Program Organization && Separate Compilation

- Separate compilation for performance reasons
    - Header files store declarations `.h`

      -public interface + private data members

    - `.cpp` files for implementation

> For each source file (.cpp) in a project, the compiler produces an object file (.o) containing: (1) the machine language translation of the code for each function; (2) storage for global objects (e.g. std::cout) (3) a list of global names defined (i.e. implemented) by this object file; (4) a list of global names used in this file that need a definition somewhere

> The linker brings all these objects files together, along with needed object files that are part of the library, to produce an executable file.

> Rules: (1) Nothing can be defined more than once (2) Every need must be satisfied by some definition (3) There must be (exactly) one main routine

- Compilation
    - Source file `Circle.cpp` and `myapp.cpp`
    - Translate each `.cpp` file into machine language code (`.o` object files)
        - `Circle.o` and `myapp.o`
    - Compiler also has two tables: names of things that the object files defines (functions) and names of things it needs

Define	Needs
Circle:: Circle	exit
area	atan

Define	Needs
main	Circle:: Circle
	area
	cout
	<<

    - pulls definitions from the library that defines a bunch of things
    - .o files and library linked by linker, produces executable file
- Don't put implementations of functions in header files
    - Suppose we put implementation in header file --> linker error

        - implementation will be compiled in `Circle.cpp` as well as `myapp.cpp`

- You don't include `.cpp` files
    - Double implementation
- Standard header files
    - All standard headers are protected against being included more than once in the same files
        - You can `#include <string>` in both the `.h` and `.cpp` file
- `Using namespace std;`
    - Do not put this in the header file generally
        - There is no way to turn it off when you put this in the header file
    - No issue in `.cpp` files
        - `.cpp` files are not included anywhere else

More Issues with Separate Compilation...

Include Guards

> Prevents double declaration/include of header files e.g. main.cpp includes app.h and student.h while student.h also includes app.h (two app defs in main-> error)

```
Point.h
class Point
{};

===
Circle.h
class Circle
{
    ...
    private:
        Point m_center;
        double m_radius;
}

===
myapp.cpp
#include "Circle.h"
int main()
{
    Circle c(-2, 5, 10);
    ...
}
```

```
//This will not compile, because Circle.h does not know what Point is, we should
include point.h in the circle file
```

```
===
Circle.h

#include "Point.h"
class Circle
{
    ...
    private:
        Point m_center;
        double m_radius;
}
```

> But what if the user also wants to use a point?

- Do NOT try switching order of `#include` in main routine file

Include Guarding

```
Point.h

//surround everything with included guard
//if the symbol has not yet been defined, then define it, then end

#ifndef POINT_INCLUDED
#define POINT_INCLUDED //this is just a symbol

class Point
{
    ...
};

#endif //POINT_INCLUDED
```

- Compiler asks: has this symbol been defined yet? If not --> mark defined, if yes, skip
- This will let the main routine know what the point is no matter how

```
main
#include "Circle.h"
#include "Point.h" //by the time the compiler gets here, point.h has already been
defined once, so it sees it, sees its marked defined, then skips everything
```

- **Everything significant** should be inside the includ guard
- Should be done with **every header file**

- Symbol should be different from any other symbol
- All standard headers have include guards

Circular Dependencies

```
Student.h

#ifndef STUDENT_INCLUDED
#define STUDENT_INCLUDED

#include "Course.h" //compiler first goes to Course.h for Course in this file

class Student
{
    ...
    void enroll(Course* cp);
    Course* m_studyLIst[10];
};

#endif

Course.h

#ifndef COURSE_INCLUDED
#define COURSE_INCLUDED

#include "Student.h" //as compiler comes here and needs Student, goes to Student.h
again, thinks its already seen the file (include guard), skips everything, we
never actually process the Student.h file

class Course
{
    int units() const;
    Student* m_roster[1000];
};

#endif

main.cpp

#include "Student.h"
void f(Student* s, Course* cp)
{
    s->enroll(cp);
}
```

- Student depends on Course, and Course depends on student --> PROBLEM!!!

Incomplete Type Declaration

```
class A; //this exists as a type, but no details

//later on define the class somewhere else
```

> Why does the compiler not need any details?

- Compiler needs to figure out the **size** of objects: how big a Student is (an array of 10 Course pointers)
  - In C++, all pointers are 4 bytes long
  - So the compiler knows the size without any other details
  - `studyList` is 40 bytes long
  - Compiler only needs to know name of the type to know the size of something

```
Student.h

#ifndef STUDENT_INCLUDED
#define STUDENT_INCLUDED

class Course;
...

Course.h

...
class Student;
```

- In main routine: cannot do incomplete type declaration
  - because we are depending that we know Student has a member function called `enroll`
  - for class Course, we can do incomplete type (because it only takes a pointer)

  ```
  #include "Student.h"
  class Course; //could also just say #include, since .cpp file

  void f(Student* s, Course* cp)
  {
      s->enroll(cp);
  }
  ```

Basically...

> If the file Foo.h defines the class Foo, when does another file you to say `#include "Foo.h" and when can you do incomplete type declaration?

You have the `#include` the header file defining a class when:

- you declare a data member of that class type

```
FirstClass y;
```

- you declare a container (e.g. an array/vector) of objects of that class type

```
FirstClass b[10];
```

- you create an object of that class type
- you use a member function of that class type
- you use the variable in any way (call a method on it, return it, etc.)

```
y.someFunc();
return(y);
```

If all the file needs to know is the *name* of the type, then you can do incomplete type declaration

- When you use the class to define a parameter to a function

```
void goober(FirstClass p1);
```

- When you use the class as the return type for a function

```
FirstClass hoober(int a);
```

- When you use the class to define a pointer or reference variable

```
void joober(FirstClass &p1);
void koober(FirstClass *p1);

void loober()
{
    FirstClass *ptr;
}

FirstClass *ptr1, *z[10];
```

Example:

```
class Blah
{
    void g(Foo f, Foo& fr, Foo* fp); //incomplete, just prototype of function , even if passing by value is copying (not in prototype)
    Foo* m_fp; //incomplete, since only pointer and we know size
    Foo* m_fpa[10]; //incomplete
    vector<Foo*> m_fpv; //incomplete

    Foo m_f; //Must include
    Foo m_fa[10]; //Must include
    vector<Foo> m_fv; //Must include
};

void Blah::g(Foo f, Foo& fr, Foo* fp)
{
    Foo f2(10,20); //must include, declaring local data member
    f.gleep(); //must include
    fr.gleep(); //must include
    fp-> gleep(); //must include
}
```

> Some illegal stuff: Circular dependencies with actual objects...

```
Struct A
{
    B b; //for this to compile, compiler has to see full declaration of B
};

Struct B
{
    int i;
    A a; //but if we put B above A the same problem will occur here
};
```

This is **illegal** anyways!!!

```
A a;
```

This `a` object contains a B object, which has an integer and an `a` object, infinity...

How big is this `a` object?

size of A = size of B

size of B = 4 (int) + size of A

hence size of A = 4 + size of A --> infinite!!!

-