Module03-05
C++ 语言基础：函数

语言基础 - 函数

- 函数
  - 声明与定义
  - 参数传递
  - 返回值
  - 函数名重载 (Overloading)
  - 默认参数
  - 未定数目参数列表
  - 递归
  - 函数指针
  - 内联函数和宏
  - main() 函数

函数 - 声明与定义

- 函数声明：
  - 函数声明的语法：
    - `type name(arguments) cv-qualifiers except-spec;`
  - 说明：
    - `type`：返回值类型，除下列两种情况外，其它任何函数必须明确指定函数的返回值类型，如果无返回值，则是 `void`
      - 类类型的构造、析构函数不指定返回值类型
      - 各种类型转换的操作符重载函数不指定返回值类型
    - `name`：一个合法的函数名称
    - `arguments`：参数列表，0 个到多个参数，参数之间用，（逗号）分隔
    - `cv-qualifiers`：`const` 限定符，只能在类类型中的函数使用
    - `except-spec`：函数将抛出的异常清单

函数 - 声明与定义

- 函数声明（续）：
  - 参数名字
    - 函数声明中的参数名字可以省略
    - 加上参数的名称更易于阅读

```
void print(const int& a, const int& b);
void print(const int&, const int&);     // 效果同上
```

- 函数声明（续）：
  - 函数声明示例：

```
class Array;

void print(const Array&);  // 形参的名字可以省略
int get(const size_t& index) throw(exception);  // 带异常清单
inline bool isOdd(const int& number);   // 内联函数 inline

class Array {
public:
    Array(); // constructor，不指定返回值类型
    Array(const size_t& len); // constructor，不指定返回值类型
    ~Array(); // destructor，不指定返回值类型

    operator int*();  // 类型转换操作符，不指定返回值类型

    size_t size() const;    // const 限定符
    // ...
};
```

函数 - 声明与定义

- 函数定义
  - 除声明的几大要素外，函数定义还须有函数体，在函数体中实现具体的动作
  - 任何函数在使用之前必须是已经被正确定义过的（且只能定义一次）
  - 如函数 isOdd() 的定义：

```
inline bool isOdd(const int& number) {
    return number % 2;
}
```

函数 - 声明与定义

- 函数定义（续）
  - 函数体内的局部变量
    - 局部变量在程序执行到变量定义处初始化
    - 函数中定义的非静态局部变量，每一次调用都需作初始化，且都拥有自己的拷贝
    - 如果是静态局部变量，其初始化仅有唯一的一次，且所有的调用均共享该静态变量（不是一次调用就拥有自己的拷贝）

- 函数定义（续）
  - 函数体内的局部变量（续）
    - 示例

```
void staticTest() {
    int n = 0;
    static int m = 0;
    cout << ": ++n == " << ++n
         << ", ++m == " << ++m << endl;
}

int main() {
    staticTest(); // ++n == 1, ++m == 1
    staticTest(); // ++n == 1, ++m == 2
    staticTest(); // ++n == 1, ++m == 3
}
```

函数 - 参数传递

- 参数传递的实质
  - 函数被调用的时，其形参将被分配储存空间，并用对应的实参对形参进行初始化（如赋值）
  - 如果传入的实参与声明的形参类型不匹配，将会进行隐式类型转换，如隐式类型转换不支持或不成功，将需显式类型转换。
- C++ 函数参数传递的方式：
  - 按值传递 (Passed by value)
  - 按引用传递 (Passed by reference)

函数 - 参数传递

- 按值传递
  - 按值传递，传入实参的副本（ copy ），函数体内对传入的值的修改不会影响实参
  - 示例：

```
void passByValue(int n) {
    n += 12;
    cout << "n: " << n << endl;
}

int main() {
    int k = 3;
    passByValue(k);                 // n: 15
    cout << "k: " << k << endl;     // k: 3
}
```

函数 - 参数传递

- 按引用传递
  - 按引用传递是传入实参的引用，函数体内对传入的引用的修改将会影响实参
  - 一个函数的参数声明为按引用传递，则表明告诉调用者，传入的实参将（可能）被修改
  - 示例：

```
void passByReference(int& n) { // 注意是按引用传递
    n += 12;
    cout << "n: " << n << endl;
}

int main() {
    int k = 3;
    passByReference(k);          // n: 15
    cout << "k: " << k << endl; // k: 15 , k的值也被修改
}
```

函数 - 参数传递

- 两种参数传递的比较
  - 按值传递将会复制一份实参的副本，函数对实参的副本操作，所以对实参是安全的
  - 按引用传递传入实参的引用，函数对该引用所作的所有改变将会影响到实参，所以实参的状态没有保护
  - 按引用传递参数，要求函数返回之前，实参不能被销毁，而按值传递则不存在这个限制
  - **重要：**传大对象尽量用按引用传递（或改用传指针的方式），按值传递过程中创建大对象副本的代价很大，如：

```
void largeObj(vector<double> vec);

int main() {
    // vec容纳100000个double型对象
    vector<double> vec(100000);
    largeObj(vec);  // 将创建vec的副本
}
```

函数 - 参数传递

- const 引用参数
  - 为什么用 const 引用参数
    - const 引用参数避免了引用传参数会修改实参，同时也避免了按值传递所带来的副本拷贝的开销
    - 引用参数不接受需要类型转换的参数、常量、文字常量，但 const 引用参数可以
  - 使用 const 引用参数
    - 在绝大多数情况下，可以用 const 引用参数替代按值传递的方式，且拥有更小的开销、类似的安全性
    - 在不需要更改实参的情况下，不要使用按引用传递的方式，以按 const 引用方式传递参数替代之

函数 - 参数传递

- 指针参数
  - 传指针实际上是按值传递的一种，但可以实现按引用传递方式类似的效果：通过指针更改实参（指针）指向的对象
  - 如果要修改指针参数本身，应该传引用如：

```
void passPonitRef(int*& pr);
```

  - 如果要禁止修改指针指向的对象，则需传入 const T* ，如：

```
void passPoint(const int* i); //  或
void passPoint(int const* i);
```

函数 - 参数传递

- 数组参数
  - 数组参数类似于指针参数，因为一旦作为参数传递，数组就退化成指针
  - 示例：

```
// 以下三种定义被编译器视为相同（同时出现将通不过编译）
void passArray(int* i) {
}
void passArray(int i[]) {
}
void passArray(int i[8]) {   // 数组的维数将被编译器忽略
}

int main() {
    int n[4] = {};
    passArray(n);
}
```

函数 - 返回值

- 关于返回值
  - 如果一个函数声明中返回值不是 void， 必须返回相应类型的值（ main() 函数可以例外 ）
  - 如果一个函数声明中返回值是 void， 则不能返回任意类型的值
  - 一个函数体内可以有多个 return 语句
  - 示例：

```
void f1() { return 12; }    // 错误，不能有返回值
void f2() { return; }       // OK
void f3() { return f2(); } // OK
int f4() { int a = 12; }    // 错误，必须返回int型的值

int f5(const int& i) {      // 多个return语句
    if (0 == i) return 8;
    if (1 == i) return 12;
    return 16;
}
```

函数 - 返回值

- 返回局部对象的引用或指针
  - 注意：返回局部自动对象的引用或指针是危险的行为！

```
int* f1() {
    int a = 12;
    return &a; // 危险！
}

int* f2() {
    int* a = new int(12);
    return a; // OK！
}

int* f3(int* a) {
     // do something...
     return a; // OK
}
```

```
int& f4() {
     int a = 8;
     return a; // 危险！
}

int& f5(int& a) {
     // do something...
     return a;  // OK
}
```

函数 - 函数名重载

- 什么是函数名重载
  - 就是在同一个作用域内，定义一些名字相同、但参数细节不同的函数，这就是函数名重载
  - 示例：

```
void mySwap(int& a, int& b);

void mySwap(double& a, double& b);

void mySwap(int& a, int& b, int& c);
```

函数 - 函数名重载

- 重载函数的调用匹配
  - 在一系列的重载函数中正确调用合适的函数，将需要通过对这些函数的参数类型、形参列表进行最佳匹配
  - 匹配规则（和次序）：
    - 精确匹配，参数类型不用转换或只要作细微的转换（如数组名到指针、函数名到函数指针、 T 到 const T ）
    - 类型提升匹配，如 short 到 int ， float 到 double...
    - 使用标准转换匹配，如 int 到 double...
    - 使用用户定义类型转换匹配
    - 匹配不确定参数个数的函数（参数列表为不确定个数）

- 调用匹配的示例：

```
void f(char a, char b);
void f(int a, int b);
void f(int a, int b, int c);
void f(double a, double b);

int main() {
    int i1 = 12, i2 = 8;
    double d1 = 23.6, d2 = 45.8;
    short s1 = 6, s2 = 8;
    char c1 = 'V', c2 = 'A';
    char c3 = 'N', c4 = 'M', c5 = 'P';

    f(i2, i1); // 精确匹配，调用 void f(int, int);
    f(c1, c2); // 精确匹配，调用 void f(char, char);
    f(d1, d2); // 精确匹配，调用 void f(double, double);
    f(s1, s2); // 类型提升，调用 void f(int, int);
    f(c3, c4, c5); // 类型提升，调用 void f(int, int, int);
    f(s1, i1);  // 标准转换，int->double，调用 void f(double,
double);
}
```

函数 - 函数名重载

- 调用匹配的二义性 (ambiguous) 示例：

```
void f2(long n);
void f2(double n);

int main() {
    f2(1L); // 精确匹配，调用 void f2(long);
    f2(1.0); // 精确匹配，调用 void f2(double);
    f2(1); // ?? 调用哪个？匹配失败，有二义性，两个f2()都符合！
}
```

函数 - 函数名重载

- 调用匹配 - 同作用域优先
  - 匹配过程从调用函数所在的作用域开始，如果该作用域下有可以匹配（包括需要类型转换）的函数，则优先调用之

```
void f(int a);

int main() {
    void f(double d);
    f(8); // 调用 void f(double);
}
```

函数 - 函数名重载

- 构成函数名重载的依据
  - 参数个数
  - 参数类型
  - 不同类型的参数在列表中出现的次序
  - const 限定符

```
class A {
public:
    void f();    //#1
    void f() const; //#2 有const限定符
    void f(int a);  //#3 参数个数与 #1 不同
    int f(int a, double d); //#4 参数个数与 #3 不同
    int f(double d, int a); //#5 参数次序与 #4 不同
    int f(); // 与 #1 名字冲突，因为返回值类型不能作为重载的依据
};
```

为什么返回值的类型不能作为函数名重载的依据？

函数 - 默认参数

- 默认参数 (Default arguments)
  - 规则：
    - 参数列表中只有最后的（一个或多个）参数提供默认参数
  - 示例：

```
void f1(int n = 0, int k); // 错误
void f2(int n = 0, int k = 0); // OK
void f3(int n, int k = 0); // OK

void display(int x, int y, bool showMenu = true);

int main() {
    display(0, 0); // 相当于调用 display(0, 0, true);
    display(200, 300, false);
}
```

函数 - 默认参数

- 默认参数与函数名重载
  - 调用时可能会产生二义性 (ambiguous)

```
void f3(int n);
void f3(int n, int k = 0);

int main() {
    f3(8); // 错误: 调用 f3(int) ? 还是 f3(int, int = 0) ?
}
```

函数 - 未定数目参数列表

- 参数类型和数量未定
  - C 标准库中一些函数的参数个数是未定的，如：
    - `int printf(const char* ...)`
    - `int fprintf(FILE*, const char* ...);`
    - `.....`
  - 这类函数的调用，如：

```
int i = 9, j = 12;
printf("%d + %d = %d\n", i, j, i + j);
```

  - 问题？
    - 如上示例，第一个参数是格式化串，期望随后跟 3 个 int 型的值，但编译器并不检查（无法检查）：随后的 3 个参数是否都存在？类型是否都匹配？（这些必须由调用者检查）

函数 - 递归

- 递归 (Recursive)
  - 一个函数直接或间接的调用自己，即为递归
  - 一个递归，必须有正确的终止条件，否则将成为无限递归（无法得到最终的值）
  - 递归的开销与效率？
  - 示例（求 n 的阶乘 n!）

```
int factorial(const int& n) {
    // 注意: n < 2 是递归的终止条件
    return n < 2 ? 1 : n * factorial(n - 1);
}
```

- 为什么要函数指针
  - 隐藏实现的细节、使接口与实现解耦，有利于针对统一接口编程
  - 可以实现回调 (Callback) 机制
- 函数指针的定义与初始化

```
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }
int f1(int a, int b) { return a * a + b * a; }

int (*fp)(int, int);   // 函数指针的定义
void demo() {
    int i = 8, j = 3;
    fp = &f1;          // 函数指针的赋值，取址符 & 可省略
    fp(i, j);          // 函数指针的使用
    fp = subtract;     // 函数指针的赋值
    fp(i, j);          // 函数指针的使用
}
```

- 函数指针作为其它函数的参数

```
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }
int f1(int a, int b) { return a * a + b * a; }

void test(int(*fp)(int, int), int k) { // test()函数一种形式
    int i = 9, j = 2;
    cout << "k: " << k << ", i + j = "
         << fp(i, j) << endl;
}
```

```
typedef int (*fp)(int, int);    // 先定义函数指针的别名
void test(fp func, int k) { // test()函数另一种形式
    int i = 9, j = 2;
    cout << "k: " << k << ", i + j = "
         << func(i, j) << endl;
}
```

```
int main() {
    test(f1, 9); // test(add, 9);  // test(subtract, 9);
}
```

函数 - 内联函数和宏

- 内联函数 (inline functions)
  - 一个函数声明为 inline，就是提示编译器在编译时将该函数的目标代码直接插入到对该函数的每个调用处，以消除函数调用的开销（参数拷贝、查找符号等）
  - 内联一般用于被频繁调用，且代码短小的函数

函数 - 内联函数和宏

- 宏 (Macro)
  - 宏的实质？
    - 编译前（预处理）文本替代
  - 宏定义的格式、使用宏

```
#define BUF_SIZE 512
#define MAX(a, b) (((a) > (b)) ? (a) : (b)) // 返回二者之较大的

int main() {
    int buf[BUF_SIZE];
    cout << MAX(12, 8) << endl;
}
```

函数 - 内联函数与宏

- C++ 中宏的替代方案
  - 以刚才的示例为例：
    - `#define BUF_SIZE 512` // 可以用如下方式：
      - `const int BUF_SIZE = 512;`
      - `static const int BUF_SIZE = 512;`
      - `enum {BUF_SIZE = 512};` // 此方案尤佳
    - `#define MAX(a,b) (((a) > (b)) ? (a) : (b))` // 可用内联函数
      - `inline int max(int a, int b) {return a > b ? a : b; }`
    - ……

函数 - main() 函数

- 关于 main() 函数
  - C++ 标准接受的 2 种形式的 main() 函数：
    - `int main() {}`
    - `int main(int argc, char* argv[]);`

      注：第二种形式中第二个参数也可以写成：
      - `char** argv`
      - `const char* argv[]`
  - `main()` 函数的返回值
    - 标准 C++ 规定 `main()` 函数的返回值类型是 `int`，不再接受 `void main()` 或 `main()` 等形式
    - `main()` 函数中可以不显式的编写 `return` 语句，如果没有显式的编写 `return` 语句，编译器认为 `main()` 函数返回 0

函数 - main() 函数

- 关于命令行参数
  - main 函数的第二种形式中：
    - int argc：表示命令行参数的个数，包括可执行文件名
    - char* argv[]：包括可执行文件名以及所有命令行参数
  - 命令行参数示例：

```
int main(int argc, char* argv[]) {
    cout << "Program's name: " << argv[0] << endl;
    for (int i = 1; i < argc; ++i) {
        cout << "Arguments#" << i << " is: " << argv[i] <<
endl;
    }
}
```

```
$ ./args_test hello xuanyuan    # 运行程序，传2个命令行参数
Program's name: ./args_test
Arguments#1 is: hello
Arguments#2 is: xuanyuan
```

函数 - Bjarne's Advices

- Bjarne's Advices
  - 少用非 const 引用参数；如果想要函数修改其参数，使用指针作为参数或返回新值
  - 使用 const 引用参数，减少参数复制
  - 广泛而一致的使用 const
  - 避免使用宏
  - 避免不确定数量的参数
  - 不要返回局部对象的引用或指针
  - 当一些函数对不同的类型执行概念上相同的操作时，使用重载
  - 在考虑函数指针时，请考虑虚函数、函数模板是否更好