

MODULE 10 PREPROCESSOR DIRECTIVES

My Training Period: hours

Abilities

- ' Able to understand and use `#include`.
- ' Able to understand and use `#define`.
- ' Able to understand and use macros and inline functions.
- ' Able to understand and use the conditional compilation – `#if`, `#endif`, `#ifdef`, `#else`, `#ifndef` and `#undef`.
- ' Able to understand and use `#error`, `#pragma`, `#` and `##` operators and `#line`.
- ' Able to display error messages during conditional compilation.
- ' Able to understand and use assertions.

10.1 Introduction

- For C/C++ preprocessor, preprocessing occurs before a program is compiled. A complete process involved during the preprocessing, compiling and linking can be read in Module W.
- Some possible actions are:
 - ' Inclusion of other files in the file being compiled.
 - ' Definition of symbolic constants and macros.
 - ' Conditional compilation of program code or code segment.
 - ' Conditional execution of preprocessor directives.
- All preprocessor directives begin with `#`, and only white space characters may appear before a preprocessor directive on a line.

10.2 The `#include` Preprocessor Directive

- The `#include` directive causes copy of a specified file to be included in place of the directive. The two forms of the `#include` directive are:

```
//searches for header files and replaces this directive
//with the entire contents of the header file here
#include <header_file>
```

- Or

```
#include "header_file"
```

```
e.g.     #include <stdio.h>
         #include "myheader.h"
```

- If the file name is enclosed in **double quotes**, the preprocessor searches in the same directory (local) as the source file being compiled for the file to be included, if not found then looks in the subdirectory associated with standard header files as specified using angle bracket.
- This method is normally used to include user or programmer-defined header files.
- If the file name is enclosed in **angle brackets** (< and >), it is used for standard library header files, the search is performed in an implementation dependent manner, normally through designated directories such as `C:\BC5\INCLUDE` for Borland C++ (default installation) or directories set in the programming (compiler) environment, project or configuration. You have to check your compiler documentation. Compilers normally put the standard header files under the `INCLUDE` directory or subdirectory.
- The `#include` directive is normally used to include standard library such as `stdio.h` and `iostream` or user defined header files. It also used with programs consisting of several source files that are to be compiled together. These files should have common declaration, such as functions, classes etc, that many different source files depend on those common declarations.
- A header file containing declarations common to the separate program files is often created and included in the file using this directive. Examples of such common declarations are structure

(`struct`) and union (`union`) declarations, enumerations (`enum`), classes, function prototypes, types etc.

- Other variation used in UNIX system is by providing the relative path as follows:

```
#include "/usr/local/include/test.h"
```

- This means search for file in the indicated directory, if not found then look in the subdirectory associated with the standard header file.

```
#include "sys/test1.h"
```

- This means, search for this file in the `sys` subdirectory under the subdirectory associated with the standard header file.
- Remember that from Module 9, `.` (dot) means current directory and `..` (dot dot) means parent directory.

10.3 The `#define` Preprocessor Directive: Symbolic Constants

- The `#define` directive creates symbolic constants, constants that represented as symbols and macros (operations defined as symbols). The format is as follows:

```
#define identifier replacement-text
```

- When this line appears in a file, all subsequent occurrences of `identifier` will be replaced by the `replacement-text` automatically before the program is compiled. For example:

```
#define PI 3.14159
```

- Replaces all subsequent occurrences of the symbolic constant `PI` with the numeric constant `3.14159`. `const` type qualifier also can be used to declare numeric constant that will be discussed in another Module.
- Symbolic constants enable the programmer to create a **name** for a constant and use the name throughout the program, the advantage is, it only need to be modified once in the `#define` directive, and when the program is recompiled, all occurrences of the constant in the program will be modified automatically, making writing the source code easier in big programs.
- That means everything, to the right of the symbolic constant name replaces the symbolic constant.
- Other `#define` examples include the stringizing as shown below:

```
#define STR "This is a simple string"
#define NIL ""
#define GETSTDLIB #include <stdlib.h>
#define HEADER "myheader.h"
```

10.4 The `#define` Preprocessor Directive: Macros

- A macro is an operation defined in `#define` preprocessor directive.
- As with symbolic constants, the macro-identifier is replaced in the program with the `replacement-text` before the program is compiled. Macros may be defined with or without arguments.
- A macro without arguments is processed like a symbolic constant while a macro with arguments, the arguments are substituted in the replacement text, then the macro is expanded, that is the `replacement-text` replaces the identifier and argument list in the program.
- Consider the following macro definition with one argument for an area of a circle:

```
#define CIR(x) PI*(x)*(x)
```

- Wherever `CIR_AREA(x)` appears in the file, the value of `x` is substituted for `x` in the replacement text, the symbolic constant `PI` is replaced by its value (defined previously), and the macro is expanded in the program. For example, the following statement:

```
area = CIR_AREA(4);
```

- Is expanded to

```
area = 3.14159*(4)*(4);
```

- Since the expression consists only of constants, at compile time, the value of the expression is evaluated and assigned to variable `area`.
- The parentheses around each `x` in the replacement text, force the proper order of evaluation when the macro argument is an expression. For example, the following statement:

```
area = CIR_AREA(y + 2);
```

- Is expanded to:

```
area = 3.14159*(y + 2)*(y + 2);
```

- This evaluates correctly because the parentheses force the proper order of evaluation. If the parentheses are omitted, the macro expression is:

```
area = 3.14159*y+2*y+2;
```

- Which evaluates incorrectly (following the operator precedence rules) as:

```
area = (3.14159 * y) + (2 * y) + 2;
```

- Because of the operator precedence rules, you have to be careful about this.
- Macro `CIR_AREA` could be defined as a function. Let say, name it a `circleArea`:

```
double circleArea(double x)
{
    return (3.14159*x*x);
}
```

- Performs the same calculation as macro `CIR_AREA`, but here the overhead of a function call is associated with `circleArea` function.
- The advantages of macro `CIR_AREA` are that macros insert code directly in the program, avoiding function overhead, and the program remains readable because the `CIR_AREA` calculation is defined separately and named meaningfully. The disadvantage is that its argument is evaluated twice.
- Another better alternative is using the **inline function**, by adding the `inline` keyword.

```
inline double circleArea(double x)
{
    return (3.14159 * x * x);
}
```

- The following is a macro definition with 2 arguments for the area of a rectangle:

```
#define RECTANGLE_AREA(p, q) (p)*(q)
```

- Wherever `RECTANGLE_AREA(p, q)` appears in the program, the values of `p` and `q` are substituted in the macro replacement text, and the macro is expanded in place of the macro name. For example, the statement:

```
rectArea = RECTANGLE_AREA(a+4, b+7);
```

- Will be expanded to:

```
rectArea = (a+4)*(b+7);
```

- The value of the expression is evaluated and assigned to variable `rectArea`.
- If the replacement text for a macro or symbolic constant is longer than the remainder of the line, a backslash (`\`) must be placed at the end of the line indicating that the replacement text continues on the next line. For example:

```
#define RECTANGLE_AREA(p, q) \
```

(p)*(q)

- Symbolic constants and macros can be discarded by using the `#undef` preprocessor directive. Directive `#undef` un-defines a symbolic constant or macro name.
- The scope of a symbolic constant or macro is from its definition until it is undefined with `#undef`, or until the end of the file. Once undefined, a name can be redefined with `#define`.
- Functions in the standard library sometimes are defined as macros based on other library functions. For example, a macro commonly defined in the `stdio.h` header file is:

```
#define getchar() getc(stdin)
```

- The macro definition of `getchar()` uses function `getc()` to get one character from the standard input stream. `putchar()` function of the `stdio.h` header, and the character handling functions of the `ctype.h` header implemented as macros as well.
- A program example.

```
#include <iostream.h>
#include <stdlib.h>
#define THREETIMES(x) (x)*(x)*(x)
#define CIRAREA(y) (PI)*(y)*(y)
#define REC(z, a) (z)*(a)
#define PI 3.14159

int main(void)
{
    float p = 2.5;
    float r = 3.5, s, t, u = 1.5, v = 2.5;

    cout<<"Power to three of "<<p<<" is "<<THREETIMES(p)<<endl;
    cout<<"Circle circumference = 2*PI*r = "<<(2*PI*r)<<endl;

    s = CIRAREA(r+p);
    cout<<"Circle area = PI*r*r = "<<s<<endl;

    t = REC(u, v);
    cout<<"Rectangle area = u*v = "<<t<<endl;
    system("pause");
    return 0;
}
```

Output :

- In another Module you we will discuss the `inline` function that is a better construct used in C++ compared to macros.

10.5 Conditional Compilation

- Enable the programmer to control the execution of preprocessor directives, and the compilation of program code.
- Each of the conditional preprocessor directives evaluates a constant integer expression. Cast expressions, `sizeof()` expressions, and enumeration constants cannot be evaluated in preprocessor directives.
- The conditional preprocessor construct is much like the `if` selection structure. Consider the following preprocessor code:

```
#if !defined(NULL)
    #define NULL 0
#endif
```

- These directives determine whether the NULL is defined or not. The expression `defined(NULL)` evaluates to 1 if NULL is defined; 0 otherwise. If the result is 0, `!defined(NULL)` evaluates to 1, and NULL is defined.
- Otherwise, the `#define` directive is skipped. Every `#if` construct ends with `#endif`. Directive `#ifdef` and `#ifndef` are shorthand for `#if defined(name)` and `#if !defined(name)`.
- A multiple-part conditional preprocessor construct may be tested using the `#elif` (the equivalent of `else if` in an `if` structure) and the `#else` (the equivalent of `else` in an `if` structure) directives.
- During program development, programmers often find it helpful to comment out large portions of code to prevent it from being compiled but if the code contains comments, `/*` and `*/` or `//`, they cannot be used to accomplish this task.
- Instead, the programmer can use the following preprocessor construct:

```
#if 0
    code prevented from compiling...
#endif
```

- To enable the code to be compiled, the 0 in the preceding construct is replaced by 1.
- Conditional compilation is commonly used as a debugging aid.
- Another example shown below: instead using the `printf()` statements directly to print variable values and to confirm the flow of control, these `printf()` statements can be enclosed in conditional preprocessor directives so that the statements are only compiled while the debugging process is not completed.

```
#ifdef DEBUG
    printf("Variable x = %d\n", x);
#endif
```

- The code causes a `printf()` statement to be compiled in the program if the symbolic constant `DEBUG` has been defined (`#defined DEBUG`) before directive `#ifdef DEBUG`.
- When debugging is completed, the `#define` directive is removed from the source file, and the `printf()` statements inserted for debugging purpose are ignored during compilation. In larger programs, it may be desirable to define several different symbolic constants that control the conditional compilation in separate sections of the source file.
- A program example.

```
#define Module10
#define MyVersion 1.1
#include <iostream.h>
#include <stdlib.h>

int main(void)
{
    cout<<"Sample using #define, #ifdef, #ifndef\n";
    cout<<" #undef, #else and #endif...\n";
    cout<<"-----\n";
    #ifdef Module10
        cout<<"\nModule10 is defined.\n";
        #else
            cout<<"\nModule10 is not defined.\n";
        #endif

    #ifndef MyVersion
        cout<<"\nMyVersion is not defined\n";
        #else
            cout<<"\nMyVersion is "<<MyVersion<<endl;
        #endif

    #ifdef MyRevision
        cout<<"\nMy Revision is defined\n"<<endl;
        #else
            cout<<"\nMyRevision is not defined!\n"<<endl;
        #endif

    #undef MyVersion
    #ifndef MyVersion
        cout<<"MyVersion is not defined\n"<<endl;
        #else
            cout<<"MyVersion is "<<MyVersion<<endl;
        #endif
}
```

```

#endif
system("pause");
return 0;
}

```

Output:

- If you check the header file definition, the conditional compilation directives heavily used as guard macro to guard the header files against multiple inclusion or filename redefinition.
- For example, create the following header file named `boolean.h` and save it in the same folder as your `main()` source file that follows. Do not compile and run.

```

//test the boolean.h header file from
//multiple inclusion or re definition
#ifndef BOOLEAN_H
    //means literal string 'int' is same as 'boolean'

const boolean FALSE = 0;
const boolean TRUE =1;

#define BOOLEAN_H
#endif

```

- This file defines the type and constants for the boolean logic if `boolean.h` has not been defined.
- Then create the following program, compile and run.

```

//Program using the user defined
//header file, boolean.h
#include <iostream.h>
#include <stdlib.h>
//notice this...
#include "boolean.h"

int main(void)
{
    //new type stored in boolean.h...

    HappyTime = TRUE;

    //if TRUE = 1, do...
    if(
        )
        cout<<"I'm happy today lor!!!"<<endl;
    //else, FALSE = 0, do...
    else
        cout<<"What a bad day...today!!!"<<endl;
    system("pause");
    return 0;
}

```

Output:

- Let say we want to define a vehicle class in header file named `vehicle.h`. By using the conditional directives, we can avoid the multiple inclusion of this file when there are multiple `#include <vehicle.h>` directives in multiple files in the same program.

```
#ifndef VEHICLE_H
#define VEHICLE_H
//The file is compiled only when VEHICLE_H is not defined.
//The first time the file is included using #include <vehicle.h>,
//the macro is not defined, then it is immediately defined.
//Next time, if the same inclusion of the vehicle.h or
//by other source file with the same vehicle.h, needs to be
//included in the same program, the macro and conditional directives
//ensure that the file inclusion is skipped...

class vehicle{...};

#endif
//end of the vehicle.h
```

- The usage of the multiple inclusions of similar header files is discussed in Module 14.

10.6 The `#error` And `#pragma` Preprocessor Directive

- The `#error` directive

```
#error tokens
```

- Prints an implementation-dependent message including the tokens specified in the directive.
- The tokens are sequences of characters separated by spaces. For example the following statement contains 6 tokens.

- In Borland C++, when a `#error` directive is processed, the tokens in the directive are displayed as an error message, preprocessing stops, and the program does not compile. For example:

```
#if (MyVAL != 2 && MyVAL != 3)
#error MyVAL must be defined to either 2 or 3
#endif
```

- Then, when you compile the following simple program:

```
//#error directive...
#include <stdio.h>
#include <stdlib.h>

#if MyVAL != 2

#endif

int main()
{
    system("pause");
    return 0;
}
//No output, error message during the
//compilation
```

- The following error message should be generated.

- Then correct the error by defining MyVal to 2 as the following program and when you rebuild, it should be OK.

```
//#error directive...
#include <stdio.h>
#include <stdlib.h>

#define
#if MyVAL != 2
#error MyVAL must be defined to 2
#endif

int main()
{
    system("pause");
    return 0;
}
//No output
```

- Another simple program example.

```
//#error directive...
#include <stdio.h>
#include <stdlib.h>

#if MyChar != 'X'
#error The MyChar character is not 'X'
#endif

int main()
{
    system("pause");
    return 0;
}
//No output, with error message during
//the compilation
```

- The following error should be generated.

- The #pragma directive,

#pragma tokens

- The tokens are a series of characters that gives a specific compiler instruction and arguments, if any and causes an implementation-defined action. A pragma not recognized by the implementation is ignored without any error or warning message.
- A pragma is a compiler directive that allows you to provide additional information to the compiler. This information can change compilation details that are not otherwise under your control. For more information on #error and #pragma, see the documentation of your compiler.
- Keyword pragma is part of the C++ standard, but the form, content, and meaning of pragma is different for every compiler.
- This means different compilers will have different pragma directives.
- No pragma are defined by the C++ standard. Code that depends on pragma is not portable. It is normally used during debugging process.
- For example, Borland #pragma message has two forms:

```
#pragma message ("text" ["text"["text" ...]])
#pragma message text
```

- It is used to specify a user-defined message within your program code.
- The first form requires that the text consist of one or more string constants, and the message must be enclosed in parentheses.
- The second form uses the text following the #pragma for the text of the warning message. With both forms of the #pragma, any macro references are expanded before the message is displayed.

- The following program example will display compiler version of the Borland C++ if compiled with Borland C++ otherwise will display "This compiler is not Borland C++" message and other predefined macros.

```

//#pragma directive...
#include <stdio.h>
#include <stdlib.h>

//displays either "You are compiling using
//version xxx of BC++" (where xxx is the version number)
//or "This compiler is not Borland C++", date, time
//console or not... by using several related
//predefined macro such as __DATE__ etc

#ifdef __BORLANDC__
#pragma message You are compiling using Borland C++ version __BORLANDC__.
#else
#pragma message ("This compiler is not Borland C++")
#endif
#pragma message time: __TIME__.
#pragma message date: __DATE__.
#pragma message Console: __CONSOLE__.

int main()
{
    system("pause");
    return 0;
}
//No output

```

- The following message should be generated if compiled with Borland C++.

- Program example compiled with VC++ / VC++ .Net.

```

//#pragma directive...
#include <stdio.h>
#include <stdlib.h>

//displays either "You are compiling using
//version xxx of BC++" (where xxx is the version number)
//or "This compiler is not Borland C++", date, time
//console or not... by using several related
//predefined macro such as __DATE__ etc

#ifdef __BORLANDC__
#pragma message You are compiling using Borland C++ version __BORLANDC__.
#else
#pragma message ("This compiler is not Borland C++")
#endif
#pragma message ("time:" __TIMESTAMP__)
#pragma message ("date:" __DATE__)
#pragma message ("file:" __FILE__)

int main()
{
    system("pause");
    return 0;
}

```

- Another program example compiled using VC++ / VC++ .Net

```
//#pragma directives...
#include <stdio.h>

#if _M_IX86 != 500
#pragma message("Non Pentium processor build")
#endif

#if _M_IX86 == 600
#pragma message("but Pentium II above processor build")
#endif

#pragma message("Compiling " __FILE__)
#pragma message("Last modified on " __TIMESTAMP__)

int main()
{
    return 0;
}
```

- The following message should be expected.

- So now, you know how to use the #pragmas. For other #pragmas, please check your compiler documentations and also the standard of the ISO/IEC C / C++ for any updates.

10.7 The # And ## Operators

- The # and ## preprocessor operators are available in ANSI C. The # operator causes a replacement text token to be converted to a string surrounded by double quotes as explained before.
- Consider the following macro definition:

```
#define HELLO    printf("Hello, " #x "\n");
```

- When HELLO(John) appears in a program file, it is expanded to:

```
printf("Hello, " "John" "\n");
```

- The string "John" replaces #x in the replacement text. Strings separated by white space are concatenated during preprocessing, so the above statement is equivalent to:

```
printf("Hello, John\n");
```

- Note that the # operator must be used in a macro with arguments because the operand of # refers to an argument of the macro.
- The ## operator concatenates two tokens. Consider the following macro definition,

```
#define CAT(p, q) p ## q
```

- When CAT appears in the program, its arguments are concatenated and used to replace the macro. For example, CAT(O,K) is replaced by OK in the program. The ## operator must have two operands.
- A program example:

```
#include <stdio.h>
#include <stdlib.h>
#define HELLO(x) printf("Hello, " #x "\n");
#define SHOWFUNC(x) Use ## Func ## x

int main(void)
{
    //new concatenated identifier, UseFuncOne
    char * SHOWFUNC(One);
    //new concatenated identifier, UseFuncTwo
    char * SHOWFUNC(Two);

    SHOWFUNC(One) = "New name, UseFuncOne";
    SHOWFUNC(Two) = "New name, UseFuncTwo";

    HELLO(Birch);
    printf("SHOWFUNC(One) -> %s \n",SHOWFUNC(One));
    printf("SHOWFUNC(One) -> %s \n",SHOWFUNC(Two));
    system("pause");
    return 0;
}
```

Output:

10.8 Line Numbers

- The #line preprocessor directive causes the subsequent source code lines to be renumbered starting with the specified constant integer value. The directive:

```
#line 100
```

- Starts line numbering from 100 beginning with the next source code line. A file name can be included in the #line directive. The directive:

```
#line 100 "file123.cpp"
```

- Indicates that lines are numbered from 100 beginning with the next source code line, and that the name of the file for purpose of any compiler messages is "file123.cpp".
- The directive is normally used to help make the messages produced by syntax errors and compiler warnings more meaningful. The line numbers do not appear in the source file.

10.9 Predefined Macros

- There are standard predefined macros as shown in Table 10.1. The identifiers for each of the predefined macros begin and end with two underscores. These identifiers and the defined identifier cannot be used in #define or #undef directive.

- There are a lot more predefined macros extensions that are non standard, compilers specific, please check your compiler documentation. The standard macros are available with the same meanings regardless of the machine or operating system your compiler installed on.

Symbolic Constant	Explanation
<code>__DATE__</code>	The date the source file is compiled (a string of the form "mmm dd yyyy" such as "Jan 19 1999").
<code>__LINE__</code>	The line number of the current source code line (an integer constant).
<code>__FILE__</code>	The presumed names of the source file (a string).
<code>__TIME__</code>	The time the source file is compiled (a string literal of the form <code>:hh:mm:ss</code>).
<code>__STDC__</code>	The integer constant 1. This is intended to indicate that the implementation is ANSI C compliant.

Table 10.1: The predefined macros.

- A program example:

```
#include <iostream.h>
#include <stdlib.h>

int main(void)
{
    cout<<"Let test the free macros, standard and compiler specific..."<<endl;
    cout<<"\nPredefined macro __LINE__ : "<<__LINE__<<endl;
    cout<<"Predefined macro __FILE__ : "<<__FILE__<<endl;
    cout<<"Predefined macro __TIME__ : "<<__TIME__<<endl;
    cout<<"Predefined macro __DATE__ : "<<__DATE__<<endl;
    cout<<"Some compiler specific __MSDOS__ : "<<__MSDOS__<<endl;
    cout<<"Some compiler specific __BORLANDC__ : "<<__BORLANDC__<<endl;
    cout<<"Some compiler specific __BCPLUSPLUS__ : "<<__BCPLUSPLUS__<<endl;

    system("pause");
    return 0;
}
```

Output :

10.10 Assertions

- This is one of the macro used for simple exception handling.
- The `assert()` macro, defined in the `assert.h` header file, tests the value of an expression. If the value of the expression is 0 (false), then `assert` prints an error message and calls function `abort()` (of the general utilities library – `stdlib.h`) to terminate program execution.
- This is a useful debugging tool, for example, testing if a variable has a correct value or not. For example, suppose variable `q` should never be larger than 100 in a program.
- An assertion may be used to test the value of `q` and print an error message if the value of `q` is incorrect. The following statement would be:

```
assert(q <= 100);
```

- If `q` is greater than 100 when the preceding statement is encountered in a program, an error messages containing the line number and file name is printed, and the program terminates.
- The programmer may then concentrate on this area of the code to find the error. If the symbolic constant `NDEBUG` (`#define NDEBUG`) is defined, that is no debugging, subsequent assertions will be ignored.
- Thus, when assertions are no longer needed, the line:

```
#define NDEBUG
```

- Is inserted in the program file rather than deleting each assertion manually.
- A program example:

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

void TestString(char *string);

void main()
{
    //first test array of char, 10 characters...
    //should be OK for the 3 test conditions...
    char test1[] = "abcdefghij";
    //second test pointer to string, 9 characters...
    //should be OK for the 3 test conditions...
    char *test2 = "123456789";
    //third test array char, empty...
    //should fail on the 3rd condition, cannot be empty...

    printf("Testing the string #1 \"%s\"\n", test1);
    TestString(test1);
    printf("Testing the string #2 \"%s\"\n", test2);
    TestString(test2);
    printf("Testing the string #3 \"%s\"\n", test3);
    TestString(test3);
}

void TestString(char * string)
{
    //set the test conditions...
    //string must more than 8 characters...
    assert(strlen(string) > 8);
    //string cannot be NULL
    assert(string != NULL);
    //string cannot be empty...
    //test3 should fail here and program abort...
}

```

Output :

- You can see from the output, `project.cpp` is `__FILE__` and line 34 is `__LINE__` predefined macros defined in `assert.h` file as shown in the source file at the end of this Module.
- For this program example, let try invoking the Borland® C++ Turbo Debugger. The steps are:
- Click Tool menu : Select Turbo Debugger sub menu : Press `Alt + R` (Run menu) : Select `Trace Into` or press `F7` key continuously until program terminate (line by line code execution) : When small program dialog box appears, press `Enter/Return` key (OK) : Finally, press `Alt+F5` to see the output window.

Figure 10.1: Borland Turbo® Debugger window.

- For debugging using Microsoft Visual C++ or .Net read [HERE](#). For Linux using `gdb`, read [HERE](#).
- Another program example.

```
//assert macro and DEBUG, NDEBUG
//NDEBUG will disable assert().
//DEBUG will enable assert().
#define DEBUG
#include <stdlib.h>
#include <iostream.h>
#include <assert.h>

int main()
{
    int    x, y;

    //Tell user if NDEBUG is defined and do assert.
    #if defined(NDEBUG)
    cout<<"NDEBUG is defined. Assert disabled,\n";
    #else
        cout<<"NDEBUG is not defined. Assert enabled.\n";
    #endif

    //prompt user some test data...
    cout<<"Insert two integers: ";
    cin>>x>>y;
    cout<<"Do the assert(x < y)\n";

    //if x < y, it is OK, else this program will terminate...
    assert(x < y);
    if(x<y)
    {
        cout<<"Assertion not invoked because "<<x<<" < "<<y<<endl;
        cout<<"Try key in x > y, assertion will be invoked!"<<endl;
    }
    else
        cout<<"Assertion invoked, program terminated!"<<endl;
    system("pause");
    return 0;
}
```

Normal Output:

Abnormal program termination output:

- If you use Microsoft Visual Studio® 6.0, the output should be more informative with dialog box displayed whether you want to abort, retry or ignore :) as shown below.

- And the output screen tries to tell you something as shown below.

- The following program example compiled using g++. For the gdb debugger, please read Module000.

```
////////testassert.cpp////////  
//DEBUG will enable assert().  
#define DEBUG  
#include <iostream>  
#include <cassert>  
using namespace std;  
  
int main()  
{  
    int    x, y;  
  
    //Tell user if NDEBUG is defined and do assert.  
    #if defined(NDEBUG)
```

```

cout<<"NDEBUG is defined. Assert disabled,\n";
#else
    cout<<"NDEBUG is not defined. Assert enabled.\n";
#endif

//prompt user some test data...
cout<<"Insert two integers: ";
cin>>x>>y;
cout<<"Do the assert(x < y)\n";

//if x < y, it is OK, else this program will terminate...
assert(x < y);
if(x<y)
{
    cout<<"Assertion not invoked because "<<x<<" < "<<y<<endl;
    cout<<"Try key in x > y, assertion will be invoked!"<<endl;
}
else
    cout<<"Assertion invoked, program terminated!"<<endl;
return 0;
}

```

```

[bodo@bakawali ~]$ g++ testassert.cpp -o testassert
[bodo@bakawali ~]$ ./testassert

```

```

NDEBUG is not defined. Assert enabled.
Insert two integers: 30 20
Do the assert(x < y)
testassert: testassert.cpp:24: int main(): Assertion `x < y' failed.
Aborted

```

```

[bodo@bakawali ~]$ ./testassert

```

```

NDEBUG is not defined. Assert enabled.
Insert two integers: 20 30
Do the assert(x < y)
Assertion not invoked because 20 < 30
Try key in x > y, assertion will be invoked!

```

-----Note-----

Program Sample

- The following program sample is the `assert.h` header file. You can see that many preprocessor directives being used here.

```

/* assert.h
   assert macro
*/

/*
 *      C/C++ Run Time Library - Version 8.0
 *
 *      Copyright (c) 1987, 1997 by Borland International
 *      All Rights Reserved.
 */
/* $Revision: 8.1 $ */
#if !defined(__DEFS_H)
#include <_defs.h>
#endif

#if !defined(RC_INVOKED)

#if defined(__STDC__)
#pragma warn -nak
#endif
#endif /* !RC_INVOKED */
#if !defined(__FLAT__)
#ifdef __cplusplus
extern "C" {
#endif

```

```

void _Cdecl _FARFUNC __assertfail( char _FAR *__msg,
                                   char _FAR *__cond,
                                   char _FAR *__file,
                                   int __line);

```

```

#ifdef __cplusplus
}
#endif
#undef assert
#ifdef NDEBUG
# define assert(p) ((void)0)
#else
# if defined(_Windows) && !defined(__DPMI16__)
#   define _ENDL
# else
#   define _ENDL "\n"
# endif
# define assert(p) ((p) ? (void)0 : (void) __assertfail( \
    "Assertion failed: %s, file %s, line %d" _ENDL, \
    #p, __FILE__, __LINE__ ) )
#endif

#else /* defined __FLAT__ */
#ifdef __cplusplus
extern "C" {
#endif

void _RTLENTY _EXPFUNC _assert(char * __cond, char * __file, int __line);

/* Obsolete interface: __msg should be "Assertion failed: %s, file %s, line %d"
*/
void _RTLENTY _EXPFUNC __assertfail(char * __msg, char * __cond,
    char * __file, int __line);

#ifdef __cplusplus
}
#endif
#undef assert

#ifdef NDEBUG
#define assert(p) ((void)0)
#else
#define assert(p) ((p) ? (void)0 : _assert(#p, __FILE__, __LINE__))
#endif

#endif /* __FLAT__ */
#if !defined(RC_INVOKED)
#if defined(__STDC__)
#pragma warn .nak
#endif
#endif

#endif /* !RC_INVOKED */

```

-----oO-----

Further reading and digging:

1. Check the best selling C/C++ books at Amazon.com.