第六章 函数

函数基础

函数是一个命了名的代码块，我们通过调用函数执行相应的代码。

函数可以有0个或多个参数，可以重载函数

调用运算符

形式：（）

- 一对圆括号，作用于一个表达式，该表达式是函数或指向函数的指针
- 圆括号内是一个用逗号隔开的实参列表，我们用实参初始化函数的形参

  > - 注意这里表述，实参是初始化形参的，所以形参和实参只是值的关系
  > - 多个实参给形参赋值的顺序是未知的

- 调用表达式的类型就是函数的返回类型

函数的形参列表

- 即使两个形参的类型一样，也必须把两个类型都写出来
- 任意两个形参不能同名

函数的返回类型

- **函数的返回类型不能是数组类型或函数类型，但可以是指向数组或函数的指针**

函数的声明（函数原型）

- 函数只能定义一次，但可以多次声明
- 函数声明无须函数体，用一个分号**代替**即可；

  函数声明实际上也可以不写形参，但是写上形参可以帮助使用者更好的了解函数的功能

  > 形参可以只写类型，不写名字

如果在头文件声明函数，那么函数函数声明的头文件应该被包含到定义函数的源文件中

局部对象

自动对象

只存在于块执行期间的对象。当块的执行结束后，块中创建的对象的值就变成未定义的

> 形参是一种自动对象。一旦函数终止，形参就被销毁

局部静态对象 `static关键字`

在程序的执行路径第一次经过对象定义语句时初始化，并且直到程序终止才被销毁，在此期间即使对象所在的函数结束执行也不会对它有影响。

> 静态局部变量初始化语句只会执行一次(实际上每次执行前编译器都会有一个if判断该初始化语句是否执行过)

分离式编译

> **通常我们编写较小的项目时代码都写在一个.cpp文件里，而较大的项目，为了结构清晰，容易修改且充分利用类封装的特性，可采用多文件编写，每个文件可视为一个编译单元最后链接起来一起编译**

基本思路：

函数/类 的声明放在头文件中，函数/类 的定义（具体实现）放在一个专门的 `.cpp` 文件中。

在主函数中，头文件需要包括函数声明头文件才可使用对应函数

场景使用

Linux下使用GCC

Clion设置

```
// CmakeList中
add_executable(temp main.cpp My_sum.h My_sum.cpp)
```

未完成

参数传递

传值参数

当形参的是一个非引用的类型时，实参的值被拷贝给形参，形参和实参是两个相互独立的对象，对变量的改动不会影响初始值

指针形参：

实际上，指针形参也是一个被赋值的结果，但是由于指针的特殊性，指向同一内存地址的指针，修改该地址的数还是会相互影响

> C++ 中建议使用引用代替指针形参

传引用参数

当形参是一个引用类型时，形参会被绑定到对应实参上，引用形参就是实参的别名。所有对引用实参的改变会影响实参。

> 引用实参并没有赋值操作，只是将一个变量绑定到另一个变量上

优点

- 拷贝大的类类型对象或容器的效率很低，使用引用可以避免赋值操作
- 有的类类型根本不支持拷贝操作（IO对象），这个时候函数只能通过引用形参来访问该类型的变量

  > 如果函数无须改变引用形参的值，最好将其声明为常量引用`const int & p`
  >
  > 原因：
  >
  > - 不支持`const对象`、字面值或者需要类型转换的对象传递给普通的引用形参
- 利用形参，函数可以返回额外的信息

const形参和实参

- 实参初始化时，会忽略顶层const

  > 实际上是发生了自动类型转换，可以忽略顶层const

数组形参

数组的两个特性：

- 不允许拷贝数组 ---> 不能以值传递的方式使用数组参数

> 尽管不能以值传递的方式传递数组，我们仍然可以将形参写出类似数组的 **形式**

- 使用数组时（通常）会将其转换为指针 ——> 实际传递的是数组首元素的指针

因为数组是以指针的形式传递给函数的，调用者应该提供额外的信息来指明数组的确切大小

- 使用标记指定数组长度，如C风格字符串
- 使用标准库规范：同时传递数组首元素和尾后元素的指针（`begin(a),end(a)`）
- 显式传递一个表示数组大小的形参

数组引用形参

```
void print(int (&array)[10])        // 维度是类型的一部分，必须要有
{

}
```

传递多维数组

- 数组第二维（以及以后所有的维度）的大小都是数组类型的一部分，不能省略

> 如何理解？
>
> 因为C++实际上并没有多维数组，所有都按一维数组来。那多出来的维数就认为是该一维数组的类型

main：处理命令行选项

```
int main(){}
/*
argv是一个数组，其元素是指向C风格字符串的指针;
    第一个元素指向程序的名字
argc 表示数组中字符串的数量
*/
int main(int argc, char *argv[]){...}
==
int main(int argc,char **argv)
```

含有可变形参的函数

C++11 编写处理不同数量实参的函数

C++11新标准提供两种主要的方法：

- 如果所有实参类型相同，可以传递一个名为 `initializer_list` 的标准库类型；
- 如果实参类型不同，可以编写一种特殊的参数，所谓的可变参数模块 **后续**

`initializer_list` ：一种标准库类型（**模版类**），用于表示特定类型的值的数组

头文件： `<initializer_list>`

提供的操作：

操作	**含义**
initializer_list lst;	默认初始化：T类型元素的空列表
initialzer_list lst{a,b,c..};	lst的元素数量和初始值一样多；lst的元素是对应初始值的副本；**列表中的元素是const**
lst2(lst) lst2=lst	拷贝或赋值一个 `initialize_list` 对象不会拷贝列表的元素；拷贝后，原始列表和副本共享元素
lst.begin()	返回指向lst中首元素的指针
lst.end()	返回指向lst中尾后元素的指针
lst.size()	返回列表中元素的数量

- 如果想向 `initializer_list` 形参中传递一个值的序列，则必须把序列放在一对花括号内

> 当初始化的时候使用的是大括号初始化，被自动构造

> 省略符形参：只用于需要与C函数交互的接口程序
>
> **省略，一般不用**

返回类型和return语句

无返回值函数

`return;`只能用在返回类型是`void`的函数中；

返回类型是`void`的函数不要求非要有返回语句；

> 实际上是在函数最后隐式执行return

有返回值的函数

- 函数返回值用于初始化（赋值）调用点的一个临时量，该临时变量就是函数调用的结果

  如果函数返回引用，则该引用仅仅是它所引对象的一个别名，不会真正的拷贝对象

- 注意：
  - 不要返回局部对象的引用或指针

    > 因为引用和指针都是和对象存在绑定关系的，函数内的局部对象在函数结束后就被销毁了，那返回的对象在调用函数中就是未定义的值

  - **函数的返回类型如果是引用的话，则函数返回的是左值，其他返回类型都是右值**

    > 因为引用实际上是对象的别名，所以可以做左值；而起来都是一个临时变量，不能做左值

- **C++11**,函数可以返回花括号包围的值的列表。

  此处的列表也用来表示对函数返回的临时变量的初始化

  - 初始化的类型由函数的返回类型决定
    - 如果是基本类型，则花括号包围的列表最多包含一个值，且不能大于目标类型的空间
    - 如果是类类型，则由类本身定义初始值如何使用

main函数返回值

- 允许main函数没有return语句，因为编译器会隐式插入一条`return 0`
- main函数返回值可以看作状态指示器
  - `<cstdlib>`定义了带个预处理变量，来表示函数允许成功（`EXiT_SUCCESS`）或失败(`EXIT_FAILURE`)
- main函数不支持递归，不能自己调用自己

返回数组指针

因为数组不能拷贝，所有函数不能返回数组，但可以返回数组的指针或引用

声明一个数组指针的函数

- 格式：`Type (*func(参数列表))[数组维度]`

```
int (*func(int i))[10];
/*
func(int i) 表示调用func函数时胎一个int类型的实参
(*func(int i)) 表示我们可以对函数调用的结果执行解引用操作
(*func(int i))[10] 表示解引用func的调用将得到一个大小是10的数组
int (*func(int i))[10] 表示数组中的元素是int类型
*/
```

使用尾置返回类型

C++11

任何函数的定义都可以使用尾置返回，但是这种形式对于返回类型比较复杂的函数最有效，比如返回类型是数组的指针或数组的引用；

- 格式：在函数形参列表后接 `-> 返回的类型` ，同时在本应该出现返回类型的地方放一个 `auto`

```
auto func(int i) -> int(*)[3]{

}
```

> `int(*)[3]` 可以看作一种数组类型，指向数组的指针
>
> **个人总结**：实际上，C++和指针或引用组合的基本类型，把正常声明时候的变量删除，就可以当作一种函数返回类型。只是函数不允许数组返回，所以 `int *[3]` 是不被支持的。

使用decltype

如果知道函数返回的指针将指向哪个数组，就可以使用 `dectype` 关键字声明返回类型

```
int odd[]={1,3,5};
int even[]={0,2,4};
decltype(odd) *arrPtr(int i){       // 这里decltype(odd)返回的是一个int[3]的数组，所以要变为指针类型 int[3]*
    return (i%2)?&odd:&even;
}
```

一个问题的分析

```
using arryT = int [3];
arryT* test(int (&p)[3]){       // 函数的返回类型是 int[3] *p，这样写是方便理解实际上还是int (*p)[3]，数组的指针

 p[1]=3;
 return &p;
}
using  namespace std;
int main(){
 int a[]={1,2,3};
 int * pp=&a;       // 报错,Clion提示&a的类似是int[3]*。
 cout<<pp;
 auto p=test(a);

}
```

理解 &a的类型是int[3]*。

对于数组来说，只有一维的概念。所以 int a[3][4] 实际上是一个基本元素是int[4]的一维数组。所以，单独a的类型应该理解为 int[4] *p。这里一定要把[4]写上去，才可以知道基本元素是4个大小的数组。当然考虑的**语言的限制**，C++需要写成 `int (*p)[4]` 才是被允许的

那么对于int a[3], a是一个int *p，这个概念容易理解。那么&a，我们把它看作一个取地址操作，应该也要有一个指针对应，此时a就是该指针的基本元素，就是一个3个大小的数组，可以理解该指针是一个数组指针

函数重载

定义重载函数

含义：如果同一作用域内的几个函数名字相同但形参列表不同（数量或类型不同），则称之为重载函数。编译器会根据传递的实参类型推断想要的是哪个函数。

- main 函数不能重载

注意：

- 一个拥有顶层const的形参无法和另一个没有顶层const的形参区别开；

  底层const是可以起到区分效果的

---

const_cast和重载

```
const string & shorterString(const string &s1,const string &s2)
```

- 如果我们想当函数接受非常量时返回普通的引用，可以利用 `const_cast` 和函数重载

```
string & shorterString(string &s1, string &s2)  // 底层const是可以重载的
{
    auto &r = shorterString(const_cast<const string&>(s1), const_cast<const string&>(s2));
    return const_cast<string&>(r);
}
```

重载和作用域

只有在同一作用域内的函数声明才能被看作是重载；

在不同作用域内无法重载函数名；

内存作用域中声明会隐藏外层作用域下所有同名的实体；

> 变量由声明除确定其作用域

重载函数匹配规则

可以互相转换的类型，编译器是如何区分的呢？

- 匹配顺序：

1. **确定候选函数**
   - 与被调用函数同名；
   - 其声明的调用点可用
2. **确定可行函数**
   - 形参数量和实参数量相同；
   - 每个实参的类型和对应形参类型相同或者能够转换成形参的类型
3. **确定最佳匹配函数**：实参类型越接近，它们就越匹配
4. **确定可行匹配**：如果没有最佳匹配，则找可行匹配

- 该函数每个实参的匹配都不劣于其他可行函数
- 该函数至少有一个实参的匹配优于其他可行函数

有且只有一个函数满足上述条件是，该函数才算是可行匹配。

> 调用重载函数应尽量避免强制类型转换；

- 函数参数匹配的优劣比较
  - 精确匹配：
    1. 实参和形参类型相同
    2. 实参从数组类型或函数类型转换为对应的指针类型
    3. 向实参添加顶层const或从实参中删除顶层const
  - 通过const转换实现的匹配

```
int a=2;
const int p = a;
```

  - 通过类型提升实现的匹配
  - 通过算术类型转换或指针转换实现的匹配

    > 所有算术类型装换的级别都一样

  - 通过类类型转换实现的匹配

特殊用途语言特性

默认实参

```
void(int i,int j=1){        // j=1是默认实参

}
```

- 调用含有默认参数的函数时，可以包含该实参，也可以省略该实参
- **一旦某个形参被赋予了默认值，它后面所有的形参都必须有默认值**

  > 尽量让不怎么使用默认值的形参出现在前面，而经常使用默认值的形参出现在后面

- 对于默认实参的多次声明，后续的声明只能为之前那些没有默认实参的形参添加默认实参，而且还要保证该形参右侧的所有形参都要有默认值

```
stirng screen(char,char,char = '');
string sceen(char,char,char ='X');       // 错误，重复声明
string sceen(char='',char='',char)       // 正确
```

- 因为默认实参是在函数括号内，所以局部变量不能作为默认实参

内联函数和constexpr函数（难）

内联函数：在调用内联函数的调用点上，程序逻辑不会转到该函数上执行，而是将其“内联地”展开执行

- 格式：在函数返回类型前加 `inline`
- 例子：

```
// shortString 声明为内联函数
cout<<shortString(s1,s2)<<endl;
==
cout<<(s1.size()<s2.size()?s1:s2)<<endl;
```

- 适用情况：内联机制用于优化规模较小、流程直接、频繁调用的胡

> 很多编译器不支持内联递归函数，而且一个75行的函数也不太可能在调用点内联地展开

constexpr函数：能用于常量表达式的函数 **难点**

- 约定：
  - 函数的返回类型及所有形参的类型都是字面值类型

    > 注意这里是字面值类型，constexpr函数返回值并非都是常量
    >
    > - 如果传入参数是一个常量表达式，则返回值也是常量表达式

  - **函数体必须有且只有一条return语句**

    > 函数体内实际上可以包含其他语句，只要这些语句在在运行时不进行任何操作即可（空语句、类型别名、using声明）

- 特性：
  - 编译器把constexpr函数的调用隐式地指定为内联函数
- 例子

```
constexpr int test(int i){
    return 1;
}
int main(){
    int i =1;
    constexpr int tt= test(i);      // 报错
    constexpr int tt=test(1);       // 正确
    return 0;
}
```

- 为什们test(i)明明返回值不是常量表达式，为什们函数的返回值有constexpr?

  可能是可以发生类型装换吧？？

和其他函数不一样，在不同的`.cpp文件中`，内联函数和constexpr函数可以中多次定义不报错，但多个定义的函数头必须完全一致。

```
//a.h里定义
inline int fun()
{
    return 1;
}
//在 b.h 里定义
inline int fun()
{
    return 100;
}
```

调试帮助

基本思想：程序可以包含一些用于调试的代码，这些代码只在Debug版本时有用，而如果编译为Release版本则忽略

assert预处理宏——断言

- 头文件：`<cassert>`
- 格式：`assert(expr)`

  首先对expr求值，如果为假，assert输出信息并终止程序的运行；如果为真，则什么都不做

> - 和预处理变量一样，宏名字在程序中必须唯一。

> 含有cassert头文件的程序不能再定义assert的变量、函数或其他实体
>
> - assert常用于检查“不能发生”的条件
> - 类型于用if判断一些某些条件

NDEBUG预处理变量

`assert`的行为依赖于一个名为`NDEBUG`的预处理变量的状态。如果定义，则assert什么也不做。

```
cc -D NDEBUG main.c     # 等价于定义NDEBUG
```

> **C++编译器预定义变量**
>
> ```
> _ _func_ _;      // 函数名字...
> _ _FILE_ _;      // 文件名字...
> _ _LINE_ _;      // 当前行号...
> _ _TIME_ _;      // 文件编译时间...
> _ _DATE_ _;      // 文件编译日期字符串字面值
> ```

函数指针

要想声明一个指向函数的指针，只需用指针替换函数名即可（记得加括号）

- 当我们把函数名作为一个值来使用时，该函数自动转换为指针

```
pf = lengthCompare();
pf = &lengthCompar();       // 等价
```

- 可以直接使用函数指针调用该函数，而无须解指针

```
bool b = pf(a,b);
bool b = (*pf)(a,b);    // 等价
```

- 指向不同的函数类型的指针之间不存在转换规则（除了`nulltrl`）

函数指针形参

形参是函数类型，实际上当指针使用

```
void useBigger(const string &s1,const string &s2,bool pf(const string &,const strign &));
```

- 调用时，直接使用对应函数名字即可
- 使用 `typedef` 定义函数指针，简化写法

```
// 格式  函数类型
typedef 返回类型 新类型名(参数表)；=====  typedef decltype(lengthCompare) Func2;
// 格式 函数指针类型
typedef 返回类型 (*新类型名)(参数表)；=====  typedef decltype(lengthCompare) *Func2;
```

> 个人认为，定义函数的类型似乎没有什么用，因为只是一种类型，最终还是需要利用其指针；（当然，类型可以用在函数形参声明上）
>
> 那既然这样，我们为何不直接定义指针类型呢

返回指向函数的指针

要想声明一个返回函数指针的函数，最简单的办法是使用类型别名

```
using F = int (int*,int);
using PF = int (*) (int *,int );        //  把实际使用的时候的变量名字去掉，保持其他形式不变，就可以认为是一种类型了

PF f1(int)
```

- 函数名作为返回类型不会自动地转换为指针，必须显示的将返回类型定义为指针

```
F f1(int);      // 错误
F *f1(int);     //
```

直接声明

```
int (*f1(int)) (int* ,int )     // 从内向外阅读 返回类型int (*?)(int*,int)  函数 f1(int)
```

> 较复杂，难以理解

使用 `auto` 和 `decltype` 用于函数指针类型

> 参考使用尾置类型