Chapter 12: Compilation & Namespaces

Instructor: Mark Edmonds

edmonds_mark@smc.edu

Compilation

- So far, when we've compiled our programs, we've compiled them from source code all the way to an executable
- However, there are multiple stages to the compilation process, and we can compile each part separately
- For instance, a class definition can be stored separately from a program (by separating the implementation of the class from the program)
  - This allows programmers to use the class in multiple programs, without recompiling the class
    * This is how the Standard Library works - it's compiled ahead of time, and you just have to include the header to have access to the class
  - This is one of the main reasons we separate the interface/header file (.h) from the implementation file (.cpp)

Compilation process

Preprocessing

- Preprocessor directives like `#include` and `#define`
- The preprocess executes the directives, but is unaware of C++ syntax
  - For instance, `#include` directs the preprocessor to go out and fetch the corresponding interface/header file and copy it at this point in the source code
    * That is, the preprocessor replaces the `#include` directives with the content of the included files
- The preprocessor outputs a single output file that is ready to be compiled

Compilation

- The compiler parses C++ source code (that now does not contain any preprocessor directives like `#include`) and converts it into *assembly code*
- The compilation process outputs *object files* that contains compiled code (in binary form) of the original source code
- You can pause the compilation process at this point if you wish

    - This is how libraries are made, including the Standard Library
    - The object files can be placed into special archives called "static libraries"
- Most of the compilation errors you've seen so far most likely occurred at this point
    - For instance, a missing semicolon `;` would be reported at this point

Linking

- The final stage of the process takes all of the object files and produces a final program that can be executed on this machine
    - This process includes linking against object files that are provided by a library, such as the Standard Library
- This stage may report errors about missing "symbols" (such a a function that was declared in a header file, but never defined in any implementation file)
- This stage may also report errors if the same function is defined twice (for instance, if you have two definitions of the `main` function, the linker will produce an error)

Namespaces

- A namespace is a collection of name definitions
    - This could be a grouping of class definitions and variable declarations
- Namespaces are important because multiple programmers may define classes and functions with the same name
    - Imagine Programmer A defined a `Car` class and Programmer B also defined a `Car` class
        * A namespace allows us to specify which `Car` class we'd like to use

`using` directive

- The Standard Library lives in the `std` namespace
- When we've been writing `using namespace std;`, we were telling the compiler "anytime you can't resolve a symbol, see if that symbol exists in the `std` namespace
    - Without this directive, we have to write `std::cout` instead of writing `cout` alone
- If we didn't write `using namespace std`, we could have defined `cout` and `cin` to behave differently
- If you don't define a namespace, then the code you write is in the global namespace
    - No need to use the `using` directive with the global namespace

Creating namespaces

- Very simple. Wrap your code with a namespace grouping

```
namespace ns1
{
  // code
  // could be a class, or functions, or anything
}
```

- For example

```
namespace ns1
{
  void hello_world(){
    cout << "Hello, World!" << endl;
  }
}
```

- And then we can use it like this:

```
ns1::hello_world();
```

Name conflicts

- If the same name is used in two namespaces, then the two namespaces cannot be used at the same time
- For example if `hello_world()` is defined in namespaces `ns1` and `ns2`, the two versions of `hello_world()` could be used in one program using any of the following schemes

```
int main(){
  {
    using namespace ns1;
    hello_world();
  }
  // the using directive will terminate when the block terminates
  {
    using namespace ns2;
    hello_world();
  }
  ns1::hello_world(); // directly using ns1 namespace
  ns2::hello_world(); // directly using ns2 namespace
}
```