第6章 函数

6.1 函数基础

1. 函数的本质是一个命名了的代码块。
2. 函数的返回值不能是数组类型或函数类型，但可以是指向数组的或函数的指针。

6.1.1 局部对象

1. 我们把只存在于块执行期间的对象成为 *自动对象*。
2. *局部静态对象 local static object* 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。

6.1.2 函数声明

1. 函数只能定义一次，但是可以声明多次。

6.2 参数传递

6.2.1 传值参数

1. 熟悉 C 的程序员常常使用指针类型的形参访问函数外部对象。在 C++ 语言中，建议使用引用类型的形参替代指针。

6.2.3 `const` 形参和实参

1. 和其他初始化过程一样，当用实参初始化形参时会忽略掉顶层 `const` 。
2. 因为顶层 `const` 被忽略掉，所以传入两个 `fcn()` 的参数可以完全一样，造成函数重载错误：

```
void fcn(const int i){}
void fcn(int i){}          // 错误：重复定义
```

6.2.4 数组形参

1. 管理指针形参有三种常用的技术：
   (1) 使用编辑指定数组长度：要求数组本身包含一个结束标记。
   (2) 使用标准库规范：传递指向数组首元素和尾后元素的指针。
   (3) 显示传递一个表示数组大小的形参。
2. 形参可以是数组的引用，且 `&arr` 两端的括号不能少：

```
void f(int (&arr)[10]){}     // 正确：arr是具有10个整数的整型数组的引用
void f(int &arr[10]){}       // 错误：将arr声明成了引用的数组
```

3. 传递多维数组：

```
// matrix 指向数组的首元素，该数组的元素是由10个整数构成的数组
void f(int (*matrix)[10], int rowsize){}

// 再次强调，*matrix两端的括号必不可少：
int *matrix[10];             // 10个指针构成的数组
int (*matrix)[10];           // 指向含有10个整数的数组的指针
```

6.2.5 `main`：处理命令行选项

1. 第二个形参 `argv` 是一个数组，它的元素是指向 C 风格的字符串的指针。

```
// 两个表达式等价
int main(int argc, char *argv[]){}
int main(int argc, char **argc){}
```

2. 假定 `main` 函数位于执行文件 prog 之内，可以向程序传递选项： `prog -d -o ofile data0` ，此时 `argc` 等于5， `argv` 应该包含如下的C风格字符串：

```
argc[0] = "prog";            // 或者 argv[0] 也可以指向一个空字符串
argc[1] = "-d";
argc[2] = "-o";
argc[3] = "ofile";
argc[4] = "data0";
argc[5] = 0;
```

3. 当使用 `argv` 中的实参时，一定要记得可选的实参从 `argv[1]` 开始； `argv[0]` 保存程序的名字，而非用户输入。

6.2.6 含有可变形参的函数

1. 为了编写处理不同数量实参的函数，c++11新标准提供了两种主要的办法：如果所有的实参函数类型相同，可以传递一个名为 `initializer_list` 的标准库函数；如果实参的类型不同，我们可以使用可变参数模板。
2. 拷贝或赋值一个 `initializer_list` 对象不会拷贝列表中的元素，拷贝后原始列表和副本共享元素。
3. 和 `vector` 不同， `initializer_list` 对象中的元素永远是常量值，我们无法改变 `initializer_list` 对象中元素的值。
4. 省略符形参是为了便于 C++ 程序访问某些特殊的 C 代码而设置的，这些代码用到了名为 `varargs` 的 C 标准库功能。省略符形参仅仅用于 C 和 C++ 通用的类型，大多数类类型的对象在传递给省略符形参时无法正确拷贝。
5. 省略符形参只能出现列表的最后一个位置，省略符对应的实参无需类型检查：

```
void foo(parm_list, ...);
void foo(...);
```

6.3 返回类型和 `return` 语句

6.3.2 有返回值函数

1. 函数的返回类型决定函数调用是否是左值，调用一个返回引用的函数得到左值，其他返回类型得到右值。
2. 允许 `main` 函数没有 `return` 语句直接结束。编译器会隐式地插入一条返回 0 的 `return` 语句。
3. 为了使 `main` 函数返回的值与机器无关， `cstdlib` 头文件定义了两个预处理变量： `EXIT_FAILURS` 和 `EXIT_SUCCESS` 。
4. `main` 函数不能递归调用自身

6.3.3 返回数组指针

1. 因为数组不能被拷贝，所以函数不能返回数组。不过，函数可以返回数组的指针或引用：

```
// 使用类型别名
typedef int arrT[10];      // arrT是一个类型别名，它表示的类型是含有10个整数的数组
using arrT = int[10];      // arrT的等价声明
arrT* func(int i);         // func返回一个指向含有10个整数的数组的指针

// 不使用类型别名
Type (*function(parameter_list))[dimension];     // 数组的维度必须跟在函数名字之后
//例如：
int (*func(int i))[10];

// 使用尾置返回类型
auto func(int i) -> int(*)[10];

// 如果知道函数返回的指针将指向哪个数组，可以使用 decltype 关键字声明返回类型。
int odd[] = {1, 2, 3};
decltype(odd) *arrPtr(int i)
{
   return &odd;
}
```

6.4 函数重载

1. 重载的前提是要在同一个作用域下。
2. `main` 函数不能重载。

```
// 一个拥有顶层 `const` 的形参无法和另一个没有顶层 `const` 的形参区分开来：
int func(double* a);
int func(double* const a);

// 底层const形参可以区分
int func(double* b);
int func(const double* b);
```

3. 调用重载函数时，如有多于一个函数可以匹配，但是每一个都不是明显的最佳选择。此时也将发生错误，称为 *二义性调用 ambiguous call*。

6.4.1 重载与作用域

1. 如果我们在内层作用域中声明了名字，它将隐藏外层作用域中声明的同名实体。在不同的作用域中无法重载函数名。

6.5 特殊用途语言特性

6.5.1 默认实参

1. 要注意由于隐式类型转换而出现的与预期不符合的函数调用情况。
2. 在给定的作用域中一个形参只能被赋予一次默认实参，函数的后续声明只能为之前那些没有默认值的形参添加默认实参，*而且该形参右侧的所有形参必须都有默认值*。

6.5.2 内联函数和 `constexpr` 函数

1. 一次函数的调用包含的工作：调用前要先保存寄存器，并在返回时恢复；可能需要拷贝实参；程序转向一个新的位置继续执行。
2. 内联函数可以避免函数调用的开销，在调用点上 “内联地” 展开。
3. `constexpr` 函数是指能用于常量表达式的函数。函数的返回类型及所有形参的类型都得是字面值类型，而且函数体中必须有且只有一条 `return` 语句。编译器会把对 `constexpr` 函数的调用替换成其结果值。为了能在编译过程中随视展开， `constexpr` 函数被隐式地指定为内联函数。
4. 内联函数和 `constexpr` 函数通常定义在头文件中。

6.5.3 调试帮助

1. `assert` 是一种预处理宏，常用于检查“不能发生”的条件。
2. `assert` 的行为依赖于一个名为 `NDEBUG` 的预处理变量的状态。如果定义了 `NDEBUG` ，则 `assert` 什么也不做。
3. 应该把 `assert` 当作调试程序的一种辅助手段，但是不能用它替代真正的运行时逻辑检查，也不能替代程序本身应该包含的错误检查。
4. 编译器定义了5个对程序调试很有用的局部静态变量：

```
_ _func_ _ 存放函数名的字符串字面值
_ _FILE_ _ 存放文件名的字符串字面值
_ _LINE_ _ 存放当前行号的整型字面值
_ _TIME_ _ 存放文件编译时间的字符串字面值
_ _DATE_ _ 存放文件编译日期的字符串字面值
```

6.6 函数匹配

1. 函数匹配的第一步是选定本次调用对应的重载函数集，集合中的函数称为 *候选函数 candidate function*。

2. 函数匹配的第二步考察本次调用提供的实参，然后从候选函数中选出能被这组实参调用的函数，这些新选出的函数称为 *可行函数 viable fuction*。
3. 函数匹配的第三步是从可行函数中选择与本次调用最匹配的函数。
4. 含有多个形参的函数匹配规则：
   ·该函数每个实参的匹配都不劣与其他可行函数需要的匹配
   ·至少有一个实参的匹配优于其他可行函数提供的匹配。
   ·如果在检查了所有实参之后没有任何一个函数脱颖而出，则该调用是错误的。编译器将报告二义性调用的信息。
5. 调用重载函数时应尽量避免强制类型转换。如果在实际应用中确实需要强制类型转换，则说明我们设计的形参集合不合理。

6.6.1 实参类型转换

1. 为了确定最佳匹配，编译器将实参类型到形参类型的转换划分了成了几个等级：
   (1) 精确匹配：
   ·*实参类型* 和 *形参类型* 相同。
   ·实参从 *数组类型* 或 *函数类型* 转换成对应的指针类型。
   · 向实参添加顶层 `const` 或者从实参中删除顶层 `const` 。
   (2) 通过 *`const` 转换* 实现的匹配。
   (3) 通过 *类型提升* 实现的匹配。
   (4) 通过 *算术类型转换* 或 *指针转换* 实现的匹配。
   (5) 通过 *类类型转换* 实现的匹配。

6.7 函数指针

1. 函数指针指向的是 *函数* 而非 *对象* 。函数指针指向某种特定的类型。函数的类型由它的返回类型和形参类型共同决定，与函数名无关。

```
// 函数声明：
bool func(int a);

// 函数指针：
bool (*pf)(int);                    //未初始化

// *pf两端的括号必不可少
bool *pf(int);                      // 声明了一个名为pf的函数，该函数返回bool*
```

2. 当我们把函数名作为值使用时，该函数自动地转换成指针。

```
pf = func;
pf = &func;                         // 等价
```

3. 直接使用指向函数的指针调用该函数，无需提前解引用指针：

```
bool b1 = pf(1);                    // 调用func()
bool b2 = (*pf)(1);                 // 等价调用
bool b3 = func(1);                  // 等价调用
```

4. 在指向不同函数类型的指针间不存在转换规则。可以给函数值赋值一个 `nullptr` 或值为 0 的整型常量表达式，表示该指针没有指向任何一个函数。
5. 如果定义了指向重载函数的指针，编译器通过指针类型决定选用哪个函数，指针类型必须与重载函数中的某一个精确匹配。
6. 和数组类似，不能定义函数类型的形参，但是形参可以是指向函数的指针。

```
void use(bool pf(int));             // 形参是函数类型，会自动转换成指向函数的指针
void use(bool (*pf)(int));          // 等价的声明，显示地将形参定义成指向函数的指针
```

7. 直接使用函数指针类型显得冗长而烦琐。类型别名 `typedef` 和 `decltype` 能简化使用函数指针的代码。下面的四个声明语句声明的是同一个函数，编译器会自动将函数类型转换成指针。

```
bool func(int a);

// Func 和 Func2是函数类型
typedef bool Func(int);
typedef decltype(func) Func2;       // 等价
void use(Func);                     // 声明1
void use(Func2);                    // 声明2

// FuncP 和 FuncP2 是指向函数的指针
typedef bool (*Funcp)(int);
typedef decltype(func) *Funcp2;     // 等价
void use(Funcp);                    // 声明3
void use(Funcp2);                   // 声明4
```

8. 与函数不能直接返回 *数组* 但能返回 *指向数组的指针* 类似，函数不能直接返回 *函数类型* 但能返回 *指向函数的指针*。
9. 必须把返回类型写成 *指针* 形式，编译器 *不会自动地* 将函数返回类型当成对应的指针类型处理。

```
// 使用类型别名声明f1:
using F = int(int*, int);          // F是函数类型，不是指针
using PF = int(*)(int*, int);      // PF是指针类型
PF f1(int);                        // 正确：PF是指向函数的指针，f1返回指向函数的指针
F *f1(int);                        // 正确：显示地指定返回类型是指向函数的指针
F f1(int);                         // 错误：F是函数类型，f1不能返回一个函数

// 直接声明f1:
int (*f1(int))(int*, int);

// 使用尾后返回类型声明f1:
auto f1(int) -> int(*)(int*, int);

// 如果明确知道返回函数是哪一个，就能使用 decltype 简化书写
// decltype返回的是函数类型而非指针，所以需要显式地加上*以表明需要返回的是指针而非函数本身
bool func(int);
decltype(func) *getFcn(const string&);
```