The Preprocessor

- The preprocessors are the directives, which give instruction to the compiler to preprocess the information before actual compilation starts.
- All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;).
- You already have seen a #include directive in all the examples. This macro is used to include a header file into the source file.

```
#include <iostream>
```

Symbolic Constants

- The #define preprocessor directive creates symbolic constants. The symbolic constant is called a macro and the general form of the directive is:

```
#define macro-name replacement-text
```

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

```
#define PI 3.14159

int main () {
      cout << "Value of PI :" << PI << endl;
      return 0;
}
```

- In this case the preprocessor swaps out the `PI` symbol with `3.14159` before the code is compiled. What the compiler see is this,

```
int main () {
      cout << "Value of PI :" << 3.14159 << endl;
      return 0;
}
```

Predefined C++ Macros

- C++ provides a number of predefined macros.
- `__LINE__` Line number of the program when it is being compiled.
- `__FILE__` File name of the program when it is being compiled.
- `__DATE__` Month/day/year that is the date of the translation of the source file into object code.
- `__TIME__` Hour:minute:second that is the time at which the program was compiled.
- Take a look at what the following cout lines print.

```
cout << "Value of __LINE__ : " << __LINE__ << endl;
cout << "Value of __FILE__ : " << __FILE__ << endl;
cout << "Value of __DATE__ : " << __DATE__ << endl;
cout << "Value of __TIME__ : " << __TIME__ << endl;
```

Conditional Inclusion

- The preprocessor supports conditional compilation of parts of source file.
- This behavior is controlled by `#if`, `#else`, `#elif`, `#ifdef`, `#ifndef`, and `#endif` directives to control a code block, where `#elif` means “else if”, `#ifdef` means “if defined”, and `#ifndef` means “if not defined”, e.g.

```
#define MY_DEBUG

#ifdef MY_DEBUG
   cout << "Debug trace" << endl;
#endif
```

- Causes the cout statement to be compiled in the program if the symbolic constant MY_DEBUG has been defined before directive #ifdef MY_DEBUG.
- You can use `#if 0` statement to comment out a portion of a program, e.g.

```
#if 0
    cout << "This line will never be compiled" << endl;
#endif
```

- In C++ the boolean `false` declaration will also work, e.g.

```
#if false
    cout << "This line will never be compiled" << endl;
#endif
```

Larger Programs

- Typically, when writing large C++ programs classes are broken into `.hpp` header files and `.cpp` body files, and of course there is always a `main.cpp` file.
- C++ compilers do exactly what you tell them to. If you tell them to include the same file more than once, then that is exactly what they will do. And if you don't handle it properly, you'll get linkage errors as a result of duplication of code in your compile.
- Header files are a good example of using `#ifndef` directive to avoid such duplication, e.g.

```
//
// myclass.hpp
//
#ifndef MyClass_H
#define MyClass_H

class MyClass
{
     public:
          void foo();
          int bar;
};
#endif
```

- In this way multiple `#include “myclass.hpp"` directives may be used in the code, such as in both `main.cpp` and the `myclass.cpp`, files and at compile time the compiler will only include one copy of the header into the compiled codebase.
- See the ChessOOP example of how a program is broken into multiple files. Note that both `main.cpp` and `chess.cpp` start with `#include “chess.hpp"`