进入到整个C++学习的重中之重——**模(mu)板**

> 程序 = 算法 + 数据结构
>
> **数据结构：**能够存储任意类型
>
> **算法：**能够操作存储任意类型数据的数据结构。

泛型编程：将【任意类型】从程序设计中抽象化出来。

	泛型编程
面向过程编程	用**模板**实现函数过程
面向对象编程	用**模板**实现类

抽象化是剥离了原本的具体形式的这样一个过程。

- 面向过程抽象化的是问题求解过程中的参数：

```
int f(int x) {
    return x * x + 2 * x + 4;
}
int main() {
    cout << 1 * 1 + 2 * 1 + 4 << endl;
    cout << 2 * 2 + 2 * 2 + 4 << endl;
    cout << 3 * 3 + 2 * 3 + 4 << endl;
    cout << 4 * 4 + 2 * 4 + 4 << endl;
    return 0;
}
```

而**类型**才是**模板**的参数。

另外只要有template关键字，这玩意就是模板。

模板函数

```
template<typename T>
T add(T a, T b) {
    return a + b;
}
int main() {
    cout << "add(3, 4) = " << add(3, 4) << endl;
    cout << "add(3.1, 4.2) = " << add(3.1, 4.2) << endl;
    return 0;
}
```

模板函数是创造函数的函数。

`typename`关键字是用来说 T 是一个类型的，除了`typename`以外，还可以打上`class`关键字。他俩的作用是一模一样的。但保不齐未来可能会有区别

模板函数是如何工作的？

编译器会**实例化出来一段代码**，在实例化出来的这段代码中，T就是int，在编译阶段自动生成的这段代码才是真正被调用的方法。

这个过程是不是很**像宏？**

这种机制可以看成是C++重载的一种高级形式。

```
nm -C a.out
```

可以通过查看可执行程序中的若干个符号的定义：

```
0000000000001100 T _start
0000000000004010 D __TMC_END__
00000000000012aa t __static_initialization_and_destruction_0(int, int)
0000000000001328 W double test1::add<double>(double, double)
0000000000001310 W int test1::add<int>(int, int)
00000000000011e9 T test1::main()
                 U std::ostream::operator<<(double)@@GLIBCXX_3.4
                 U std::ostream::operator<<(int)@@GLIBCXX_3.4
                 U std::ostream::operator<<(std::ostream& (*)(std::ostrea
m&))@@GLIBCXX_3.4
                 U std::ios_base::Init::Init()@@GLIBCXX_3.4
                 U std::ios_base::Init::~Init()@@GLIBCXX_3.4
0000000000004040 B std::cout@@GLIBCXX_3.4
                 U std::basic_ostream<char, std::char_traits<char> >& std
::endl<char, std::char_traits<char> >(std::basic_ostream<char, std::char_
traits<char> >&)@@GLIBCXX_3.4
0000000000002008 r std::piecewise_construct
```

只有当模板中的**参数推导不存在冲突**。才能正确生成实例。

方法一：可以显示性地告诉编译器T的类型：

```
cout << add<double>(3, 4.2) << endl;
```

方法二：增加类型。

`decltype(3 + 4.2)` ：根据表达式来推导类型。**但是并没有进行表达式的执行**。

```
template<typename T, typename U>
decltype(T() + U()) add(T a, U b) {
    return a + b;
}
```

但是上面这段代码有bug：如果没有默认构造函数，

```
class A {
public:
    A() = delete;
    A(int x) : x(x) {}
    int x;
};

class B {
public:
    B() = delete;
    B(int x) : x(x) {}
    int x;
};
class C {
public:
    C() = delete;
    C(int x) : x(x) {}
    int x;
};
C operator+(const A &a, const B &b) {
    return C(a.x + b.x);
}
C operator+(const B &b, const A &a) {
    return C(a.x + b.x);
}
ostream &operator<<(ostream &out, const C &c) {
    out << "Class C.x = " << c.x;
    return out;
}
template<typename T, typename U>
decltype(T() + U()) add(T a, U b) {  //默认构造被删除，程序会报错。
    return a + b;
}
int main() {
    A a(56);
    B b(78);
    cout << "add(A, B) = " << add(a, b) << endl;
    return 0;
}
```

解决方法：**auto关键字/返回值后置**

返回值后置

C++11提出的新的语法结构，**返回值后置**：允许返回值写到函数后面去，但是由于前面也不能空

着，可以用auto关键字来进行占位。

```
//当代码到了后面这一部分的时候是可以访问a,b变量的。但是在前面是访问不到的，这是最大的差别：
template<typename T, typename U>
auto add(T a, U b) -> decltype(a + b) {
    return a + b;
}
```

为什么难，难在模板知识在之前所有知识的结合应用。

模板类

在类的前面加上template关键字。

```
//可以打印任意类型的模板类的函数对象：
template<typename T>
class PrintAny {
public:
    PrintAny(ostream &out) : out(out) {}
    PrintAny &Print(T a) {
        out << a;
        return *this;
    }
    PrintAny &endl() {
        cout << std::endl;
        return *this;
    }
private:
    ostream &out;
};

int main() {
    PrintAny<int> pint(cout);
    PrintAny<double> pdouble(cout);
    PrintAny<string> pstring(cout);
    pint.Print(3).endl();
    pdouble.Print(3.3).endl();
    pstring.Print("hello world").endl();
    return 0;
}
```

但我们希望能传任意类型的参数，所以实际上我们要的不是模板类，我们要的是模板成员属性方

法。

```
//可以打印任意类型的模板类的函数对象：
class PrintAny {
public:
    PrintAny(ostream &out) : out(out) {}
    template<typename T>
    PrintAny &Print(T a) {
        out << a;
        return *this;
    }
    PrintAny &endl() {
        cout << std::endl;
        return *this;
    }
private:
    ostream &out;
};

int main() {
    PrintAny p(cout);
    p.Print(3).endl().Print(3.3).endl().Print("hello world").endl();
    return 0;
}
```