函数一些需要注意的点

尽量使用常量引用

1. 若引用对象不需改变，最好采用const限定避免意外的bug。
2. 实参传给形参时会忽略掉形参顶层的const，意思是说若形参是const，实参就算不是const也可以传给形参。因此下面两个函数在编译时被认为是同一个函数：

```
void func(const int i){}
void func(int i){}
```

数组形参

1. 数组不允许拷贝：从实参到形参的拷贝；使用数组时（通常）会将其转换成指针。因此无法采用值传递的方式使用数组参数，**数组会被转换为指针**。所以当我们为函数传递一个数组时，**实际上传递的是指向数组首元素的指针**。
   尽管不能以值传递的方式传递数组，但可以把形参写成类似数组的形式：

```
//以下三种方式等价
void func(const int*);
void func(const int[]);
void func(const int[10]);   //这里的10表示我们期望数组含有多少元素，实际上并不一定需要10个元
```

2. 数组引用形参：

```
void print(int (&arr)[10]);
```

   上面的括号是必须的，**表示arr是具有10个整数的整型数组的引用**。所以这一用法也限制了数组的大小。
3. 传递多维数组时，实际上多维数组是数组的数组，所以**第二维数组（以及后面所有维度）的大小是整个数组类型的一部分，不能省略**。
   等价的定义：

```
void func(int (*matrix)[10]);   //指向含有10个整数的数组的指针
void func(int matrix[][10]);
```

main函数处理命令行选项

1. main函数的两个参数：

```
//两种表示方式
int main(int argc,char *argv[]);
int main(int argc,char **argv);
```

如命令行输入prog -d -o ofile data0，则argc=5表明有5个参数。其中argv的第一个元素指向**程序的名字**或者一个空字符串。接下来的元素依次传递命令行提供的实参。argv的最后一个元素之后的元素保证为0。
argv[0]="prog",argv[0]="-d",argv[0]="-o"...

含有可变形参的函数

1. 若函数的实参数量未知但是全部实参的类型都相同，则可以使用initializer_list类型作为形参。initializer_list是C++11的新标准，是一种标准库类型，用于表示某种特定类型的值的数组。实际上initializer_list是一种模板类型。如：

```
void func(initializer_list<int> listofint);
func({1,2,3});
func({1,2});
```

initializer_list包含begin和end对象，以及size。且initializer_list的值是**只读的**，故也不可声明为**非常量**引用。
若用vector代替initializer_list则在重载时可能出现问题：

```
void func(initializer_list<int> const &args);   //编译器最先匹配
//若无initializer_list，当调用func({1,2,3})时下面两者一样，产生冲突
void func(vector<int> const &args);
void func(list<int> const &args);
```

vector是可以修改的，所以必须在堆heap上分配空间来存储数据，initializer_list是不可改的，所以编译器可能可以将其存储在栈上来提高性能。
此外initializer_list是指针语义，里面的元素并不会被拷贝。如下：

```
int sum(initializer_list<int> const &nums){
    if(nums.begin()==nums.end()) return 0;
    initializer_list<int> next_seq(nums.begin()+1,nums.end());
    return *nums.begin()+sum(next_seq);
}
```

由于initializer_list是指针语义，所以递归调用时不用浅拷贝。

函数的返回值

1. 有的编译器无法检测出有返回值的函数是否定义了正确的return语句
2. 返回一个值的方式和初始化一个变量或者形参的方式完全一样。
3. 不要返回局部对象的引用或者指针
4. C++11新标准允许函数返回花括号包含的值的列表，值的类型根据返回类型决定。若有多个值，则返回类型需为vector；若返回类型为内置类型，则可以返回{一个值}
5. 如果控制到达main函数的结尾处而没有return语句，编译器将隐式地插入一条返回0的return语句，表示执行成功
6. 返回数组指针：
   - 使用类型别名：

     ```
     typedef int arr[10];    //arr是一个类型别名，表示的类型是含有10个int的数组
     using arr=int[10];  //与上面的声明等价，故*arr表示一个指向含有10个整数的数组的指针
     ```

   - 声明一个返回数组指针的函数：

     ```
     int *p1[10];    //表示p1是一个包含10个指针的数组
     int (*p2)[10];  //表示p2是一个指针，指向含有10个整数的数组
     ```

     和上面的声明一样，要定义一个返回数组指针的函数则数组的维度必须跟在函数名字之后，形式如此：**数组元素类型 (*函数名(参数列表))[数组维度]**。
     例如 int (*func(int i))[10]，可以这样理解:
     - func(int i) 表示调用func需要一个int类型的实参；
     - (*func(int i)) 表示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的调用结果将得到一个大小是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新标准定义的

     ```
     auto func(int i)-> int(*)[10]; //返回一个指针，指向含有10个整数的数组
     ```

     为了表示真正的返回类型跟在形参列表后面，用auto声明返回类型，而用->指明实际的返回类型。
   - 使用decltype：

     ```
     int arr[]={1,3,5,7,9};
     decltype(arr) *func(int i){ return &odd};
     ```

     *func表明函数返回的是指针，decltype(arr)表示指针所指的对象与arr的类型一致。因此func返回的是一个指针，指向含有5个整数的数组。

函数重载

1. **main函数不能重载**。

2. 对于重载函数来说，它们应该在**形参数量**或者**形参类型**上有所不同。
3. 形参中函数const时需注意：void func(const int)和void func(int)是一样的；
   而void func(const int&)和void func(int&) 还有void func(const int*)和void func(int*)则是不同的函数。编译器会优先选择非常量的版本
4. C++中名字查找发生在类型检查之前，一旦在当前作用域中找到了所需的名字，编译器就会忽略掉外层作用域的同名实体。

特殊的语言特性

1. 当设计含有默认实参的函数时，使用默认实参的形参只能出现在参数列表的后面。
2. 多次声明同一个函数也是合法的，不过在给定的作用域中一个形参只能被赋予一次默认实参，否则编译出错。同一个作用域下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，且该形参右边的所有形参都必须有默认值。

调试

1. assert宏（定义在cassert头文件中）：assert(expr)首先对expr表达式求值，若表达式为假则assert输出信息并终止程序的执行，否则assert什么也不做。输出信息如下：
   Assertion failed!
   Program: 文件路径
   File: 文件名 出错行数
   Expression: 出错的表达式
2. assert的行为依赖于一个名为NDEBUF的预处理变量的状态，若定义了NDEBUG则assert什么也不错。默认状态下没有定义NDEBUG则assert将执行运行时检查。用NDEBUG也可以编写自己的调试代码：

```
void function(){
    #ifndef NDEBUG  //若无定义NDEBUG则定义
        cerr<<__func__;
    #endif
}
```

   C++编译器定义了5个对程序调试很有用的名字：
   - __func__：当前调试的函数的名字；
   - __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；

函数匹配

1. 寻找最佳的匹配。若有如下重载函数：

```
void func(int a,int b){};
void func(double a,double b){}
func(1,2.1);
```

此时将会编译出错，因为重载函数有二义性。寻找最佳匹配时，对第一个实参来说最佳匹配是第一个函数；对第二个实参来说最佳匹配是第二个函数，有二义性。
因此虽然可以这样声明函数，但调用时却可能出错。

函数指针

1. 函数指针指向的是函数而非对象，指向的是某种特定类型。**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**。声明函数指针时括号必不可少————(*函数名)
2. 当把函数名作为一个值使用时，函数名自动转为函数指针，所以下面的赋值语句是等价的：

```
bool func(const string& s1,const string& s2){return s1==s2};
bool (*pf)(const string& s1,const string& s2);  //声明一个函数指针
pf=func;
pf=&func;

//可以直接含函数指针调用函数
bool b1=pf("hello","hi");
```

3. 指向不同函数类型的指针间不存在转换规则。但可以将指针赋值为空指针。
4. 函数指针作为形参：

```
//下面二者等价
void func(bool pf(const string& s1,const string& s2));
void func(bool (*pf)(const string& s1,const string& s2));
```

5. 可以借助typedef和decltype简化对函数指针类型的描述
6. 函数可以返回一个函数指针，但不可以返回函数类型。

```
using F=int(int,int);
using PF=int(*)(int,int);

PF func(int);    //正确
F *func(int);    //正确
F  func(int);    //错误

int (*func(int))(int,int);  //也可以如此显式声明
auto func(int)->int (*)(int,int);   //尾置返回类型
```

最后显式声明的解读：按照由内到外的顺序阅读声明，由于func有形参列表所以func是一个函数，func前面有*说明func是一个指针；指针的类型本身也包含形参列表所以指针指向函数，函数的返回类型是int。