多态

多态

一种形式，多种状态

多态的类型

静态多态（函数重载、运算符重载）

动态多态（虚函数）

虚函数

快乐的动物园

进动物园之前：**zoo-1.cpp**

进动物园之后：**zoo-2.cpp**

注意：赋值兼容规则（P270）

问题：dog、cat都当作animal看待，缺乏个性。

解决方法1：静态多态

基类不变；修改派生类，增加不同的成员函数。

```
say_hello_dog_version() { ... }

say_hello_cat_version() { ... }
```

改动太大，调用也不方便。

解决方法2：动态多态

派生类不变；修改基类，把成员函数申明为虚函数。

```
virtual void say_hello() { ... }
```

只需要这一点改动，即可解决问题：**zoo-3.cpp**

分析原因，对比两种情况下的函数调用指令：

```
(gdb) x/10i $pc
call   0x7ff7c0d02b40 <_ZN6animal9say_helloEv>
call   *%rdx
```

幕后黑手 —— vtable

在调试器中观察：

```
(gdb) p nobody
(gdb) p sizeof(nobody)
(gdb) p wangcai
(gdb) p sizeof(nobody)
(gdb) p jiafei
(gdb) p sizeof(jiafei)
```

问题1： 多出多少字节？多出的字节在什么位置？

问题2： 多出的字节有什么作用?

vptr：指向vtable

vtable：存放虚函数的地址

问题3： 如何查看vtable的内容?

获得vtable的地址：

```
(gdb) p wangcai
... 0x7ff7df1d4670 <vtable>
```

查看vtable的内容：

```
(gdb) help x
... 1xb 1xh 1xw 1xg ...
(gdb) x/1xg 0x7ff7df1d4670
... 0x7ff7df1d2a70
```

对比成员函数：

```
(gdb) p wangcai.say_hello
... 0x7ff7df1d2a70 <dog::say_hello()>
```

可知它是dog类的成员函数say_hello的地址。

同样的方法，查看nobody对象：

```
...
(gdb) p nobody
... 0x7ff7df1d4690 <vtable>
(gdb) x/1xg 0x7ff7df1d4690
... 0x7ff7df1d2b80
(gdb) p nobody.say_hello
... 0x7ff7df1d2b80 <animal::say_hello()>
```

同样的方法，查看jiafei对象：

```
...
(gdb) p jiafei
... 0x7ff7df1d4650 <vtable>
(gdb) x/1xg 0x7ff7df1d4650
... 0x7ff7df1d2950
(gdb) p jiafei.say_hello
... 0x7ff7df1d2950 <cat::say_hello()>
```

结论： vtable中存放了各对象自己的虚函数的地址。

思考： 当存在多个虚函数时，如何查看（zoo-4.cpp）？

```
(gdb) p wangcai
... 0x7ff7f37f4690 <vtable>
(gdb) x/3xg 0x7ff7f37f4690
0x00007ff7f37f2b10
0x00007ff7f37f2ab0
0x0000000000000000
(gdb) p wangcai.say_hello
(gdb) p wangcai.say_food
```

结论： 每个对象的vptr字段指向该对象的vtable，vtable中保存了该对象的所有虚函数的地址。

问题： 是否每个对象都有一个独立的vtable?

```
dog dog1("Dog1", 2021);
dog dog2("Dog2", 2022);
cat cat1("Cat1", "BLACK");
cat cat2("Cat2", "WHITE");
```

思考：

现实中，存在animal类型的对象吗？

需要对animal类型进行实例化吗？

需要调用animal类的say_hello函数吗？

需要定义animal类的say_hello函数吗？

animal类的say_hello函数多余吗？能删除吗？

解决方法： 纯虚函数

纯虚函数

纯虚函数的定义

```
virtual void say_hello() = 0;
```

抽象类： 拥有纯虚函数的类称为抽象类。抽象类不能被实例化，只能作为其它类的基类（zoo-5.cpp）。

一个抽象类的派生类需要实现这个抽象类的全部纯虚函数，才能被实例化。否则，这个派生类仍然是抽象类，仍然不能实例化。

接口类： 当一个抽象类的所有成员函数都是纯虚函数时，这个抽象类又称为接口类。

模板

模板：一种更加抽象的静态多态。

场景：多个函数，功能相似，参数类型不同。

解决方法：函数重载

addone-1.cpp

利用强制类型转换，可少写几个函数，减少代码量。

删除第1、2、3个函数.....

删除第2、3、4个函数.....

删除第2、3个函数.....

删除第3个函数.....

都不够简化，也不够优雅。

解决方法：函数模板

函数模板

addone-2.cpp

函数模板最终会被“实例化”为模板函数。

一个函数模板，会被“实例化”为多个模板函数 —— 多态。

一个函数模板“实例化”为多个模板函数的过程在编译时完成的，运行时不会改变。 —— 静态多态。

用nm观察函数模板实例化的结果。

```
nm test.exe | find "addone"
004027e0 T __Z6addoneIcET_S0_
0040280c T __Z6addoneIdET_S0_
0040283c T __Z6addoneIfET_S0_
00402860 T __Z6addoneIiET_S0_
```

结论：一个函数模板“实例化”为多个模板函数的过程在编译时完成的。

函数模板 vs 函数重载

addone-1.cpp

函数重载：一个函数有多个实现 —— 多态。
函数与实现之间的对应在编译时完成，运行不会改变。 —— 静态多态。
用nm观察函数重载的结果。

```
nm test.exe | find "addone"
004015c0 T __Z6addonec
0040162e T __Z6addoned
0040160b T __Z6addonef
004015e9 T __Z6addonei
```

总结

如果把函数重载，看作你“手动”实现多个功能相似的函数，那么，可以把函数模板，看作由编译器“自动”帮你实现多个功能相似的函数。

函数模板也可以和函数重载配合使用，适应更加灵活的应用场景。

类模板

场景：多个类，功能相似，参数类型不同。

解决方法1：手工实现多个类（不够简化、不够优雅）

解决方法2：使用类模板，让编译器“自动”帮你实现多个类

node.cpp

用nm观察类模板实例化的结果。

```
nm test.exe | find "node"
004028a0 T __ZN4nodeIfE6appendEPS0_
004028e4 T __ZN4nodeIfE8get_nextEv
004028f8 T __ZN4nodeIfE9get_valueEv
00402908 T __ZN4nodeIfE9set_valueEf
00402920 T __ZN4nodeIfEC1Ev
00402940 T __ZN4nodeIiE6appendEPS0_
00402984 T __ZN4nodeIiE8get_nextEv
00402998 T __ZN4nodeIiE9get_valueEv
004029a8 T __ZN4nodeIiE9set_valueEi
004029c0 T __ZN4nodeIiEC1Ev
```

每个类都拥有自己的构造函数和成员函数版本。

总结

函数模板、类模板，都是由编译器“自动”帮你实现功能相似的代码。

标准模板库

泛型程序设计（Generic Programming）是一种程序设计风格，

泛型允许程序员在编写代码时使用一些以后才指定的数据类型。

《设计模式（Design Patterns）》一书将泛型称为：

参数化类型（Parameterized Type）。

很多程序设计语言和编译器都支持泛型，

泛型在不同的程序设计语言中有不同的叫法。

Ada、Delphi、Eiffel、Java、C#、F#、Swift、Visual Basic .NET：泛型（Generics）

ML、Scala、Haskell：参数多态（Parametric Polymorphism）

C++、D：模板

使用泛型，可编写出不依赖于具体数据类型的程序，

从而将算法从特定的数据结构中抽象出来。

实现时：算法和数据结构分离，减少重复的代码量。

使用时：算法和数据结构组合，适应不同的应用场景。

C++ 的STL（Standard Template Library），即标准模板库，是泛型程序设计最成功的应用案例。

STL 是一个工业级的、高效的C++程序库。

STL 它包含了诸多在计算机科学领域里所常用的基本数据结构（如数组、向量、链表、队列、栈、树）和算法（如排序、查找）。这些数据结构和算法组合在一起，为我们的软件开发提供了良好的支持。

STL于1998年正式加入 C++ 标准。

Bjarne Stroustrup的《The C++ Programming Language》一书中，有1/3以上的篇幅讲述STL。

进一步的学习

The C++ Programming Language（Bjarne Stroustrup）

C++ Primer（Stanley B. Lippman）

课后复习

课程MOOC

www.icourse163.org/course/UESTC-1001774006

第九章 继承、派生与多态

第十章 模板、命名空间和异常处理

码图作业

matu.uestc.edu.cn

第9章 作业1、作业2

第10章 作业1、作业2