CS 3460

Introduction to Object Oriented C++

Origins of C++

- As I’ve noted before, C++ started out by creating a “C with classes” (Bjarne Stroustrup)
- Therefore, classes were an early big focus of the language
- Recent updates to the language have made relatively small changes to classes; most changes are elsewhere
- Concept & syntax of a class in C++ is similar, but not exactly the same, as Java
- Trivia: Only difference between `struct` & `class`
  - default visibility for `struct` is `public`
  - default visibility for `class` is `private`

Basic Class Structure

- In Java, a class is defined in a single .java file
- (conventionally) In C++, a class is split in two sections
  - Declaration in .hpp file
  - Implementation in .cpp file
- The C++ language doesn’t require this, but it is recommended, as you’ll see over time

Basic Class Structure – Class Declaration

- General form of class declaration looks like

```
class [class name]
{
  [access specifier]:
     [members]
};
```

- (optional) [access specifier]
  - `public`, `protected`, `private`
- (optional) [members]
  - any data field or method declarations
- Don’t forget that semi-colon, Java doesn’t require it

Basic Class Structure – Class Declaration

- General form of class declaration looks like

```
class [class name]
{
   [access specifier]:
        [members]
};
```

- Differences from Java
  - No visibility modifier before class keyword
  - Compiler doesn’t enforce or require a specific filename
    - But common practice follows this approach
  - That trailing semi-colon!

Basic Class Structure – Class Definition

- General form of class implementation looks like

```
[return type] [class name]::[method name]([parameters])
{
      … method body …
}
```

- `[return type]` Return type; may be `void`, even `auto`
- `[class name]` Name of the class the method belongs to
- `::` Scope resolution operator
- `[method name]` Name of the method

Basic Class Structure – Include Processing

- `#pragma once`
  - Compiler directive
  - Tells compiler, after it has been processed, don’t process it again
- Legacy code you might see something like…

```
#ifndef _PERSON_HPP_
#define _PERSON_HPP_
    … header file code goes here …
#endif
```

Basic Class Structure – Access Specifiers

- Similar to Java, but not the same
  - `public` : all code has visibility
  - `protected` : only class and derived classes have visibility
  - `private` : only class has visibility
  - No such thing as *default* visibility
- Declare visibility for groups of class members, rather than every member
- Can have any number of visibility groups, even multiple of the same type

Basic Class Structure – Constructors

- Same/similar to Java
  - Have the same name as the class; no return type
  - Default constructor has no parameter
  - Any number of overloaded constructors
  - Same compiler rules for when it does or doesn’t write the default constructor
- Direct Initialization / Member Initializer Lists
  - Executes before the body of the constructor
  - Can be initialized with hard-coded value, calling a function, etc.
  - Not to be confused with `std::initializer_list`

```
Person::Person(std::string nameFirst, std::string nameLast, unsigned short age) :
    m_nameFirst(nameFirst),
    m_nameLast(nameLast),
    m_age(age)
    {
    }
```

Basic Class Structure – Methods

- Declared in the header file
- Implementation can also be done in the header file
  - Common practice for one-liners
- Why not do all implementations in header file
  - (coming soon) Translation Unit
  - Causes compiler to do more work than necessary!

Code Demo – Basic Class Structure

Class Usage – Where Objects Exist

- In Java, all class instances are heap allocated
- In C++, an instance might be on the stack or heap allocated
- Stack Allocated

```
Person p1("Lisa", "Smith", 22);
```

- Heap Allocated

```
Person* p2 = new Person("Larry", "Jones", 33);
… use p2 …
delete p2;
```

Code Demo – Class Usage

Translation Unit & Separate Compilation

- Have seen the use of `#ifndef`, `#define`, `#pragma`, `#include`
  - Compiler directives
  - Programmer is giving instructions to the compiler for how C++ files/code is turned into executable instructions

Translation Unit

- After preprocessing, the result is called a *translation unit*
- Compiler takes this and creates object code
- Only info available is that in the translation unit
  - Translation unit must be self-contained
  - Essential to reduce the work of the compiler by writing separate header/implementation files
  - Only include headers that are necessary

Separate Compilation

- Code only knows of type and function declarations
  - Doesn’t need access to the implementations
- Type and function declarations are placed in header files
- Implementations are placed in implementation files
- They can be separately compiled; simple!
  - Header file in multiple translation units
  - Implementation in one translation unit