Comp151

Compilation and Separate Compilation

Quick and Dirty Compilation in Unix

Recall the program *Name_Equivalence1.cpp* from the last lecture.

```
//Name_Equivalence1.cpp

#include <iostream.h>
class X {public: int a;};
void main()
{
  X x1, x2;
  x1.a = 1;
  x2.a = 2;

  cout << " x1.a = " << x1.a << ", x2.a = " << x2.a << endl;
  x2 = x1;
  cout << " x1.a = " << x1.a << ", x2.a = " << x2.a << endl;
}
```

Quick and Dirty Compilation in Unix (cont)

We will use g++ to compile c++ programs under Unix.

- *g++ foo.cpp*
  compiles *foo.cpp* and leaves executable in *a.out*

```
228> g++ Name_Equivalence1.cpp
229> a.out
          x1.a = 1, x2.a = 2
          x1.a = 1, x2.a = 1
```

- *g++ -o foobar foo.cpp*
  compiles *foo.cpp* and leaves executable in *foobar*

```
230> g++ -o Name_Equivalence1 Name_Equivalence1.cpp
231> Name_Equivalence1
          x1.a = 1, x2.a = 2
          x1.a = 1, x2.a = 1
```

Motivation: "Divided We Win"

We have a program "program.cpp" that uses a class called `Picture` to manipulate "character pictures": for example, it permits framing them, and horizontally or vertically gluing them together.

It is useful to keep the implementation of the `Picture` class in a separate file "picture.cpp", because:

- This makes it easy to reuse it in another (application) program
- Two programmers can work easily together: one implements `Picture` and the other writes the main program, "program.cpp".
- When the program is changed, only "program.cpp" needs to be compiled again, so the compilation is faster.
  In large software projects this makes a huge difference!

Remark: By convention, C++ program files usually have the suffix: ".cpp", ".cc", ".C", or ".cxx".

Class Header File: ".h"

- Since we don't want the user who writes "program.cpp" to know the details of the `class Picture` (which might be a commercial secret), we need to separate the class interfaces (declarations) from the class implementation.
- On the other hand, the main program "program.cpp" also needs to know about the definition of `class Picture` and its methods before it can be compiled.
- The solution is to describe the `class Picture` in 2 files:
  - class header file, "picture.h" — containing the interface
  - class implementation file, "picture.cpp" — containing the implementation (of constructors and all methods)

Class Header File: ".h" ..


```
/* picture.h */
class Picture
{
    // ...
    Picture* frame(const Picture&);
}
```

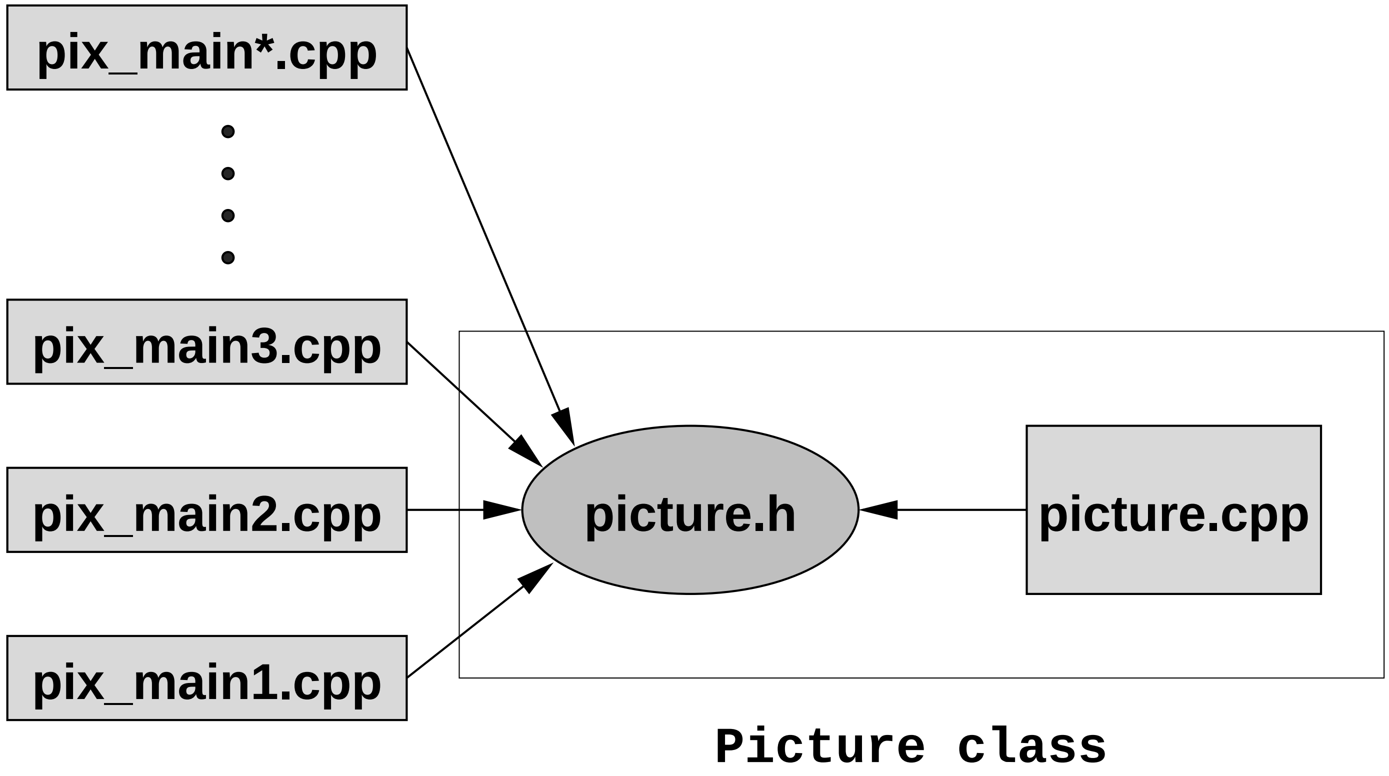
```
/* picture.cpp */
#include "picture.h"

Picture* frame(const Picture& x)
{
    // codes to frame a picture ...
}
```

```
/* program.cpp */
#include "picture.h"

int main()
{
    // manipulate pictures...
}
```

Class Header File: “.h” ...



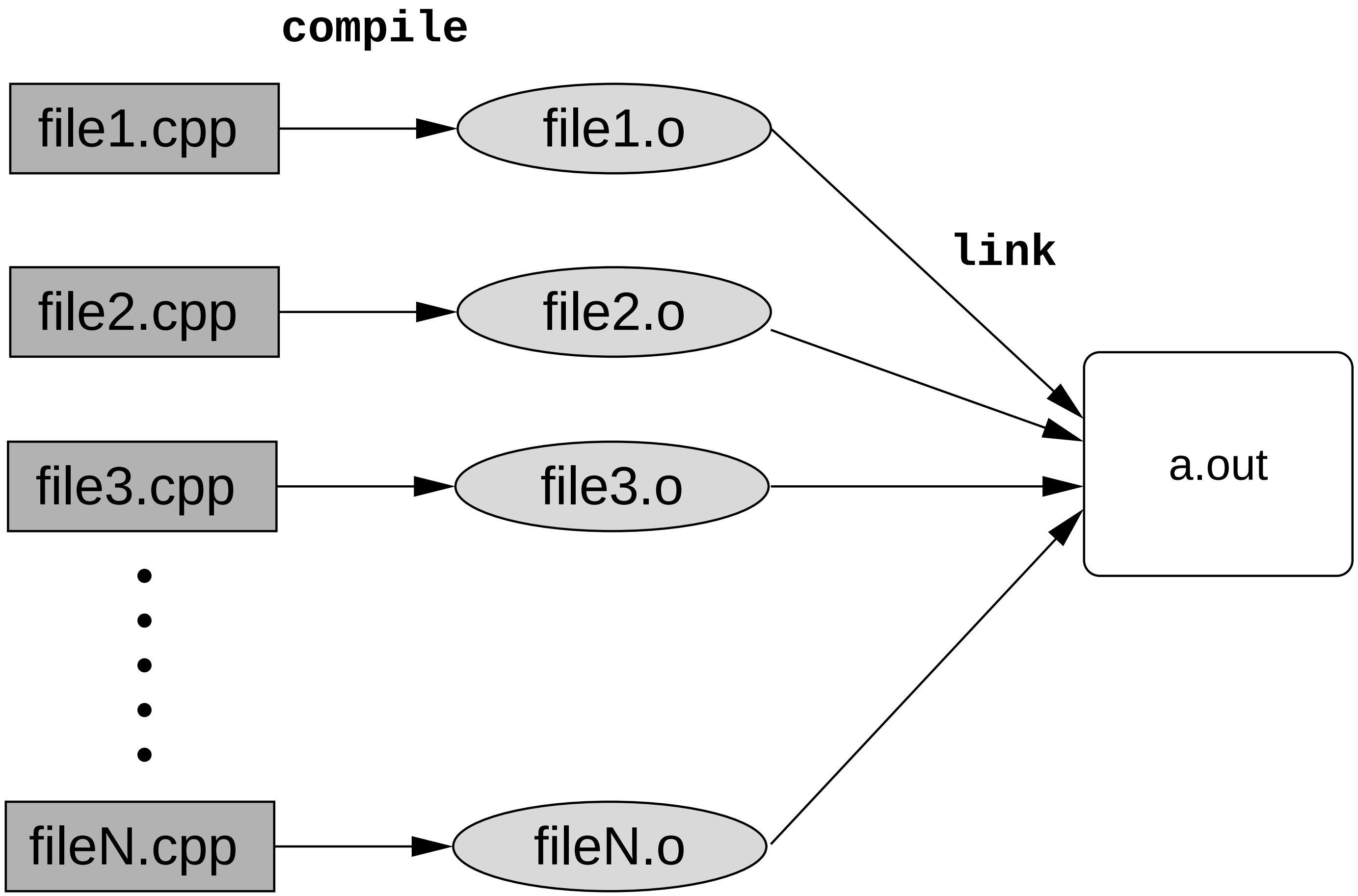
Separate Compilation

We can compile the program with g++ as follows:

```
1> g++ -c program.cpp
2> g++ -c picture.cpp
3> g++ -o program program.o picture.o
```

- "g++" has many options; "man g++" for details.
- The "-c" option on lines (1) and (2) create the object files "program.o" and "picture.o". They can't run on their own.
- Line (3) creates the executable program called "program" (with the "-o" option) by linking the object files together.
- Linker: is a program that binds together separately compiled pieces of code.

Linking Object Files



Down and Dirty

For small programs that do not have many files

```
1> g++ -c program.cpp
2> g++ -c picture.cpp
3> g++ -o a.out program.o picture.o
```

gives the same result as

```
g++  program.cpp picture.cpp
```

That is, the execuable code is created and stored in *a.out*.

```
//Date.h
//Header File for Date

class Date
{
public:
        //Basic Constructor
        // sets date to be d/m/y
        Date(int m, int d, int y);

        //Default Constructor
        Date();

        //Set Date
        // sets date to be d/m/y
        void set(int m, int d, int y);

        //Print Date
        void print();

private:
         int month, day, year;
};
```

```
//Date.cpp
//Implementation File for Date

#include <iostream.h>                    //Library file so uses <>
#include "Date.h"                        //User Defined so uses ""

Date::Date(int m, int d, int y)
{ month =m; day =d; year =y;}

Date::Date()
{}

void Date::set(int m, int d, int y)
{ month =m; day =d; year =y;}

void Date::print()
{cout << day << "/";
 cout << month << "/" <<year << endl;
}
```

```
//CheckDate.cpp
//uses Date Class

#include <iostream.h>                    //Library file so uses <>
#include "Date.h"                        //User Defined so uses ""

void main()
{ Date Today(2,4,2001), When;

  cout << "Today is "; Today.print();

  When.set(6,1,2002);
  cout << "When is "; When.print();
}
```

An Example

	Date.h:	Declaration of Date class
Given	*Date.cpp:*	Definitions of Date class
	CheckDate.cpp:	test program using Date class

Both

```
g++ Date.cpp CheckDate.cpp
```

```
g++ -c Date.cpp
g++ -c CheckDate.cpp
g++ -o a.out Date.o CheckDate.o
```

result in the same executable *a.out.* Running *a.out* gives

```
Today is 4/2/2001
When is 1/6/2002
```

Separate Compilation ..

- If "CheckDate.cpp" is changed but "Date.cpp" is not, then the first line of the 3 line compilation is unnecessary and you just need:

  ```
  g++ -c Checkdate.cpp
  g++ -o a.out Date.o Checkdate.o
  ```

  This can save a lot of time!

- The separate compilation process can be simplified using "`gmake`" on a "`Makefile`". This will 'automatically' check which files have been changed and need to recompiled before linking and which have not been changed (so their old object files can be used). You will lean more about this in the lab.

Preprocessor Directives: #include

Besides statements allowed in a programming language, some useful features are added via directives which are handled by a program called a preprocessor before the source code is compiled.

- In C++, preprocessor directives begin with a # sign in the very first column.
- The `#include` directive reads in the contents of the named file.

```
#include <standard_file.h>
#include "my_file.h"
```

- Angle brackets (< >) are used to include standard header files which are searched for in the standard library directories.
- Quotes (`"  "`) are used to include user-defined header files which are searched for first in the current directory.
- `"g++ -I"` may be used to change the search path.

Libraries

To produce a working executable, the linker needs to include the codes for functions that are *declared* in the standard C++ header files (iostream.h, string.h, etc.). The corresponding codes can be found in the standard C++ libraries.

- A library is a collection of object files.
- The linker selects object codes from the libraries that contain the definitions for functions used in the program files, and includes them in the executable.
- Some libraries are used automatically by the C++ linker, such as the standard C++ library. Other libraries have to be specified during the linking process with the "-l" option.

  e.g. To link with the standard math library "libm.a",

```
g++ -o myprog myprog.o -lm
```

Library Example

```
//Use_sqrt
//Illustrates the use of the math library

#include <iostream.h>
#include <math.h>

void main()
{ cout << "The Square Root of 5 is " << sqrt(5.0) << endl; }
```

To compile/link this program you need to invoke

```
g++ -o Use_sqrt Use_sqrt.o -lm
```

Without the *-lm* and the `include <math.h>` the compilation *should* fail.

#ifndef, #define, #endif


```
/* program.h */      /* b.h */            /* c.h */
#include "b.h"       #include "a.h"       #include "a.h"
#include "c.h"       #include "d.h"       #include "e.h"
...                  ...                  ...
```

- Since #include directives may be nested, the same header file may be included twice!
  - multiple processing ⇒ waste of time
  - re-definition of #define constants/macros
- Thus, the need of conditional directives

```
#ifndef PICTURE_H
#define PICTURE_H
// object declarations, class definitions, functions
#endif  // PICTURE_H
```