SESSION

26

C++ Preprocessor

Session Checklist

- ✔ Assigning names to commonly used constants
- ✔ Defining compile-time macros
- ✔ Controlling the compilation process

30 Min. To Go

The programs in Session 25 used the `#include` preprocessor directive to include the definition of classes in the multiple source files that made up our programs. In fact, all of the programs that you have seen so far have included the stdio.h and iostream.h file to define the functions that make up the standard C++ library. This session examines the `#include` directive along with other preprocessor commands.

The C++ Preprocessor

As a C++ programmer, you and I click the Build command to instruct the C++ compiler to convert our source code into an executable program. We don't typically worry about the details of how the compiler works. In Session 25, you learned that the build process consists of two parts, a compile step that converts each of our .cpp files into machine-language object code and a separate link step that combines these object files with those of the standard C++ library to create an .exe executable file. What still isn't clear is that the compile step itself is divided into multiple phases.

The compiler operates on your C++ source file in multiple passes. Generally, a first pass finds and identifies each of the variables and class definitions, while a subsequent pass generates the object code. However, a given C++ compiler makes as many or as few passes as it needs — there is no C++ standard.

Even before the first compiler pass, however, the C++ preprocessor gets a chance. The C++ processor scans through the .cpp file looking for lines that begin with a pound (#) sign in the first column. The output from the preprocessor, itself a C++ program, is fed to the compiler for subsequent processing.

Note

The C language uses the same preprocessor so that anything said here about the C++ preprocessor is also true of C.

The #include Directive

The `#include` directive includes the contents of the named file at the point of the insertion. The preprocessor makes no attempt to process the contents of the .h file.

Tip

The include file does not have to end in .h, but it can confuse both the programmer and the preprocessor if it does not.

Note

The name following the `#include` command must appear within either quotes (" ") or angle brackets (< >). The preprocessor assumes that files contained in quotes are user defined and, therefore, appear in the current directory. The preprocessor searches for files contained in angle brackets in the C++ compiler directories.

The include file should not include any C++ functions because they will be expanded and compiled separately by the modules that include the file. The contents of the include file should be limited to class definitions, global variable definitions, and other preprocessor directives.

20 Min. To Go

The #define Directive

The #define directive defines a constant or macro. The following example shows how the #define is used to define a constant.

```
#define MAX_NAME_LENGTH 256

void fn(char* pszSourceName)
{
    char szLastName[MAX_NAME_LENGTH];

    if (strlen(pszSourceName) >= MAX_NAME_LENGTH)
    {
        // ...source string too long error processing...
    }

    // ...and so it goes...
}
```

The preprocessor directive defines a parameter MAX_NAME_LENGTH to be replaced at compile time by the constant value 256. The preprocessor replaces the name MAX_NAME_LENGTH with the constant 256 everywhere that it is used. Where we see MAX_NAME_LENGTH, the C++ compiler sees 256.

Tip

This example demonstrates the naming convention for #define constants. Names are all uppercase with underscores used to divide words.

When used this way, the #define directive enables the programmer to assign meaningful names to constant values; MAX_NAME_LENGTH has greater meaning to the programmer than 256. Defining constants in this fashion also makes programs easier to modify. For example, the maximum number characters in a name might be embedded throughout a program. However, changing this maximum name length from 256 characters to 128 is simply a matter of modifying the #define no matter how many places it is used.

Defining macros

The `#define` directive also enables definitions macros — a compile-time directive that contains arguments. The following demonstrates the definition and use of the macro `square()`, which generates the code necessary to calculate the square of its argument.

```
#define square(x) x * x
void fn()
{
    int nSquareOfTwo = square(2);
    // ...and so forth...
}
```

The preprocess turns this into the following:

```
void fn()
{
    int nSquareOfTwo = 2 * 2;
    // ...and so forth...
}
```

Common errors using macros

The programmer must be very careful when using `#define` macros. For example, the following does not generate the expected results:

```
#define square(x) x * x
void fn()
{
    int nSquareOfTwo = square(1 + 1);
}
```

The preprocessor expands the macro into the following:

```
void fn()
{
    int nSquareOfTwo = 1 + 1 * 1 + 1;
}
```

Because multiplication takes precedence over addition, the expression is interpreted as if it had been written as follows:

```
void fn()
{
    int nSquareOfTwo = 1 + (1 * 1) + 1;
}
```

The resulting value of `nSquareOfTwo` is 3 and not 4.

Fully qualifying the macro using a liberal dosage of parentheses helps because parentheses control the order of evaluation. There would not have been a problem had `square` been defined as follows:

```
#define square(x) ((x) * (x))
```

However, even this does not solve the problem in every case. For example, the following cannot be made to work:

```
#define square(x) ((x) * (x))
void fn()
{
    int nV1 = 2;
    int nV2;
    nV2 = square(nV1++);
}
```

You might expect the resulting value of `nV2` to be 4 rather than 6 and of `nV1` to be 3 rather than 4 caused by the following macro expansion:

```
void fn()
{
    int nV1 = 2;
    int nV2;
    nV2 = nV1++ * nV1++;
}
```

Macros are not type safe. This can cause confusion in mixed-mode expressions such as the following:

```
#define square(x) ((x) * (x))
void fn()
{
    int nSquareOfTwo = square(2.5);
}
```

Because nSquareOfTwo is an int, you might expect the resulting value to be 4 rather than the actual value of 6 (2.5 * 2.5 = 6.25).

C++ inline functions avoid the problems of macros:

```
inline int square(int x) {return x * x}
void fn()
{
    int nV1 = square(1 + 1); // value is two
    int nV2;
    nV2 = square(nV1++)   // value of nV2 is 4, nV1 is 3
    int nV3 = square(2.5) // value of nV3 is 4
}
```

The inline version of square() does not generate any more code than macros nor does it suffer from the traps and pitfalls of the preprocessor version.

10 Min. To Go

Compile Controls

The preprocessor also provides compile-time decision-making capabilities.

The #if directive

The most C++-like of the preprocessor control directives is the #if statement. If the constant expression following an #if is nonzero, any statements up to an #else are passed on to the compiler. If the constant expression is zero, the statements between the #else and an #endif are passed through. The #else clause is optional. For example, the following:

```
#define SOME_VALUE 1
#if SOME_VALUE
int n = 1;
#else
int n = 2;
#endif
```

is converted to

```
int n = 1;
```

A few operators are defined for the preprocessor. For example:

```
#define SOME_VALUE 1
#if SOME_VALUE - 1
int n = 1;
#else
int n = 2;
#endif
```

is converted to the following:

```
int n = 2;
```

Tip

Remember that these are compile-time decisions and not run-time decisions. The expressions following the `#if` involve constants and `#define` directives—variables and function calls are not allowed.

The #ifdef directive

Another preprocessor control directive is the `#ifdef`. The `#ifdef` is true if the constant next to it is defined. Thus, the following:

```
#define SOME_VALUE 1
#ifdef SOME_VALUE
int n = 1;
#else
int n = 2;
#endif
```

is converted to the following:

```
int n = 1;
```

However, the directive

```
#ifdef SOME_VALUE
int n = 1;
#else
int n = 2;
#endif
```

is converted to the following:

```
int n = 2;
```

The `#ifndef` is also defined with the exact opposite definition.

Using the #ifdef/#ifndef directives for inclusion control

The most common use for the `#ifdef` is inclusion control. A symbol cannot be defined twice. The following is illegal:

```
class MyClass
{
    int n;
};
class MyClass
{
    int n;
};
```

If `MyClass` is defined in the include file myclass.h, it would be erroneous to include that file twice in the same .cpp source file. You might think that this problem is easily avoided; however, it is not uncommon for one include file to include another include file, such as in the following example:

```
#include “myclass.h”
class mySpecialClass : public MyClass
{
    int m;
}
```

An unsuspecting programmer might easily include both mycl ass.h and myspecial-class.h in the same source file leading to a doubly defined compiler error.

```
// the following does not compile
#include “myclass.h”
#include “myspecialclass.h”
void fn(MyClass& mc)
{
    // ...might be an object of class MyClass or MySpecialClass
}
```

This particular example is easily fixed; however, in a large application the relationships between the numerous include files can be bewildering.

Judicious use of the #ifdef directive avoids this problem by defining the myclass.h include file as follows:

```
#ifndef MYCLASS_H
#define MYCLASS_H
class MyClass
{
    int n;
};
#endif
```

When myclass.h is included the first time, MYCLASS_H is not defined and #ifndef is true. However, the constant MYCLASS_H is defined within myclass.h. The next time that myclass.h is encountered during compilation MYCLASS_H is defined and the class definition is avoided.

Using the #ifdef to include debug code

Another common use for the #ifdef clause is compile-time inclusion of debug code. Consider, for example, the following debug function:

```
void dumpState(MySpecialClass& msc)
{
    cout << "MySpecialClass:"
        << "m = " << msc.m
        << "n = " << msc.n;
}
```

Each time this function is called, dumpState() prints the contents of the MySpecialClass object to standard output. I can insert calls throughout my program to keep track of the MySpecialClass objects. After the program is ready for release, I need to remove each of these calls. Not only is this tiresome, but it runs the risk of reintroducing errors to the system. In addition, I might be forced to reinsert these calls in the program should I need to debug the program again.

I could define some type of flag that controls whether the program outputs the state of the program's MySpecialClass objects, but the calls themselves introduce overhead that slows the function. A better approach is the following:

```
#ifdef DEBUG
void dumpState(MySpecialClass& msc)
{
```

```
    cout << "MySpecialClass:"
        << "m = " << msc.m
        << "n = " << msc.n;
}
#else
inline dumpState(MySpecialClass& mc)
{
}
#endif
```

If the parameter `DEBUG` is defined, then the debug function `dumpState()` is compiled. If `DEBUG` is not defined, then a do-nothing inline version of `dumpState()` is compiled instead. The C++ compiler converts each call to this inline version to nothing.

If the inline version of the function doesn't work, perhaps because the compiler does not support inline functions, then use the following macro definition:

```
#define dumpState(x)
```

Both Visual C++ and GNU C++ support this approach. Constants may be defined from the project settings without adding a `#define` directive to the source code. In fact, `_DEBUG` is automatically defined when compiling in debug mode, the constant.

REVIEW

The most common use for the preprocessor is to include the same class definitions and function prototypes in multiple .cpp source files using the `#include` directive. The `#if` and `#ifdef` preprocessor directives enable control over which lines of code are compiled and which are not.

- The name of the file included via the `#include` directive should end in .h. To do otherwise confuses other programmers and might even confuse the compiler. Filenames included in quotes (" ") are assumed to be in the current (or some other user specified) directory, whereas filenames included in brackets (< >) are assumed to reference one of the standard C++-defined include files.
- If the constant expression after an `#if` directive is nonzero, then the C++ commands that follow are passed to the compiler; otherwise, they are not passed. The `#ifdef x` directive is true if the `#define` constant `x` is defined.

- All preprocessor directives control which C++ statements the compiler "sees." All are interpreted at compile time and not at execution time.

QUIZ YOURSELF

1. What is the difference between `#include "file.h"` and `#include <file.h>`? (See "The #include Directive.")
2. What are the two types of `#define` directive? (See "The #define Directive.")
3. Given the macro definition `#define square(x) x * x`, what is the value of square(2 + 3)? (See "The #define Directive.")
4. Name a benefit of an inline function compared to the equivalent macro definition. (See "Common Errors Using Macros.")
5. What is a common use for the `#ifndef` directive? (See "Using the #ifdef/#ifndef Directives for Inclusion Control.")