University of Toronto

CSC 488 / CSC2107 Compilers and Interpreters **Winter 2022**

Tutorial 3: The introduction to C++

C++ is an object-oriented language. Instead of data structures and separate program structures, both data and program elements are combined into one structure called an object. The content of this tutorial is closely related to CSC488 assignment C++ data structures.

Class

Here is a simple example of Class.

```
class MyClass {        // The class
  public:              // Access specifier
    int myNum;         // Attribute (int variable)
    string myString;   // Attribute (string variable)
    void myMethod() {  // Method/function defined inside the class
      cout << "Hello World!";
    }
};
int main() {
  MyClass myObj;      // Create an object of MyClass
  myObj.myMethod();   // Call the method
  return 0;
}

```

this way of creating a class object will create the object in the stack.

Constructor & Destructor

```
class Car {         // The class
    public:           // Access specifier
      string brand;   // Attribute
      string model;   // Attribute
      int year;       // Attribute
      char *_text;
      // Constructor with parameters
      Car(string x, string y, int z) { // Constructor with parameters
        brand = x;
        model = y;
        year = z;
        _text = new char[100];
      }
      //Or you can do
      //Car(string x, string y, int z):brand(x),model(y),year(z) {
      //  _text = new char[z];
      //}
      //Destructor
      ˜Car(){
        // Deallocate the memory that was previously reserved
        // for this string.
```

String is a class in c++.

```
        delete[] _text;
      }
};

int main() {
    // Create Car objects and call the constructor with different values
    Car carObj1("BMW", "X5", 1999);
    Car carObj2("Ford", "Mustang", 1969);
    ...
    return 0;
}

```

Since the String is a class in c++, the attributes, model and brand will be deconstructed by calling their deconstructor automatically.

Access specifier

In Class, there are three access specifiers:

- public - members are accessible from outside the class
- private - members cannot be accessed (or viewed) from outside the class
- protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

Inheritance

- derived class (child) - the class that inherits from another class
- base class (parent) - the class being inherited from

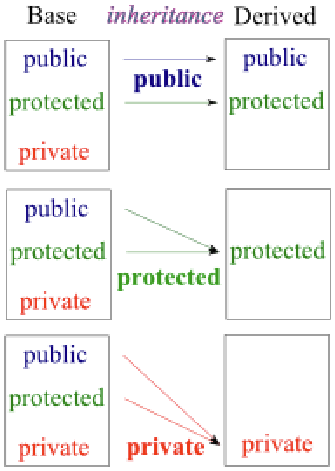
Here is a simple example:

```
// Base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};

// Derived class
class Car1: public Vehicle {
  public:
    string model = "X";
};

// Derived class
class Car2: public Vehicle {
  public:
    string model = "Y";
};

int main() {
  Car1 myCar1;
  myCar1.honk();
  Car2 myCar2;
  cout << myCar2.brand + " " + myCar2.model;
  return 0;
}
```

Access Control and Inheritance (under public inheritance):

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Difference among public, protected and private inheritance:



Function overriding

Here is a simple example:

```
// C++ program to demonstrate function overriding
class Base {
    public:
    void print() {
        cout << "Base Function" << endl;
    }
};
class Derived : public Base {
    public:
    void print() {
        cout << "Derived Function" << endl;
    }
};
int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

prints Derived method

How to call base function which is overridden?

```
class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};
class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;

        // call overridden function
        Base::print();
    }
};
int main() {
    Derived derived1;
    derived1.print();
    derived1.Base::print();

    // pointer of Base type that points to derived1
    Base* base1 = &derived1;
    // calls member function of Based class
    base1->print();
    return 0;
}
```

This will call the parent (or base) class’s print method.

Since the type of the “base1” is a Base pointer.

We found `base1->print()` is not actually overridden.

But sometimes, we want the base1 to print the child’s method, how do we do that?

Virtual function and overriding

Here is a simple example:

```
class Base {
    public:
    virtual void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
    public:
    void print() override{
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* base1 = &derived1;

    // calls member function of Derived class
    base1->print();

    return 0;
}
```

Here, we have declared the `print()` function of Base as virtual.
So, this function is overridden even when we use a pointer of Base type that points to the Derived object derived1.
From Official Cppreference website: ”As opposed to non-virtual functions, the overriding behavior is preserved even if there is no compile-time information about the actual type of the class.”

Here is a application how to use virtual function.

```
class Animal {
  private:
    string type;

  public:
    // constructor to initialize type
    Animal() : type("Animal") {}

    // declare virtual function
    virtual string getType() {
        return type;
    }
};

class Dog : public Animal {
  private:
    string type;

  public:
    // constructor to initialize type
    Dog() : type("Dog") {}

    string getType() override {
        return type;
    }
};

class Cat : public Animal {
  private:
    string type;

  public:
    // constructor to initialize type
    Cat() : type("Cat") {}

    string getType() override {
        return type;
    }
};

```

we can override the parent’s implementation by the keyword “override”.

```cpp
void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;
}

int main() {
    Animal* animal1 = new Animal();
    Animal* dog1 = new Dog();
    Animal* cat1 = new Cat();

    print(animal1);
    print(dog1);
    print(cat1);

    return 0;
}
```

It will call their own getType() method.

What value will be printed in stdout as following?

```cpp
class A{
    public:
    virtual int getType();
    virtual int getVal();
};
class B: public A{
    public:
    int getType() override;
    int getVal() override;
};
int A::getType(){
    return this->getVal();
}
int A::getVal(){
    return 3;
}
int B::getType(){
    return A::getType();
}
int B::getVal(){
    return 4;
}

int main(){
    B b;
    A* ptr = &b;
    cout<<ptr->getType();
    return 0;
}
```

Since these two overriden functions do not have any implementation body, they will invoke the default constructor.

refers to the current object that calls this method. So since in this case, the object that calls this method is of type class B, “this”refers to class “B” and so the “getVal()” of class B will be called, which returns 4.

ptr is an object of B. So ptr->getType() calls the B’s getType() method, which it then calls A::getType(). Then inside the A::getType(), it says whoever called me is “this” and so it calls the this.getVal(). Here “this” is an object of B, which means we call B’s getVal() which returns 4.

Unique Pointer

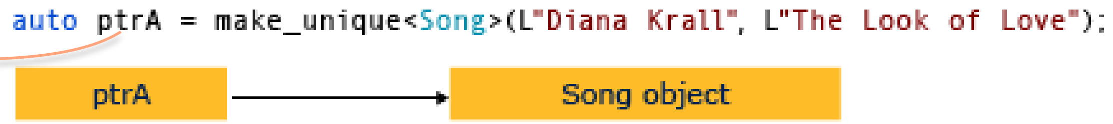
std::unique_ptr is a smart pointer that owns and manages another object through a pointer and disposes of that object when the unique_ptr goes out of scope. Unique pointer will be used in Assignment 5 and 6. Note that:

- It cannot be copied to another unique_ptr

- It can only be moved using `std::move`

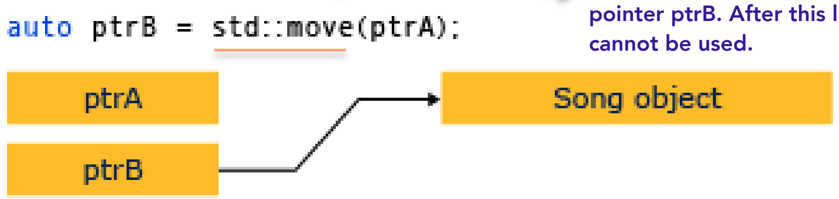
Here shows unique pointer behavior.

ptrA's type is explicitly a unique pointer to the object of class Song.



```
auto ptrA = make_unique<Song>(L"Diana Krall", L"The Look of Love");
```

returns the pointer that the ptrA is referring to and changes the ownership of the pointer to the pointer ptrB. After this line ptlA is useless and cannot be used.



```
auto ptrB = std::move(ptrA);
```

In general, `std::move` will convert lvalue to rvalue reference. Here is an example of `std::unique_ptr`.

std::move change the ptrA which is a lvalue to a rvalue reference so that it can used to be asainged to prtB.

lvalue == a value in memory

rvalue == a value in cpu's registers and do not have a memory allocate for them

```
int main()
{
    std::unique_ptr<int> valuePtr(new int(15));
    std::unique_ptr<int> valuePtrNow(std::move(valuePtr));
    std::unique_ptr<int> valuePtrThen = std::move(valuePtrNow);
    std::cout << "valuePtrThen = " << *valuePtrThen << "\n";
}
```

Notes:

- Before doing following assignments, try to get the class and its derived structure of skeleton code.
- Get to know how to visit each AST node.
- A3 will be released after Feb.6.