53 Templates in C++

1. 什么是模板

C++中的*templates*（模板）要比其它语言中的泛型要更强大。模板有点像*macro*（宏），可以让做基本任何事。而泛型却非常受制于类型系统和其它因素。

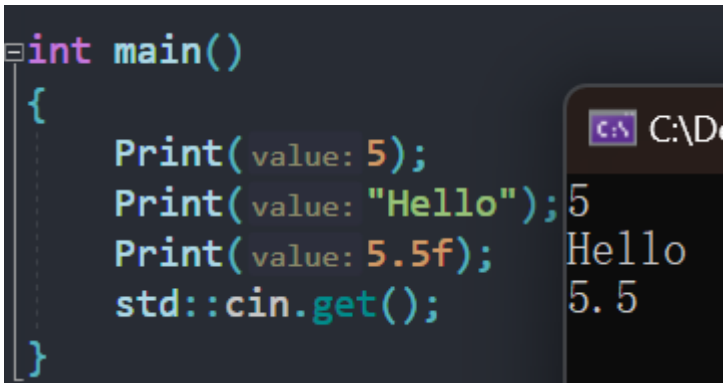
模板允许你定义一个根据你的用途进行编译的模板，你可以让编译器基于一套规则帮你写代码。例如我写一个函数时，可以在这里面使用模板，我实际做的类似创建了一个*blueprint*（蓝图），当我决定要调用这个函数时，我可以根据我要如何使用这个函数来指定特定的参数，这个参数决定了要放入到模板中的实际代码。

```C++
#include <iostream>
#include <string>

void Print(int value)
{
    std::cout << value << std::endl;
}

void Print(std::string value)  //一次函数重载
{
    std::cout << value << std::endl;
}

int main()
{
    Print(5);
    Print("Hello");
    Print(4.3f);   // 这里要是再想打印一个float类型还得再重载一次
    std::cin.get();
}
```

我们最终要手动定义三个Print重载函数，如果我们只需要定义一次这个函数呢?

2. 创建模板

```C++
template<typename T>  // 通常叫T
void Print(T value)
{
    std::cout << value << std::endl;
}
```



`template`表明这里定义了一个模板，会在编译阶段被评估，所以这并不是一个真的函数，只有我们实际调用它的时候，这些函数才被真正地创建。

`<typename T>`是模板参数，这个例子中我们用选择typename作为模板参数的类型，T作为名字，实际上是模板的参数名称，可以替换在例子中出现的类型。

```C++
Print<int>(5.5f); //可以指定类型，这里得到的结果是：5
```

再重复一遍，模板只有在它被调用时才会创建，否则它根本就不存在。你甚至在里面有语法错误，只要不调用都不会影响编译。（视频中的版本是这样的，但目前模板中的错误就算不调用也会影响编译。）

实际上，编译时会对模板进行填空：

```C++
template<typename T>
void Print(T value)
{
    std::cout << value << std::endl;
}

void Print(int value)  //等价于复制粘贴了一个这个函数，再把T换成了int，得到了长一个真正的函数
{
    std::cout << value << std::endl;
}
```

模板绝不仅仅局限于类或者任何东西，也不会局限于函数，你可以基于模板创建一整个类。而且事实上，大量的C++标准模板库同样完全使用了模板。
这里有一个例子，这里我们不能用类型作为模板参数:

```C++
class Array //我想要一个数组，它的大小是在编译时决定的
{
private:
    int m_Array[];
};
```

这里不能直接输入一个变量size之类的，因为这是一个栈分配的数组，所以在编译时就必须知道它，而模板正是在编译时评估处理的。

```C++
template<int N>
class Array
{
private:
    int m_Array[N];
public:
    int GetSize() const { return N; }
};

int main()
{
    Array<5> array;
    std::cout << array.GetSize() << std::endl; // 5

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

所以可以看到，我们不是只能使用类型，我们可以使用整数或者其它的数据类型，来指定我们想要如何生成一个类和整个类。

让我们刚进一步，我想让数组的类型也是可变的，因此我希望能够在编译时指定这个数组实际包含的类型，我可以添加另一个模板参数。

```C++
template<typename T,int N>
class Array
{
private:
    T m_Array[N];
public:
    int GetSize() const { return N; }
};

// 调用
Array<int,5> array0;
Arary<std::string,50> array1;
```

可以看到我们创建的是一个可以自动创建的类，实际上这里与标准数组类在C++标准模板库中的工作方式非常相似（有2个模板参数，type和size）。这有点像是C++的*meta programming*（元编程），相较于代码运行时的实际编程，我们像是在编译器在编译时进行编程。

3. 哪里用模板？哪里不用

本部分完全是主观的，仅供参考。
很多游戏工作室或软件公司实际上禁止使用模板，但模板非常有用，比如在日志系统或者其它的使用场景下，你想记录每一种可能的类型，难道你真的要为每个函数都进行重载吗？你可以用模板自动完成，也就是你可以让编译器基于一些规则为你写代码。这是说，你可以深入使用模板来做一些很好的事，但如果你的模板变得非常复杂，你开始让它为你生成一个完整的元语言就有点过头了。因为这里有一个平衡点，当模板变得越来越复杂时，没人能搞懂它是做什么的了，你得花大量时间弄清楚哪些代码已经被编译了以及你的模板到底发生了什么而不能工作。这种情况说明你做的过头了。

Cherno的观点是：手动做一些事，自己写代码实际上会让你和你的团队受益更多，而不是试图创建庞大的模板魔法般地为你做所有事。所以模板不应该被完全禁止，但也不应该滥用。比如游戏引擎的日志系统和材质系统肯定会用到模板，当你有一个可以包

含各种不同类型的统一缓冲区时，模板在一定程序上是非常有用的。