Code Libraries

Lecture 8

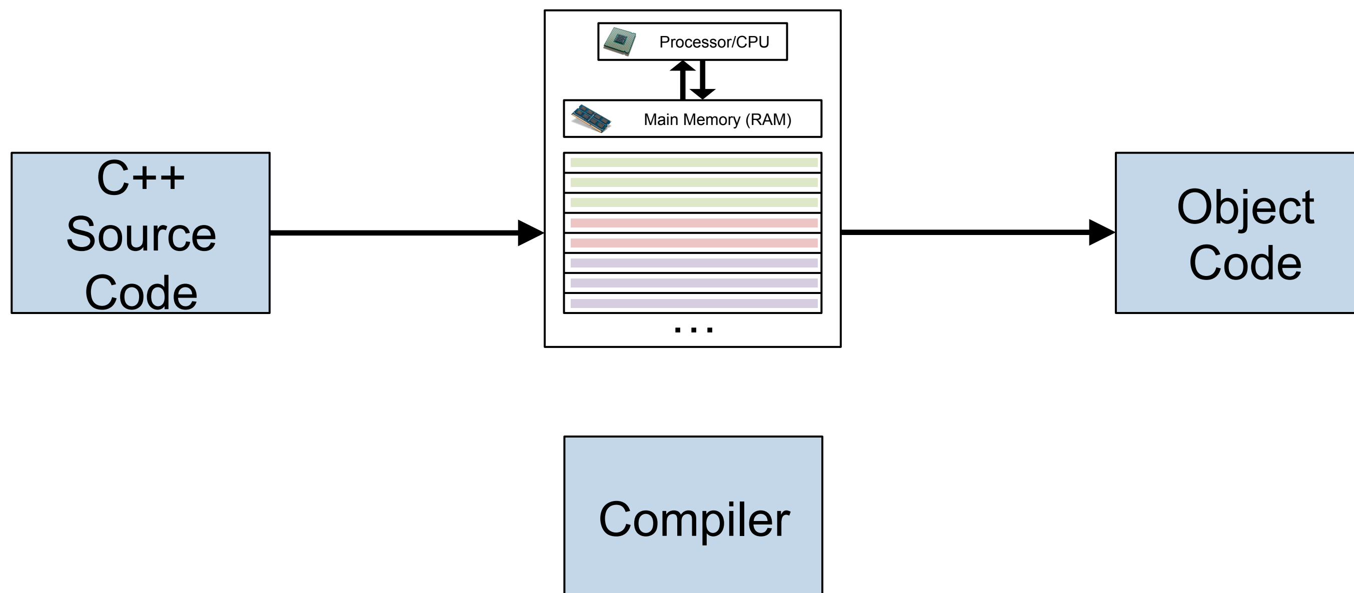
Outline

- Context
- Headers and includes
- Type Definitions
- Templates
- Namespaces

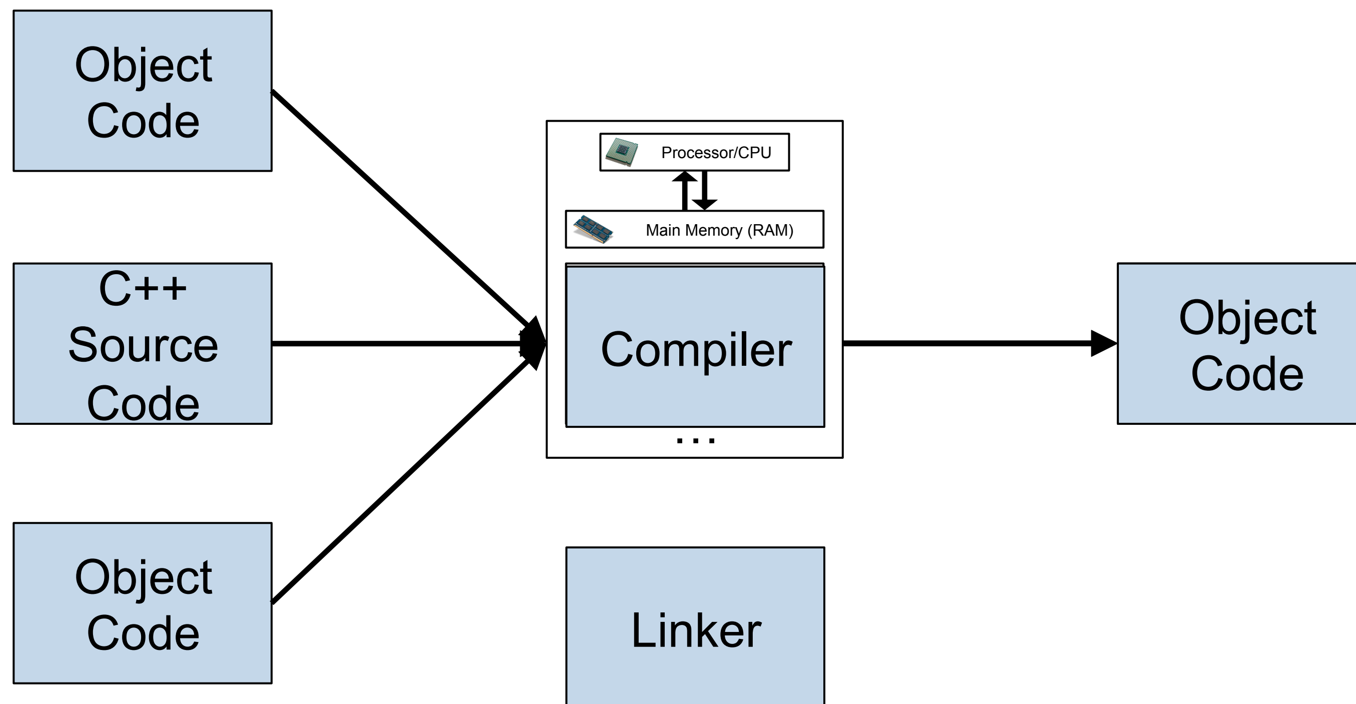
Separate Compilation

- Ever since we started C++, you have been incorporating external code into your programs via `#include`
- These standard libraries organize code into modules that are useful in a variety of programs
- You also have the ability break up *your* programs into multiple files

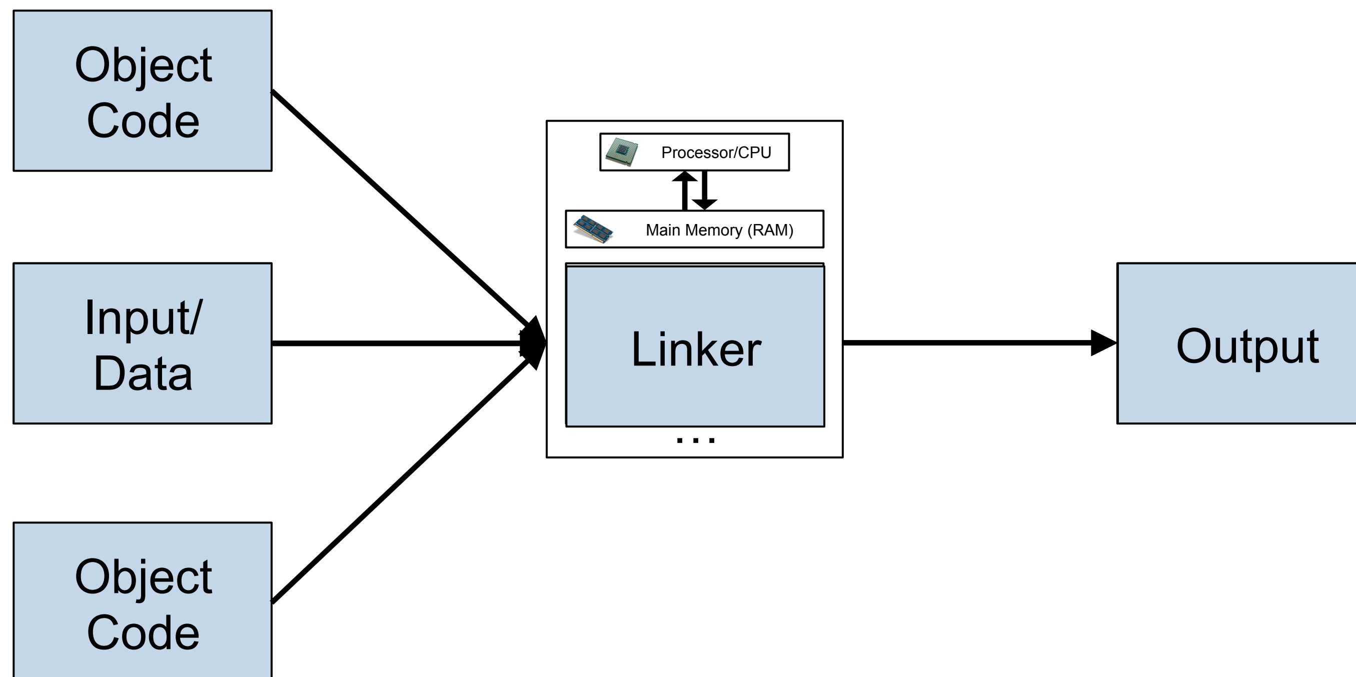
C++ Compilation Process (1)



C++ Compilation Process (2)



C++ Compilation Process (3)



Why `#include`?

- **Speed**. As your programs get more complex it can take a long time to compile. If you break into multiple pieces, you only need to recompile those pieces that change.
- **Organization**. Easier to find/maintain code, easier to re-use across projects.
- **Modularity**. Separate the *interface* (what a class does) from the *implementation* (how it does it).

Problem with Separate Compilation

myclass.cpp

```
class MyClass
{
public:
	void foo();
};

void MyClass::foo()
{
	// do stuff
}
```

main.cpp

```
int main()
{
	MyClass a;
	return 0;
}
```

```
> g++ -c -o myclass.o myclass.cpp
> g++ -c -o main.o main.cpp
main.cpp:3:2: error: unknown type name 'MyClass'
        MyClass a;
```

Enter Header Files

- Header files allow you to separate declaration from definition
- The result is that when compiling each object file, the compiler knows that function(s)/class(es) exist, and linking brings together the implementation

Separate Compilation

myclass.h

```
class MyClass
{
public:
	void foo();
};
```

myclass.cpp

```
#include "myclass.h"

void MyClass::foo()
{
	// do stuff
}
```

main.cpp

```
#include "myclass.h"

int main()
{
	MyClass a;
	return 0;
}
```

Same

```
> g++ -c -o myclass.o myclass.cpp
> g++ -c -o main.o main.cpp
> g++ -o prog myclass.o main.o

> g++ -o prog myclass.cpp main.cpp
```

What is `#include` Doing?

- The `#include` command acts as a form of copy/paste, telling the compiler to replace the line with the entire contents of the included file
- Thus, all source files that `#include` a header file get the full contents of that file
- In general, you should not `#include` source files

File Names

- Header files use a `.h__` extension
  - `.h` / `.hxx` / `.hpp` are common
- C++ source files use a `.c__` extension
  - `.cpp` / `.cxx` / `.cc` are common
- C source files use a `.c` extension
  - Some tools look for this extension, so don't use `.c` unless you are programming in C

`#include ""` vs. `<>`

- Surrounding characters indicate to the compiler where to find the header file
- The angle brackets (`<>`) typically indicate built-in/standard that are kept in a pre-defined location
- The double quotes (`""`) typically indicate programmer libraries that are kept in the current directory, or a list of alternatives

Additional Include Locations (1)

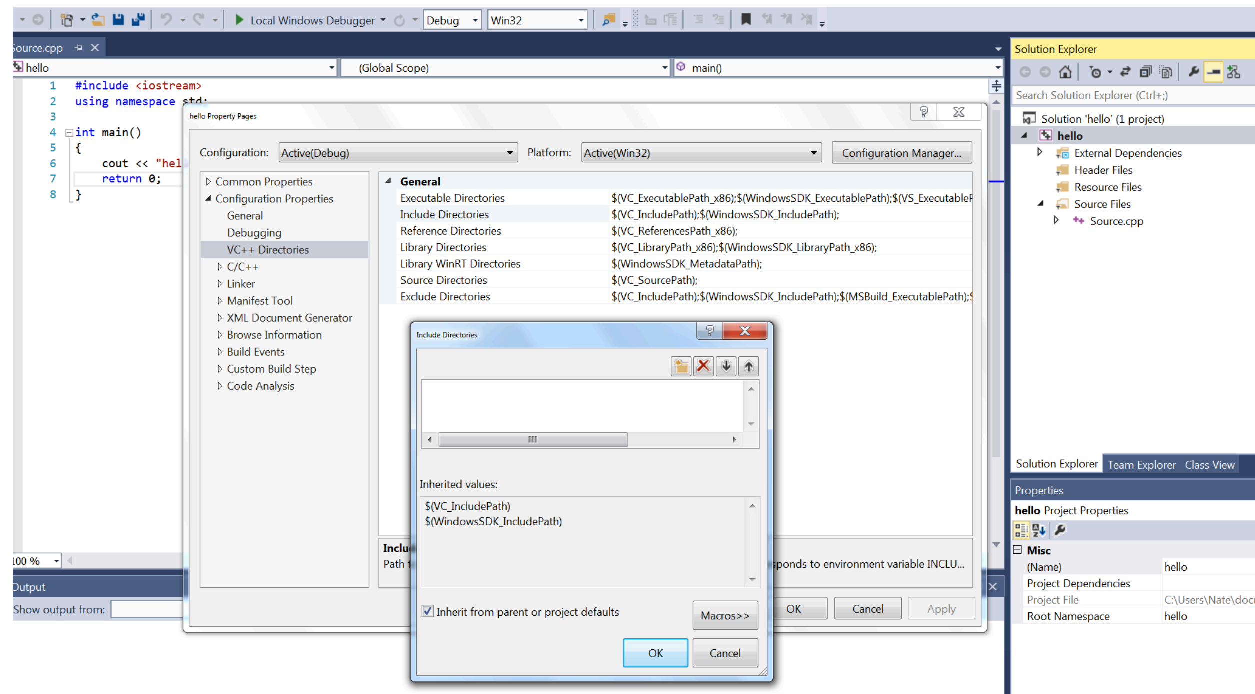
Each compiler has a way to indicate alternative search locations for header files

```
> g++ -c -o myclass.o myclass.cpp
myclass.cpp:1:10: fatal error: 'myclass.h' file not found
#include "myclass.h"

> g++ -c -o myclass.o -I/path/to/includes myclass.cpp
```

Additional Include Locations (2)

Each compiler has a way to indicate alternative search locations for header files



Include Guards

- A best practice in C++ is to guard header files against multiple inclusion from the *same* file (results in *already defined* error)
- Can occur by simple mistake

```
#include "myclass.h"
#include "myclass.h"
```

- Or by including multiple header files with a common dependency

Unguarded Headers

x.h

```
class X
{
};
```

a.h

```
#include "x.h"

class A
{
	X x;
};
```

b.h

```
#include "x.h"

class B
{
	X x;
};
```

main.cpp

```
#include "a.h"
#include "b.h"

int main()
{
	A a;
	B b;
	return 0;
}
```

```
> g++ -o prog main.cpp
In file included from main.cpp:2:
In file included from ./b.h:1:
./x.h:1:7: error: redefinition of 'X'
class X
```

Include Guard

Basic idea: each header gets a unique identifier, check for it before including. Headers get included at most once!

```
// x.h

#ifndef __X_H
#define __X_H

class X
{
};

#endif
```

Guarded Headers

x.h

```
#ifndef __X_H
#define __X_H

class X
{
};

#endif
```

a.h

```
#ifndef __A_H
#define __A_H

#include "x.h"

class A
{
	X x;
};

#endif
```

b.h

```
#ifndef __B_H
#define __B_H

#include "x.h"

class B
{
	X x;
};

#endif
```

main.cpp

```
#include "a.h"
#include "a.h"
#include "b.h"
#include "b.h"

int main()
{
	A a;
	B b;
	return 0;
}
```

```
> g++ -o prog main.cpp
```

Forward Declaration

- To avoid circular dependencies, it is best to avoid **#include** within header files when possible
- A forward declaration is a way of stating basic information about an identifier (e.g. class, structure, function) without having the complete definition
  - We have been doing this with functions since COMP128!
  - For a class/structure:

```
class X;
struct Y;
```

Guidelines

If in header for class **A**, considering class **B**…

- **Do Nothing**. **A** makes no references at all to **B**
- **Do Nothing**. The only reference to **B** is in a `friend` declaration
- **Forward Declare**. **A** contains a **B** pointer or reference
```
B* myb;
```
- **Forward Declare**. One or more functions has a **B** object/pointer/reference as a parameter, or as a return type
```
B MyFunction(B myb);
```
- `#include "b.h"`. **B** is a parent class of **A** (later)
- `#include "b.h"`. **A** contains a **B** object
```
B myb;
```

Fix This Code!

x.h

```
class X
{
};
```

a.h

```
#include "x.h"

class A
{
	X foo(X x);
	X *x;
};
```

b.h

```
#include "x.h"

class B
{
	X x;
	void foo(X& x);
};
```

main.cpp

```
#include "a.h"
#include "b.h"

int main()
{
	A a;
	B b;
	return 0;
}
```

Answer

x.h

```
#ifndef __X_H
#define __X_H

class X
{
};

#endif
```

a.h

```
#ifndef __A_H
#define __A_H

class X;

class A
{
      X foo(X x);
      X *x;
};

#endif
```

b.h

```
#ifndef __B_H
#define __B_H

#include "x.h"

class B
{
      X x;
      void foo(X& x);
};

#endif
```

main.cpp

```
#include "a.h"
#include "b.h"

int main()
{
      A a;
      B b;
      return 0;
}
```

Libraries

- Header files, when combined with their implementation files, form a library
- Each header can contain one or more classes, as well as non-class functions
  - Just provide a declaration in the header

Type Definitions

- The C++ **typedef** keyword allows you to create a new name (or alias) for an existing data type

```
typedef existing_type alias;
```

- Type definitions are common in header files and provide both clarity and ease of modification

```
typedef int dept_id_t;
dept_id_t getDepartmentID();
dept_id_t id = 314;
```

Templates

- Often times you write a function for a specific data type, but it could actually be used more generally with other data types
- Templates allow you to tell C++ that one or more data types can be changed in a function/class implementation

Example

```
#include <iostream>
using namespace std;

void swap_value(char& v1, char& v2)
{
    char temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}

int main()
{
    char a = '1', b = '2';

    cout << a << " " << b << endl; // 1 2
    swap_value( a, b );
    cout << a << " " << b << endl; // 2 1

    return 0;
}
```

What if I want to swap two integers or doubles?

Function Templating

- Preface the function with…

```
template <typename T>
```

- The T can be any identifier, and becomes a pattern that gets used anywhere you wish inside the function
  - You will also see `<class T>`, it doesn't actually matter
  - At compile time, whenever the function is called with a new type, C++ makes a whole new function copy
- In C++, you **cannot** have a separate function declaration for a templated function
  - So the entire function goes into a header file if part of a library

Example

```
#include <iostream>
using namespace std;

template <typename T>
void swap_value(T& v1, T& v2)
{
	T temp;
	temp = v1;
	v1 = v2;
	v2 = temp;
}

int main()
{
	char a = '1', b = '2';
	double e = 3.14, pi = 2.7;

	cout << a << " " << b << endl; // 1 2
	swap_value( a, b );
	cout << a << " " << b << endl; // 2 1

	cout << e << " " << pi << endl; // 3.14 2.7
	swap_value( e, pi );
	cout << e << " " << pi << endl; // 2.7 3.14

	return 0;
}
```

Class Templating

- An entire class can be specialized for a particular data type
- This is common for collections of items
  - We saw this for **`vector<type>`**
- Again, preface the class and remember to include templated member function definitions in the header file

Example

wrapper.h

```
#ifndef __WRAPPER_H
#define __WRAPPER_H

template <class T>
class wrapper
{
public:
     wrapper(T init_val) { x = init_val; }

     T get() { return x; }
     void set(T new_val) { x = new_val; }

private:
     T x;
};

#endif
```

main.cpp

```
#include <iostream>
#include "wrapper.h"
using namespace std;

int main()
{
     wrapper<int> x( 7 );
     wrapper<double> y( 7.11 );

     cout << x.get() << " "
          << y.get() << endl; // 7 7.11
     y.set( x.get() );
     cout << x.get() << " "
          << y.get() << endl; // 7 7

     return 0;
}
```

Name Conflicts

- If a program uses classes and functions written by different programmers, it may be that the same name is used for different things
- A **namespace** is a method by which to group together classes and functions to avoid naming conflicts
- All functions/classes are in the **global** namespace unless otherwise indicated

Creating a Namespace

Use the namespace keyword to create a scope within which all functions, classes, etc. are part of the namespace

```
namespace friendly
{
   void smile()
   {
      cout << ":)" << endl;
   }
}
```

Using Namespace Contents

- To make use of a function/class within a namespace, use the scope resolution operator **`namespace::item`**

- The **`using`** keyword tells C++ that within the current scope, it should look in the indicated namespace for unrecognized items

Example (1)

```
#include <iostream>

namespace friendly
{
    void smile()
    {
        std::cout << ":)" << std::endl;
    }
}

int main()
{
    friendly::smile();

    return 0;
}
```

Example (2)

```
#include <iostream>
using namespace std;

namespace friendly
{
    void smile()
    {
        cout << ":)" << endl;
    }
}

int main()
{
    friendly::smile();

    return 0;
}
```

Example (3)

```
#include <iostream>

namespace friendly
{
    void smile()
    {
        std::cout << ":)" << std::endl;
    }
}

int main()
{
    using namespace friendly;

    smile();

    return 0;
}
```

Example (4)

```
#include <iostream>
using namespace std;

namespace friendly
{
    void smile()
    {
        cout << ":)" << endl;
    }
}

int main()
{
    using namespace friendly;

    smile();

    return 0;
}
```

Wrap Up

- Header files allow you to break your code into multiple files, improving speed, organization, and code modularity
  - Name your header/source files appropriately, and `#include` them using the appropriate form
  - Always use include guards
  - Try to minimize `#include` commands within header files to save yourself headaches later

- The `typedef` keyword allows you to provide intuitive, easy-to-modify aliases for data types

- Templates allow you to abstract away the data types your functions/classes can operate upon

- Namespaces allow you to group together functions/classes to avoid naming conflicts