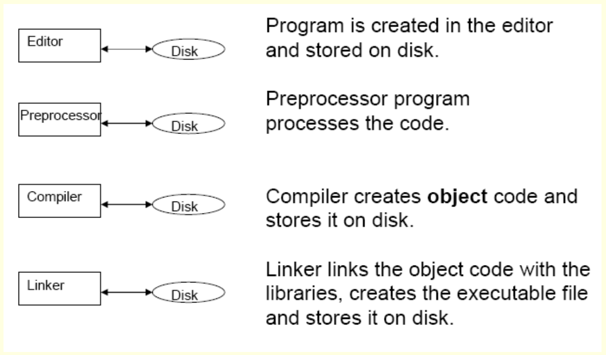
Lecture 6.1

The Preprocessor and Linking

A typical C++ environment



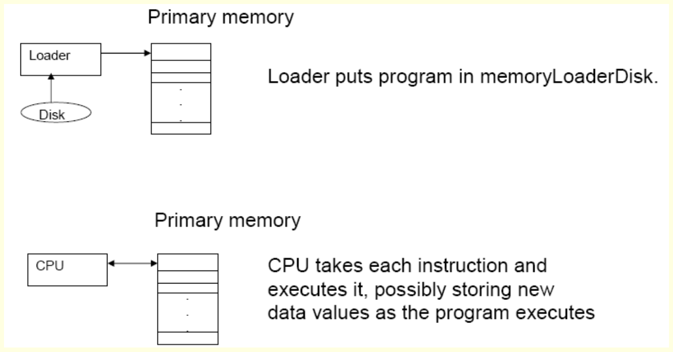
Program is created in the editor and stored on disk.

Preprocessor program processes the code.

Compiler creates **object** code and stores it on disk.

Linker links the object code with the libraries, creates the executable file and stores it on disk.

A typical C++ environment (cont)



Loader puts program in memoryLoaderDisk.

CPU takes each instruction and executes it, possibly storing new data values as the program executes

What is the preprocessor?

- Program that executes automatically **BEFORE** the compiler's translation phase begins.
- Obeys special commands called ***preprocessor directives***
- Indicate that certain manipulations need to be performed on the program before compilation.

What is the preprocessor? (cont)

- These manipulations usually consist of including other text files in the file to be compiled and performing various text replacements.
- The preprocessor is invoked by the compiler before the program is converted to machine language.

What is linking?

- C++ programs typically contain references to functions defined elsewhere
- The object code produced by the C++ compiler typically contains “holes” due to these missing parts.
- A linker links the object code with the code for the missing functions to produce an executable image (with no missing pieces).

Loading, the next phase

- Before a program can be executed, the program must first be placed in memory.
- Done by the loader, which takes the executable image from disk and transfers it to memory.
- Additional components from shared libraries that support the program are also loaded.

Execution phase, the end

- The computer, under the control of its CPU, executes the program one instruction at a time.

Preprocessor directives

- **#include**

  Causes a *copy* of a specified file to be included in place of the directive.
- #include <filename> (**<** and **>**) are used

  for a standard library header file. The preprocessor searches for the file in a implementation – dependent predesigned location.

#include

- #include “filename”
  - Used for programmer defined header files.

- The preprocessor searches for the file first in the same directory as the file being compiled, then in the in the same location as the standard library header files.

What happens when you use #include?

```
/*********** main file ************/
#include “myExample.h”

int main( ){
        int x;
        myExample one;
        x=one.calculateX( );
}
/****end of main file**************/
```

/*********myExample.h***********/
/***this is myExample declaration/**public interface******/

```
class myExample{
        private:
        int a,b;
        public:
        int getA( );
        int getB( );
        void setA (int value);
        void setB (int value);
        int calculateX( );
};
/*****end of myExample.h**********/
```

After preprocessing

```
class myExample{
   private:
        int a,b;
   public:
        int getA( );
        int getB( );
        void setA (int value);
        void setB (int value);
        int calculateX( );
};
int main( ){
   int x;
   myExample one;
   x=one.calculateX( );
}
```

What happened to the class implementation?

- So far it has not been included in the code
- This code can be compiled, but it will have some holes that will need to be filled by the linker in the next phase.

Class Implementations

- Class implementations should be in a **separate** file.
- Traditionally the file has the same name as the class with a **.cpp** extension
- Just as the header file is usually named with the same name of the class with a **.h** extension

MyExample implementation

(MyExample.cpp)
#include "myExample.h" ← Since this is a separate file, the header must be included here too. This file (myExample.cpp) will be compiled **separately** from the main file.

```
myExample::myExample( ) { a = 0; b = 0; }
int myExample::int getA( ) { return a; }
int myExample::int getB( ); { return b; }
void myExample::void setA (int value); { a = value; }
void myExample::void setB (int value); { b = value;}
int myExample:: calculateX( ); { return (a + b); }
```

More about implementations

- Implementations should NOT be included in header files
- their .cpp files should NEVER be included as headers
- Classes are meant to be reused.
- You don't want to give the public access to your code.
- You want to let other people USE your class, not modify it.

Even more about implementations

- This implementation file CAN be compiled on its own, without the need of a main (or driver) file.
- The compiler will generate an ***object*** file if the compilation was successful.
- This object file will have the same name as the .cpp file except that it will have a **.o** extension.

For example

compiling myExample.cpp will produce myExample.o

Even more about implementations (cont)

- The linker the one that generates the executable file.
- If we only compile myExample.cpp there will be NO executable file.
- The executable file will be generated once the linker ***links*** the object code of the main file with the object code of myExample file

Linking and Dev C++

- In DevC++ you usually don't have to worry about specifying that you want to link several files together
- You have to include them as part of a project.
- When compiling the project the linking will be done automatically.

#define

- This preprocessor directive creates symbolic constants.
- Symbolic constants are normal *constants* represented by symbols instead of being declared with a data type.
- For example,

```
#define SIMB_CONST 99
    is equivalent to,
const int SIMB_CONST = 99
```

#define (cont)

- Traditionally capital letters are used for constant identifiers.
- There's no = sign after SIMB_CONST used with #define.

  Otherwise every time you use SIMB_CONST in your program, it would be replaced by =99 instead of just 99.

Advantages of using #define over const

- You can check whether a symbolic constant has been defined or not (with the use of conditional compilation directives)

Conditional Compilation

- Enables the programmer to control the execution of preprocessor directives and the compilation of program code.
- Each conditional preprocessor directive evaluates a constant integer expression that will determine if the code will be compiled.

Conditional Compilation (cont)

- They work pretty much like a normal *if* statement.

For example

```
#ifndef X //or #if ! defined X
#define X
……. // definition of X
#endif
```

Conditional Compilation Example

```
#ifndef MYEXAMPLE_H
#define MYEXAMPLE_H

class myExample{
   private:
        int a,b;
   public:
        int getA( );
        int getB( );
        void setA (int value);
        void setB (int value);
        int calculateX( );
};
#endif
```

Final recommendation

- Good programming practice!

  Use the name of the header file with the period replaced by an underscore when you are using conditional compilation preprocessor directives.