Upcasting and downcasting

Friday, May 7, 2021 10:37 AM

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.

II. Const keyword

- The const keyword is a promise you made to C++ that you won't change anything to that variable.

example:

```cpp
int main()
{
    const int x =2;
    x++;
}
```

Const before and after the * symbol in pointers

```cpp
int main()
{
    int y = 9;
    int *x = new int;
    x = &y;
    (*x)++;
    cout << *x <<'\n';
}
```

What we are doing is creating a pointer to heap of type int and pointing it to y then incrementing the value it is pointing to.

```cpp
int main()
{
    int y = 9;
    const int *x = new int;
    x = &y;
    (*x)++;
    cout << *x <<'\n';
}
```

The word const before the * means that the address that it is pointing to &y is now read-only. Therefore we can't change y's value through the x pointer anymore

```cpp
int main()
{
    int y = 9;
    const int *x = new int;
    x = &y;
    // (*x)++;
    y++;
    cout << *x <<'\n';
}
```

No error

1. Note: const int *x and int const *x are identical C++ just care if the const is before or after *.
2. Now we put const behind *:

```cpp
int main()
{
    int y = 9;
    int * const x = new int;
    x = &y;
    (*x)++;
    cout << *x <<'\n';
}
```

1.

2. What we are doing is creating a pointer that can only point to that specific memory address, it is locked and can't change what it's pointing to &y.
3. We can still change the value it is pointing to:

```cpp
int main()
{
    int y = 9;
    int * const x = &y;
    // x = &y;
    (*x)++;
    cout << *x <<'\n';
}
```

4.
```
10
```

5. Const with class and method

- Usually used with the get() method, only works in a class.

```cpp
using namespace std;

class Shape
{
    private:
    int x;
    public:
    Shape(int x): x(x){}
    int get_x() const
    {
        return this->x;
    }
};

int main()
{
    Shape S1(3);
    cout <<S1.get_x()<<'\n';
}
```