# 1.函数基础：

通过调用运算符来执行函数，调用运算符的形式是一对圆括号，它作用于一个函数或指向函数指针的表达式；圆括号之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，实参是形参的初始值。传入的实参要能隐式转换为形参而且参数数量一致。

定义空形参列表：

void f1(){}//隐式定义

void f2(void){}//显示定义

## 1.1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，块中创建的自动对象的值就变成未定义的了。形参就是一种自动对象。如果变量定义本身含有初始值，就用这个初始值进行初始化；否则，如果变量定义本身不含初始值，执行默认初始化，这意味着内置类型的未初始化局部变量将产生未定义的值。

局部静态对象：在程序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。如果局部静态变量没有显式的初始值，它将执行值初始化，内置类型的局部静态变量初始化为0。

## 1.2函数声明：

函数只能定义一次，但可以声明多次。如果函数用不到，则可以只有声明没有定义。函数的声明可以省略形参列表。函数声明也称作函数原型。

如果把函数声明放在头文件中能确保同一函数的所有声明保持一致，而且一旦我们想改变函数的接口，只需改变一条声明语句。

## 1.3分离式编译：

分离式编译允许我们把程序分割到几个文件中去，每个文件独立编译，然后链接。分离出来的文件不用重复包含头文件。

# 2.参数传递：

传递指针的形式，如果在子函数修改指针，不能改变真正的指针，如果修改指针所对的值，能改变真正的指针。

## 2.1const形参和实参：

当用实参初始化形参时会忽略掉顶层const。

void fun(const int i){}

void fun(int i){}//错误：重复定义了fun，顶层const被忽略掉了

C++语言中允许我们定义若干具有相同名字的函数，不过前提是不同的函数形参列表或返回值。

## 2.2const形参和实参：

void reset(int \*p);

void reset(int &p);

int I = 0;

const int ci = I;

reset(&ci);//错误，不能用指向const int对象的指针初始化int\*

reset(i);//正确

reset(ci);//错误，不能把普通引用绑定到const对象ci上

reset(42);//错误，不能把普通引用绑定到字面值上

想要调用引用版本的reset，只能使用int类型的对象，不能使用字面值、求值结果为int的表达式、需要转换的对象、const int类型的对象；想要调用指针版本的reset，只能用int\*。

使用引用而非常量引用会极大地限制函数所能接受的实参类型，不能把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。

## 2.3数组形参：

不允许直接拷贝数组，而是转换成数组首元素指针的形式（传入数组的引用可以破除，但是也限制了函数的可用性），数组名就是0号元素的地址。当编译器检查传入参数时，只看类型对不对，不看是否是数组。以下三种形式等价：

void print(const int \*a);

void print(const int a[]);

void print(const int a[10]);//这里的维度表示我们期望数组含有多少元素，实际不一定。

数组尺寸一开始未知，有如下三种方式：

2.3.1使用标记指定数组长度：

void print(sonst char \*cp)

{

if(cp)//若cp不是空指针

while(\*cp) //若cp所指字符不是空字符（即\0）

cout<<\*cp++;

}

2.3.2使用标准库规范：传递指向数组的首元素和尾后元素的指针

2.3.3显式传递一个表示数组大小的形参：

对于多维数组的传递，数组第二维以及第二维以上的维度的大小都是数组类型的一部分，不能省略，有如下方式：

void print(int (\*matrix)[10],int row);//指向含有10个整数的数组的指针

void print(int matrix[][10],int row);

## 2.4main：处理命令行选项

假定main函数位于可执行文件prog之内，我们可以向程序传递下面的选项：

prog –d –o ofile data0

这些命令行选项通过两个可选的形参传递给如下编写的main函数。

int main(int argc,char \*argv[])//第二个相当于char \*\*argv

argc表示数组中字符串的数量，argv最后一个指针之后的元素值保证为0.如下：

argv[0] = “prog”;

argv[1] = “-d”;

argv[2] = “-o”;

argv[3] = “ofile”;

argv[4] = “data0”;

argv[5] = 0;

当使用argv中的实参时，一定要从argv[1]开始，argv[0]保存程序的名字。

## 2.5含有可变形参的函数：

2.5.1initializer\_list:

如果所有实参类型相同，可以传递一个名为initializer\_list（定义在同名头文件中）的标准库类型，如果实参不同，可以编写可变参数模板。

initializer\_list是一种标准库类型，用于表示某种特定类型的值的数组。

|  |  |
| --- | --- |
| initializer\_list<T> lst; | 默认初始化，T类型元素的空列表 |
| initializer\_list<T> 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 |
| lst2(lst); | 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 |
| lst2 = lst; |
| lst.size(); |  |
| lst.begin(); |  |
| lst.end(); |  |

initializer\_list中的元素永远是const。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堆花括号内，也可以同时拥有其他形参：

if(expected != actual)

error\_msg(int(42),{“functionX”, expected,actual });

else

error\_msg(int(42),{“functionX”,”okay”});//msg的形参就是initializer\_list类型

2.5.2省略符形参：

为了便于C++程序访问某些特殊的C代码而设置的，这些代码使用了名为varargs的C标准库功能，仅仅用于C和C++通用的类型。省略符形参只能出现在形参列表的最后一个位置。

void fun(int a,…);//省略部分不需要类型检查，形参后面的逗号是可选的。

# 3.返回值类型和return语句：

## 3.1无返回值函数：

返回void的函数不要求非得有return语句，因为在这类函数的最后一句后面会隐式执行return。

## 3.2有返回值函数：

有返回值函数如果按无返回值函数返回，编译器会报错，反之亦然。有返回值的函数要求得有return语句（main语句除外，会自动补return 0），但是编译器也许能检测到这个错误，也许不能，如果没检测到，则运行时的行为将是未定义的。

返回的值用于初始化调用点的一个临时量，该临时量就是函数调用的结果。

不要返回局部对象的引用和指针，因为函数终止意味着局部变量的引用将指向不再有效的内存区域。

函数的返回值类型决定函数调用是否是左值，调用一个返回值引用的函数得到左值，其他返回值得到右值。我们能为返回类型是非常量引用的函数的结果赋值,如果返回值是常量引用，我们不能给调用的结果赋值。

char &get\_val(string &str,string::size\_type ix)

{

return str[ix];

}

int main()

{

string s("a value");

get\_val(s,0) = 'A';

cout<<s<<endl;

return 0;

}

C++11新标准规定函数可以返回花括号包围的值的列表，类似于其他返回结果，此处的列表也用来对表示函数返回的临时量进行初始化。如果列表为空，临时量执行值初始化；否则，返回的值由函数的返回类型决定。

对于main函数，返回值0表示执行成功，返回其他值表示执行失败，其中非0的具体含义依据机器而定。为了使返回值与机器无关，stdlib.h定义了两个预处理变量EXIT\_FAILURE和EXIT\_SUCCESS。

## 3.3返回数组指针：

要求明确数组的尺寸。

3.3.1使用类型别名：

要想定义一个返回数组的指针或引用的函数比较麻烦，但是有一些方法可以简化这一任务，最直接的方法就是使用类型别名。

typedef int art[10];//表示含有10个整数的数组

using arrT = int[10];// 表示含有10个整数的数组

arrT\* func(int i);///返回一个指向10个整数的数组的指针，因为我们无法返回数组，所以返回数组的指针。

3.3.2声明一个返回数组指针的函数：

Type (\*function(parameter\_list))[dimension];

Type表示元素类型，dimension表示数组大小，function是函数。

3.3.3使用尾置返回类型：

这种形式对于返回类型比较复杂的函数最有效，尾置返回类型跟在形参列表后面并以一个->符号开头。为了表示函数真正的返回类型跟在形参列表之后，我们在本该出现返回类型的地方放置一个auto。

auto fun(int i) -> int(\*)[10];//返回一个指向10个整数的数组

3.3.4使用decltype：

如果我们知道函数返回的指针指向哪个数组，就可以使用decltype关键字声明返回值。

string str[10];

decltype(str) &fun();//要是引用得加上引用符号

# 4.函数重载：

main函数不能重载。不允许两个函数除了返回类型外其他所有的要素都相同，如果出现，则只有第一个是正确的。有时候两个形参列表看起来不一样，但实际上是相同的：

Record lookup(const Account &acct);

Record lookup(const Account &);//省略了形参的名字，形参的名字只起到帮助记忆的作用，没有它并不影响形参列表的内容。

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；一个拥有底层const的形参可以通过区分其指向的是常量对象还是非常量对象可以实现函数重载。

尽管函数重载能在一定程度上减轻我们为函数起名字、记名字的负担，但是最好只重载那些确实非常相似的操作，有时给函数起不同名字能使得程序更容易理解。

函数匹配也叫重载确定。有多个函数可以匹配，但是每个都不是最佳选择，此时将发生错误，叫二义性调用。

对于重载，我们要注意其作用域，如果我们在调用函数的层中有相同名字的变量声明、有相同名字的函数声明，它将隐藏外层作用域中声明的同名实体，剩下的工作就是检查函数调用是否有效了。

# 5.特殊用途语言特性：

## 5.1默认实参：

函数中多次调用它们都被赋予一个相同的值，这个反复出现的值被称为函数的默认实参。调用含有默认实参的函数时，可以包含该实参，也可以省略。一旦某个形参被赋予了默认值，其后所有的形参都必须有默认值，这会让调用的时候有些理解上的问题，否则就是不合理但合法的默认实参。

多次声明一个函数是合法的，不过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。

string screen(int ht,int wid = 80,char backgrnd = ‘ ’);//通过声明函数来设置默认实参，ht没有默认值

函数调用时实参按其位置解析，默认实参负责填补函数调用缺少的尾部实参。

window = screen(,,’?’);//错误只能省略尾部的实参

局部变量不能作为默认实参，用作默认实参的名字在函数声明所在的作用域内解析，而这些名字的求值过程发生在函数调用时。

//wd、def、ht的声明必须出现在函数之外，为全局变量

int wd = 80;

char def = ‘ ’;

int ht();

string screen(int = ht(),int = wd,char = def); void f()

{

def = ‘\*’;//该变了实参的默认值

int wd = 100;//隐藏了外层定义的wd，但是没有改变默认值

window = screen();//调用screen(ht(),80,’\*’);

}

## 5.2内联函数：

调用函数比求等价表达式的值要慢一些，大多数机器上，一次函数调用包含着一系列工作：调用前线保存寄存器，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运行。而内联函数可以避免函数调用的开销，内联函数是把它在每个调用点上内联地展开，在编译时将函数体嵌入在每一个调用处，类似宏替换。内联函数有函数的结构，在编译后却没有函数的性质。

只需要在函数返回值前加inline就可以声明称内联函数。一般来说，内联机制用于优化规模较小、流程直接、频繁调用的函数。

## 5.3constexpr函数：

用于常量表达式的函数。要求函数的返回类型及所有形参的类型都得是字面类型，而且函数体中必须有且只有一条return语句。执行初始化任务时，编译器把对constexpr函数的调用替换成其结果值。为了在编译过程中随时展开，constexpr函数被隐式地指为内联函数。

constexpr函数体内也可以包含其他语句，只要这些语句在运行时不执行任何操作就行。比如可以有空语句、类型别名、using声明等。

constexpr int new\_sz(){return 42;}//constexpr函数

constexpr int foo = new\_sz();//正确，foo是一个常量表达式

constexpr int scale(int cnt){return new\_sz()\*cnt;}//如果cnt是常量表达式，则scale(cnt)也是常量表达式

int I = 2;

int a[scale(i)];//错误，如果scale的实参是常量表达式时，它的返回值也是常量表达式，反之不然

因为二者都可以在程序中多次定义，它的多次定义必须完全一致，所以内联函数和constexpr函数通常定义在头文件中。

## 5.4调试帮助：

5.4.1assert预处理宏：

定义在assert.h头文件中，是一个预处理变量，宏名字在程序内必须唯一，行为类似于内联函数。

assert(expr); 如果表达式为假，assert输出信息并终止程序的执行；如果表达式为真，assert什么也不做。常用于检查不能发生的条件。

5.4.2NDEBUG预处理变量：

①assert依赖于NDEBUG预处理变量，如果定义了NDEBUG，则assert什么也不做，默认状态下没有定义NDEBUG，assert将执行。我们可以用#define语句定义NDEBUG。

②使用NDEBUG编写自己的条件调试代码，如果NDEBUG未定义，则执行#ifndef和#endif之间的代码。

预处理器定义了几个对于程序调试很有用的名字，输到cerr流中：

①\_\_func\_\_：存放当前函数的名字

②\_\_FILE\_\_：存放文件名的字符串字面值

③\_\_LINE\_\_：存放文件编译时间的字符串字面值

④\_\_DATE\_\_：存放文件编译日期的字符串字面值

# 6.函数匹配：

当几个重载函数的形参数量相等或形参类型可以转换得来时，需要做一下几个工作：

①确定候选函数：选定本次调用对应的重载函数集。

②确定可行函数：从候选函数中选出能被这组实参调用的函数。要求参数数量相等、类型相等或可以隐式转换。如果没找到，编译器报错。

③寻找最佳匹配：

实参类型与形参越接近，匹配越好。如果要匹配含有多个形参的函数，则需要找满足如下条件的函数：

该函数的每个实参的匹配都不劣于其他可行性函数需要的匹配。

至少有一个实参的匹配由于其他可行函数提供的匹配。

如果在检查了所有实参后没有任何一个函数脱颖而出，则编译器报告二义性调用的信息。

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成了几个等级，顺序如下：

①精确匹配：实参和形参类型相同、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（顶层有和没有都一样）。

②通过const转换实现的匹配：

如果重载函数的区别在于它们的引用或指针类型的形参是否引用了const，则当调用发生时编译器通过实参是否是常量来决定选择哪个函数。

③通过类型提升实现的匹配：

小整型一般会提升到int类型更大整数类型。有时传进来的实参即使是一个很小的整数值，也会直接将它提升成int类型，此时使用short版本反而会导致类型转换。

④通过算术类型转换或指针转换实现的匹配：

所有算数类型转换的级别一样。

void manip(long);

void manip(float);

manip(3.14);//3.14的类型是double，它既能向float转换也能向long转换，所以产生二义性

⑤通过类类型转换实现的匹配

# 7.函数指针：

函数指针指向的是函数而非对象，函数的类型由它的返回类型和形参类型共同决定。想要声明一个可以指向该函数的指针，只需要用指针替换函数名即可。

bool lengthcompare(string &,string &);

函数类型为：bool (string&,string &)，该函数的指针可以为：

bool (\*pf)(const string &,const string &);//返回值为bool的函数的指针，是指针

bool \*pf(const string &,const string &);//返回值为bool的指针的函数，是函数

当我们把函数名作为值使用时，该函数自动地转换为指针。还能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，也可以指向nullptr或0表示该指针没有指向任何一个函数。在指向有不同返回值、形参的函数的指针间不存在转换规则。

pf = lengthcompare;//声明函数指针，

bool b = pf(“hello”,”goodbye”);//调用函数

bool b = (\*pf)(“hello”,”goodbye”);//一个等价的调用

如果定义了指向函数的指针，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。

void ff(int\*);

void ff(unsigned int);

void (\*pf1)(unsigned int) = ff;

void (\*pf2)(int) = ff;//错误，形参不匹配

double (\*pf3)(int\*) = ff;//错误，返回值不匹配

和数组类似，虽然不能定义函数类型的形参，但是形参可以指向函数的指针，此时形参看起来是函数类型，实际上确实当成指针来用。

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

void useBigger(const string &s1,bool (\*pf(const string &,const string &));

PS:直接使用函数指针类型显得冗长而繁琐，类型别名和decltype能让我们简化。

//以下两个声明等价：

typedef bool(\*func)(const string &,const string &);

typedef decltype(lengthcompare) \*func;//注意decltype返回函数类型，不会将函数类型自动转换成指针类型。

我们可以直接把函数作为实参使用，此时它会自动转换成指针，

useBigger(s1,lengthcompare);

和数组类似，虽然不能返回一个函数，但是可以返回函数类型的指针。返回函数类型不会自动转换成指针，我们必须显式地将返回类型指定为指针，而形参函数类型可以。

PS：想要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);//函数类型

using PF = int (\*)(int\*,int);//函数指针类型

PF f1(int);//正确

F f1(int);//错误，F是函数类型，f1不可能返回一个函数

F \*f1(int);//正确

我们也可以用下面的形式直接声明f1：

int (\*f1(int))(int \*,int);//右边的括号是形参列表，f1是个函数，最前面的int代表函数返回值是int，

还可以以一种尾置返回类型的方式声明一个返回函数指针的函数：

auto f1(int) -> int(\*)(int\*,int);

如果我们明确知道返回的函数是哪一个，就能使用decltype简化书写函数指针返回类型的过程。但要牢记decltype作用于某个函数时，它返回函数类型而非指针类型。

int sumlength(string &,string &);

decltype(sumlength) \*getfun(const string &);//返回sumlength类型的函数