函数

目录：
函数指针
参数传递
返回值
内联函数 inline
函数的重载

参数传递：

1.pass-by-value（值传递）:

在 C 语言中，参数传递的方式是值传递，例如：

```
int gcd(int v1, int v2) {
    while (v2) {
        int temp = v2;
        v2 = v1 % v2;
        v1 = temp;
    }
    return v1;
}
```

这段代码在函数中修改了 v1 和 v2 的值，假设 main 中这样写：

```
int a = 10;
int b = 25;
cout << gcd(a, b) << endl;
```

函数调用完之后，a 和 b 的值没有发生改变，因为值传递的过程是一个 copy 值的过程，函数的形参在调用的时候是实参的一个副本，它的改动与原来的变量无关。

看一个例子，如何交换两个数，我们知道这样的代码是无效的：

```
void swap(int a, int b){
```

```
    int temp = a;
    a = b;
    b = temp;
}
```

正确的代码是这样的：

```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

这段代码起作用的原因是拷贝的不是要交换的两个数字，而是两个地址。

注意：**不存在一种地址传递方式叫做指针传递!!!**

练习：　写一个 swap 程序，交换两个 int* 变量。

总结交换两个变量的模板：

假设要交换的变量为 TYPE 类型，那么采用值传递交换需要加一级指针，如下：

```
void swap(TYPE *a, TYPE *b) {
    TYPE temp = *a;
    *a = *b;
    *b = temp;
}
```

看下面的代码：

```
void test(char *p){
    p = new char[100];
}
```

```
int main(int argc, char **argv) {
    char *p = NULL;
    test(p);
    strcpy(p, "hello");
}
```

这段代码存在什么问题？ 如何修正？

1.传递 p 的指针

2.采用返回值

3.采用引用

值传递的缺点：

值传递既然是值拷贝，那么每次函数调用都伴随着变量的复制，如果变量比较大，就会增加程序运行的开销。

2.pass-by-reference（引用传递）:

前面交换两个数字，我们采用了把指针传递进入的办法，这里还可以直接采用引用：

```
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}
```

这段程序之所以能够起到交换的目的，是因为这里采用了引用传递，函数调用时并没有去 copy 两个变量的副本。

采用传递引用的方式交换两个变量的模板为：

```
void swap(TYPE &a, TYPE &b){
    TYPE temp(a);
    a = b;
```

```
        b = temp;
    }
```

引用传递和值传递的区别：

引用传递避免了对象的复制！这很大程度上减少了函数调用的开销。

引用传递的用处：

正因为引用传递有着这样的好处，对于标准库类型变量，我们应该尽可能采用引用作为形参的类型。例如有个函数，输入十个数字，保存在一个数组中，我们可以这样写：

```
void input_num(vector<int> &vec){
    int n = 10;
    int temp;
    while(n--){
        cin >> temp;
        vec.push_back(temp);
    }
}
```

所以，引用形参可以当做输出参数。

采用 const 保护引用形参的值：

如果传递一个字符串，可以这样写：string &s 但是传递引用，使得在函数体内修改 s 的值变为可能（以前值传递是修改的副本，没有副作用），这可能使得代码比较危险，于是我们采用 const 给 string 加上一层保护功能：

```
bool is_short(const std::string &s1, const std::string &s2)
{
    return s1.size() < s2.size();
}
```

这里的 const string &s 有两重含义：

1.避免复制 s，造成开销

2.加上 const 保护 s 的值，防止恶意修改。

实际在编程中，如果一个参数采用的是非 const 引用，这通常意味着**希望**我们修改它的值，如果是 const 引用，那么仅可以读取它的值。

我们总结下**参数传递的原则**：

1.对于原生数据类型，例如 int，如果不需要改变，采用简单的值传递即可，需要改变参数就加上 &符号。

2.对于标准库类型或者用户自定义类型，如果不需要改变，我们使用 const 引用的形式，例如 const A &a，如果需要改变，我们采用 A &a.

内联函数：

1.避免了函数调用的开销

2.相对于宏进行了语法检查

注意点：内联函数应该放入到头文件中。

例如我们可以将 swap 写为内联，减少开销，只需在函数开头加上 inline 关键字即可

例如：

```
inline void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}
```

这里可以看做我们写了一个高级的宏函数。

函数的返回值：

每条路径都要有返回值：

```
bool str_subrange(const std::string &str1, const std::string &str2)
{
    if(str1.size() == str2.size())
    {
        return str1 == str2;
    }
    std::string::size_type size = (str1.size() < str2.size())? str1.size() : str2.size();
    std::string::size_type i = 0;
    //look at each element up to size of smaller string
    while(i != size)
    {
        if(str1[i] != str2[i])
            return ;
    }
    //error
}
```

上面的代码是错误的，因为有的情况下没有返回值。

返回 value 类型：

例如：

```
string make_plural(size_t ctr, const string &word, const string *&ending){
    return (ctr == 1)? word: word + ending;
}
```

返回的是 string 值，这里返回的是一个副本，因此同样存在对象的复制。

返回 reference 类型：

下列代码的作用是求两个字符较短的一个：

```
const string &shorterString(const std::string &s1,
                            const std::string &s2)
{
    return s1.size() < s2.size() ? s1 : s2;
}
```

这里注意的是：因为 s1 和 s2 都是 const 引用，所以这里返回引用必须加上 const。

再看：

```
char &get_val(std::string &str, std::string::size_type ix)
{
    return str[ix];
}
```

这里返回的就是一个引用。

注意：返回引用的函数可以作为一个**左值**。

例如：

```
char &get_val(std::string &str, std::string::size_type ix)
{
    return str[ix];
}


int main(int argc, char **argv) {
    string s = "hello";
    get_val(s, 2) = 's';
    cout << s << endl;
}
```

千万不要返回局部对象的引用或者指针！！！

例如：

```
const std::string &manip(const std::string &s)
{
    std::string ret = s;
    return ret;
}
```

上例中的 ret 是个局部对象，离开这个函数后就被销毁了，此时函数返回一个它的引用，实际上是引用了一块**非法**的内存区域。

函数重载：

名称相同，但是参数不同的函数成为函数的重载。在应用程序中，如果需要使用不同的参数调用具有特性名称的函数，函数重载就是很好的解决方案。

```
string sum(const string &a, const string &b){
    return a + b;
}
int sum(int a, int b){
    return a + b;
}
```

这里是两个求和的函数，实际调用时就可以根据参数的类型和个数来辨别应该调用哪个函数。

函数的返回值不能作为函数重载的依据！例如：

```
int sum(int a, int b){
    return a + b;
}

void sum(int a, int b){
    std::cout << (a+b) << std::endl;
}
```

函数的唯一标示包括函数名和形参表，不包括返回值。

后面我们接触到类的成员函数后，函数的唯一标识实际上还可能包含类名和 const 属性。

关于为什么 C 语言没有函数重载，我们到类和对象部分再详细解释。

函数指针

下面是一个冒泡排序，可以自定义排序规则：

```
inline void swap(int &a, int &b) {
    int temp(a);
    b = a;
    a = temp;
}

void bubble_sort(vector<int> &vec, bool (*cmp)(int, int)) {
    typedef vector<int>::size_type pos;
    for (pos ix = 0; ix < vec.size() - 1; ++ix) {
        for (pos iy = 0; iy < vec.size() - ix - 1; ++iy) {
            if (cmp(vec[ix], vec[iy])) {
                swap(vec[ix], vec[iy]);
            }
        }
    }
}
```

定义一个函数指针：

```
bool (*pf) (const std::string &, const std::string &);
```

这个函数指针的类型是什么？

```
bool (*) (const std::string &, const std::string &);
```

可以采用这种手段简化函数指针的定义：

```
typedef bool (*cmpFun) (const std::string &, const std::string &);
```

这句话并没有定义一个新的函数指针变量，而是把函数指针的名称简化为 cmpFun