Lecture Notes 2

Basic Python to C++ Classes

- C++ `struct` – The `struct` is the closest equivalent to Python `class` because members are by default `public`
- C++ `class` – The `class` in C++ has default `private` access to members, so has slightly different default behavior than the Python `class`
- Definition Example
  - Python `class`

```
class Name:
    def __init__(self):
        self.DataMember = 4
```

  - C++ `struct`

```
struct Name{
    int DataMember = 4; // allowed in C++11
};
```

- Instantiation Example
  - Python `class`

```
# Assumes Name is defined as above
MyName = Name()
```

  - C++ `struct`

```
// Assume Name is defined as above
Name MyName;
```

- Constructor Example
  - Python `class`

```
class Name:
    def __init__(self, param):
        self.DataMember = param
```

  - C++ `struct`

```
// Typically in header file
struct Name{
    int DataMember;
    Name(int param); // Constructor
};

// Typically in cpp file
Name::Name(int param){
    DataMember = param; // Python "self" is implied
    // could also do this->DataMember = param;
}
```

- Member Function Example
  - Python `class`

```
class Name:
    def __init__(self, param):
        self.DataMember = param

    def foo(self, param):
        self.DataMember += param
```

  - C++ `struct`

```
// Typically in header file
struct Name{
    int DataMember;
    Name(int param); // Constructor
    void foo(int param); // Member function
};

// Typically in cpp file
Name::Name(int param){
    DataMember = param; // Python "self" is implied
    // could also do this->DataMember = param;
}

void Name::foo(int param){
    DataMember += param;
}
```

- Static Member Example
  - Python `class`

```
class Name:
    DataMember = 4
```

  - C++ `struct`

```
// Typically in header file
struct Name{
    static int DataMember;
};
// Typically in cpp file
int Name::DataMember = 4;
```

- Static Member Function Example
  - Python `class`

```
class Name:
    DataMember = 4

    def foo(param):
        Name.DataMember += param
```

  - C++ `struct`

```
// Typically in header file
struct Name{
    static int DataMember;
    static void foo(int param);
};

// Typically in cpp file
int Name::DataMember = 4;
void Name::foo(int param){
    DataMember += param;
}
```

- Inheritance Example
  - Python `class`

```
class Base:
    def __init__(self, param):
        self.DataMember = param

class Derived(Base):
    def __init__(self, param, param2):
        Base.__init__(self, param)
        self.DataMember2 = param2
```

  - C++ `struct`

```
// Typically in header file
struct Base{
    int DataMember;
    Base(int param); // Constructor
};

struct Derived : Base{
    int DataMember;
    Derived(int param, int param2); // Constructor
};

// Typically in cpp file
Base::Base(int param){
    DataMember = param;
}

Derived::Derived(int param, int param2):Base(param){
    DataMember2 = param2;
}
```

- C++ Access Specifiers – Unlike Python, C++ can limit access to class/struct members
  - `public` – Members are accessible from anywhere
  - `private` – Members are only accessible from within the class member functions
  - `protected` – Members are only accessible from within the class member functions, or derived class member functions
  - Access Example

```
struct Name{
    int PubDataMember;      // Default is public
    private:   // All members are private after
        int PrivDataMember;
    protected: // Now all are protected after
        int ProtDataMember;
    public:    // Now all are public again after
        int OthPubDataMember;
};

class OtherName{
    int PrivDataMember;     // Default is private
    public:    // All members are public after
        int PubDataMember;
    protected: // Now all are protected after
        int ProtDataMember;
    private:   // Now all members are private after again
        int OthPrivDataMember;
};
```

Python to C++ Functions

- Function Signatures vs. Body
  - Python has only the function body definition

```
def foo(param):
    # do something
    return result
```

  - C++ Often splits the Signature from Body

```
// This is typically put in header
int foo(int param);

// This is typically put in cpp file
int foo(int param){
    // do something
    return result;
}
```

- Function Overloading
  - Python variables are dynamically typed, so single function can handle multiple types

```
def foo(param):
    if isinstance(param, int):
        # do something with int param
        return int_result
    elif isinstance(param, float):
        # do something with float param
        return float_result
    elif isinstance(param, str):
        # do something with string param
        return str_result
```

  - C++ is statically typed, so multiple functions need to be written to handle different types

```
// Compiler will call this one if foo has int argument
int foo(int param){
    // do something with int param
    return int_result;
}

// Compiler will call this one if foo has float argument
float foo(float param){
    // do something with float param
    return float_result;
}
```

```
// Compiler will call this one if foo has string argument
std::string foo(std::string param){
    // do something with string param
    return str_result;
}
```

- Default Arguments
  - Python Example

```
# The default argument for param is 0
def foo(param=0):
    return param + 3

# Call location can provide the argument or not
x = foo(6) # x will be 9
x = foo()  # x is now 3 because same as foo(0)
```

  - C++ Example

```
// The default argument for param is 0
int foo(int param=0);

// The default argument is not repeated in the body
int foo(int param){
    return param + 3;
}

// Call location can provide the argument or not
x = foo(6); // x will be 9
x = foo();  // x will be 3 because same as foo(0);
```

- Named Arguments
  - Python Example

```
# The default for both
def foo(param1=0, param2=0):
    return param1 - param2

# Call location can name to change positional
foo(1,2) # param1 = 1, param2 = 2
foo(param2 = 3) # param1 = 0 default, param2 = 3
foo(param2 = 4, param1 = 5) # param1 = 5, param2 = 4
```

  - C++ – This behavior is possible, but extremely complicated to emulate