L3 封装与接口

L3 封装与接口

By 曹菡雯（计 03）、赵晨阳（软 01）、罗华坤（化 93）、李晨宇（材 01）

readme

这一部分主要是 L3 的部分课堂笔记整理，由小组四名同学共同完成。在课件的基础上，我们尽力做到了对于部分 PPT 都有全面的解读，同时联系前后课程的内容与课程作业，对于课件中的一些操作也有一定的扩展。

5 月 2 日更新：将原文档中所有代码重新进行了编写，有助于阅读。
5 月 3 日更新：修改了 5.1 节的笔误

如果阅读时间不够充足，建议阅读课堂的扩展部分。
3.3.2 优先匹配调用
5.1.2 追踪返回类型的函数
5.5 auto 字符例题
6.1 private 与 overload 的先后

一、函数重载

1.1 定义与意义

同一名称的函数，有两个以上不同的函数实现， 被称为“函数重载”。

1.2 区别方法

多个同名的函数实现之间，必须保证至少有一个函数参数的类型有区别，“这些同名函数的形式参数的个数、类型或者顺序必须不同”——返回值、参数名称等不能作为区分标识。

```
float f(int s) {return s / 2.0;}
int f(int s) {return s * 2;}
int main(){
cout << f(3) << endl;

//编译器应该调用哪个函数呢？

return 0;
}
```

为什么返回值不能作为区别？
编译器无法识别调用哪一个函数。

1.3 自动类型转换

1.3.1 定义与性质

如果函数调用语句的实参与函数定义中的形参数据类型不同，且两种数据

类型在 C++中可以进行自动类型转换（如 int 和 float，float 在自动转换成 int 时是向下取整），则实参会被转换为形参的类型。（关于自动类型转换在 L6 中有进一步阐述）

例如：

```
#include <iostream>
using namespace std;
void print(float score) {
cout << "score = " << score << endl;
}
int main() {
int a = 1;
print(a); // 此时会将 a 转换为 float 型
return 0;
}
```

又例如：

自动类型转换也可以通过自定义的类型转换运算符来完成。（在 L6 中阐述）

1.3.2 优先匹配调用

当函数重载时，会优先调用类型匹配的函数实现，否则才会进行类型转换。

例子：

```
#include <iostream>
using namespace std;
void print(int score) { cout << score << endl; }
void print(float score) { cout << score << endl; }
int main() {
float a = 1.0;
print(a);
return 0;
}
输出结果：1
```

这里看上去和优先匹配调用相互矛盾，其实并不是，这是由于 float 输出精度的问题。我们稍作修改：

```
#include <iostream>
#include <iomanip>
using namespace std;
void print(int score)
{ cout << "int = " << score << endl; }
void print(float score)
{ cout << "float = " <<fixed << setprecision(2)<<score << endl; }
int main()
```

```
{ float a = 1.0;
print(a);
// float = 1
return 0; }
```

输出结果；float=1.00

这里调用了 float 使用的控制精度的库，<iomanip>

二、参数缺省值

2.1 定义

函数参数可以在定义时设置默认值（缺省值），这样在调用该函数时，若不提供相应的实参，则编译自动将相应形参设置成缺省值。例如：

```
#include <iostream>
using namespace std;
void print(const char* msg = "hello") {
cout << msg << '#';
}
int main {
cout << "Beijing...";
print();
return 0;
}
```

输出：Beijing...hello#

2.2 语法

缺省值必须放在没有缺省值的参数之后，有多个缺省值时同理

2.3 缺省值保护

如果因为函数缺省值，导致了函数调用的二义性，编译器将拒绝代码。也就是说，参数类型可以发生自动类型转换的函数重载是合法的，因为编译器有调用的优先级；但是缺省值带来的参数类型重复却是不合法的。如下面代码，会导致编译不通过。

```
void fun(int a, int b=1) {
cout << a + b << endl;
}
void fun(int a) {
cout << a << endl;
}
//测试代码
```

```
fun(2);//编译器不知道该调用第一个还是第二个函数
```

例子：

int fun(int a) { ... }
选项中的函数不会与上述函数产生歧义的是（多选）

A **int fun(int b) { ... }**

B **float fun(int a) { ... }**

C **float fun(float a) { ... }**

D **int fun(int a, int b=1) { ... }**

E **int fun(int b, int a) { ... }**

A.不可根据形参名字与有无形参来区别函数。B.不可根据返回值类型不同来区别函数。D.缺省值造成二义性，编译失败。

例子二：

```
#include <iostream>
using namespace std;
int fun(int a=1) { return a+1; }
float fun(float a) { return a; }
int fun(int a, int b) { return a+b; }
int main() {
float a = 1.5;
int b = 2;
cout << fun(fun(a, b)) + fun(fun(a), b) << endl;
return 0;
}
```

fun(fun(a, b))：里层的 fun(a,b)调用 int fun(int a, int b)，直接对 float a 进行强制类型转换并向下取整，fun(1,2)，故而返回了 3 且为 int 类型。调用 int fun(int a=1)，返回 4。

fun(fun(a), b)：里层 fun(a)调用 float fun(float a)，返回 float 1.5，接着 fun(1.5, 2)，此处仅可以调用 int fun(int a, int b)，故而强制类型转换且向下取整，调用 fun(1,2)。返回 3。

综上，答案为 7。

三、auto 关键字与 decltype

3.1 作用与意义

3.1.1 自动确定变量的类型

由编译器根据上下文自动确定变量类型

例如：

```
auto i = 3;
//i 是 int 型变量
auto f = 4.0f;
//f 是 float 型变量
auto a('c');
//这句话等价于 auto a='c';

//a 是 char 型变量
auto b = a;
//b 是 char 型变量
auto *x = new auto(3);
//x 是 int*
```

3.1.2 追踪返回类型的函数

可以将函数返回类型的声明信息放到函数参数列表后进行声明。

```
普通函数声明形式
int func(char* ptr, int val);
追踪返回类型的函数声明形式
auto func(char* ptr, int val) -> int;
追踪返回类型在原本函数返回值的位置使用 auto 关键字
```

在这一例子中，auto 关键字并没有实质作用，然而在泛型编程中，auto 有着巨大的作用。

在模板类型推导过程中，比如下列代码：

```
template<typename T1, typename T2>
decltype(t1+t2) Sum(T1&t1, T2& t2)
{return t1+t2;}
```

在上面这个函数定义中，decltype 无法推导出 t1+t2 的类型，因为编译器是从左向右处理的，当处理到 decltype 的时候，编译器还不知道 t1+t2 的类型。追踪返回函数就是为了解决这个问题而生的，上面的函数我们可以声明如下：

```
template<typename T1, typename T2>
auto Sum(T1&t1, T2& t2) -> decltype(t1+t2)
{return t1+t2;}
```

这样的话，decltype 就可以根据 t1，t2 的类型推导出函数 Sum 的返回类型。

3.1.3 auto 的进一步阐述

auto 并不能代表一个实际的类型声明，只是一个类型声明的“占位符”。使用 auto 声明的变量必须马上初始化，以让编译器推断出它的类型，并且在编译时将 auto 占位符替换为真正的类型。

3.2 auto 其他性质

auto 变量必须在编译期确定其类型

auto 变量必须在定义时初始化 ： auto a; //错误，未初始化

同一个 auto 关键字应将变量推导为同一类型：
auto b4 = 10, b5 = 20.0, b6 = 'a';
//错误,没有推导为同一类型

参数不能被声明为 auto ： void func(auto a) {…} //错误

auto 并不是一个真正的类型。不能使用一些以类型为操作数的操作符，如 sizeof 或者 typeid：cout << sizeof(auto) << endl;//错误

3.3 decltype

配合 auto 一同使用，主要用于泛型编程

```
#include <bits/stdc++.h>
using namespace std;
struct
{
   char *name;
} anon_u;
struct
{
   int d;
   decltype(anon_u) id; //没有告诉 id 的类型，用 decltype 自动推导

} anon_s[100];          //匿名的 struct 数组
int main()
{
   decltype(anon_s) as;//注意 as 的类型。
cin >> as[0].id.name;
   return 0;
}
```

第一个 decltype 的理解：编译器根据 anon_u 的结构推导出一个类型，并

创建了这个类型的新变量 id。

第二个 decltype 的理解：编译器根据 anon_s 的结构推导出了一个类型，这个类型是某个匿名的结构体数组。并创建了这个结构体数组类型的新变量 as，as 也是一个结构体数组。（如果 decltype 括号里面的是一个数组，那么推导出的类型也是个数组，而不是这个数组里每一个元素的类型）

配合 auto 推导出返回值类型

```
auto func(int x, int y) -> decltype(x+y)
{
return x+y;
}
```

C++14 中不再需要显式指定返回类型

```
auto func(int x, int y)
{ return x+y; }
```

3.4 auto 的优势

用于代替冗长复杂、变量使用范围专一的变量声明。我们现在学习的类型都并不复杂，随着模板的学习，类型会越发复杂。

```
std::vector<std::string> vs;
for (std::vector<std::string>::iterator
i = vs.begin(); i != vs.end(); i++)
{ //... }
等价于：
std::vector<std::string> vs;
for (auto i = vs.begin(); i != vs.end();i++)
{ //.. }
```

有时候我们不能直接确定模板函数的返回值的类型，则可在定义模板函数时，用于声明依赖模板参数的变量类型。

```
template <typename _Tx, typename _Ty>
void Multiply(_Tx x, _Ty y)
{ auto v = x*y; //临时变量
std::cout << v; }
```

结合 auto 和 decltype，自动追踪返回类型

```
template <typename _Tx, typename _Ty>
auto multiply(_Tx x, _Ty y)->decltype(x*y)
{
return x*y;
}
```

5.5 auto 字符例题

以下语句能够编译的有（多选）

A `auto x = new int[10];`

B `auto *x = new int[10];`

C `auto x = "123";`

D `auto *x = "123";`

提交

该题非常详细的解答在 https://github.com/thu-coai/THUOOP/issues/12

以下能够在C++14下正确编译的有（多选）

A `for(auto x : "123") { ... }`

B `void f(auto x) { ... }`

C `auto f(int x) { ... }`

D `auto f(auto x) { ... }`

题目并不难，但是此处的遍历这很有讲究。

```
#include <iostream>
using namespace std;
int main() {
for(auto x : "123") {cout<<x<<'-';}
   return 0;
}
```

输出 1-2-3--

注意到 3 之后有两个-。我们更换循环方式：

```
#include <iostream>
using namespace std;
int main() {
for(auto x : {1,2,3}) {cout<<x<<'-';}
```

```
    return 0;
}
输出：1-2-3-
```

其实是字符数组结尾的\0 也被遍历了，但是无法输出。

补充：\0 和空格的区别：

```
【1】从字符串的长度:——>空字符的长度为 0,空格符的长度为 1
    char a[] = "\0";
    char b[] = " ";
    cout << strlen(a) << endl;   //0
    cout << strlen(b) << endl;   //1

【2】
    char crr[] = "a b";   //输出是 a b

    char brr[] = "a\0b";  //输出是 a，因为遇到'\0'代表结束

    cout << strlen(crr) << endl; //3
cout << strlen(brr) << endl; //1

【3】输出到屏幕上，'\0'什么都没有，而空格是空格。
```

四、封装与内联函数

4.1 private 与 overload 的先后

```
#include <iostream>
using namespace std;
class A
{ private: int a;
void f(int i=2)
{ a = i; }
public:
void f(int i, int j=2)
{ a = i + j; }
int get_a() { return a; }
};
int main()
{ A aa;
aa.f(1);
```

```
cout << aa.get_a() << endl;
return 0; }
```

这段代码：call to member function 'f' is ambiguous

不过给人的感觉是，我的 void f(int i=2)；是一个 private 函数，理论上在 main 里面是无法访问的，所以不应该会发生调用。

实际上，编译器看到 a.f 的时候会先找出所有的 f，判断调用正确之后才判断 private 权限是否正确。也就是说，函数调用优先匹配参数对应性，再判断权限合理性。编译为汇编语言后的程序里两个 f 是不分是否是 private 的，private 的语法检查在编译之后。

4.2 内联函数

4.2.1 定义与意义

函数调用要进行一系列准备和后处理工作(压栈、跳转、退栈、返回等)，所以函数调用是一个比较慢的过程。如果对于一个简单的函数进行大量的调用，会降低程序效率。

比较下面两种实现方式，函数比等价的表达式效率更低。

```
cout << max(a, b) << endl;
cout << (a > b ? a : b) << endl;
```

使用内联函数，编译器自动产生等价的表达式。

```
inline int max(int a, int b) {
   return a > b ? a : b; }
cout << max(a, b) << endl;
```

上述代码等价于

```
cout << (a > b ? a : b) << endl;
```

4.2.2 内联函数和宏定义的区别

在 L1 2.2.3.3 带参数宏中我们提到过，由于带参数宏具有高度歧义性，故而往往被内联函数替代。

宏定义只是拷贝代码到被调用的地方。
内联函数则是生成和函数等价的表达式。

内联函数可以执行类型检查，进行编译期错误检查。
内联函数可调试，而宏定义的函数不可调试。
在 Debug 版本，内联函数没有真正内联，而是和一般函数一样，因此在该阶段可以被调试。
在 Release 版本，内联函数实现了真正的内联，增加执行效率。

宏定义的函数无法操作私有数据成员。

4.2.3 内联函数的注意事项

避免对大段代码使用内联修饰符。

内联修饰符相当于把该函数在所有被调用的地方拷贝了一份，所以大段代码的内联修饰会增加负担。（代码膨胀过大）

避免对包含循环或者复杂控制结构的函数使用内联定义。

因为内联函数优化的，只是在函数调用的时候，会产生的压栈、跳转、退栈和返回等操作。所以如果函数内部执行代码的时间比函数调用的时间长得多，优化几乎可以忽略。

不可以将内联函数的声明和定义分开（不同于大多数函数将生命和定义分别写在头文件和源文件里）

编译器编译时需要得到内联函数的实现，因此多文件编译时内联函数先需要将实现写在头文件中，否则无法实现内联效果。

定义在类声明中的函数默认为内联函数。（但函数一般都不定义在类声明内）一般构造函数、析构函数都被定义为内联函数。

内联修饰符更像是建议而不是命令。

编译器“有权”拒绝不合理的请求，例如编译器认为某个函数不值得内联，就会忽略内联修饰符。

编译器会对一些没有内联修饰符的函数，自行判断可否转化为内联函数，一般会选择短小的函数。