详解char

字符的表现形式有两种：unsigned char 和 signed char.

char的类型是unsigned char 还是 signed char 取决于编译器。

int i = 128;//1 0000 0000

char c = i;// 0000 0000 -> 0111 1111 -> 1000 0000 = -128

printf("c = %d\n", c);

输出结果是 -128.

vs2017把char类型当做 signed char 类型。

i ：00000000 00000000 00000000 10000000

c：10000000

计算机如何解释c的8位二进制表示的十进制整数？

符号位是1，说明表示的是负数，这是补码。补码-1取反等于原码，所以

原码是 10000000 ----->  01111111 ----->10000000 即 -128

(符号位不变，其他位取反得到反码，反码＋1得到补码)

基本数据类型的类型转换

C++不会直接把两种不同类型的值相加，而是先根据类型转换的法则转换成同一类型后再相加

算术转换

1, char, unsigned char, signed char, short, unsigned short

char, unsigned char, signed char, short, unsigned short 提升成int 类型或 unsigned int 类型。

wchar\_t  char16\_t  char32\_t 提升成int，unsigned int ，long，unsigned long，long long， unsigned long中最小的一种类型。

2, int  ,unsigned int , long , unsigned long, long long ,unsigned long long

    不看类型！只看所占字节宽度！

    类型所占字节宽窄不同：窄字节转成宽字节

    类型所占字节宽窄相同：有符号转成无符号

举例：

   long 和 unsigned int，如果long是4字节则 long 转换成 unsigned int ；如果long是8字节，则unsigned int 转换成 long 。

 unsigned int 和 int  转成int 转换成 unsigned int 。

3，整型与浮点型：整型转成浮点型。

4，float 转换成 double 。

无符号整数的雷区

切勿混合使用无符号类型和有符号类型，否则产生极大的错误。严格来说，无符号数就是地雷，格外小心。

有符号数转换成无符号数注意点

1，

unsigned int u = 10;

int b = -42;

std::cout << b + b << std::endl;//-48

std::cout << u + b << std::endl;//4294967264

2，

无符号整数，必须大数 - 小数；否则，得到一个“意外”的整数。

3，

for(unsigned int i = 10; i >= 0 ; --i) ; //死循环

#include<iostream>

using namespace std;

int main() {

unsigned int u1 = 10;

unsigned int u2 = 20;

cout << u2 - u1 << endl;//10

cout << u1 - u2 << endl;//4294967286

unsigned int u3 = 1;

int i = -2;

cout << u3 + i << endl;//4294967295

system("pause");

}

常量表达式和constexpr

Q：如何检测一个表达式或者对象是常量？

A：int a[表达式];编译通过是，否则，不是。

------------------------------------------------------

什么是常量表达式？

值不能被修改，编译期就能计算出其值。 或

值不能被修改，使用常量表达式初始化。

const int a = 20;//是

const int b = a + 1;//是

int c = 20;//否，值可被修改

const int d = f( );//否，编译器无法计算出其值

---------------------------------------------------------

Q：数组必须使用常量表达式指定大小，那么只要是const变量，即可作为数组的大小，对吗？

A：错，const修饰的变量，表示其运行期不能被修改，但不代表它是常量表达式。

-----------------------------------------------------------------------------------------

（1）const int val = f();

（2）constexpr int val = 100;

Q：（1）和（2）的区别？

A：（1）const变量既可以用字面值，常量表达式，也可以用函数的返回值等编译期不能计算出结果的表达式初始化，它表示val运行期不能被修改，是常变量，是种数据类型。未必能做数组的大小。

（2）constexpr变量，严格说已经不是变量了，既表示不能被修改，也表示它一定是常量，必须用编译期能计算出值的常量表达式初始化。一定能做数组的大小。

Q：const，constexpr，常量，三者的关系。

A：const可能是常量，constexpr一定是常量。

constexpr指针和引用

constexpr 指针 必须用0或nullptr初始化 或 存储位置固定的变量地址，例如，函数体外定义的变量。函数体内的变量地址不固定，程序运行过程中可能发生变化。

int x = 1;

int y = 2;

int main() {

const int \*p = &x;

constexpr int \*pp = &y;

\*p = x \* 100;//error

\*pp = y\*100;//OK

}

constexpr函数

constexpr修饰的函数，如果其传入的参数可以在编译时期计算出来，那么这个函数就会产生编译时期的值。但是，传入的参数如果不能在编译时期计算出来，那么constexpr修饰的函数就和普通函数一样了。不过，我们不必因此而写两个版本，所以如果函数体适用于constexpr函数的条件，可以尽量加上constexpr。而检测constexpr函数是否产生编译时期值的方法很简单，就是利用std::array需要编译期常值才能编译通过的小技巧。这样的话，即可检测你所写的函数是否真的产生编译期常值了。



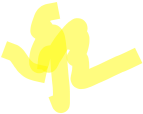
int f() {

return 3;

}

constexpr int g(int i) {

return i;



}

int main() {

int x = 100;

 const int y = 100;



 constexpr int size = f();//error







const int size1 = f();



 array<int, size1> a;//error

const int size2 = g(10);



array<int, size2> aa;//ok



constexpr int size3 = g(10);



array<int, size3> aaa;//ok

constexpr int size4 = g(x);//error

constexpr int size4 = g(y);//ok

}

引用

一维数组

int a[10]

a是数组名，a是指针常量，类型是数组元素类型，值是数组第一个元素的地址（第一个byte的地址），存储于静态存储区，编译器为其附有数组长度属性。

a用于表达式时，a是指针常量，普通的一个指针。sizeof和&除外。

sizeof(a)，整个数组总字节数目。&a,数组指针。

int \*c = &a[0] 等价于 int \*c = a;

a不是指针变量，++a或a = ?;//错误，常量不能被修改

array[index] 等价于 \*(array+index)

int \*c = a+2;//OK

二维数组和数组指针

int a[2][3] = {{1,2,3},{4,5,6}};

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 |

a是数组名，数组成员是含有3个元素的数组，是数组指针常量。

a的类型是int(\*a)[3]类型，a的值是a[0][0]的第一个字节的地址。

a+1的类型是int(\*a)[3]类型，a+1的值是a[1][0]的第一个字节的地址。

\*a的类型是int\*类型，\*a的值是a[0][0]的第一个字节的地址。

\*a+1的类型是int\*类型，\*a的值是a[0][1]的第一个字节的地址。

a[1][1] 相当于 \*(\*(a+1)+1)。

向函数传递二维数组，必须指定第二维的大小。int(\*a)[10]

存储区和初始化

const

1，const修饰成员函数

const修饰的成员函数不能修改数据成员，不能返回类数据成员的引用，除非返回类成员的const引用，只能调用const成员函数。成员函数能用const修饰就必须用。

void fun（）const 和 void fun（）构成重载函数。

const对象只能调用const函数，非const对象优先调用非const重载的fun函数。

2，const修饰数据成员

声明时const int val = 100;或者定义时在初始化列表中初始化（必须用初始化列表）。

声明时static const int val;const int class::val = 100；

3，const修饰返回值

函数参数

const int \* ptr;

const

auto和decltype

static

静态数据区的内存默认初始化为0x00，所以全局变量和static局部变量默认初始化为0；

static修饰全局变量和全局函数，该变量和该函数只能在本源文件中使用，其他源文件不得使用，避免了命名冲突。

static修饰局部变量，存储在静态数据区，程序运行时存在，生存期是整个程序运行期间，但是作用域不变。定义时必须初始化，且只初始化一次，函数下次被调用，局部变量是在上次函数运行的结果的基础上运算的。

static修饰类的数据成员，静态数据成员。为所有对象共享，只有一份，任何对象都能修改之。

如果定义成public，则对象可通过类型+域作用符访问。一般定义静态数据成员的访问权限为private，定义public static函数访问它。静态成员在类中声明，但必须在类外定义(.cpp文件)，定义方式是int 类名::变量名 = value.static const int val，则定义方式是const int类名::val = 100;声明只代表占用了标识符，但内存中并未开辟空间，在.cpp文件定义后才开辟内存。

static修饰成员函数，静态函数，不得用const修饰，不能是虚函数。静态函数不能调用非静态成员，非静态函数能调用静态成员。

初始化列表

只能被初始化列表初始化的情况？

初始化列表提高效率的原理？

命名空间

作用域：局部变量可以屏蔽全局变量、嵌套层次中的内层变量可以屏蔽外层变量，从而被屏蔽的全局或外层变量在其倍屏蔽的区域内是不可见的。屏蔽全局变量的局部代码块不算是全局变量的作用域。

命名冲突：两个相同的标志符，作用域有重叠。标志符的重叠频繁发生在第三方库中，他们使用了相同名字的全局标志符，这将导致编译错误。

多个标准库，含有相同的标志符，如果没有命名空间，则在main中发生冲突，编译错误。有了命名空间，在使用一个标志符时，加上命名空间，就能区分出唯一的标志符。

如果存在命名冲突的可能，尽可能在成员前面加上命名空间的名字和二元作用域分辨运算符（::）。

一般情况下，对偶尔使用的命名空间成员，应该使用命名空间的作用域解析运算符来直接给名称定位。而对一个大命名空间中的经常要使用的少数几个成员，提倡使用using声明，而不应该使用using编译指令。只有需要反复使用同一个命名空间的许多数成员时，使用using编译指令，才被认为是可取的。不在头文件中使用using指令和using声明。

std1头文件的命名空间std1有标志符i；

std2头文件的命名空间std2有标志符i；

使用std1.和istd2.i区分开。

命名空间就像在程序中垒起一堵墙，把标志符围起来，每个标志符增加修饰词，使其独一无二。

但这仍旧不能100%保证唯一性，两个第三方供应商可能在命名他们的命名空间时使用了相同的标志符。

#include<iostream>

using namespace std;

int integer1 = 98;

namespace Example {

const double PI = 3.1415926;

const double E = 2.71828;

int integer1 = 8;

void printValues();

namespace Inner {

enum Years{FISCAL1 = 1990,FISCAL2,FISCAL3};

}

}

namespace {

double InUnnamed = 88.92;

double likeStatic();

}

namespace Example {

class Cat {};

int integer2 = 222;

}

int main() {

cout << "InUnnamed = " << InUnnamed << endl;

cout << "(global)integer1 = " << integer1 << endl;

cout << "(Example)integer1 = " << Example::integer1 << endl;

cout << "FISCAL3 = " << Example::Inner::FISCAL3 << endl;

Example::printValues();

}

void Example::printValues() {

cout << "i am namespace Example" << endl;

}

1. 无名命名空间

无名命名空间的标志符，就像加了static修饰一样，仅限于本文件使用。

1. 相同命名空间名

自动合并到同一个命名空间。[C++](https://baike.baidu.com/item/C%2B%2B" \t "_blank)[标准程序库](https://baike.baidu.com/item/%E6%A0%87%E5%87%86%E7%A8%8B%E5%BA%8F%E5%BA%93)中的所有标识符都被定义于一个名为std的namespace中。

异常机制

（1）异常的处理的目的

让一个函数在发现了自己无法处理，或不知道如何妥善的最优化的处理的错误时，抛出（throw）一个异常，然后它的（直接或者间接）调用者能够处理这个问题。 将问题检测和问题处理相分离。

在C语言的世界中，对错误的处理总是围绕着两种方法：一是使用整型的返回值标识错误；二是使用errno宏（可以简单的理解为一个全局整型变量）去记录错误。抛出异常是第三种更好的错误处理机制。

（2）try catch

catch处理器的参数是引用，是throw出的异常对象，它并没有复制和创建新的异常对象。catch处理器可以捕获异常并进行不彻底的异常处理，重新抛出异常。try{ }语句块若没有抛出异常，try语句块执行完毕，忽略所有的catch处理器，继续往下执行。try{ }语句块若抛出异常，说明try{ }语句块内调用了可能抛出异常的函数E，且函数E抛出了异常，

第一步：销毁函数E所有的局部变量。

第二步：暂停并跳出try语句块的执行流

第三步：逐一向下匹配catch处理器，遇到第一个该类或基类的异常引用，执行该catch处理器。

第四步：忽略剩下的所有catch处理器，紧接着执行最后一个catch处理下面的第一条语句。

如果没有匹配的处理器，会对try语句块所在函数进行堆栈展开。

（3）堆栈展开

函数B是可能抛出异常的函数，函数A调用函数B，如果函数A没把函数B try并catch，那么函数A也变成了可能会抛出异常的函数。如果函数A把try{调用函数B的语句}catch（except &e）,那么函数A是一个不会抛出异常的函数。

函数C抛出异常，销毁函数C的所有局部变量，如果函数B没有捕获C，销毁函数B的所有局部变量，如果调用函数B的函数A没有捕获异常，销毁函数B的所有局部变量，如果捕获了，则执行catch处理器。直至main函数，如果main函数也没捕获，则终止整个程序。

//Cat.h

#include<iostream>

class Cat{

private:

int id;

public:

Cat(int id) {

this->id = id;

}

~Cat() {

std::cout << id << " cat is destroied\n";

}

};

#include<iostream>

#include<stdexcept>

#include"Cat.h"

#include<cstdlib>

using namespace std;

void f3() {

Cat c3(3);

throw runtime\_error("runtime\_error in f3");

}

void f2() {

Cat c2(2);

f3();

}

void f1() {

Cat c1(1);

f2();

}

int main() {

Cat c0(0);

try {

f3();

}

catch (runtime\_error &e) {

cout << e.what() << endl;

}

try {

f1();

}

catch (runtime\_error &e) {

cout << e.what() << endl;

}

system("pause");

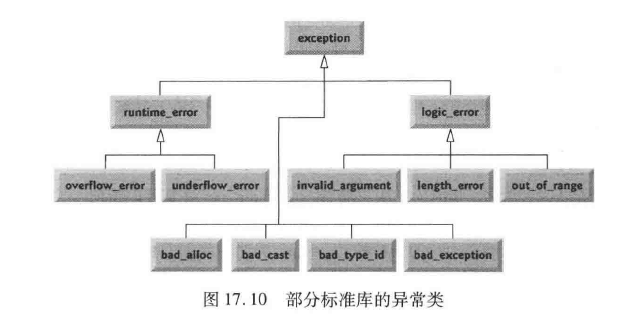
}

3 cat is destroied  
runtime\_error in f3  
3 cat is destroied  
2 cat is destroied  
1 cat is destroied  
runtime\_error in f3  
请按任意键继续. . .

C++11，不会抛出异常和调用抛出异常的函数的函数，应该显示的声明为noexcept

标准库的异常层次结构

except(<exception>)  runtime\_error和logic\_error(<stdexcept >)



eg1：列出几个常见的异常例子

有C++运算符抛出的标准异常：new抛出的bad\_alloc，dynamic\_cast抛出的bad\_cast异常，typeid抛出的bad\_typeid异常。

eg2：请给出理由，说明为什么异常处理的技术不应该用于普通的程序控制。

一般的小错误，应该函数内自行处理，或者通过返回值检测函数是否执行成功，最后再考虑使用异常处理机制。

异常与继承

catch处理器的参数若是基类引用，一旦接受派生类异常，可以很好的发挥多态性来处理异常。

思考题？为什么静态函数不能用const修饰。