Reading assignments

Monday, May 10, 2021 8:56 AM

I. Constructor, destructor and object copying

-**Constructor** are methods that are called when an object is created, and tells C++ what to do when creating a new object

- **A destructor** is called when an object is deleted. Has the form ~class_name()
- We would like to call to a destructor when we want to deallocate some pointers used in the object to save memory.
- **Copy constructor** is a constructor but we use it when copying one object to another object of the same type. There are two behaviours of copying object:
  - Copy Constructor: when copy happens on the same line as instantiation
    ClassName obj1 = obj2;
  - Copy Operator: when copy happens after instantiation

Syntax for user-defined copy constructor:

ClassName (const ClassName& other){}

It's a constructor that take another object as parameter

II. Naming convention & getters setters

```
//global with camel case and underscore
const int global_car = 69;

class SomeClass{
private:
//pointers start with p
void* pSomething;
int bruh=22;

public:
    //method names start with verbs
    bool isString(){}
    bool hasChildren(){}
    void showThings(){}
    int getting_bruh()
    {
        return bruh;
    }
    void setting_bruh(int n)
    {
        this->bruh= n;
    }

    };
```

III. Virtual Inheritance / Multiple Inheritance

- In C++ we can inherit multiple class at once:

```
3   class Dad
4   {
5   public:
6       Dad()
7       {
8           cout<< "From the Dad"<<endl;
9
10      }
11  };
12  class Mom
13  {
14  public:
15      Mom(){
16      cout<<"From the Mom"<<endl;
17      }
18  };
19  class Child: public Dad,public Mom
20  {
21  public:
22      Child()
23      {
24          cout<<"This is the Child"<<endl;
25      }
26  };
27  int main()
28  {
29      Child child;
30  }
```

- Line 24 shows us that we can do multiple inheritance using the comma to separate the classes.
- Creating a Child object

```
int main()
{
    Child child;
}
```

- Ouput :

```
From the Dad
From the Mom
This is the Child
```

- So the constructor for Mom and Dad are both called by the Son

- If Mom and Dad both inherit a same class

```
class Human
{
public:
    Human()
    {
        cout<<"This is a human"<<endl;
    }

};
class Dad: public Human
{
public:
    Dad()
    {
        cout<< "From the Dad"<<endl;

    }
};
class Mom: public Human
{
public:
    Mom(){
    cout<<"From the Mom"<<endl;
    }
};
```

- If we then now run the Child class object again:

```
This is a human
From the Dad
This is a human
From the Mom
This is the Child
```

- We can see that the Human constructor are called twice.
- This is because it is needed to create both the mom and dad object for the Child

- Problem occurs when creating a function:

```
class Human
{
public:
    Human()
    {
        cout<<"This is a human"<<endl;
    }
    void greetings()
    {
        cout<<" Human say hello"<<endl;
    }

};
```

Using this function by the Child object:

```
43        child.greetings();
44    }
45
```

ogs & others

Build messages | CppCheck/Vera++ | CppCheck/Vera++ messages | Cscop

File	Line	Message
C:\Users\bu...	43	error: request for member 'greetings' is ambiguous

The problem has occured since the compiler doesn't know where should the **greetings()** come from, Dad or Mom, 'cause both of them have **greetings()**. This just created a little confusion in the program

- To solve the problem of abiguous, we just need to tell C++ to do with the **greetings()** function and the core thing is a bout the **Human** class.

```
class Dad: public virtual Human
{
public:
    Dad()
    {
        cout<< "From the Dad"<<endl;

    }
};
class Mom: public virtual Human
{
public:
    Mom(){
    cout<<"From the Mom"<<endl;
    }
};
```

By putting the virtual keyword in dad's and mom's inheritance. We are saying that both should just have the same **Human** class object
- Output:

```
This is a human
From the Dad
From the Mom
This is the Child
 Human say hello
```

- Obviously, the **Human** constructor was just called 1 time, so there is just 1 function of greetings(), which could not confuse the compiler anymore.

Function and class template

- template is a keyword in C++, it represents abtract types of data like int, float, class,.....
- Why would we use template?
- Instead of writing overloading function for every methods for every types, we just need to write one template for all

Function template

```
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void swap(float &a, float &b) {
    float temp = a;
    a = b;
    b = temp;
}
```

these are two the same function but different in the functions' types

```
template<typename T>
void swap(T &a, T &b) {
    T temp = a;
    a = b;
    b = temp;
}
```

instead of writing 2 functions, we just need 1 template function

```
#include <iostream>
using namespace std;
template<typename T>
void Swap(T &a, T &b) {
    T temp = a;
    a = b;
    b = temp;
}
int main()
{
    int a = 3, b = 5;
    Swap(a, b);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
}
```

```
a: 5
b: 3
```

```
#include <iostream>
using namespace std;
template<typename T, typename X>
void Swap(T &a, X &b) {
    T temp = a;
    a = (T)b;
    b = (X)temp;
}
int main()
{
    int a = 3;
    float b = 5.5f;
    Swap(a, b);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
}
```

```
a: 5
b: 3
```

Class template

Example: creating a class called Point containing x,y with whatever type

```cpp
template<class Type>
class Point {
private:
    Type x;
    Type y;
public:
    Point(Type x, Type y){
        this.x = x;
        this.y = y;
    }
    Point(){}
    void printPoint() {
        cout << "x: " << x;
        cout << "y: " << y;
    }
};

int main()
{
    Point<int> p(5, 10);
    p.printPoint();
}
```

```
x: 5
y: 10
```

Upcasting and down casting

Upcasting is converting a derived-class reference or pointer to a base-calss

- Upcasting allows us to treat a derived type as though it were its base type.
- it is always allowed for public inheritance, without explicit type cast

Downcasting is the opposite process, converting a base-class pointer to a derived-class pointer.

- Downcasting is not allowed without an explicit type cast

```cpp
class Parent {
public:
  void sleep() {}
};

class Child: public Parent {
public:
  void gotoSchool(){}
};

int main( )
{
  Parent parent;
  Child child;

  // upcast - implicit type cast allowed
  Parent *pParent = &child;

  // downcast - explicit type case required
  Child *pChild =  (Child *) &parent;

  pParent -> sleep();
  pChild -> gotoSchool();

  return 0;
}
```

As in the example, we derived **Child** class from a **Parent** class, adding a member function, **gotoSchool()**. It wouldn't make sense to apply the **gotoSchool()** method to a **Parent** object.

Because a **Parent** isn't a **Child** (a **Parent** need not have a **gotoSchool()** method), the downcasting in the above line can lead to an **unsafe** operation.

Here, this example upcast, a pointer to parent created by copied from a reference of current "child". So this pointer from the "child" could be used as a pointer in the "parent" class

And because "Parent" is not a "Child", we could not generate a "Child" pointer as a copied from a "Parent" pointer. Hence, we must force the pointer from "Parent" to be a type of "Child" pointer, this pointer now becomes a 'fake' type pointer which could be copied to type Child pointer.

C++ provides a special explicit cast called **dynamic_cast** that performs this conversion

- Downcasting is the opposite of the basic object-oriented rule, which states objects of a derived class, can always be assigned to variables of a base class.

- Because **implicit upcasting** makes it possible for a base-class pointer to refer to a base-class object,there is the need for **dynamic binding**. That's why we have **virtual** member functions.

-
  1. **Pointer (Reference)** type: known at **compile** time.
  2. **Object** type: not known until **run** time.

Dynamic Casting

The **dynamic_cast** operator answers the question of whether we can **safely** assign the address of an object to a pointer of a particular type.

```cpp
#include <string>

class Parent {
public:
  void sleep() {
  }
};

class Child: public Parent {
private:
  std::string classes[10];
public:
  void gotoSchool(){}
};

int main( )
{
  Parent *pParent = new Parent;
  Parent *pChild = new Child;

  Child *p1 = (Child *) pParent;  // #1
  Parent *p2 = (Child *) pChild;  // #2
  return 0;
}
```

Type cast #1 is not safe because it assigns the address of a base-class object (**Parent**) to a derived class (**Child**) pointer. So, the code would expect the base-class object to have derived class properties such as **gotoSchool()** method, and that is false. Also, **Child** object, for example, has a member **cl**Type case #2, however, is safe because it assigns the address of a derived-class object to a base-class pointer. **asses** that a **Parent** object is lacking.

```cpp
void f(Parent* p) {
  Child *ptr = dynamic_cast<Child*>(p);
   if(ptr) {
    // we can safely use ptr
  }
}
```

In the code, if **(ptr)** is of the type **Child** or else derived directly or indirectly from the type **Child**, the **dynamic_cast** converts the pointer **p** to a pointer of type **Child**. Otherwise, the expression evaluates to **0**, the null pointer.

Void keyword

- Void as a return type tells the function to not expect any returns.
- Void as a parameter tells the function not to expect any parameter, though this is implied implicitly.
- Void pointers can only store(or point) to a memory address but can't dereference them because they don't know what data type it is.
-