

28 Virtual Functions in C++

1. 虚函数

虚函数可以让我们在子类中重写方法

举例：B是A的子类，如果我们如果在A类里新建一个方法并把它标记为虚函数，我们可以再B类中重写这个方法让它去做其他事情。

```
C++

#include <iostream>
#include <string>

class Entity
{
public:
    std::string GetName() { return "Entity"; }
};

class Player : public Entity
{
private:
    std::string m_Name;
public:
    Player(const std::string& name)    // 这里25课学到的构造函数
        : m_Name(name) {}

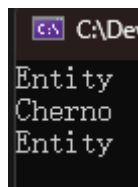
    std::string GetName() { return m_Name; }
};

int main()
{
    Entity* e = new Entity();
    std::cout << e->GetName() << std::endl;

    Player* p = new Player("Cherno");
    std::cout << p->GetName() << std::endl;

    Entity* entity = p;
    std::cout << entity->GetName() << std::endl;

    std::cin.get();
}
```



```
C:\Dev
Entity
Cherno
Entity
```

这里虽然我们指向的是一个Entity *pointer*，但是它实际上是一个Player类的实例。

一个更好的例子

C++

```
void PrintName(Entity* entity)
{
    std::cout << entity->GetName() << std::endl;
}

int main()
{
    Entity* e = new Entity();
    PrintName(e);                                //Entity
    Player* p = new Player("Cherno");
    PrintName(p);                                //Entity

    std::cin.get();
}
```

原因是如果我们在类中正常声明函数或者方法，当它去调用这个方法的时候，它总会去调用属于这个类型的方法

这里我们希望C++编译器知道我们要调用的第二个传入的其实是一个Player，所以请调用Player的GetName。这就是要用到`virtual function`虚函数的地方了。

虚函数引入了一种要动态分派的方法，一般通过`vtable`(虚表)来实现编译。

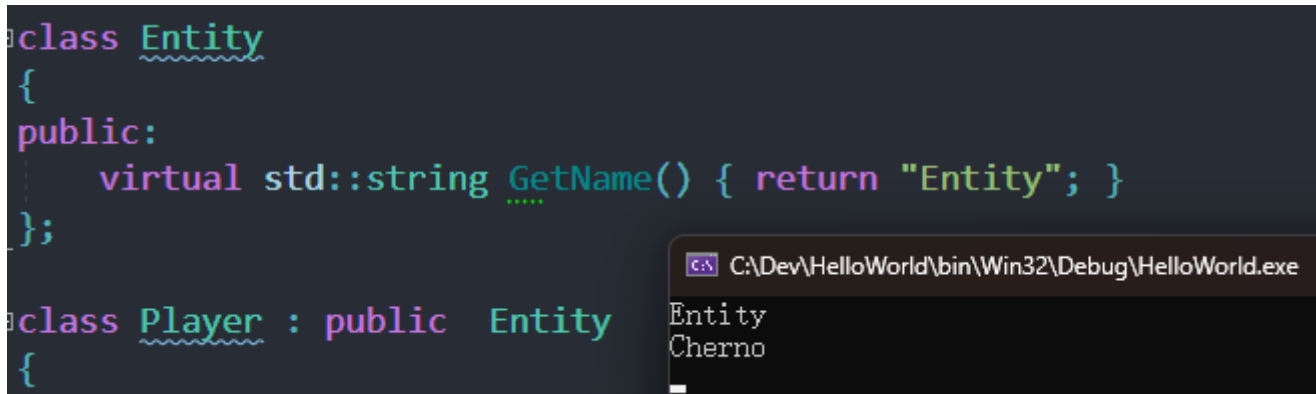
虚表就是一个包含类中所有虚函数映射的列表，通过虚表我们就可以在运行时找到争取的被重写的函数。

简单来说，就是如果你想重写一个函数，那么你必须把基类中的原函数设置为虚函数。

C++

```
class Entity
{
public:
    virtual std::string GetName() { return "Entity"; }    //设置为虚函数方法
};
```

现在就可以正确打印Entity和Cherno了



The screenshot shows a C++ IDE with a dark theme. The code editor displays the following code:

```
class Entity
{
public:
    virtual std::string GetName() { return "Entity"; }
};

class Player : public Entity
{
    // ... (code is partially obscured) ...
}
```

Below the code editor, a terminal window shows the output of the program:

```
C:\Dev\HelloWorld\bin\Win32\Debug\HelloWorld.exe
Entity
Cherno
```

C++11新标准允许给被重写的函数用`override`关键字标记

C++

```
std::string GetName() override { return m_Name; }
```

不加也没报错，但是加了可读性更好，让我们直观地知道这个函数被重写了，还能帮助我们预防bug，比如拼写错误等。

```
std::string Getname() override { return m_Name; }
```

Function 'std::string Player::Getname()' has the 'override' specifier but does not override a base class member

比如这里N写成小写n，就有报错，因为基类中没有一个这样的函数给我们重写

2. runtime

虚函数不是没有成本的，有两种虚函数运行时的花费。

一种就是需要额外的内存用来存储虚表，这样我们就可以分配到正确的函数，基类里还有一个指针成员指向虚表。

另一种就是每次调用虚函数的时候，我们必须要遍历虚表去找到最终要运行的函数，这也是额外的性能损失。

除非是CPU很差的嵌入式平台，不会告诉你说：因为性能问题而别去用虚函数，因为它造成的影响很小，你可能根本察觉不到。