深蓝学院
shenlanxueyuan.com

# C++ 基础

## 第 6 章：函数


主讲人 李伟

微软高级工程师

《C++ 模板元编程实战》作者


# 目录

# 函数基础

- 函数：封装了一段代码，可以在一次执行过程中被反复调用。
  - 函数头
    - 函数名称—标识符，用于后续的调用
    - 形式参数—代表函数的输入参数
    - 返回类型—函数执行完成后所返回的结果类型
  - 函数体
    - 为一个语句块（block），包含了具体的计算逻辑
- 函数声明与定义
  - 函数声明只包含函数头，不包含函数体，通常置于头文件中
  - 函数声明可出现多次，但函数定义通常只能出现一次（存在例外）

43：函数基础，有时间可以重新看一下

# 函数基础（续）

- 函数调用
  - 需要提供函数名与实际参数
  - 实际参数拷贝初始化形式参数
  - 返回值会被拷贝给函数的调用者
  - 栈帧结构 stack free 栈的地址是由高到低排列的
    栈帧结构有个链接可以点 argu是实参，para是形参
- 拷贝过程的（强制）省略
  - 返回值优化
  - C++17 强制省略拷贝临时对象
- 函数的外部链接

# 函数详解——参数

- 函数可以在函数头的小括号中包含零到多个形参
  - 包含零个形参时，可以使用 void 标记
  - 对于非模板函数来说，其每个形参都有确定的类型，但形参可以没有名称
  - 形参名称的变化并不会引入函数的不同版本
  - 实参到形参的拷贝求值顺序不定，
- 函数传值、传址、传引用
  底层逻辑是拷贝初始化
- 函数传参过程中的类型退化
- 变长参数
  - initializer_list
  - 可变长度模板参数
  - 使用省略号表示形式参数

```
#include <initializer_list>
void fun(std::initializer_list<int> par) {};
int main() {
    fun({1,2,3,4});
}
// 将initializer_list变成返回值是非常非常
危险的，这个返回值在fun函数结束的时候就销毁了
```

```
void fun(int) { } //对于预留参数，我们可以不定义名称。方便以后扩展
```

```
void fun(int, int y) {
    std::cout << y;
}
int main() {
    fun(1, 2);
}
```

c++17之前是编译器自行选择

C++17 强制省略复制临时对象

```
void fun(int z, int y) {
   std::cout << y;
}
int main() {
   fun(1, int{}); // 会省略复制
临时对象
}
```

```
void fun(int z, int y) {// 用1来拷贝初始化
z，和用2来拷贝初始化y的顺序不确定
    std::cout << y;
}
int main() {
    fun(1, 2);
}
```

里面包含了两个指针，第一个指针指向元素的开头，第二个指向了元素结尾的下一个元素

1. 传递的类型必须完全相同
2. 一般不传递指针和引用

为防止类型退化，可以如下：

```
void fun(int (&par)[3])
{
};

int main()
{
    int a[3];
    fun(a);
}
```

不建议按左侧的形式写。其实编译器都会编译成右侧的形式。如果按右侧写可能会对程序阅读产生困扰
下面的编写方式也可

```
void fun(int par[])
{
};
```

```
void fun(int x = 0) {
    std::cout << x << '\n';
}
```

我们可以在声明也可以在定义中写。不过只能写一遍，不能写2遍

```
void fun(int x, int y = 2, int z = 3);

void fun(int x, int y, int z)
{
    std::cout << x + y + z<< '\n';
}

int main()
{
    fun(1);
}
```

# 函数详解——参数（续）

#include<header.h> 会在编译的时候把header拷贝过来，放在cpp文件的前面

- 函数可以定义缺省实参
  - 如果某个形参具有缺省实参，那么它右侧的形参都必须具有缺省实参
  - 在一个翻译单元中，每个形参的缺省实参只能定义一次

    优点：可以在不同翻译单元中对缺省实参进行不同的定义
  - 具有缺省实参的函数调用时，传入的实参会按照从左到右的顺序匹配形参

    最不重要的参数一般放在最后面。什么是不重要的参数：一般含有缺省值，并且缺省值会满足一般意义的需求
  - 缺省实参为对象时，实参的缺省值会随对象值的变化而变化

    通常不建议大家这么做，会对其它程序的阅读者产生困扰。建议还是在头文件中使用缺省实参的方式
- main 函数的两个版本
  - 无形参版本
  - 带两个形参的版本 有链接可以点

    http://jbwyatt.com/244/web/args.htm

下面形式也是合法的

```
void fun(int x, int y, int z = 3);
void fun(int x, int y = 2, int z);
void fun(int x = 1, int y, int z);

void fun(int x, int y, int z)
{
    std::cout << x + y + z << '\n';
}
```

```
int x = 3;
void fun(int y = x)
{
    std::cout << y << '\n';
}

int main()
{
    fun();      // fun(x)
}
```

int main(int argc, char* argv[])

```
int main(int argc, char* argv[])
{
    if (argc != 3)
    {
        std::cerr << "Usage: " << argv[0] << " param1 param2\n";
        return -1;
    }
    // ...
}
```

```
test1@test1-UX31A:~/demo/demo/Debug$ ./demo
Usage: ./demo param1 param2
```

# 函数详解——函数体

- 函数体形成域：
  - 其中包含了自动对象（内部声明的对象以及形参对象）
  - 也可包含局部静态对象
- 函数体执行完成时的返回
  - 隐式返回 返回类型为void的函数
  - 显式返回关键字： return
    - return; 语句
    - return 表达式；
    - return 初始化列表；
  - 小心返回自动对象的引用或指针
  - 返回值优化（RVO）—— C++17 对返回临时对象的强制优化

```
return {1,2,3,4};
```

```
int& fun() {
    int x = 3;
    return x;
}
```

named return value optimization

```
Local static variable
void fun() {
    static int x = 0;
++x
}
static int x = 0;
```

1. 生存周期：从首次调用fun函数走到这一行开始，到整个程序结束为止。
2. 在函数内部可见
3. 如果有多个线程调用fun函数，局部初始化对象只会被初始化一次

3特性会引来问题。需要引入互锁

```
int& fun()
{
    static int x = 3;
    return x;
}

int main()
{
    int& res = fun();
}
```

这种情况的返回引用是可以接受的，因为这个返回的是一个局部静态对象。局部静态对象在程序结束的时候才会被销毁

```
struct Str {
    Str() = default;
    Str(const Str&) {
        std::cout << "copy constructor is called";
    }
};
Str fun() {
    Str x;
    return x;
}
int main() {
    Str res = fun();
}
// 如果是没有优化的编译器，则拷贝构造函数会被调用2次。1. return x的时候会构造出一个临时的对象，会调用拷贝构造函数。2.临时的对象会用来拷贝初始化res.
```

# 函数详解——返回类型

- 返回类型表示了函数计算结果的类型，可以为 void
- 返回类型的几种书写方式
  - 经典方法：位于函数头的前部
  - C++11 引入的方式：位于函数头的后部
  - C++14 引入的方式：返回类型的自动推导
    - 使用 constexpr if 构造“具有不同返回类型”的函数
- 返回类型与结构化绑定（C++ 17）
- [[nodiscard]] 属性（C++ 17）

```
auto fun(int a, int b) -> void
```

场景：
1. 元编程或者泛型编程
2. 类的成员函数

```
auto fun(int a , int b) {
    std::cout << a << b;
}
```

```
decltype(auto) fun(int a, int b) //
decltype(auto) 不会造成类型退化
```

```
constexpr bool value = false;
auto fun() {
    if constexpr (value) {
        return 1;
    } else {
        return 1.0;
    }
}
// 编译器具有不同的返回类型
```

不是所有的数据结构都能进行结构化绑定。

```
[[nodiscard]] int fun(int a, int b) {
    // 会生成warning
    return a + b;
}
int main() {
    fun(2, 3); //其实你根本没必要调用fun()
}
```

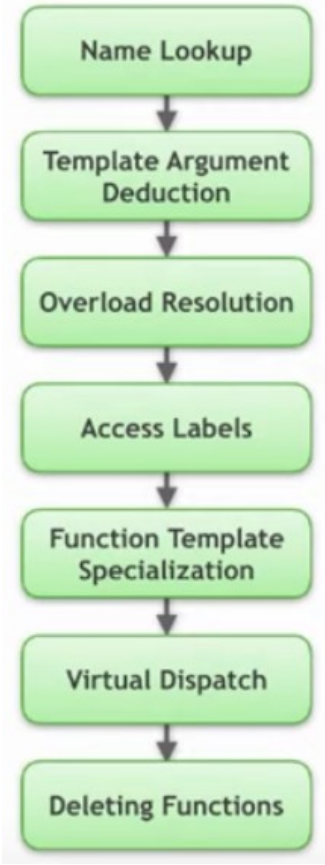
```
struct Str {
    int x;
    int y;
};
Str fun() {
    return Str{};
}
int main() {
    auto [res1, res2] = fun();
}
```

```
struct Str {
    int x;
    std::string y;
};
Str& fun() {
    static Str inst;
    return inst;
}
int main() {
    auto& [res1, res2] = fun();
}
```

# 函数重载与重载解析

- 函数重载：使用相同的函数名定义多个函数，每个函数具有不同的参数列表
  - 不能基于不同的返回类型进行重载
- 编译器如何选择正确的版本完成函数调用？
  - 参考资源： Calling Functions: A Tutorial 这有个链接
- 名称查找
  - 限定查找（qualified lookup）与非限定查找（unqualified lookup）
  - 非限定查找会进行域的逐级查找—名称隐藏（hiding）
  - 查找通常只会在已声明的名称集合中进行
  - 实参依赖查找（Argument Dependent Lookup: ADL）
    - 只对自定义类型生效

编译器完成函数调用的流程



# 函数重载与重载解析

重载解析是个链接

- 重载解析：在名称查找的基础上进一步选择合适的调用函数
  - 过滤不能被调用的版本 (non-viable candidates)
    - 参数个数不对
    - 无法将实参转换为形参
    - 实参不满足形参的限制条件
  - 在剩余版本中查找与调用表达式最匹配的版本，匹配级别越低越好（有特殊规则）
    - 级别 1： 完美匹配 或 平凡转换（比如加一个 const ）
    - 级别 2： promotion 或 promotion 加平凡转换
      int -> double double -> float
      float -> double
      short -> int
      true -> int
    - 级别 3： 标准转换 或 标准转换加平凡转换
    - 级别 4*： 自定义转换 或 自定义转换加平凡转换 或 自定义转换加标准转换
    - 级别 5*： 形参为省略号的版本
    - 函数包含多个形参时，所选函数的所有形参的匹配级别都要优于或等于其它函数

带星号的级别不是标准内部所定义的。而是上页中的参考资源自己定义的

```
void fun(int) {};
void fun(const int) {};
// 上面这两个函数会被叫做重定义,因为这两个函数都属于级别1, redefinition
// 特殊情况：
void fun(int&) {}; // 使用场景，既可以对函数读，也可以对函数写
void fun(const int &) {}; // 只能对传入的形参读，不能写
// 不会报错。虽然是同一级别
int main() {
    fun(3); // 选择void fun(const int &), 因为int&是一个左值引用，左值引用只能引用左值，而3是个右值。直接在第一步过滤就筛选掉了
    int x = 3;
    fun(x); // 选择void fun(int
}
```

# 函数相关的其它内容

- 递归函数：在函数体中调用其自身的函数
  - 通常用于描述复杂的迭代过程（示例）
- 内联函数 / constexpr 函数 (C++11 起 ) / consteval 函数 (C++20 起 )
- 函数指针
  - 函数类型与函数指针类型
  - 函数指针与重载
  - 将函数指针作为函数参数
  - 将函数指针作为函数返回值
  - 小心： Most vexing parse

内联函数：调用函数都会设计到栈帧的概念。我需要新开辟一个空间，里面存储实参的值等等。结束后还涉及栈帧的销毁，地址的跳回

```
inline void fun() {
    std::cout << "hi\n";
}
// inline只是对编译器的一个建议
```

链接

```
constexpr函数
constexpr int x = 3;//x是常量表达式，他能在编译器被确定
constexpr int fun() {
//可以在编译期，也可以在运行期被调用，如果想在编译期调用，就不能含有运行期调用的逻辑
    return 3;
}
constexpr int fun(int x){
    return x + 1;
}
constexpr int z = fun(3);// 编译期调用
int main() {
    int x = fun();
    int y;
    std::cin >> y;
    fun(y); //运行期被调用
}
```

```
consteval 函数只能在编译期被调用，不能在运行期被调用
consteval int fun(int x) {
    return x + 1;
}
```

std::function类模板，可以接受函数类型作为模板参数。
可以参考cppreference

```
std::function<void(int)> f_display = print_num;
```

函数指针,可以构造高阶函数

```
using K = int[3];
K* a; // a是一个指针，指向int[3]的对象
using F = int(int);
F* fun; //fun是一个变量，变量是一个函数指针类型
int add(int val) {
    return val + 1;}
int sub(int val) {
    return val - 1;}
int X(F* child, int val){
    auto tmp = (*child)(val);
    return tmp * tmp;
}
int main(){
    F* fun = &add;
    fun = &sub;
    std::cout << X(fun,3) << std::endl;
    // result is 4
}
```

c++11引入{}做初始化用途之一
就是为了解决most vexing parse

Most 是一个链接

```
using F = int(int); // F是一个函数类型，用来接收int,返回Int
F fun; // 声明函数fun
int fun(int val); //本行和上一行是等价的
int(int) fun; //这个语句是不合理的
int fun(int val) // 函数类型：int(int)，用来声明一个函数
函数和数组有很多相似的地方
using K = int[3];
K a = {}; //定义了一个数组a， aggregate initialization
int a[3]; // 类型：int[3]，用来声明和定义数组
```

深蓝学院
shenlanxueyuan.com

```
using F = int(int);
void FunWithArr(F val);
void FunWithArr(F* val);
// 上面这两个函数是完全等价的
```

# 感谢聆听！

## Thanks for Listening