Object-oriented Programming with C++
Session 5

Robert Palmere

2021

Topics:

- Inheritance
- Encapsulation (i.e. private, public, protected keywords)
- Overloading (operators and functions)
- Templates
- Namespaces
- Abstract Classes
- Implementations

Now that we understand object-oriented programming in the context of Python, we will take a look at this programming paradigm, again, using C++.

```
#include <iostream>
class Polygon{};

int main(){
   Polygon A;
   return 0;
}
```

If the above program is compiled and ran, it will appear as if nothing happened. However, we declared and efined the class Polygon with no data members. The compiler provides a default constructor for instantiating the class. The destructor is called prior to the end of the program. Nothing is printed for these occurrences so it appears that nothing happened although the program should have ran successfully. If we want to overwrite the constructor, we can do this by using the class name within the scope of the class definition.

```
class Polygon{
   public:
      Polygon(){
         std::cout << "Polygon created." << std::endl;
      }
      ~Polygon(){
         std::cout << "Polygon destroyed." << std::endl;
      }
};
```

Rerunning the program with these changes will make what is happening in the program clearer.

Inheritance

Since we already covered inheritance in the Python session we will just make note of the syntax difference required for inheritance in C++ as well as dealing with encapsulation.

```
class Rectangle: public Polygon {
   public:
      int x = 0;
      int y = 0;
};

int main(){
   Rectangle A;
   return 0;
}
```

When we create an instance of a *Rectangle* called 'A', we notice that the inherited base class *Polygon* constructor / destructor are called. This is because *Rectangle* is a type of *Polygon* which does not have its own constructor / destructor. These are instead overwritten by its base class which it herits from. Notice that *Polygon* is publicly inherited which means that its attributes are accessible from within the *Rectangle* definition. Since *Rectangle* is a derived class of *Polygon* (i.e. it inherits from *Polygon*), we can now add functions to the base class and access them through the derived class instance.

```
include <iostream>

class Polygon{
   public:
      Polygon(){
```

```
            std::cout << "Polygon created." << std::endl;
        }
        ~Polygon(){
            std::cout << "Polygon removed." << std::endl;
        }
        void display (void) {
            std::cout << "Polygon has no display." << std::endl;
        }
};

class Rectangle: public Polygon {
   public:
      int x = 0;
      int y = 0;
};

int main(){
   Rectangle A;
   A.display();
   return 0;
}
```

The derived class, in this case *Rectangle*, can also overload the function from the base class. Overloading here means that the base class function prototype is the same but the parameters / functionality of the function is changed depending on how it is being called. In the next example we provide a default function for display within a *Polygon* and then overload this function within *Rectangle* to print a rectangle to the terminal.

```
class Polygon{
   ...
   void display(void){
      std::cout << ``Polygon has no display.'' << std::endl;
   }
}

class Rectangle: public Polygon{
   ...
   for (int i = 0; i < x; i++){
         std::cout << "-";
      }
      std::cout << std::endl;

      for (int j = 0; j < y; j++){
          for (int i = 0; i < x; i++){
              if (i == 0 || i == (x-1)){
                  std::cout << "|";
```

```
            }
            else{
                std::cout << " ";
            }
        }
        std::cout << std::endl;
    }

    for (int i = 0; i < x; i++){
        std::cout << "-";
    }
    std::cout << std::endl;
}
```

Despite understanding the basics of inheritance, to better appreciate object-oriented programming and the power of inheritance, a simple, annotated, example of how inheritance / polymorphism can be used to limit the code we have to write and make future changes easier to implement is provided in 1-classes7.cpp. Some interesting things can happen when using class pointers as was seen in 1-classes7.cpp which brings us to the *virtual* keyword. This keyword allows the compiler to not bind the function at compile time. Instead, during runtime, the derived class of which the function is called from is the deciding factor as to what instructions to carry out with overloaded functions. When using this keyword, only the definition of the derived class can be changed, not the function prototype.

```
#include <iostream>

class Base{
   public: virtual void print(void){ std::cout << "Hello from base
        class" << std::endl; }
};

class Derived: public Base {
   public: void print(void){ std::cout << "Hello from derived
        class" << std::endl; }
};

int main(){

   Base* B = new Base();
   Derived* D = new Derived();

   B->print();
   D->print();
```

```
   Base* Other = D;

   Other->print();

   delete B;
   delete D;

   return 0;
}
```

Without the *virtual* keyword in the base class *print()* function, we will only call the base class function despite pointing to the derived class (since the type of the pointer is *Base*). Overloading is not only permitted for functions, but also for operators.

```
#include <iostream>

class Base{
   public:
      int count;
      Base() : count(0) {}

      void operator ++ (){
           count = count + 1;
      }
};

int main(){

   Base A;
   Base B;

   ++A;
   ++B;

   std::cout << A.count << std::endl;
   std::cout << B.count << std::endl;

   return 0;
}
```

By specifying the *operator* keyword and which operator we wish to overload, we can now iterate the attribute *count* of class *Base* using the usual *++* operator. We introduced the topic of encapsulation earlier when we inherited a base class publicly. Additionally, we can alter these permissions using *private*, *friend*, and *protected* keywords. Simple annotated examples are provided in 1-classes10.cpp

- 1-classes13.cpp. Lastly, we will briefly introduce templates in C++. Templates are useful when we want to enable multiple types as arguments to a function without rewriting code / overloading. The template is expanded at runtime, like a macro, and will substitute the appropriate type for the function arguments.

```
#include <iostream>
#include <string>

template <class T>
void print(T& a){
   std::cout << a << std::endl;
}

int main(){

      int x = 0;
   double y = 1.234;

   std::string s = "Hello World";

   print<int>(x);
   print(x);

   print(y);
   print(s);

   return 0;
}
```

Templates are declared above a function (or class if we want all functions of that class to have access to template parameters). The template specifies parameter T as a typename (i.e. class), the identity of which is not known until runtime. We can call the function and explicitly state the typename expected, *print<int>(x)* but this is not required.