To:
Tenouk

MODULE 10
THE C/C++ PREPROCESSOR DIRECTIVES 1

My Training Period: hours

C/C++ preprocessor directives abilities that should be acquired:

- 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.

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 C/C++ - Compile, link and running programs.
- 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

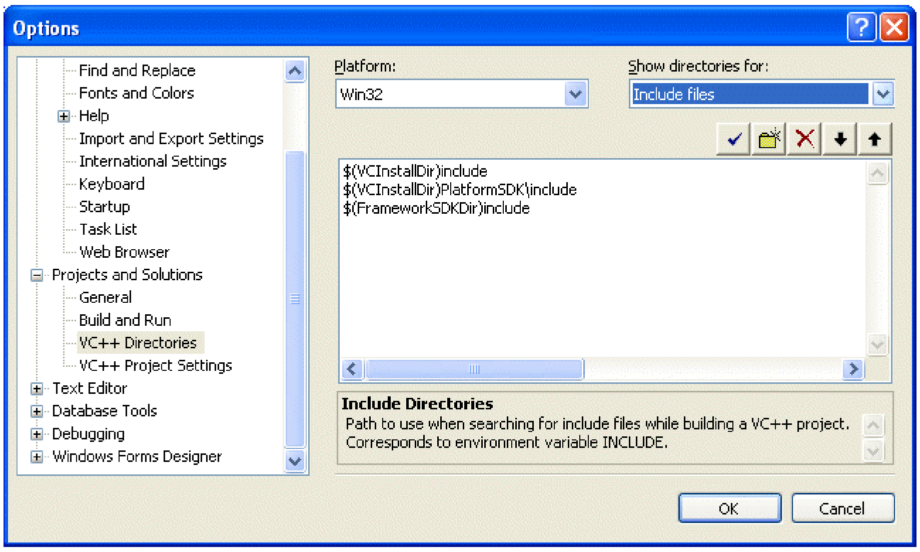
- The #include directive causes copy of a specified file to be included in place of the directive. The two forms of the #include directives are:

```
// 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 project options (compilers/IDEs) configuration. You have to check your compiler documentation. Compilers normally put the standard header files under the INCLUDE or INC directory.



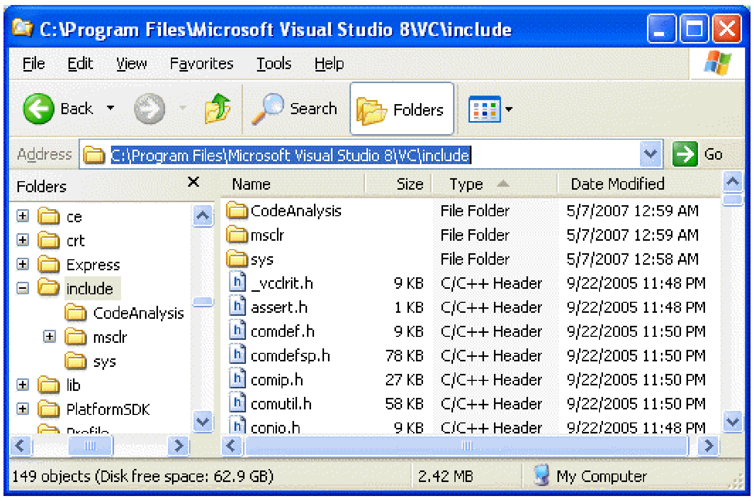
Platform: Win32

Show directories for: Include files

```
$(VCInstallDir)include
$(VCInstallDir)PlatformSDK\include
$(FrameworkSDKDir)include
```

Include Directories
Path to use when searching for include files while building a VC++ project. Corresponds to environment variable INCLUDE.

- 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.


C:\Program Files\Microsoft Visual Studio 8\VC\include
File Edit View Favorites Tools Help
Back Search Folders
Address C:\Program Files\Microsoft Visual Studio 8\VC\include Go
Folders
ce
crt
Express
include
CodeAnalysis
msclr
sys
lib
PlatformSDK
Profile

Name	Size	Type	Date Modified
CodeAnalysis		File Folder	5/7/2007 12:59 AM
msclr		File Folder	5/7/2007 12:59 AM
sys		File Folder	5/7/2007 12:58 AM
_vcclrit.h	9 KB	C/C++ Header	9/22/2005 11:48 PM
assert.h	1 KB	C/C++ Header	9/22/2005 11:48 PM
comdef.h	9 KB	C/C++ Header	9/22/2005 11:50 PM
comdefsp.h	78 KB	C/C++ Header	9/22/2005 11:50 PM
comip.h	27 KB	C/C++ Header	9/22/2005 11:50 PM
comutil.h	58 KB	C/C++ Header	9/22/2005 11:50 PM
conio.h	9 KB	C/C++ Header	9/22/2005 11:48 PM

149 objects (Disk free space: 62.9 GB)   2.42 MB   My Computer

- 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.
- We can also use the relative path as used in UNIX system 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 a circle area:

```
#define  CIR_AREA(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 a 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 (creating and dismantling the stack), 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, undefine 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:

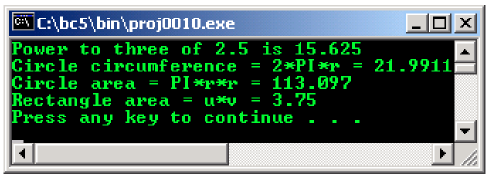
```
#define    getchar()    getc(stdin)
```

- 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>
using namespace std;
#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;
   return 0;
}
```

Output:



```
C:\bc5\bin\proj0010.exe
Power to three of 2.5 is 15.625
Circle circumference = 2*PI*r = 21.9911
Circle area = PI*r*r = 113.097
Rectangle area = u*v = 3.75
Press any key to continue . . .
```

- In another Module 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) respectively.
- 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.

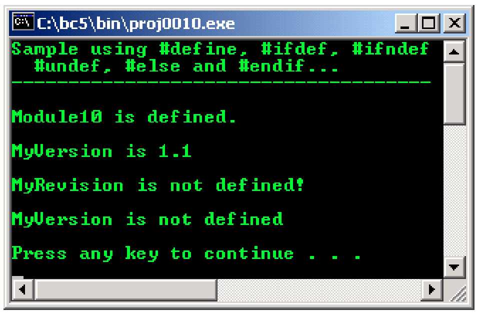
```
#ifdef DEBUG
   printf("Variable x = %d\n", x);
#endif
```

- 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>
using namespace std;

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
   return 0;
}
```

Output:



```
Sample using #define, #ifdef, #ifndef
  #undef, #else and #endif...
-------------------------------------

Module10 is defined.

MyVersion is 1.1

MyRevision is not defined!

MyVersion is not defined

Press any key to continue . . .
```

- 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.

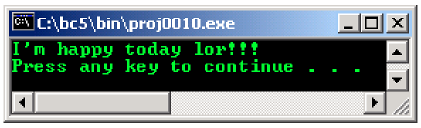
```
// testing the boolean.h header file from
// multiple inclusion or re-definition
#ifndef BOOLEAN_H
typedef int boolean;  // means literal string 'int' is same as 'boolean'

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>
using namespace std;
// notice this...
#include "boolean.h"

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

```
    HappyTime = TRUE;

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

Output:



```
I'm happy today lor!!!
Press any key to continue . . .
```

- 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 project.

```
#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 project, 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.

C & C++ programming tutorials

Further related preprocessor directives reading and digging:

1. A complete process involved during the preprocessing, compiling and linking can be read in C/C++ - Compile, link and running programs.
2. Check the best selling C / C++ books at Amazon.com.