# C/C++基础知识

## 关键字及类型

### float类型的存储方式

### float 数据类型

(1)123456.0f

转化1、整数部转化为二进制表示：1 11100010 01000000.0

转化2、将小数点向左移，一直移到离最高位只有1位:1.11100010010000000 \* ( 2 ^ 16 )

转化3、指数位(30-23位):16, 11100010010000000后面补零补够23位:11100010010000000000000

转化4、指数位+127:143:10001111, 由于符号位0

最后结果01000111 11110001 00100000 00000000 转化16进制:47 F1 20 00,最后翻过来：00 20 F1 47(小端存储)

0.0f对应的十六进制是00 00 00 00

(2) //可以利用联合体把内存中4个字节转化浮点数

union FloatBits

{

float fValue;

unsigned char szBits[4];

};

1. 浮点型比较

float f1 = 1.200000012;

Float f2= 1.199999999;

if (abs(f2 - f1) > 0.000001); //精确度，则认为f2与f1不相等。

1. 浮点型做除法取整

float f1 = 0.019999771118166976;

float f2= 0.02;

错误例子：如果直接(int)(f1/f2)则为0。

正确例子：

Int nPresion = 3; //精度

While(nPresion-- > 0)

{

f1\*=10;

f2\*=10;

}

f1 += 0.5;

f2 += 0.5

(int)f1/(int)f2则为1。

### explicit关键字

explicit一般用在类的单参数构造函数前面, 作用是禁止隐式转换。

例如:

class CTest

{public:

explicit CTest(int i);};

CTest test = 10; //禁止,这是隐式类型转换

CTest test(10); //允许

### volatile关键字

该关键字的作用:编译器的优化会从寄存器读值，导致不正确。而volatile修饰的变量不会使用编译器的优化， 每次直接从地址里读值。

例如:

int i = 10;

int a = i;

printf( "i= %d", a );

\_\_asm //修改内存值，不然编译器知道

{ mov dword ptr [ebp-4], 20h }

int b = i;

printf( "i= %d", b );

然后，在调试版本模式运行程序，输出结果如下：

i = 10, i = 32

然后，在release版本模式运行程序，输出结果如下：

i = 10, i = 10 若给 i变量修饰volatile, 则都输出i = 10, i = 32

### inline 关键字

可使编译器强制生成内联:\_\_forceinline

在这种简单函数情况下，使用内联节省指针在堆栈中移动的时间。

缺点是增加了文件的大小。

循环中调用内联函数时， 函数只展开一次。

内联函数应该和调用者放在同一实现文件中。

### static关键字

静态局部变量:在函数体内初始化一次，多次调用此函数，维持此变量不变如,也就是作用返回是是局部的， 生命周期在退出程序后结束。

fun()

{static a = 0;a++;}调用5次， a的值为5；

### const 关键字

(1)全局const存在常量区是常量。

(2)局部const不存在常量区是变量，可修改。

(3)const int i = 5; //局部const

int \*p = (int\*)&i;

\*p = 6;

int g = i; //i变量存放是的6，但还是g = 5,因为编译器对局部const的所有i进行了5替换(类似宏替换)。

(4)const 修饰函数参数时， 不能构成重载。

(5)void fun(int) const 与 void fun(int), 构成重载。

### mutable关键字

用来修改常对象中的成员变量， mutable的反义词就是const。

### typename 关键字

typename关键字告诉了编译器把一个特殊的名字解释成一个类型,在模板中可以代替class

解决歧义---类型名指示符

//该T::int \*p;存在歧义，编译器默认Data是一个变量， 则\*变成了乘号

template<class T>

class A

{public:

T::Data \*p;};

//使用typename告诉T::Data是-一个类型名， p指向T::Data类型的对象的指针。

template<class T>

class A

{public:

typename T::Data \*p;};

### typedef 关键字

typedef 类型 定义名; //typedef int INT; 函数指针类型：typedef void(\*fun)(void); 其中fun相当于定义名。

类型说明只定义了一个数据类型的新名字而不是定义一种新的数据类型。定义名表示这个类型的新名字。

### new关键字

malloc分配失败后，返回空指针。

new分配失败后，不返回空指针。

new(std::nothrow) ,分配失败后，返回空指针。

只能处理new分配失败, 回调函数必须是参数和返回值都为空

my\_new\_handler Old\_Handler\_= set\_new\_handler(my\_new\_handler);

//设置new分配失败的处理函数my\_new\_handler

void my\_new\_handler()

{

//错误处理(1.直接退出程序，2.释放一部分内存。3.抛出bad\_alloc异常或其派生异常)。可抛出给自定义异常类。

}

重载new的三种形式:

void\* operator new(std::size\_t size) throw(std::bad\_alloc); //内存分配失败抛出bad\_alloc异常，用new\_handler()捕获,(推荐)

void\* operator new(std::size\_t size, const std::nothrow\_t&) throw(); //不抛出异常， 分配失败返回0。

void\* operator new(std::size\_t size, void\* ptr) throw(); //在指定的内存地址分配空间。char buf[20]; void\*p = new(buf)char[50]; p指针等于buf。

静态成员operator new 则调用new创建该类时，则调用该类的静态成员operator new，若没则调用全局operator new。

### final

class A final:表示类A不会被继承，一般不用final修饰类， 除非一定不会继承或安全设计。

### \_\_super::

\_\_super:function()调用父类的函数。

### 类型转换

(1) static\_cast不能转换掉f的const、volatile、或者\_\_unaligned属性。并不做运行时类型检查。

int i;

float f = 1.23;

i = static\_cast <int>(f);

(2) reinterpret\_cast 以二进制形式进行转换。

int i;

char \*p = "This is an example.";

i = reinterpret\_cast<int>(p);

## 指针及内存管理

指针变量(函数指针变量除外)可以比较、相减、指向同一地址，但不能相加。

### 函数指针

函数指针保存的是代码段内的地址信息,函数指针不能做加减运算。

|  |  |  |
| --- | --- | --- |
|  | 声明指针变量类型 | 定义 |
| 非成员函数指针 | typedef void (\_cdecl \*fun)(); //调用约定可选， 默认和编译器设置相同  fun f = function;//赋值,这里使用\*或者&或者不用，指针值都一样。  f();//使用 | void (\_cdecl \*fun)() = function  fun(); |
| 成员函数指针 | typedef void (\_cdecl C::\*fun)();  fun f = &C::fun;//必须只能加&  C cla;  (cla.\*f)(); //使用 | C cla;  void (\_cdecl C::\*fun)() = &C::function;  (cla.\*fun)();//使用 |

函数指针指向类成员函数:C++11

#include <functional>

typedef std::function<void()> fp; //<返回值(参数)> //函数指针类型

//绑定全局或类静态函数。

fp f2 = g\_fun;

f2();

/\*绑定类成员函数, 对于参数要使用占位符 std::placeholders::\_1) \*/

fpi f = std::bind(&A::A\_fun\_int, this, std::placeholders::\_1);

f(5);

fpi f = std::bind(&A::A\_fun\_int, this, std::placeholders::\_1, std::placeholders::\_2);

f(5, 50);

### this指针

this指针:即对象的地址。每个成员函数的第一个隐藏参数就是this指针。

若成员函数使用默认的调用约定：则将this指针存入ecx寄存器再传递给成员函数。

### 虚函数指针

虚函数表指针位置:位于包含(virturl关键字)的类的第一个数据成员,通过获取成员偏移量可得知。\*this即”继承第一个类”的虚函数表指针，也就是一个数组的起始地址。\*(this+4)”继承第二个类”的虚函数表指针。

虚函数表:即存放虚函数的”函数指针”的数组，虚函数指针则是数组起始地址, 数组值的顺序根据虚函数声明的顺序而定。 多重继承中， 虚函数表为二维数组,虚函数表指针 = (this)[继承第n-1个类][函数位置],    = \*(\* ((int\*\*)this+0) + 0)

### 数组

char szAccount[10] = {0};//小数组初始化

char szAccount[100000000]; //大数组应该用memset初始化

编译器不优化情况下，对于{0}的初始化，通常是一个字节逐一赋值为0，而memset在64位平台下很可能是一次8个字节。故当数组大于10个字节时，两者的性能差距在2~8倍，根据数组大小而定。

1. char sz[2][40];

sz[1]="string1"; //错误，不能从const char\* 转换char[40];

2.  char \*sz[2];sz[1]=“string2”  //正确

char \*sz[2]={“string1”,“string2”};//正确

### 变量的内存分配及划分

成员变量: 先声明的成员变量在低地址，后声明的成员变量在高地址。

局部变量: 先声明的局部变量在高地址，后声明的局部变量在低地址。(PUSH 堆栈)

堆变量  : 先申请的堆变量在低地址，后声明的堆变量在高地址。

初始化的全局变量(或全局常量): 先申请的堆变量在低地址，后声明的堆变量在高地址。

未初始化的全局变量：随机分配地址。

全局/静态区: 全局、静态、类静态变量。

常量区: 比如局部和全局(char\* str="ABCD"、const char\* str="ABCD")、以及全局的const int i = 10;

代码区:

栈区: 存放局部变量，包括局部和类中的const int i = 10;

堆区: 存放堆变量，包括堆和堆类中的const int i = 10;

## 结构体

### C结构体与C++结构体区别

C结构体不允许有函数，访问权限只有public,不能继承。

### C++结构体与类的区别

第一结构体成员函数与变量默认访问属性为公有， 类成员函数默认访问属性为私有。

第二C++结构体的继承默认是public，而c++类的继承默认是private。

其他C++结构体与类一样。

### 结构体的对齐

#pragma pack(n):按n个字节对齐。 默认对齐是按成员变量所占空间最大字节数对齐，比如double(8)。

1. 如果 n 大于结构体某成员变量所占字节最大值, 则使用该变量所占字节数对齐。

2. 如果 n 小于结构体成员变量所占字节最大值， 则使用n字节数对齐。

   例如:

struct stTest1   { short a; char d;   short b[20];    char c;}; //pack(4)与pack(2)结果都46

struct stTest2    { char \*b; char d;   short c;   struct  stTest1 stT1;  char  e;};//pack(4) 与pack(2)结果都56

   struct stTest3    { struct  stTest1 stT1;   struct stTest2 stT2;  char f;};//pack(4)结果为108， pack(2)结果为102。等价于将stT1与stT2中各个成员变量依次放入。

### 获取成员变量的偏移量

方法一:unsigned long ulOffset = (unsigned long)&((类名或结构名\*)NULL)->成员变量名。

方法二:使用系统自带宏定义offsetof(类名，成员变量名)

### 嵌套类

class A

{

 public:

 class  B : public C

{

public:

 void fun1();

}m\_a;

};   //A类中有B类型的变量m\_a。

void A::B::fun1()

 {

  printf("A"); //fun1函数中的this指针是B类型的。

 }

## 函数

### 函数参数和返回值

函数调用者:负责实参和返回值的临时空间分配内存空间,并不会未形参分配内存。

函数本身:为形参分配内存(在栈里)。

参数值传递:若是对象则会调用拷贝构造。

返回值:将值赋值给临时内存空间(函数调用者分配),  比如int n = fun(),fun()返回值赋值给临时内存空间， 临时内存空间的值在赋值给变量n。若没临时内存空间则不能出现直接调用fun()。 注意，返回值类型是类类型则:如下。

  class c = fun()//fun()等价；只发生1次拷贝构造。

  class d; d = fun() // class c = fun(); d = c 等价；先拷贝构造，后拷贝赋值。

  函数class fun(class c)调用完后会产生2次析构。形参和临时对象都被释放了。

### 函数调用约定

cdecl 按从右至左的顺序压参数入栈,由调用者清除堆栈，c/c++默认使用

stdcall（pascal） 按从右至左的顺序压参数入栈,由被调的函数清除堆栈 WINDOWS API函数

fastcall 是把函数参数列表的前三个参数放入寄存器eax,edx,ecx,其他参数从左到右顺序入栈。

### 匿名函数(C++11)

[capture](参数类型+名称)->返回类型{函数体} //返回值和参数都可以省略，[capture](){函数体}。

[capture]:

[] //未定义变量.试图在Lambda内使用任何外部变量都是错误的.

[x, &y] //x 按值捕获, y 按引用捕获.

[&] //用到的任何外部变量都隐式按引用捕获

[=] //用到的任何外部变量都隐式按值捕获,例如多线程中就算外部变量被释放都可以继续使用，因为是按值来的， 若是按引用的，可能此时外部变量已释放，到时线程中非法访问。

[&, x] //x显式地按值捕获. 其它变量按引用捕获

[=, &z] //z按引用捕获. 其它变量按值捕获

### 可变参数及可重入函数

可变参数: 根据从右到左的调用约定入栈， 因此可知参数的个数多少。

string fun(char\* strFormat, …)

{

  char out[4096] = {0};

  va\_list valist; //， va\_list（char\*类型）

  va\_start(valist, strFormat);   //指向strFormat

  vsprintf\_s(out,sizeof(out), strFormat, valist); //该函数的strFormat可能会使out溢出

va\_end(valist);//释放指针

}

int Fsum(int n, ...)

{

int sum = 0;

va\_list pArg = NULL;

va\_start(pArg, n);

for (int i = 0; i < n; i++)

{

int num = va\_arg(pArg, int); //可变参数全是int类型

sum += num;

}

va\_end(pArg);

return sum;

}

可重入函数:

(1)如何编写可重入的函数？

答：在函数体内不访问那些全局变量，不使用静态局部变量，坚持只使用局部变量，写出的函数就将是可重入的。如果必须访问全局变量，记住利用互斥信号量来保护全局变量。

(2)如何将一个不可重入的函数改写成可重入的函数？

答：把一个不可重入函数变成可重入的唯一方法是用可重入规则来重写它。其实很简单，只要遵守了几条很容易理解的规则，那么写出来的函数就是可重入的。

例如在2个线程中调用一个函数。该函数就是可重入函数。

### 构造函数与析构函数

构造函数:

执行完构造函数分两步， 第一步执行初始化即初始化列表(变量的分配空间顺序与声明有关，与初始化列表无关)，父类的构造函数在初始化表中调用，若没显示调用则调用默认父类构造。此时也完成虚函数指针的初始化工作。//

第二步执行函数体。 因此const成员变量、引用类型成员变量以及类成员对象变量的构造函数，必须在初始化列表中完成。

1. 无返回类型，无返回值。
2. 函数中不要调用虚函数。(因为创建子类对象时，由于先调用父类的构造函数，若父类构造函数中使用了覆盖子类的虚函数， 而子类的虚函数中调用了子类的成员变量，此时子类并未构建完成，因此会在运行期出错。)
3. 类类型变量（如:string）的初始化在初始化表里进行比在函数体进行效率高。

析构函数:

1. 无返回类型，无返回值，无参数。
2. 函数中不要调用虚函数。(因为虚函数不生效，只会调用本类的函数)
3. 不要让异常逃离析构。

try

{fun();}

catch(...)

{记录失败}

4.delete只看指针类型的，如果析构不是虚函数就只调用delete的类型的析构函数,所以为了让子类析构调用，所以只需要定义父类析构为虚析构就行。

### 重载及重载操作符

const参数会产生重载， 而临时对象或者无名对象具有常性会调用const参数拷贝构造或const参数赋值构造。

拷贝构造或赋值拷贝:参数和返回类型必须加入引用。如果没有引用则编译报错， 由于临时对象的原因会造成无限调用拷贝构造或拷贝赋值函数。

class A{

public:

operator int () const{return i;} //当对象当右值时则调用该操作符重载。

A(){}

public:

int i;

int g;};

A a;a.i = 10;a.g = 20;

int i = a; //i =10

①  禁用拷贝构造和拷贝赋值

Class Child

{private:

 //只声明不实现,当成员函数或友元函数调用时会发出连接期错误

  Child(const Child&); //拷贝构造

  Child& operator=(const Child);//拷贝赋值};

② 深度拷贝

在默认的构造函数中new内存给一变量， 在拷贝构造或拷贝赋值时候，不会在调用默认构造函数，所以在析构的时候调用delete会对同一成员指针变量2次释放。所以深度拷贝，就是在拷贝构造或拷贝赋值函数里面new内存,保证delete2次释放是不同的内存空间。

深拷贝与浅拷贝的区别在于拷贝时是否重新分配空间，深拷贝重新分配空间。

重载操作符：

// 前++

Complex& operator++ (void) //返回引用支持连续++

{

 m\_r++;

 m\_i++;

 return \*this;

}

// 后++

const Complex operator++ (int) //const防止连续++

{

 Complex old = \*this;

 m\_r++;

 m\_i++;

 return old;

}

### 重载、覆盖(重写)、隐藏的区别

重载:  1、相同的范围(同一个类中)。2、函数名称相同。3、函数参数不同。4、const有无也会构成重载。5、virtual可有可无。

覆盖:  1、不同的范围(基类与子类)。 2、函数名称相同。3、函数参数相同。4、基类必须有virtual关键字。

隐藏①:  1、不同的范围(基类与子类)。 2、函数名称相同。3、函数参数不同。4、virtual可有可无。(勿与重载混淆)

隐藏②:  1、不同的范围(基类与子类)。 2、函数名称相同。3、函数参数相同。4、基类没有virtual。(勿与覆盖混淆)

## 类的特性

### 封装性

### 继承性

只继承类型，不是继承对象，若创建子类对象：会根据继承的类型来开辟内存空间并与访问属性无关。因此new 子类对象就是子类this指针， new父类对象就是父类类型this指针。

### 多态性

例如： Father \*fat = new Child;

 fat->fun(); //虚函数表指针指针需要在运行期才分配内存(动态联编),然后查找虚函数表进行函数调用,若fun函数不在虚函数表中或无虚函数表，则直接根据fat类型调用相关fun()函数。

① 多态破坏了访问属性

由于虚函数表中存放的都是虚函数，所以外界可以通过取出虚函数表中的虚函数指针然后再调用该虚函数。函数指针 =  \*(\*(int\*\*)this + 函数位置”即数组位置或虚汗表位置”) 或 =  (\*(int\*\*)this)[位置]

② 多态中，无论继承了多少个子类，对象中始终只有一个虚函数表，先放入父类虚函数，在放入子类虚函数。

因为父类的构造函数只允许调用父类的函数或虚函数。 由于先构造父类在构造子类，子类并未构造完成，则不会调用子类虚函数或成员函数。

## 宏

### 基本用法

## 是连接符号 连接两个宏

#是把名字代替成双引号括起来

#@是把名字代替成单引号括起来

#define s5(a) supper\_ ## a   s5(printf)("hello owrld"); ==>> supper\_ printf("hello owrld");

#define s(p) #p ==>> #define s(p)  “p”

#define s(p) #p ==>> #define s(p)  ‘p’

### 常见宏

1. far

int far 或者 void far 表示要编译器给声明的这些量分配在 远处的存储器(RAM)中，即分配到 64K 以外的 存储地址中。64K 作为一个“段”(Segment),是 PC 机 80x86 时期的老概念。超过64K 就是 “远”， FAR 就是 “远”de 英文。编译器对near（近）和far（远）的处理略有不同。新的编译器默认应当是远。

2.\_\_FILE\_\_  当前源文件名称, \_\_FUNCTION\_\_ 当前函数名, \_\_LINE\_\_当前行号(Linux中\_\_PRETTY\_FUNCTION\_\_当前函数名)。

## 模板

### 类模板和函数模板区别

类模板: 类中成员或类还是还可以声明其他类型的模板。成员函数声明定义都必须放入头文件。

template<class T>

class A

{

public:

void fun(T a){};

private:

T \*ptr;

int a;

};

函数模板:

template<class T>

void fun(T a);

使用类模板时需要

A<具体类型> concrete;

### 模板的实例化和具体化区别

template<class T>

void Swap(T &a, T &b)

{...}

模板具体化:优先级高于未具体化模板

template<> void Swap<具体类型>(具体类型 &a, 具体类型 &b)

{

...

}

编译器会将具体化函数解释为:void Swap(具体类型 &a, 具体类型 &b);因此也可用它代替。

模板实例化:该模板可有可无。编译器会隐式实例化。

template void Swap<具体类型>(具体类型 &a, 具体类型 &b)没有函数体。

### 模板用作参数(模板中模板)

a:template<class T>

b:template<template<class T> class T1>

使用：a必须是个模板类。

template<class T>

class human

{

private:

T age;

}

template<template<class T> class T1>

class people

{

T1<int> t1; //human.age是int型。

}

### 模板优先级

在参数类型和参数个数相同情况下：普通函数 > 具体化模板 > 模板；否则按最接近模板类型的模板来匹配。

模板的默认值:template<class T, class T1 = int>

### 数组模板

template<class T, int n>

T a[n];

## 异常

### 为什么要使用异常

原因一: 是错误处理和工作代码分开， 不用每个地方用if判断，增强代码可读性。

原因二: 构造与析构没有返回值。

### 异常抛出列表

throw() //表示不抛出任何异常

throw(异常1,异常2) //只抛出这2种异常。

throw 类名()

//throw用在已经出错的地方, 抛出的错误进入对应的catch

### try-catch-throw 基本应用

catch(…)//表示处理任何异常。

在该结构中声明的变量，不能在该结构之外使用。

例子:

class Cwrong(){}

void fun()

{if(bool) {出错:throw Cwrong}}

try{ fun()} catch(Cwrong){错误处理1} catch(CClass){错误处理2}

### 异常类的继承

class Child : public Father

try{throw Child()} //抛出子类异常。

catch(Father){} //会触发Father, 不会触发Child了， 要想触发Child得放在Father之前。

catch(Child){}

抛出父类异常不会进入子类异常处理。

自己定义的类若要继承标准异常类需要重写what函数和析构函数。

### 异常类成员函数调用

try{throw A()}

catch(A a){a.fun()}

抛出子类对象， 进入父类处理， 父类类型的子类对象调用的虚函数是子类的函数。

try{throw Child()}

catch(Father f){} 等价于 Father f = Child();

若类不支持拷贝赋值 则使用catch(Father &f){} 引用。

### 资源分配即初始化(RAII)

RAII指: 资源的分配和销毁用类封转起来。

原因: 因为在动态的内存在发生异常时，不会自动释放。 所以使用类来管理资源的分配与释放， 因为发生异常时，会自动调用构造与析构函数。

### 标准异常类

1.exception <exception>

所有标准异常类的父类

2.bad\_alloc <new>

new或new[] 分配内存失败时抛出的异常类

3.bad\_cast <typeinfo>

使用dynamic\_cast转换引用失败的时候

4.bad\_typeid <typeinfo>

使用typeid操作符，操作一个NULL指针，而该指针是带有虚函数的类，这时抛出bad\_typeid异常

5.bad\_exception <exception>

这是个特殊的异常，如果函数的异常抛出列表里声明了bad\_exception异常，当函数内部抛出了异常抛出列表中没有的异常，这是调用的unexpected函数中若抛出异常，不论什么类型，都会被替换为bad\_exception类型

6.ios\_base::failure <ios>

io操作过程出现错误

7.logic\_error <stdexcept>

逻辑错误，可以在运行前检测的错误

子类:

(1)length\_error:

试图生成一个超出该类型最大长度的对象时，例如vector的resize操作

(2)domain\_error:

参数的值域错误，主要用在数学函数中。例如使用一个负值调用只能操作非负数的函数

(3)out\_of\_range:

超出有效范围

(4)invalid\_argument:

参数不合适。在标准库中，当利用string对象构造bitset时，而string中的字符不是’0’或’1’的时候，抛出该异常

8.runtime\_error <stdexcept>

运行时错误，仅在运行时才可以检测的错误

子类:

(1)range\_error:

计算结果超出了有意义的值域范围

(2)overflow\_error:

计算上溢

(3)underflow\_error:

计算下溢

# C/C++常用函数

## 进程相关

### void exit(int)

功能:退出程序,Linux下:main函数返回时调用exit函数,类似exit(main(argc,argv))。

参数:0:表示正常退出程序,1或-1表示非正常退出程序。

返回值:无

### int atexit((void \_cdecl \*fun)())

功能:程序结束前调用的某个函数。

参数:调用的函数指针。

返回值:无

## 线程管理

### \_beginthreadex

功能:创建线程, 在使用了C++运行时间库，如1 使用malloc()和free()，或是new和delete

2 使用stdio.h或io.h里面声明的任何函数

3 使用浮点变量或浮点运算函数

4 调用任何一个使用了静态缓冲区的runtime函数，比如:asctime(),strtok()或rand()。则要调用该函数创建线程

参数一(IN void\*):安全属性，NULL为默认安全属性

参数二(IN unsigned):指定线程堆栈的大小。如果为0，则线程堆栈大小和创建它的线程的相同。一般用0

参数三(IN unsigned ( \_\_stdcall \*start\_address )( void \* )):指定线程函数的地址，也就是线程调用执行的函数地址(用函数名称即可，函数名称就表示地址)

参数四(IN void\*):传递给线程的参数的指针，可以通过传入对象的指针，在线程函数中再转化为对应类的指针

参数五(IN unsigned):线程初始状态，0:立即运行；CREATE\_SUSPEND：suspended（悬挂）

参数六(OUT unsigned\*):用于记录线程ID

返回值(unsigned long): 线程句柄， 失败返回0。

### 

## 内存相关

### void\* memcpy(void \*dest, const void \*src, size\_t n)

功能:拷贝字符串按字节拷，不受’\0’影响

返回值: 与第一个参数是一样的。

第三个参数:要拷贝的字节数, 不会覆盖字节数以外的字符。可以大于参数二的大小。

注意 ：建议vs使用memcpy\_s函数

errno\_t memcpy\_s(void \*dest,size\_t numberOfElements,const void \*src,size\_t count );

第二个参数的大小一定要”大于等于”最后一个参数大小, 与第一个参数的大小无影响, 并且最后一个参数要小于等于第一个参数的大小。

例如: char str[5] = {0};

memcpy\_s(str, 10, "12344444",8); //错误

memcpy\_s(str, 5, "12344444",6); //错误

memcpy\_s(str,10, "12344444",5); //正确

memcpy\_s(str, 5, "12344444",5); //正确

memcpy\_s(str, 2, "12344444",2); //正确

### int memset(char\*, int, int)

功能:对内存清理。针对编译器优化对memset函数可能不执行，Windows下建议使用SecureZeroMemory函数或使用#Pragma指令取消优化。

## 时间相关

time\_t 等价于 long long类型

struct tm {

int tm\_sec; /\* seconds after the minute - [0,59] \*/

int tm\_min; /\* minutes after the hour - [0,59] \*/

int tm\_hour; /\* hours since midnight - [0,23] \*/

int tm\_mday; /\* day of the month - [1,31] \*/

int tm\_mon; /\* months since January - [0,11] \*/ 获得月-1

int tm\_year; /\* years since 1900 \*/ 获得年-1900

int tm\_wday; /\* days since Sunday - [0,6] \*/

int tm\_yday; /\* days since January 1 - [0,365] \*/

int tm\_isdst; /\* daylight savings time flag \*/

};

//比如2013年10月， 在tm中 tm\_mon = 9 , tm\_year = 113

相关函数。

time\_t time(NULL) //获取当前本地时间的秒:从1970年1月1号0时0分0秒到现在本地时间的秒数(包含时区)

struct tm\* localtime(time\_t\*); //将秒数转换tm的本地时间。参数应该包含时区。

struct tm \*gmtime(time\_t \*clock);//vc下建议使用gmtime\_s(struct tm \*, time\_t \*);

将秒数转换tm的UTC时间。

time\_t mktime(struct tm \*timeptr);将时间转换从1970年1月1号0时0分0秒到参数时间的秒数。

## 文件相关

### int rename(char \* oldname, char \*newname)

功能:给文件重命名,完成文件移动。

参数:

返回值:0成功，-1失败

### int remove(const char\* filename)

功能:删除文件或目录。

参数:

返回值:0成功，-1失败

### int \_mkdir

功能:创建目录。只能一级一级目录的创建,<direct.h>

返回值:存在目录返回-1。 不存在则创建返回0

### 介绍

fwrite函数将数据写入缓冲区， 若缓冲区满或者调用fclose函数则将缓冲区数据写入文件, 若出现掉电等问题导致缓冲区数据消失，应每次接收数据时，使用 fflush函数使缓冲区写入文件。

使用int fseek(FILE \*stream, long offset, int origin)函数移动文件指针，最后一个参数取值(SEEK\_CUR, SEEK\_END, SEEK\_SET)

判断文件结尾: fseek(fp, 0, seek\_end),int i = ftell(fp), fseek(fp, 0, seek\_set),比较文件大小while(i != ftell(fp))

写入文件的函数有:fsprintf(fp, “%s”, data), fwrite(buffer , size, count，fp),

读取文件的函数：fgets(buffer, count, fp)每次获取一行

写入或者读取用文本模式的话， 写入文件时一旦遇到"换行"(ASCII 10)或者"回车"(13),会自动转换"回车-换行"组合(13\r,10\n)写入文件。 读取文件时若遇到"回车-换行"组合(13,10)会自动转换"换行"(10)。

结论:写入或读取文件，采用统一的模式。 在位图等可能存在13、10组合，所以采用二进制读取则不会出错。

## 格式化及转换

### printf

多线程环境下，禁止std::cout与printf混用

### int snprintf(char\*, unsigned int, const char\* format, ...)

功能:格式化成字符串类型。

参数一:每次调用前，会清零。

### unsigned long strtoul(const char\*, char\*\*, int)

功能:将字符串转换长整型,一般用于大数字，如time\_t类型

参数一:要转换的字符串。

参数二:

参数三: 若为10， 则表示转换10进制的长整型。

### int atoi(const char\*)

功能:将字符串转换有符号整型。从参数的第一个非空字符(包含整数，正负号)算起， 直到遇见非整型数据(包含空格及符号等)为止则不再转换。

### 格式化

hh和整数转换说明符一起使用，表示一个short int 或者unsigned short类型的数值 ，示例：%hhu，%hhx，%6.4hhd

(2)%.\*f \*表示的是精度， %.\*f比%.df的好处是：%.\*f要可输入2个参数(比如第一个整数，第二个浮点数)，%.df只能是1个参数（比如:整数,只能代替精度位数了。）。

(3)%.5s .5是精度5, 只需要格式化5个字符， 可防止缓冲区溢出。

(4)%u 匹配无符号整型

(5)%m.nf m表示数字所占位数，当数字位数超过m,则m则无效。

n表示小数精确位数。 四舍五入，不足补零。

例如%10.8f 12345.1 格式化后:12345.10000000

(6)%llu 64位无符号整型

## IO相关

### int sscanf(const char\*, const char\* format,..)

调用sscanf\_s("1:2:3","%d:%d:%d", &a,&b,&c);后 a= 1, b= 2, c= 3

## 随机数相关

### void srand(unsigned int)

功能:设置种子。

参数:种子，//若种子一样，随机序列也就一样。 种子一般为time(NULL)，

### int rand()

功能:根据设置好的种子，返回随机数。不过是伪随机数。1.Unix/Linux平台建议采用读取”/dev/random”或”/dev/urandom”文件的方式来获取随机数；

2.Windows平台推荐使用CryptGenRandom()来生成随机数；

## 字符串相关

### char\* strstr(const char\*, const char\*)

功能:在一个字符串中查找另一个字符串

参数一:在该字符串中查找。

参数二:要查找的字符串。

返回值:找到则返回参数二在参数一的位置的指针, 找不到返回NULL。

### strtok

strtok是一个线程不安全的函数，因为它使用了静态分配的空间来存储被分割的字符串位置

## string类相关

### 初始化

string str(10,0); 初始化10个字节都为0。

### int length()

功能:获取本字符串长度，长度变化不会受cpy,cat之类函数的影响， 会受+, += ,=重载操作符的影响。计算遇到'\0'且后面与前面不同，则终止。

### string &erase(int, int)

功能:删除指定字符。

参数一:要删除字符所在字符串位置(0开始)

参数二:要删除的字符的个数。

返回值:删除后的结果。

### int find(const char\*)

功能:查找指定字符串。

参数:要查找的字符串。

返回值:字符串在本字符的位置(0开始),-1或string::npos为找到。

## 正则表达式相关(C++11)

### 匹配规则

(表达式) : 分成子表达式

[数字] : 表示取其中任意数字， 如[0-9]可取0到9之间任何数字、[57]只可取5或者7数字。

{数字} : 表示使用多少个字符， 比如[0-9]{8}、\\d{8}使用0-9之间的8个数字。

? : 表示前面的子表达式出现0或1次。

| : 表示或，\\.|\_表示.或\_有个出现匹配。(13[0-9]|15[09]|17[6-8]|18[0-9]|14[57])。

+ : 表示重复一次或多次。

\* : 表示任意字符。

\ : 转义符号， 注意在字符串中要使用\\, 比如\\d十进制数。

转义字符d表示十进制。

转义字符w表示一个单词。

转义字符.表示匹配.

使用include <regex>

### std::regex\_search

功能： 在指定匹配规则搜索指定字符串。

(in)参数一(std:string)：是否符合匹配规则的字符串。

(out)参数二(std::smatch)：匹配的个数。

(in)参数三(const std::tr1::regex)：匹配规则。const std::tr1::regex rule("(13[0-9]|15[09]|17[6-8]|18[0-9]|14[57])[0-9]{8}");

返回值(bool)：true成功匹配。

### 

# C/C++常用技术原理

## 池化技术

### 内存池

编写一个内存管理类(内存池)， 一次性向系统内存申请大块内存，在根据用户需要从这片大内存中申请及回收。注意:内存池中申请时注意加锁防止多线程中申请内存。

好处:

1.非常少(几没有) 堆碎片

2.比通常的内存申请/释放(比如通过malloc, new等)的方式快

3.检查任何一个指针是否在内存池里

4.写一个”堆转储(Heap-Dump)”到你的硬盘(对事后的调试非常有用)

5.某种”内存泄漏检测(memory-leak detection)”：当你没有释放所有以前分配的内存时，内存池(Memory Pool)会抛出一个断言(assertion).

### 线程池

## 智能指针

原理：一个用来智能管理指针的类，通过类变量的作用域来调用其析构函数来释放指针。

资源分配即初始化RAII(Resource Acquisition Is Initialization)。

### auto\_ptr

说明：C++98中引入，有缺陷，目前使用率低，不建议使用。

实现方案一：拷贝构造、赋值构造，设置原来对象的指针为NULL，把指针权限交给新对象(旧对象不可再使用);析构函数通过指针是否为空来释放。缺点：旧对象移交指针权限后不能再使用。

实现方案二：拷贝构造、赋值构造通过设置bool变量来使其旧对象能继续使用;析构函数通过bool变量值来决定是否释放指针。缺点：新对象释放后，旧对象不可再使用。

代码：在auto\_ptr.cpp中。

### unique\_ptr

说明：C++11中引入，用来代替auto\_ptr, 类似于Boost库中scoped\_ptr。

实现：既然auto\_ptr拷贝构造和赋值构造会引发问题，就将其两个函数私有化, 但是const的拷贝构造和赋值构造不私有化，因为允许临时对象的存在。

unique\_ptr<int> uPtr = unique\_ptr<int>(new int(1)); //编译通过，因为声明赋值一条语句直接调用的是构造函数。

unique\_ptr<int> uPtr;

uPtr = unique\_ptr<int>(new int(1)); //编译通过，因为调用的是重载的const的赋值构造函数。

### shared\_ptr

说明：C++11中引入，为了使其旧对象还能使用，则使用shared\_ptr。

实现：

1.一个模板指针T\* ptr，指向实际的对象。

2.一个引用次数(必须new出来的，不然会多个shared\_ptr里面会有不同的引用次数而导致多次delete)。

3.重载operator\*和operator->，使得能像指针一样使用shared\_ptr。

4.重载copy constructor，使其引用次数加一。

1. 重载operator=，如果原来的shared\_ptr已经有对象，则让其引用次数减一并判断引用是否为零(是否调用delete)。然后将新的对象引用次数加一。

6.重载析构函数，使引用次数减一并判断引用是否为零(是否调用delete)。

### weak\_ptr

说明：C++11中引入，为解决shared\_ptr的循环引用导致内存泄露问题，weak\_ptr使其不进行计数。

class AA

{

public:

shared\_ptr<AA> m\_pa; //需要AA对象释放后才释放m\_pa变量。需要改为weak\_ptr<AA> m\_pa。

}

int main()

{

shared\_ptr<AA> pa(new AA);

pa->m\_pa = pa; //m\_pa指针的释放需要AA对象释放后才能释放，此时pa的引用计数为

cout << pa.use\_count << endl; //此时引用计数为2,pa析构时的计数为1，而pa->m\_pa又需要AA析构时才能析构。

}

# C++11特性

### auto关键字

自动推断变量类型。

注意:

1. auto类型变量必须初始化,以及堆变量初始化。

auto\* p = new auto(100);

1. 函数参数和模板参数不能使用auto类型。
2. 不能用于类型转换或sizeof等操作。

### 数组(tuple)

std::tuple<std:string, int, void\*> tp;//表示这类型的数组可以放入字符串、整型、指针。

//等价于一个结构体。

std:make\_tuple函数，功能创建一个tuple。

//例如: std::tuple<std:string, int> tp = std::make\_tuple(“a”, 1);

std:tie函数,功能创建一个tuple的引用

//例如: std::tuple<std:string, int> tp = std::tie(“a”, 1);

### 智能指针

成员函数:

1.long use\_count() const; //返回某个指针的引用计数

2.T\* get() const; //获取智能指针中的指针。

用法:

1. 同一个指针不能传入多个智能指针中,导致会对同一指针释放多次。

int \*pInt = new int(5);

shared\_ptr<int> temp1(pInt); //引用计数为1。

shared\_ptr<int> temp2(pInt); //引用计数为1。

1. 释放非new的指针。

class A

{public:

void operator()(FILE\* pFile)

{if (pFile != NULL)

fclose(pFile);}

}

shared\_ptr<FILE> fp(fopen("A.txt", "r"), A());//将第一个当做参数传递第二个。

1. 封装this指针

class A: public enable\_shared\_from\_this<A>

{

public:

shared\_ptr<A> Get()

{

return shared\_from\_this(); //若使用shared\_ptr<A> (this);则会导致多次释放。

}

}

shared\_ptr<A> pA(new A());

shared\_ptr<A> pB = pA->Get();

1. 在多线程中使用shared\_ptr时， 如果存在拷贝或赋值操作，可能由于同时访问引用计数而导致计数无效。解决方法是向每个线程中传递公共的week\_ptr, 线程中需要使用shared\_ptr时，将week\_ptr转换成shared\_ptr即可。

### 线程管理

1. 创建线程:<thread>

std::thread()

1. 线程同步:

std::condition\_variable m\_cv; //m\_cv条件变量对象。

std::condition\_variable::wait(std::unique\_lock<mutex>&)

std::condition\_variable::notify\_one() //唤醒一个wait中的线程。若多个线程在wait中，则随机唤醒一个。

std:mutex

std:unique\_lock

# STL

## vector

### 查找元素

1. 查找

std::vector<INT> m\_list;

m\_list.push\_back(10);

m\_list.push\_back(20);

m\_list.push\_back(30);

std::vector<INT>::iterator iter = std::find(m\_list.begin(), m\_list.end(), 10);

if (iter != m\_list.end())

{TRACE(\_T("找到了"));}

else

{TRACE(\_T("未到了"));}

1. 对象的查找

bool CPerson::operator==(const CPerson &rhs) const

{return (age == rhs.age);}

CPerson cp\_to\_find; // 要查找的对象

cp\_to\_find.age = 50;

vector<CPerson>::iterator it = find(list.begin(), list.end(), cp\_to\_find); // 查找

1. 函数指针条件查找

vector<ModuleObserverCtx\*> m\_vecObservers;

auto iterObserver = std::find\_if(m\_vecObservers.begin(), m\_vecObservers.end(),

[=](ModuleObserverCtx\* pMKOCtx)

{return(pObserObject == pMKOCtx->m\_pObserverObject);});//pObserObject 要查找的值

### 删除元素

## list

### 查找元素

和vector一样

### 获取元素

front()

返回值(元素的类型)： 容器第一个元素。

back()

返回值(元素的类型): 容器最后一个元素。

### 删除元素

void pop\_front() 删除容器内第一个元素

删除所有指定元素方法一:

std::list<std::string>::iterator Itlist;

for (Itlist = m\_listTop.begin(); Itlist != m\_listTop.end();)

{

if ((\*Itlist) == strSessionID)

{

Itlist = m\_listTop.erase(Itlist);

}

else

{

++Itlist;

}

}

删除所有指定元素方法(容器中有多个相同删除的元素时，效率低于一， 若只有少量时，则效率高于一)二:

std::remove\_if 并不会真正删除元素，会把符合条件的元素移动到末尾，在使用

erase(iterator, vector.end()); //移除所有要删除的元素

auto iter = std::remove\_if(m\_vecObservers.begin(), m\_vecObservers.end(),

[=](ModuleObserverCtx\* pCtxItem)

{

bool b = (pObserObject == pCtxItem->m\_pObserverObject);

if (b)

{ delete pCtxItem;

pCtxItem = 0;}

return b;

});

if (iter != m\_vecObservers.end())

{m\_vecObservers.erase(iter,m\_vecObservers.end());}

## map

map<string, int> mapName; //string:键值， int:键值对应的值。

键值会自动排序。string会按字母排序。

### 插入元素

(1)用insert方法插入pair对象

enumMap.insert(pair<int, Cstring>(1, “One”)); //如果1存在则不会插入。

(2) 用insert方法插入value\_type对象：

enumMap.insert(map<int, Cstring>::value\_type (1, “One”));

(3) 用数组方式插入值：

enumMap[1] = "One"; //若1不存在时， 则会创建enumMap[1] = "", 在进行= "One"赋值。 若是对象， 则会多进行一次构造以及拷贝赋值操作。

### 删除元素

删除指定的元素:

map<string, int>::iterator iter = ma.begin();

for (; iter != ma.end();)

{

if (iter->second == 1)

{ma.erase(iter++); //不会返回下一个迭代器,所以要后++,这条语句执行完毕后iter就是非法指针}

else

{++iter;}

}

删除第一个指定的Key值:

pos = mapTest.find(keyValue);

if (pos != mapTest.end())

{

mapTest.erase(pos);

}

删除所有指定的Key值:

ma.erase("a");

## Set

只有唯一键值，自动排序。

## 迭代器

迭代器声明: 容器类型::iterator 变量名 例如: map<int, string>::iterator map\_it;

在使用迭代器的时候使用前置++比后置++效率高,因为重载++函数，后置++要返回一个临时的对象。