still not found, the preprocessor checks the S:\SOURCE\HEADERS directory.

The `#include <file>` method of file inclusion is often used to include *standard* headers such as stdio.h or stdlib.h. This is because these headers are rarely (if ever) modified, and they should always be read from your compiler's standard include file directory. The `#include "file"` method of file inclusion is often used to include *nonstandard* header files that you have created for use in your program. This is because these headers are often modified in the current directory, and you will want the preprocessor to use your newly modified version of the header rather than the older, unmodified version.

Cross Reference:

V.3: How can you avoid including a header more than once?

V.4: Can a file other than a .h file be included with `#include`?

V.14: Can include files be nested?

V.15: How many levels deep can include files be nested?

V.13: Can you define which header file to include at compile time?

Answer:

Yes. This can be done by using the `#if`, `#else`, and `#endif` preprocessor directives. For example, certain compilers use different names for header files. One such case is between Borland C++, which uses the header file alloc.h, and Microsoft C++, which uses the header file malloc.h. Both of these headers serve the same purpose, and each contains roughly the same definitions. If, however, you are writing a program that is to support Borland C++ and Microsoft C++, you must define which header to include at compile time. The following example shows how this can be done:

```
#ifdef __BORLANDC__
#include <alloc.h>
#else
#include <malloc.h>
#endif
```

When you compile your program with Borland C++, the `__BORLANDC__` symbolic name is automatically defined by the compiler. You can use this predefined symbolic name to determine whether your program is being compiled with Borland C++. If it is, you must include the alloc.h file rather than the malloc.h file.

Cross Reference:

V.21: How can you tell whether a program was compiled using C versus C++?

V.32: How can you check to see whether a symbol is defined?

V.14: Can include files be nested?

Answer:

Yes. Include files can be nested any number of times. As long as you use precautionary measures (see FAQ V.3), you can avoid including the same file twice.

In the past, nesting header files was seen as bad programming practice, because it complicates the dependency tracking function of the MAKE program and thus slows down compilation. Many of today's popular compilers make up for this difficulty by implementing a concept called *precompiled headers*, in which all headers and associated dependencies are stored in a precompiled state.

Many programmers like to create a custom header file that has `#include` statements for every header needed for each module. This is perfectly acceptable and can help avoid potential problems relating to `#include` files, such as accidentally omitting an `#include` file in a module.

Cross Reference:

V.3: How can you avoid including a header more than once?

V.4: Can a file other than a .h file be included with `#include`?

V.12: What is the difference between `#include <file>` and `#include "file"`?

V.15: How many levels deep can include files be nested?

V.15: How many levels deep can include files be nested?

Answer:

Even though there is no limit to the number of levels of nested include files you can have, your compiler might run out of stack space while trying to include an inordinately high number of files. This number varies according to your hardware configuration and possibly your compiler.

In practice, although nesting include files is perfectly legal, you should avoid getting nest-crazy and purposely implementing a large number of include levels. You should create an include level only where it makes sense, such as creating one include file that has an `#include` statement for each header required by the module you are working with.

Cross Reference:

V.3: How can you avoid including a header more than once?

V.4: Can a file other than a .h file be included with `#include`?

V.12: What is the difference between `#include <file>` and `#include "file"`?

V.14: Can include files be nested?

V.16: What is the concatenation operator?

Answer:

The concatenation operator (##) is used to concatenate (combine) two separate strings into one single string. The concatenation operator is often used in C macros, as the following program demonstrates:

```
#include <stdio.h>

#define SORT(x) sort_function ## x

void main(void);

void main(void)
{

     char* array;
     int   elements, element_size;

     ...

     SORT(3)(array, elements, element_size);

     ...

}
```

In the preceding example, the SORT macro uses the concatenation operator to combine the strings sort_function and whatever is passed in the parameter x. This means that the line

```
SORT(3)(array, elements, element_size);
```

is run through the preprocessor and is translated into the following line:

```
sort_function3(array, elements, element_size);
```

As you can see, the concatenation operator can come in handy when you do not know what function to call until runtime. Using the concatenation operator, you can dynamically construct the name of the function you want to call, as was done with the SORT macro.

Cross Reference:

V.1: What is a macro, and how do you use it?

V.17: How can type-insensitive macros be created?

V.17: How can type-insensitive macros be created?

Answer:

A type-insensitive macro is a macro that performs the same basic operation on different data types. This task can be accomplished by using the concatenation operator to create a call to a type-sensitive function based on the parameter passed to the macro. The following program provides an example:

```
#include <stdio.h>

#define SORT(data_type) sort_ ## data_type

void sort_int(int** i);
void sort_long(long** l);
void sort_float(float** f);
void sort_string(char** s);
void main(void);

void main(void)
{

     int** ip;
     long** lp;
     float** fp;
     char** cp;

     ...

     sort(int)(ip);
     sort(long)(lp);
     sort(float)(fp);
     sort(char)(cp);

     ...

}
```

This program contains four functions to sort four different data types: `int`, `long`, `float`, and `string` (notice that only the function prototypes are included for brevity). A macro named `SORT` was created to take the data type passed to the macro and combine it with the `sort_` string to form a valid function call that is appropriate for the data type being sorted. Thus, the string

```
sort(int)(ip);
```

translates into

```
sort_int(ip);
```

after being run through the preprocessor.

Cross Reference:

V.1: What is a macro, and how do you use it?

V.16: What is the concatenation operator?

V.18: What are the standard predefined macros?

Answer:

The ANSI C standard defines six predefined macros for use in the C language:

Macro Name	*Purpose*
`__LINE__`	Inserts the current source code line number in your code.
`__FILE__`	Inserts the current source code filename in your code.
`__DATE__`	Inserts the current date of compilation in your code.
`__TIME__`	Inserts the current time of compilation in your code.
`__STDC__`	Is set to 1 if you are enforcing strict ANSI C conformity.
`__cplusplus`	Is defined if you are compiling a C++ program.

The `__LINE__` and `__FILE__` symbols are commonly used for debugging purposes (see FAQs V.19 and V.20). The `__DATE__` and `__TIME__` symbols are commonly used to put a time stamp on your compiled program for version tracking purposes (see FAQ V.28). The `__STDC__` symbol is set to 1 only if you are forcing your compiler to conform to strict ANSI C standards (see FAQ V.30). The `__cplusplus` symbol is defined only when you are compiling your program using the C++ compiler (see FAQ V.21).

Cross Reference:

V.1: What is a macro, and how do you use it?

V.24: What is the `__FILE__` preprocessor command?

V.26: What is the `__LINE__` preprocessor command?

V.28: What are the `__DATE__` and `__TIME__` preprocessor commands?

V.19: How can a program be made to print the line number where an error occurs?

Answer:

The ANSI C standard includes a predefined macro named `__LINE__` that can be used to insert the current source code line number in your program. This can be a very valuable macro when it comes to debugging your program and checking for logic errors. For instance, consider the following portion of code:

```
int print_document(char* doc_name, int destination)
{

     switch (destination)
     {
          case TO_FILE:

               print_to_file(doc_name);
               break;

          case TO_SCREEN:

               print_preview(doc_name);
               break;

          case TO_PRINTER:

               print_to_printer(doc_name);
               break;

          default:

               printf("Logic error on line number %d!\n", __LINE__);
               exit(1);

     }

}
```

If the function named `print_document()` is passed an erroneous argument for the `destination` parameter (something other than `TO_FILE`, `TO_SCREEN`, and `TO_PRINTER`), the default case in the `switch` statement traps this logic error and prints the line number in which it occurred. This capability can be a tremendous help when you are trying to debug your program and track down what could be a very bad logic error.

Cross Reference:

V.18: What are the standard predefined macros?

V.20: How can a program be made to print the name of a source file where an error occurs?

V.21: How can you tell whether a program was compiled using C versus C++?

V.28: What are the `__DATE__` and `__TIME__` preprocessor commands?

V.20: How can a program be made to print the name of a source file where an error occurs?

Answer:

The ANSI C standard includes a predefined macro named `__FILE__` that can be used to insert the current source code filename in your program. This macro, like the `__LINE__` macro (explained in FAQ V.19), can be very valuable when it comes to debugging your program and checking for logic errors. For instance, the following code builds on the example for FAQ V.19 by including the filename as well as the line number when logic errors are trapped:

```
int print_document(char* doc_name, int destination)
{

     switch (destination)
     {
          case TO_FILE:

               print_to_file(doc_name);
               break;

          case TO_SCREEN:

               print_preview(doc_name);
               break;

          case TO_PRINTER:

               print_to_printer(doc_name);
               break;

          default:

               printf("Logic error on line number %d in the file %s!\n",
                         __LINE__, __FILE__);
               exit(1);

     }
}
```

Now, any erroneous values for the `destination` parameter can be trapped, and the offending source file and line number can be printed.

Cross Reference:

V.18: What are the standard predefined macros?

V.19: How can a program be made to print the line number where an error occurs?

V.21: How can you tell whether a program was compiled using C versus C++?

V.28: What are the `__DATE__` and `__TIME__` preprocessor commands?

V.21: How can you tell whether a program was compiled using C versus C++?

Answer:

The ANSI standard for the C language defines a symbol named `__cplusplus` that is defined only when you are compiling a C++ program. If you are compiling a C program, the `__cplusplus` symbol is undefined. Therefore, you can check to see whether the C++ compiler has been invoked with the following method:

```
#ifdef __cplusplus                    /* Is __cplusplus defined? */
#define USING_C FALSE                 /* Yes, we are not using C */
#else
#define USING_C TRUE                  /* No, we are using C */
#endif
```

When the preprocessor is invoked, it sets `USING_C` to `FALSE` if the `__cplusplus` symbol is defined. Otherwise, if `__cplusplus` is undefined, it sets `USING_C` to `TRUE`. Later in your program, you can check the value of the `USING_C` constant to determine whether the C++ compiler is being used.

Cross Reference:

V.18: What are the standard predefined macros?

V.19: How can a program be made to print the line number where an error occurs?

V.20: How can a program be made to print the name of a source file where an error occurs?

V.28: What are the `__DATE__` and `__TIME__` preprocessor commands?

V.22: What is a pragma?

Answer:

The `#pragma` preprocessor directive allows each compiler to implement compiler-specific features that can be turned on and off with the `#pragma` statement. For instance, your compiler might support a feature called *loop optimization*. This feature can be invoked as a command-line option or as a `#pragma` directive. To implement this option using the `#pragma` directive, you would put the following line into your code:

```
#pragma loop_opt(on)
```

Conversely, you can turn off loop optimization by inserting the following line into your code:

```
#pragma loop_opt(off)
```

Sometimes you might have a certain function that causes your compiler to produce a warning such as `Parameter xxx is never used in function yyy` or some other warning that you are well aware of but choose to ignore. You can temporarily disable this warning message on some compilers by using a `#pragma` directive to turn off the warning message before the function and use another `#pragma` directive to turn it back on after the function. For instance, consider the following example, in which the function named `insert_record()` generates a warning message that has the unique ID of 100. You can temporarily disable this warning as shown here:

```
#pragma warn -100    /* Turn off the warning message for warning #100 */

int insert_record(REC* r)   /* Body of the function insert_record() */
{

    /* insert_rec() function statements go here... */

}

#pragma warn +100 /* Turn the warning message for warning #100 back on */
```

Check your compiler's documentation for a list of `#pragma` directives. As stated earlier, each compiler's implementation of this feature is different, and what works on one compiler almost certainly won't work on another. Nevertheless, the `#pragma` directives can come in very handy when you're turning on and off some of your compiler's favorite (or most annoying) features.

Cross Reference:

V.2: What will the preprocessor do for a program?

V.23: What is `#line` used for?

V.23: What is *#line* used for?

Answer:

The `#line` preprocessor directive is used to reset the values of the `__LINE__` and `__FILE__` symbols, respectively. This directive is commonly used in fourth-generation languages that generate C language source files. For instance, if you are using a fourth-generation language named "X," the 4GL compiler will generate C source code routines for compilation based on your 4GL source code. If errors are present in your 4GL code, they can be mapped back to your 4GL source code by using the `#line` directive. The 4GL code generator simply inserts a line like this into the generated C source:

```
#line 752, "XSOURCE.X"

void generated_code(void)
{

...

}
```

Now, if an error is detected anywhere in the `generated_code()` function, it can be mapped back to the original 4GL source file named XSOURCE.X. This way, the 4GL compiler can report the 4GL source code line that has the error in it.

When the `#line` directive is used, the `__LINE__` symbol is reset to the first argument after the `#line` keyword (in the preceding example, 752), and the `__FILE__` symbol is reset to the second argument after the `#line` keyword (in the preceding example, `"XSOURCE.X"`). All references hereafter to the `__LINE__` and `__FILE__` symbols will reflect the reset values and not the original values of `__LINE__` and `__FILE__`.

Cross Reference:

V.2: What will the preprocessor do for a program?

V.22: What is a pragma?

V.24: What is the _ _*FILE*_ _ preprocessor command?

Answer:

See the answer to FAQ V.20.

Cross Reference:

V.18: What are the standard predefined macros?

V.19: How can a program be made to print the line number where an error occurs?

V.20: How can a program be made to print the name of a source file where an error occurs?

V.21: How can you tell whether a program was compiled using C versus C++?

V.28: What are the `__DATE__` and `__TIME__` preprocessor commands?

V.25: How can I print the name of the source file in a program?

Answer:

See the answer to FAQ V.20.

Cross Reference:

V.18: What are the standard predefined macros?

V.19: How can a program be made to print the line number where an error occurs?

V.20: How can a program be made to print the name of a source file where an error occurs?

V.21: How can you tell whether a program was compiled using C versus C++?

V.28: What are the `__DATE__` and `__TIME__` preprocessor commands?

V.26: What is the _ _*LINE*_ _ preprocessor command?

Answer:

See the answer to FAQ V.19.

Cross Reference:

V.18: What are the standard predefined macros?

V.19: How can a program be made to print the line number where an error occurs?

V.20: How can a program be made to print the name of a source file where an error occurs?

V.21: How can you tell whether a program was compiled using C versus C++?

V.28: What are the `__DATE__` and `__TIME__` preprocessor commands?

V.27: How can I print the current line number of the source file in a program?

Answer:

See the answer to FAQ V.19.

Cross Reference:

V.18: What are the standard predefined macros?

V.19: How can a program be made to print the line number where an error occurs?

V.20: How can a program be made to print the name of a source file where an error occurs?

V.21: How can you tell whether a program was compiled using C versus C++?

V.28: What are the `__DATE__` and `__TIME__` preprocessor commands?

V.28: What are the *__DATE__* and *__TIME__* preprocessor commands?

Answer:

The `__DATE__` macro is used to insert the current compilation date in the form "mm dd yyyy" into your program. Similarly, the `__TIME__` macro is used to insert the current compilation time in the form "hh:mm:ss" into your program. This date-and-time-stamp feature should not be confused with the *current* system date and time. Rather, these two macros enable you to keep track of the date and time your program was last *compiled*. This feature can come in very handy when you are trying to track different versions of your program. For instance, many programmers like to put a function in their programs that gives compilation information as to when the current module was compiled. This task can be performed as shown here:

```
#include <stdio.h>

void main(void);
void print_version_info(void);

void main(void)
{
```

```
    print_version_info();

}

void print_version_info(void)
{

    printf("Date Compiled: %s\n", __DATE__);
    printf("Time Compiled: %s\n", __TIME__);

}
```

In this example, the function `print_version_info()` is used to show the date and time stamp of the last time this module was compiled.

Cross Reference:

V.18: What are the standard predefined macros?

V.19: How can a program be made to print the line number where an error occurs?

V.20: How can a program be made to print the name of a source file where an error occurs?

V.21: How can you tell whether a program was compiled using C versus C++?

V.29: How can I print the compile date and time in a program?

Answer:

See the answer to FAQ V.28.

Cross Reference:

V.18: What are the standard predefined macros?

V.19: How can a program be made to print the line number where an error occurs?

V.20: How can a program be made to print the name of a source file where an error occurs?

V.21: How can you tell whether a program was compiled using C versus C++?

V.28: What are the `__DATE__` and `__TIME__` preprocessor commands?

V.30: How can you be sure that a program follows the ANSI C standard?

Answer:

The ANSI C standard provides a predefined symbol named `__STDC__` that is set to 1 when the compiler is enforcing strict ANSI standard conformance. If you want your programs to be 100 percent ANSI conformant, you should ensure that the `__STDC__` symbol is defined. If the program is being compiled with

non-ANSI options, the __STDC__ symbol is undefined. The following code segment shows how this symbol can be checked:

```
...

#ifdef __STDC__
     printf("Congratulations! You are conforming perfectly to the ANSI
     ➥standards!\n");
#else
     printf("Shame on you, you nonconformist anti-ANSI rabble-rousing
     ➥programmer!\n");
#endif

...
```

Cross Reference:

V.1: What is a macro?

V.24: What is the __FILE__ preprocessor command?

V.26: What is the __LINE__ preprocessor command?

V.28: What are the __DATE__ and __TIME__ preprocessor commands?

V.31: How do you override a defined macro?

Answer:

You can use the #undef preprocessor directive to undefine (override) a previously defined macro. Many programmers like to ensure that their applications are using their own terms when defining symbols such as TRUE and FALSE. Your program can check to see whether these symbols have been defined already, and if they have, you can override them with your own definitions of TRUE and FALSE. The following portion of code shows how this task can be accomplished:

```
...

#ifdef TRUE             /* Check to see if TRUE has been defined yet */
#undef TRUE             /* If so, undefine it */
#endif

#define TRUE 1          /* Define TRUE the way we want it defined */

#ifdef FALSE            /* Check to see if FALSE has been defined yet */
#undef FALSE            /* If so, undefine it */
#endif

#define FALSE !TRUE     /* Define FALSE the way we want it defined */

...
```

In the preceding example, the symbols TRUE and FALSE are checked to see whether they have been defined yet. If so, they are undefined, or overridden, using the #undef preprocessor directive, and they are redefined in the desired manner. If you were to eliminate the #undef statements in the preceding example, the compiler

would warn you that you have multiple definitions of the same symbol. By using this technique, you can avoid this warning and ensure that your programs are using valid symbol definitions.

Cross Reference:

V.1: What is a macro, and how do you use it?

V.10: Is it better to use a macro or a function?

V.16: What is the concatenation operator?

V.17: How can type-insensitive macros be created?

V.18: What are the standard predefined macros?

V.31: How do you override a defined macro?

V.32: How can you check to see whether a symbol is defined?

Answer:

You can use the `#ifdef` and `#ifndef` preprocessor directives to check whether a symbol has been defined (`#ifdef`) or whether it has not been defined (`#ifndef`). Many programmers like to ensure that their own version of `NULL` is defined, not someone else's. This task can be accomplished as shown here:

```
#ifdef NULL
#undef NULL
#endif

#define NULL (void*) 0
```

The first line, `#ifdef NULL`, checks to see whether the `NULL` symbol has been defined. If so, it is undefined using `#undef NULL` (see FAQ V.31), and the new definition of `NULL` is defined.

To check whether a symbol has not been defined yet, you would use the `#ifndef` preprocessor directive. See FAQ V.3 for an example of how you can use `#ifndef` to determine whether you have already included a particular header file in your program.

Cross Reference:

V.3: How can you avoid including a header more than once?

V.8: How are portions of a program disabled in demo versions?

V.33: What common macros are available?

Answer:

See the answer to FAQ V.18.

Cross Reference:

V.1: What is a macro, and how do you use it?

V.18: What are the standard predefined macros?

V.24: What is the `__FILE__` preprocessor command?

V.26: What is the `__LINE__` preprocessor command?

V.28: What are the `__DATE__` and `__TIME__` preprocessor commands?