# C++ Note v1



承接《C++ Note v1》

本文档主要包含：C++ 11标准、C++静态库/动态库、编译/链接等内容

# std::initializer\_list<>

//初始化列表用于构造类的对象

**enum** Gender {boy,girl};

**class** People

{

**public**:

    People(std::initializer\_list<std::pair<string, Gender>> initlist)

        :data(initlist) {}

**private**:

    std::vector<std::pair<string, Gender>> data;

};

//初始化列表用于函数形参

**int** add(std::initializer\_list<**int**> lst)

{

**int** sum = 0;

**for** (auto i: lst)   sum += i;      //或者

  //for (auto i = lst.begin(); i != lst.end(); i++)   sum += \*i;

**return** sum;

}

**int** main()

{

    People ship{ {"zha",boy},{"zhou",girl},{"ning",girl} };

    std::initializer\_list<std::pair<string, Gender>> initlist{ { "zha",boy },{ "zhou",girl } };

People pen(initlist);

    cout << add({ 1,3,5,7 }) << endl;

    cout << add({}) << endl;

}

返回一个初始化列表

vector<int> func{ return {1, 3}; }

int a[] = {7,6,1,3,8};

cout << \*std::max\_element(a, a + 5) << endl; //最大值，8

cout << \*std::min\_element(a, a + 5) << endl; //最小值，1

std::vector<int> vec{ 1,3,5 }; //或std::vector<int> vec2={ 1,3,5 }

vec = { 2,4,6 };

std::initializer\_list<int> init{4,7,10};

vec = init;

std::initializer\_list<string> str{ "zh","lx" };

for (auto i : str) cout << i << " "; 在VS2015下能打印出zh lx

这个写法是合法的。底层数组的生存期会延续到上面的 str 变量的作用域结尾

std::initializer\_list<string>\* p=new std::initializer\_list<string>{ "zh","lx" };

for (auto i : \*p) cout << i << " "; //什么都打印不出

initializer\_list 所引用的底层数组的生存期类似临时对象。用 new 表达式创建 initializer\_list 时，底层数组的生存期不会随着 initializer\_list 延长，而是只到完整表达式结尾。所以那 3 个 std::string 在输出之前就已经被析构了。

Initializer lists may be implemented as a pair of pointers or pointer and length. Copying a std::initializer\_list does not copy the underlying objects.

The underlying array is a temporary array of type const T[N], in which each element is copy-initialized (except that narrowing conversions are invalid) from the corresponding element of the original initializer list. The lifetime of the underlying array is the same as any other temporary object, except that initializing an initializer\_list object from the array extends the lifetime of the array exactly like binding a reference to a temporary (with the same exceptions, such as for initializing a non-static class member). The underlying array may be allocated in read-only memory.

{

std::initializer\_list<Widget> lst{ 1, 2 };

getchar();//2个构造

}

{

Widget a(1);

Widget b(2);

std::initializer\_list<Widget> lst{ a, b };

getchar(); //2个构造、2个拷贝构造

}

{

Widget a(1);

Widget b(2);

std::initializer\_list<Widget> \*lst = new std::initializer\_list<Widget>{ a, b };

getchar();//2个构造、2个拷贝构造、2个析构

}

{

std::initializer\_list<Widget> lst2;

{

Widget a(1);

Widget b(2);

std::initializer\_list<Widget> lst1{ a, b };

lst2 = lst1;

}

getchar();//2个构造、2个拷贝构造、4个析构

cout << (\*lst2.begin()).m\_val << endl; //-858993460

}

# 算法：

C++17起：

在变量的初始化中，初始化器表达式为与变量类型为同一类类型的纯右值（忽略 cv 限定）时：

T x = T(T(T())); // 仅调用一次 T 的默认构造函数以初始化 x

在 return 语句中，运算数为与函数返回类型为同一类类型的纯右值（忽略 cv 限定）时：

T f() {

return T();

}

f(); // 仅调用一次 T 的默认构造函数

std::string的容量取值为15、15+16、15+2\*16、15+3\*16……即使用reserve(n)设置，n小于当前容量，则设置无效；n大于当前容量，则容量修改为以上数值中>=n且最接近n的数

std::vector:: reserve(n)，n小于当前容量，则设置无效；n大于当前容量，则容量修改为n

string::resize(n)，n大于当前size，则增加size至n，增加的是空字符（ascii码为0）；n小于size，则截断string至size为n

无论string还是vector，只要内存重分配，则所有迭代器失效

string s3 = "abcdefg";

string::iterator iter = s3.begin()+3;//d

string::iterator j = s3.begin() + 4;//e

s3.erase(iter);

cout << s3 << endl; //abcefg

cout << \*iter << endl; //e

cout << \*j << endl; //f

vector<char> vec{ 'a','b','c','d','e' };

vector<char>::iterator vi = vec.begin() + 3;

vec.erase(vi);

for (auto i : vec)cout << i << " "; cout<<endl; //a b c e

//cout << \*vi << endl; //指针已失效

string::erase(string::iterater) 测试如上

vector:: erase(vector::iterater)，指向删除点和删除点之后的元素的迭代器全部失效

算法sort：

#include<algorithm> //sort

#include<functional> //std::greater<>

string str("efgabc");

sort(str.begin(), str.end(),std::greater<char>()); //由大到小排，默认由小到大

std::vector::assign()赋值函数，清除以前的，重新赋值

assign(input\_iterator start,input\_iterator end)

std::vector::assign(size\_t num, const TYPE& val)

std::string和std::vector的[]越界崩溃，at()越界抛出异常，二者都返回引用

std::string和std::vector和std::list的back()、front()返回引用

算法swap：

template<typename T>

void swap(T& a,T&b) {

T temp(std::move(a));

a = std::move(b);

b = std::move(temp);

}

vector<int> v1{ 30,40,50,60,70 };

vector<int> v{3,4,5,6,7};

std::swap(v1, v);

std::string、std::vector和std::list都重载了<、>、>=、<=、==、!=，逐元素比较，返回bool

算法std::random\_shuffle()

template <class RandomAccessIterator>

void random\_shuffle (RandomAccessIterator first, RandomAccessIterator last);

vector<int> v1{ 3,4,5,6,7 };

std::random\_shuffle(v1.begin(), v1.end()); //7 4 6 5 3

std::random\_shuffle(v1.begin(), v1.end()); //7 6 5 3 4

std::list为双向链表，#include<list>

std::forward\_list为单向链表，#include<forward\_list>

forward\_list只能在链表头部添加删除，只能获取.begin()迭代器，只能.front()获取首元素

std::forward\_list<std::string> lst = {"3","5"};

lst.push\_front("4"); //单向列表forward\_list只有push\_front没有push\_back，只有pop\_front没有pop\_back

for (const auto& i : lst)

{

cout << i << " "; //4 3 5

}

std::list增加任何元素都不会使迭代器失效。删除元素时，除了指向当前被删除元素的迭代器外，其它迭代器都不会失效

std::list<int> lst{ 1,3,5,7 };

std::list<int>::iterator iter = lst.begin();

std::list<int>::iterator iter2 = ++iter;

iter--;

lst.erase(iter);

//cout << \*iter << endl; //error, 迭代器失效

for (const auto& i : lst) cout << i << " "; cout << endl;//3 5 7

cout << \*iter2 << endl; //3

void list::remove(const \_Ty& \_Val)删除链表中所有==val的元素，如果存储的是class，则需要类对operator ==重载。

list::remove()删除后，list::size()改变

算法template<class \_FwdIt, class \_Ty>

inline \_FwdIt remove(\_FwdIt \_First, \_FwdIt \_Last, const \_Ty& \_Val)并不改变容器(vector、list……)的size()，而是调用赋值运算符重载函数“覆盖”，返回的是未移去的最后一个元素的下一个位置.

std::list<**int**> lst{ 1,3,5,3};

lst.remove(3);

**for** (**const** auto& i : lst) cout << i << " "; cout << endl; //1 5

lst = { 1,3,5,3};

std::remove(lst.begin(),lst.end(),3);

**for** (**const** auto& i : lst) cout << i << " "; cout << endl;//1 5 5 3

lst = { 1,3,5,5,3,7,3,9 };

std::remove(lst.begin(), lst.end(), 3);

**for** (**const** auto& i : lst) cout << i << " "; cout << endl;//1 5 5 7 9 7 3 9

lst = { 1,3,5,5,3,7,3,9 };

lst.erase(std::remove(lst.begin(), lst.end(), 3),lst.end());  //std::remove()与erase()配合使用

**for** (**const** auto& i : lst) cout << i << " "; cout << endl;//1 5 5 7 9

算法：template<class \_InIt,class \_Ty>

inline \_InIt std::find(\_InIt \_First, \_InIt \_Last, const \_Ty& \_Val)

如果找到，返回第一个匹配元素的迭代器，如果没找到，则返回\_Last迭代器

vector::resize(n)如果需要扩容，则之后的容量是n

vector<int> vec(5);

vec.resize(27);

cout << vec.capacity() << endl; //27

string::resize(n)如果需要扩容，则之后的容量取值

string vec(5,'a');

vec.resize(27);

cout << vec.capacity() << endl; //31,因为只能取值15、31、47

['dek]

std::deque段连续内存，每段连续，段与段之间指针

vector的push\_back, pop\_back操作时间复杂度为O(1), 头部插入和删除操作时间复杂度为O(N)，因此vector未提供pop\_front()和push\_front()功能

deque提供push\_back()，push\_front()， pop\_back()， pop\_front()，时间复杂度都为O(1);





list < deque随机访问性能 < vector

vector < deque插入删除性能 < list

deque与vector的接口相比，没有capacity()和reserve()，多了push\_front()和pop\_frot()，其它函数相同

map查找效率高于list；插入效率低于list，因为要排序

map插入一个pair

std::map<string, int> m1{ { "a",3 },{ "b",4 },{"c",5} };

std::map<string,int>::iterator res

=m1.insert(m1.begin(), {"b",6}); //map中已有{"b",4}，因此插入失败，返回的res指向{"b",4}

std::map<string, int>::iterator res1

= m1.insert(m1.begin(), { "d",6 }); //插入成功，返回的res指向{"d",6}

std::pair<std::map<string, int>::iterator,bool> res2=m1.insert({ "b",6 }); //返回的pair.first指向{"b",4}，pair.second为false

std::pair<std::map<string, int>::iterator, bool> res3 = m1.insert({ "e",6 }); //返回的pair.first指向{"e",6}，pair.second为true

关联容器insert()在可以指定位置插入，可以不指定的位置。

指定的位置，仅仅是一个提示作用，帮助它从什么地方开始搜寻正确的安插位置。标准保证如果提示正确，那么插入将在分摊的常数时间内发生，而不是对数时间

iterator map::lower\_bound(key)查找key的位置，如果原map中没有key(返回的是end()或者返回的iterator->first不等于key)，则可调用map::insert(返回的迭代器,{key,value})常数时间插入

 对于关联容器(如map, set,multimap,multiset)，删除当前的iterator，仅仅会使当前的iterator失效，只要在erase时，递增当前iterator即可。这是因为map之类的容器，使用了红黑树来实现，插入、删除一个结点不会对其他结点造成影响

STL的map、multimap、set、multiset都有三个比较特殊的函数，lower\_bound、upper\_bound、equal\_range。

原型如下：

iterator lower\_bound (const value\_type& val) const; //返回关键字>=value的迭代器

iterator upper\_bound (const value\_type& val) const; //返回关键字>value的迭代器

pair<iterator,iterator> equal\_range (const value\_type& val) const;返回两个迭代器，组合为一个std::pair，第一个迭代器是lower\_bound的返回值，第二个迭代器是upper\_bound的返回值

对诸如set、map这种关键字唯一、且有序的集合而言：关键字val在集合中不存在，则lower\_bound和upper\_bound二者返回结果一样；如果关键在val在集合中存在，lower\_bound返回val关键字本身的迭代器，upper\_bound返回关键字val下一个元素迭代器

对multiset、multimap这类关键字不唯一、且有序的集合而言：关键字val不在集合中，这种情况下与set、map一致；关键字val出现在集合中，但是是唯一的，这种情况和set、map情况类似；关键字val出现在集合中，出现多次，这种情况下lower\_bound返回第一个出现关键字val对应的迭代器，upper\_bound返回位于关键字val对应位置后第一个不是val的位置的迭代器。

std::map<const string, int> //, std::greater<string>

m1{ { "a",3 },{ "d",4 },{"f",5} };

for\_each(m1.begin(), m1.end(), [](const std::pair<string, int>& i) {cout << i.first << " " << i.second << ", "; }); cout << endl;

cout << m1.lower\_bound("d")->second << endl; //4

cout << m1.lower\_bound("e")->second << endl; //5

cout << m1.equal\_range("d").first->second << endl;//4

cout << m1.equal\_range("d").second->second << endl; //5

std::multimap<int, int> mmp{ {1,100},{2,200},{2,300},{2,400},{5,500} };

auto rep=mmp.equal\_range(2);

cout << rep.first->second << endl;//200

cout << rep.second->second << endl; //500

算法std::upper\_bound()和std::lower\_bound()：

std::lower\_bound()、std::upper\_bound()两个函数用于记录有序数据集中给定关键字val在当前集合中区间范围

两个函数的用法类似，在一个左闭右开的有序(从小到大)区间里进行二分查找，需要查找的值由第三个参数给出。   
    对于upper\_bound来说，返回的是被查序列中第一个大于查找值的迭代器，lower\_bound则是返回的是被查序列中第一个大于等于查找值的迭代器

不过除此之外，这两个函数还分别有一个重载函数，可以接受第四个参数。如果第四个参数传入greater<Type>()，其中Type改成对应类型如greater<int>()，则在一个左闭右开的从大到小到大区间里进行二分查找，那么upper\_bound则返被查序列中第一个<查找值的指针，lower\_bound则返回被查序列中第一个<=查找值的指针。

最后说一点使用的注意事项，先看这么一句话“ The elements in the range shall already be sorted according to this same criterion (operator< or comp), or at least partitioned with respect to val”。简单来说，如果你用上述两个函数三个参数的那种形式，记得那个左闭右开的区间要为非递减的顺序，如果你给第四个参数传入greater<Type>()，则区间为非递增的顺序

二分查找的函数有三个，lower\_bound()、upper\_bound()和binary\_search()

binary\_search() (起始地址，结束地址，要查找的数值) 返回的是是否存在这么一个数，是一个bool值

数组无序时，查找结果无意义：

cout << "-------------------" << endl;

std::vector<int> v{ 10,20,30,30,20,10,10,20 };

std::vector<int>::iterator low, up;

low = std::lower\_bound(v.begin(), v.end(), 20); //

up = std::upper\_bound(v.begin(), v.end(), 20); //

std::cout << "lower\_bound at position " << (low - v.begin()) << '\n';//1

std::cout << "upper\_bound at position " << (up - v.begin()) << '\n'; //8

# 变长参数模板：

template<class...Args>

void expand(Args... args)

{

auto readarg = [](auto t) {return t; };

int arr[] = {readarg(args)...};

for (const auto& i : arr) cout << i << " "; cout << endl;

}

template<class... T>

void f(T... args) //打印变参（可变模板参数）的个数

{

cout << sizeof...(args) << endl; // func(2, "3.5", 6.7)则打印3

}

**template**<**class** T>

**void** get\_params(std::unique\_ptr<T[]>& params, **int** idx)

{

**return**;

}

**template**<**class** T, **class**... Args>

**void** get\_params(std::unique\_ptr<T[]>& params, **int** idx, T head, Args... rest)

{

params[idx] = head;

if (sizeof...(rest) == 0) return;

    get\_params(params, idx + 1, rest...);//如果没有void get\_params(std::unique\_ptr<T[]>& params, int idx)模板，则该句报错，应输入4个参数，却提供了2个. 因为上一句中的if是运行期的if，不是编译期的if，当rest数量为0，编译器仍然会编译下面那句递归调用

}

**int** main()

{

**int** len = 5;

    std::unique\_ptr<**int**[]> res = std::make\_unique<**int**[]>(len);

    get\_params(res, 0, 50, 60, 70, 80, 90);

**for** (**int** i = 0; i < len; i++)

    {

        cout << res[i] << " ";  //50 60 70 80 90

    }

}

下述代码会报错

//报错：二进制“ == ”: 没有找到接受“std::pair<const \_Kty, \_Ty>”类型的左操作数的运算符(或没有可接受的转换)

std::map<string, int,> m1{ { "a",3 },{ "b",4 },{"c",5} };

auto re = std::find(m1.begin(),m1.end(),std::pair<string,int>("b",4));

因为std::map<string, int>的value\_type是std::pair<const string, int>，find的第三个参数的类型改成std::pair<const string, int>

# stl::list::splice()：

splice是list中特有的拼接方法。可以在常数时间内从list的一个区域拼接到另一个list的一个区域

* void splice (iterator position, list& x);//将整个链表x拼接到迭代器position所指位置
* void splice (iterator position, list& x, iterator i);//将链表x的i迭代器位置的元素拼接到迭代器position所指位置
* void splice (iterator position, list& x, iterator first, iterator last);// 将链表x的[first, last)范围的元素拼接到迭代器position所指位置

C++11之前，为了实现splice()的O(1)复杂度，取折衷，把size方法设计成了O(N)。如果size()设计为O(1)复杂度，那么做splice时就需要遍历first和last间的长度（然后把链表A保存的链表长度减去first和last（待移动的元素）之间的长度）！于是作者考虑到size方法设计为O(N)，就无需在splice方法执行时做遍历了

C++11标准要求是所有容器的.size()为O(1)复杂度

# stl::list:: merge()

合并两个有序的链表，作为参数的链表将为空

std::list<int> lst1 = { 1,3,5,7};

std::list<int> lst2 = { 2,4,6,8 };

lst1.merge(lst2); //没有第二个参数，默认为由小到大

for (const auto& i : lst1) cout << i << " "; cout << endl;//1 5 5 7 9

for (const auto& i : lst2) cout << i << " "; cout << endl;//空

1）merge()是将两个有序的链表合并成另一个有序的链表，如果有一个链表不是有序则报错

2）两个链表中的内容排序顺序与合并时采用的排序顺序必须一致，如果不一致，也会报错，说链表不是有序的。如想要降序合并两个链表，那么合并前的两个链表也必须是按降序排列的。

std::list<int> lst1 = { 7,5,3,1};

std::list<int> lst2 = { 8,6,4,2};

lst1.merge(lst2,std::greater<int>());

for (const auto& i : lst1) cout << i << " "; cout << endl;//8 7 6 5 4 3 2 1

for (const auto& i : lst2) cout << i << " "; cout << endl;//空

# STL：

STL容器string、vector、deque、list、map、multimap、unordered\_map、unordered\_multimap、set、multiset、unordered \_set、unordered\_multiset

非标准容器slist、rope、hash\_set、hash\_map、hash\_multiset、hash\_multimap

标准非STL容器数组、bitset、stack、queue、valarray、priority\_queue

所有STL容器都typedef了value\_type，为其元素类型。如std::list中

typedef typename \_Mybase::value\_type value\_type;

# std::advance：

std::multimap<string, string> header = { {"a","aa"},{"b","bb"},{"b","bbb"} ,{"c","cc"} };

auto pos = header.find("d");

cout << ( pos== header.end()) << endl; //1

//std::advance(pos,2); //程序运行崩溃，不能超过范围

# 反向迭代器：

反向迭代器与普通迭代器在物理位置上保持了一一对应，即rbegin()对应普通迭代器的end()位置，rend()对应其begin()位置。物理位置即反向迭代器保存的指针指向的位置。



反向迭代器的逻辑位置等于其物理位置的前一个位置，即反向迭代器的operator \*和operator->等操作访问的都是其物理位置的前一个位置的元素。

反向迭代器riter转正向迭代器：reverse\_iterator::base()

算法reverse\_iterator pos=std::find(reverse\_iterator first, reverse\_iterator last, val)要判断有没有找到，必须在转为正向迭代器之前判断if(pos==last)

vector<int> vec{ 1,3,5,7,9 };

vector<int>::reverse\_iterator res = std::find(vec.rbegin(), vec.rend(), 9);

cout << \*res << endl; //9

if (res.base() == vec.end()) cout << "not find" << endl; //not find

反向迭代器继承普通迭代器，有数据成员\_RanIt current，主要函数实现：

**template**<**class** \_RanIt >

**class** reverse\_iterator: **public** iterator

{

**protected**:

    \_RanIt current; // the wrapped iterator

**public**:

    \_RanIt base() **const** { **return** (current); }

    reference operator\*() **const** {

        \_RanIt \_Tmp = current;

**return** (\*--\_Tmp);

    }

    pointer operator->() **const** {

**return** (pointer\_traits<pointer>::pointer\_to(\*\***this**));  //即return &(operator\*());

    }

    ......

};

# assign、=、swap与迭代器失效：

赋值操作符首先删除其左操作数容器中的所有元素，然后将右操作数容器的所有元素插入到左边容器中：

c1=c2;//赋值c2给c1

c1.erase(c1.begin(),c1.end());

c1.insert(c1.begin(),c2.begin());

//赋值后，左右两边的容器相等，尽管赋值前两个容器的长度可能不相等，但赋值后两个容器都具有右操作数的长度

c1=c2;//赋值c2给c1包括两步骤：

1. c1.erase(c1.begin(),c1.end());
2. c1.insert(c1.begin(),c2.begin());

//赋值后，左右两边的容器相等，尽管赋值前两个容器的长度可能不相等，但赋值后两个容器都具有右操作数的长度

赋值和assign操作使左操作数容器的所有迭代器失效。swap操作则不会使迭代器失效。

完成swap操作后，尽管被交换的元素已经存放在另一个容器中，但迭代器跟踪元素

vector<int> vec1 = { 3,4,5,6 };

auto i = vec1.begin() + 2;//5

auto i2 = vec1.begin() + 3;//6

vector<int> vec2 = { 10,20,30 };

auto j = vec2.begin() + 2;//30

std::swap(std::move(vec1), std::move(vec2));

cout << \*i << " " << \*j << endl; //5 30

cout << vec1.size() << endl; //3

cout << \*i2 << endl; //6

vector::clear()改变size不改变capacity

list<int>::iterator it = mylist.begin();

advance (it,5);

# 插入迭代器：

STL中主要有三种迭代器配接器：

* Insert Iterators (安插型迭代器)
* Stream Iterators（流迭代器）
* Reverse Iterators（逆向迭代器）

Insert Iterator 分类

* Back Inserters（安插于容器最尾端）。Back Inserters的内部调用push\_back()，也只有能提供push\_back()的成员函数的容器才能使用back insert迭代器，STL中有vector、deque、list、string。
* Front  Inserters（安插于容器最前端）。front inserts的内部调用push\_front()在STL容器中能使用该迭代器的容器有deque、list；
* General Inserters（一般性安插器）。简称inserts，内部调用insert()，所有的STL容器都提供insert(pos,val)成员函数

Insert迭代器，也称为inserts，内部会重载“=”，用来将“赋值新增”操作行为转换为“安插新值”操作。不同的insert迭代器会调用容器内部的push\_back(),push\_front(),insert()成员函数。



\_Myt& operator=(\_Valty&& \_Val){ // push value into container

iter = container->insert(iter, \_STD forward<\_Valty>(\_Val));

++iter;

return (\*this);

}

\_Myt& operator=(const \_Valty& \_Val){ // push value into container

container->push\_front(\_Val);

return (\*this);

}

\_Myt& operator=(const \_Valty& \_Val){ // push value into container

container->push\_back(\_Val);

return (\*this);

}

无论是vector还是list还是deque，插入单元素的insert(pos,val)函数返回一个迭代器，迭代器指向插入的元素

|  |  |
| --- | --- |
| 算式 | 效果 |
| \*iter | 无效果，传回insert迭代器本身\*this |
| iter++ | 无效果 |
| ++iter | 无效果 |
| iter=value | 调用重载的=运算符 |

{

\*dst\_iter=\*src\_iter;

dst\_iter++;

\*src\_iter;

}

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

vector<int> vec = { 10,20,30 };

std::copy(data, data + 5, std::inserter(vec,vec.begin()+1));

for (auto i : vec) cout << i << " "; cout << endl; //10 1 2 3 4 5 20 30

deque<int> de = { 10,20,30 };

std::copy(data, data + 5, std::inserter(de, de.begin() + 1));//10 1 2 3 4 5 20 30

for (auto i : de) cout << i << " "; cout << endl;

list<int> lst = { 10,20,30 };

list<int>::iterator pos = ++lst.begin();

std::copy(data, data + 5, std::inserter(lst, pos));//10 1 2 3 4 5 20 30

for (auto i : lst) cout << i << " "; cout << endl;

std::insert\_iterator<list<int>> it(lst,pos);

std::front\_insert\_iterator<list<int>> fit(lst);

std::back\_insert\_iterator<list<int>> bit(lst);

# 迭代器失效：

vector<int> myvec{ 10,20,30 };

auto posvec = myvec.begin() + 1;

auto resvec=myvec.insert(posvec, 1);

//cout << \*posvec << endl; //失效

cout << \*resvec << endl; // 1

deque<int> mydeq{ 10,20,30 };

auto posdeq = mydeq.begin() + 1;

auto resdeq = mydeq.insert(posdeq, 1);

//cout << \*posdeq << endl; //失效

cout << \*resdeq << endl; // 1

list<int> mylst{ 10,20,30 };

auto poslst = ++mylst.begin();

auto reslst = mylst.insert(poslst, 1);

cout << \*poslst << endl; //20

cout << \*reslst << endl; // 1

cout << "-------------" << endl;

deque<int> yourdeq{ 10,20,30 };

auto pos2 = yourdeq.begin();

auto res = yourdeq.insert(pos2, 1);

//cout << \*pos << endl; //失效

cout << \*res << endl; // 1

cout << "-------------" << endl;

deque<int> herdeq{ 10,20,30 };

auto posherdeq = herdeq.begin();

herdeq.push\_front(1);

cout << \*posherdeq << endl; //失效

# std::copy()：

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

list<int> v = { 10,20,30 };

std::copy(data,data+5,std::front\_inserter(v));

for (auto i : v) cout << i << " "; cout << endl; //5 4 3 2 1 10 20 30

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

list<int> v = { 10,20,30 };

std::copy(data,data+5,std::inserter(v,v.begin()));

for (auto i : v) cout << i << " "; cout << endl; //1 2 3 4 5 10 20 30，deque与list一致

与copy类似，std::set\_union()、set\_intersection()、set\_difference()、set\_ symmetric\_difference()、merge()的第5个参数是存放结果的容器的迭代器，必须保证容器有足够size，否则用插入迭代器

template <class InputIterator, class OutputIterator>

OutputIterator copy(InputIterator from\_begin, //beginning of source

InputIterator from\_end, //end of source

OutputIterator to\_pos) //beginning or dest

第一个参数是要拷贝元素的首地址，第二个参数是要拷贝的元素中最后一个元素的下一个位置，第三个参数是拷贝的目的地址

如果是int a[4] = {10,20,30,40};

std::vector<int> vec(4);

std::copy(a, a + 4, vec.begin());则MSVC报错，认为不安全，可能写溢出

当内存发生局部重叠的时候，memmove保证拷贝的结果是正确的，memcpy不保证拷贝的结果的正确。

void \*memcpy(void \*dest, const void \*src, size\_t n); 源和目的的内存区域不能重叠

void \*memmove(void \*dest, const void \*src, size\_t n); 源和目的的内存区域可以重叠

一般情况下，memcpy()函数比for循环快一两倍

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

由src指向地址为起始地址的连续n个字节的数据复制到以dest指向地址为起始地址的空间内

system("cls")：控制台清屏

std::copy\_backward() 其复制过程是从最后的元素开始复制，直到首元素复制出来。也就是说，复制操作是从last-1开始，直到first结束。这些元素也被从后向前复制到目标容器中，从result-1开始，一直复制last-first个元素

copy\_backward ( BidirectionalIterator1 first,BidirectionalIterator1 last,BidirectionalIterator2 result)；

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

int arr1[6] = { 0 };

std::copy\_backward(arr, arr + 5, arr1 + 5);

for (int i = 0; i < 6; i++)

{

cout << arr1[i] << endl; //1 2 3 4 5 0

}

list<int>::iterator it;

it = firstlist.begin(); advance(it, 3);

std::copy(secondlist.begin(), secondlist.end(), std::inserter(firstlist, it));

//std::insert\_iterator< list<int> > insert\_it(firstlist, it);

//copy(secondlist.begin(), secondlist.end(), insert\_it);

模板函数insert\_iterator<\_Container> std::inserter(\_Container& \_Cont,

typename \_Container::iterator \_Where)

{ // return insert\_iterator

return (insert\_iterator<\_Container>(\_Cont, \_Where));

}

list<int> ilist = {0,1,2,3,4,5};

vector<int> iVec;

copy(ilist.begin(), ilist.end(), back\_inserter(iVec));//采用迭代器 back\_inserter

或者

iVec.resize(ilist.size());

copy(ilist.begin(), ilist.end(), iVec.begin());//如果目标空间不足，则运行崩溃

# STL容器的区间成员函数：

应该使用STL容器的区间成员函数，代替其单元素兄弟。

在下面的，参数类型iterator意思是容器的迭代器类型，也就是container::iterator；参数类型InputIterator意思是可以接受任何输入迭代器

* 所有STL容器都有区间构造函数

container::container(InputIterator begin,//区间的起点

InputIterator end)

* 所有STL容器都有区间插入insert函数

container::insert(iterator position,//区间插入的位置

InputIterator begin,//插入区间的起点

InputIterator end);//插入区间的终点

关联容器使用它们的比较函数来决定元素要放在哪里，所以也有无position参数的重载

* 所有STL容器都有区间删除erase函数

iterator container::erase(iterator begin,iterator end)，返回的迭代器指向被删除元素的下一 个

* STL序列容器string、vector、deque、list都有区间赋值assign函数

container::assign(InputIterator begin, InputIterator end)

# STL容器的max\_size()成员

容器增长的上限，即最大size，可用max\_size()成员函数查看

std::map<int, int> m1;

cout << m1.max\_size() << endl; //461168601842738790

std::map<int, double> m2;

cout << m2.max\_size() << endl; //384307168202282325

std::vector<int> vec;

cout << vec.max\_size() << endl; //4611686018427387903

# map::operator[]和map::insert()

map <int, Widget> m;

m[key] = 1.50;

[key]运算符，key如果不存在，则调用Widget的无参构造函数，新建pair<key,Widget()>添加到map中，然后返回key对应的value（即新构造的Widget()）的引用。调用Widget(1.50)，拷贝赋值。

key如果存在，则直接返回key对应的value的引用。

对map <int, Widget>，只要调用[key]，即使key已存在map中，也需要Widget提供无参构造函数。因为运行时才能判断key是否已存在map中，但编译时，仍然需要编译key不存在map中的分支，因此编译器需要找到Widget的无参构造函数，Widget没有提供，则报错

# map:: key\_comp()

std::map:: key\_comp() 返回比较key的函数对象，即返回类模板map的第三个模板参数的()

std::map<string, int> ma = { {"a",1}, {"b",2} };

cout << ma.key\_comp()("a", "b") << endl; //1

std::map<string, int,std::greater<string>> mb= { { "b",2 },{ "a",1 } };

cout << mb.key\_comp()("a", "b") << endl; //0

# string、vector、deque的shrink\_to\_fit()

vector和string都有capacity()、reserve()成员

deque无capacity()、reserve()

C++ 11提供vector::shrink\_to\_fit()成员函数，实现容量“收缩到合适（shrink to fit）”

vector拷贝构造不拷贝容量，移动构造容量跟源vector保持一致

vector<int> vec{1,3,5};

vec.reserve(1024);

vector<int> vec2(vec); //vec2.capacity()为3

vector<int> vec3(std::move(vec)); //vec3.capacity()为1024

借助拷贝构造不拷贝容量的特性，可实现vetcor容量“收缩到合适（shrink to fit）”

vector<int> vec{1,3,5};

vec.reserve(1024);

vec.swap(vector<int>(vec)); //拷贝构造匿名对象，容量为3，交换，交换借助移动语义，实现为temp=std::move(lhs)，lhs=std::move(rhs)，rhs=std::move(temp)，因此连容量也交换，最终vec的容量为3.

vector的拷贝赋值函数，如果容量足够，则容量不变，不会收缩或扩张至与源vector一致

vector<int> vec{1,3,5};

vec.reserve(1024);

vector<int> vec2;

vec2.reserve(32);

vec2 = vec;

cout << vec2.capacity() << endl; //32

vector的移动语义，即将vector的3个指针数据成员交换，因此容量也被移动，迭代器不会失效，跟着移动

# 去除容器特定元素：

去除一个容器中有特定值的所有对象：

* 如果容器是vector、string或deque，使用erase-remove惯用法。
* 如果容器是list，使用list::remove。
* 如果容器是标准关联容器，使用它的erase成员函数。

去除一个容器中满足一个特定判定式的所有对象：

* 如果容器是vector、string或deque，使用erase-remove\_if惯用法。
* 如果容器是list，使用list::remove\_if。
* 如果容器是标准关联容器，使用remove\_copy\_if和swap，或写一个循环来遍历容器元素，当你把迭代器传给erase时记得后置递增它。

除了删除对象之外，做某些事情（如打印删除的元素）：

* 如果容器是标准序列容器，写一个循环来遍历容器元素，每当调用erase时记得都用它的返回值更新你的迭代器。
* 如果容器是标准关联容器，写一个循环来遍历容器元素，当你把迭代器传给erase时记得后置递增它

std::remove(iterator first,iterator last, const T &key) //first必须在last前，或者first==last，如果first实际位置在last后，则崩溃

std::find()、std::find\_if()

std::remove()，std::remove\_of()

//去掉vector中值小于x而出现在至少和y一样大的最后一个元素之后的所有元素

auto func=[](vector<int>& data, const int x, const int y)

{

vector<int>::iterator start;

vector<int>::reverse\_iterator temp = std::find\_if(data.rbegin(), data.rend(), [y](int i) {return i >= y; });

if (temp == data.rend()) return;

else start = --(temp.base());

data.erase(std::remove\_if(++start, data.end(), [x](int i) {return i < x; }), data.end());

};

不能简化为vector<int>::reverse\_iterator temp = std::find\_if(data.rbegin(), data.rend(), [y](int i) {return i >= y; });data.erase(std::remove\_if(temp.base(), data.end(), [x](int i) {return i < x; }), data.end());，如果没找到至少和y一样大的最后一个元素，则temp为data.rend()，直接转化为temp.base()，则丢失了未找到这一信息，导致出错

# 关联容器的比较函数必须严格弱序

STL要求有序区间的算法的比较函数必须严格弱序

std::map的模板参数

template<class \_Kty,

class \_Ty,

class \_Pr = less<\_Kty>,

class \_Alloc = allocator<pair<const \_Kty, \_Ty> > >

std::less<T>是模板类，重载了()运算符，可产生函数对象

constexpr bool operator()(const \_Ty& \_Left, const \_Ty& \_Right) const return (\_Left < \_Right);，

std::less\_equal调用\_Left <= \_Right

std::greater调用\_Left > \_Right

std:: greater\_equal调用\_Left >= \_Right

std::less<Widget>调用Widget的成员函数operator <，或者bool operator < (const Widget& lhs,const Widget& rhs)

用于排序关联容器和sort算法的比较函数必须是严格的弱序化(strict weak ordering)的

严格弱序的三条要求

* 两个关键字不能同时“严格弱序”于对方
* 如果a“严格弱序”于b，且b“严格弱序”于c，则a必须“严格弱序”于c
* 如果存在两个关键字，任何一个都不“严格弱序”于另一个，则这两个关键字是相等的

std::map<int,string,std::less\_equal<int>> m{{10,”a”},{10,”b”}}，该map对象的构造会使程序崩溃，因为map::key\_comp()(key1,key2)和map::key\_comp()(key2,key1)都为真，无法确定key1和key2如何排序

考虑multiset<int, less\_equal<int>>s;//s仍然以“<=”排序

s.insert(10);//插入10A

s.insert(10);//插入10B

现在，s里有两个10的拷贝，调用s.equal\_range(10)，希望得到一对指出包含这两个拷贝的范围的迭代器。但那是不可能的。equal\_range，虽然叫这个名字，但不是指示出相等的值的范围，而是等价的值的范围。在这个例子中，s的比较函数说10A和10B是不等价的，所以不可能让它们同时出现在equal\_range所指示的范围内。

std::map<>的默认比较函数是std::less<键类型>

关联容器的find()、insert()等成员函数，判断是否等价，由模板参数中的比较函数定义

如果!map::key\_comp()(x,y)&&!map::key\_comp()(y,x)，则等价，即在容器的排列顺序中，x既不在y前边，y也不在x前边

* 对关联容器的成员函数map::find(key)，用!map::key\_comp()(x,y)&&!map::key\_comp()(y,x)的方式判断等价
* 对算法std::find(map::begin(), map::end(),key)调用key所属类型的operater ==，判断相等

# lambda作为函数参数、返回值，不能作为模板参数：

函数可以返回lambda表达式

template<class MEM\_FUNC>

auto your\_mem\_fun(MEM\_FUNC f)

{

auto lam = [f](auto x) {return (x.\*f)(); };

return lam;

}

使用for\_each对容器中的每个对象调用其Widget::test

作为函数参数时，lambda表达式可以替换函数对象

作为模板类型参数时，如std::set<string,std::less<int>>，不能用lambda表达式或者decltype(lambda表达式)替换仿函数，因为lambda表达式不是类型参数，decltype(lambda表达式)没有无参构造函数

auto comp = [](const string\* lhs, const string\* rhs){return \*lhs < \*rhs;};

struct comp\_struct

{

bool operator()(const string\* lhs, const string\* rhs) const{return \*lhs < \*rhs;}

};

set <string\*, comp\_struct> ssp;

//set <string\*, comp> ssp;// error，comp非类型

//set <string\*, decltype(comp) > ssp; //error，尝试引用已删除的函数（即没有无参构造函数）

# map键不可修改

map<K,V>或multimap<K,V>类型的value\_type是std::pair<const K,V>。因为键的类型const K，不能改变。试图修改键是未定义的行为（即使用const\_cast映射掉它的常量性）

对于set<T>或multiset<T>类型的对象来说，储存在容器里的元素类型是T，并非const T。

对std::set<Widget>，Widget有数据成员id、rank，set根据id排序，可以修改set中某个元素的rank，但修改其id则是未定义行为，因为id是set的键。

但某些STL实现，set < T >::iterator的operator\* 返回一个const T类型，这样没法修改rank，需要(const\_cast<Employee&>(\*iter)).set\_rank(10);或者erase该元素，insert新的符合要求的元素

**class** Employee

{

**private**:

**int** id;

**int** rank;

**public**:

    Employee(**int** id\_, **int** rank\_) :id(id\_), rank(rank\_){}

**void** set\_rank(**int** x){rank=x; }

**int** get\_rank() **const**  { **return** rank; }

**int** get\_id() **const** { **return** id; }

};

**struct** comp

{

**bool** operator ()(**const** Employee& lhs, **const** Employee& rhs) { **return** lhs.get\_rank() < rhs.get\_rank(); }

};

set<Employee, comp> perpons{ Employee(1,1),Employee(2,2) ,Employee(3,3) };

set<Employee, comp>::iterator iter = ++perpons.begin();

(**const\_cast**<Employee&>(\*iter)).set\_rank(10);

**for** (auto& i : perpons) cout << i.get\_rank() << " "; cout << endl; //1 10 3

map<string, int> word\_count;

for (const auto& kv : word\_count)

{

// kv.first, kv.second

}

如果你想把 auto 的类型写出来，一定不要忘了 key 是 const。

for(const std::pair<const std::string, int>& kv : word\_count) {}

如果你写成了 pair<string, int>，会造成额外的拷贝：

for (const std::pair<std::string, int>& kv : word\_count) {}// 低效的写法：

# const\_iterator转为iterator：

箭头表示隐式转换

含base()的箭头表示通过调用base()成员函数转转



从iterator到const\_iterator、从iterator到reverse\_iterator和从reverse\_iterator到const\_reverse\_iterator可以进行隐式转换。并且，reverse\_iterator可以通过调用其base成员函数转换为iterator。const\_reverse\_iterator也可以类似地通过base转换成为const\_iterator

const\_iterator可以隐式转换为iterator

const\_iterator转为iterator::

vector<int>::const\_iterator citer =;

vector<int>::iterator res=vec.begin();

std::advance(res, std::distance((decltype(citer))res, citer));

对随机访问迭代器，string、vector、deque是常数时间复杂度，对双向迭代器（所有其它STL容器）而言，这是线性时间的操作。

模板函数std::distance的声明为：

template<class \_InIt> inline

\_Iter\_diff\_t<\_InIt> distance(\_InIt \_First, \_InIt \_Last)

std::distance(res, citer)会报错，由于res是iterator类型，citer是const\_iterator类型，且模板参数推导时不考虑构造函数引起的隐式类型转换，因此导致模板参数不能推导出。必须std::distance((vector<int>::const\_iterator )res, citer)

**class** C {};

**class** It

{

**public**:

    It() = **default**;

    It(C) {}

**bool** operator==(**const** It& rhs);

};

C c;

It i;

i == c;  //c先隐式转化为It

//c == i;//error:没有与这些操作数匹配的 "==" 运算符

vector<**int**> vec{ 1,3,5,7,2,4,6,8 };

vector<**int**>::const\_iterator citer = vec.begin();

vector<**int**>::iterator it = vec.begin();

citer== it; //okay

it == citer; //oka

\_Vector\_iterator继承了\_Vector\_const\_iterator

因此iterator可隐式转换为cosnt\_iterator

template<class \_Myvec>

class \_Vector\_iterator: public \_Vector\_const\_iterator<\_Myvec>

\_Vector\_const\_iterator提供了public bool \_Vector\_const\_iterator::operator==(const \_Myiter& \_Right) const

以下代码

citer==it和it == citer都可编译通过，都调用bool \_Vector\_const\_iterator::operator==

# STL排序算法：

STL排序算法按照需要更少资源（时间和空间）的顺序排序如下：

partition、partial\_sort、stable\_partition、sort、nth\_element、stable\_sort

sort、stable\_sort、partial\_sort、nth\_element要求随机访问迭代器

对partition和stable\_partition传入双向迭代器即可，任何STL容器皆可使用

STL无partial\_sort和nth\_element的稳定版本

传给排序算法的比较函数必须严格弱序，排序算法通过等价判断两个元素是否相同

我们总结一下你的排序选择：

* 如果你需要在vector、string、deque或数组上进行完全排序，使用sort或stable\_sort。
* 如果你有一个vector、string、deque或数组，只需要排序前n个元素，应该用partial\_sort。
* 如果你有一个vector、string、deque或数组，你需要鉴别出第n个元素或你需要鉴别出正前的n个元素，而不用知道它们的顺序，nth\_element是你应该注意和调用的。
* 如果你需要把标准序列容器的元素或数组分隔为满足和不满足某个标准，你大概就要找partition或stable\_partition。
* 如果你的数据是在list中，你可以直接使用partition和stable\_partition，你可以使用list的sort来代替sort和stable\_sort。如果你需要partial\_sort或nth\_element提供的效果，你就必须间接完成这个任务。一个间接的方法是把元素拷贝到一个支持随机访问迭代器的容器中，然后对它应用需要的算法。另一个方法是建立一个list::iterator的容器，对那个容器使用算法，然后通过迭代器访问list元素

把最好的20个元素按顺序放在widgets的前端：

partial\_sort(widgets.begin(),widgets.begin()+20, widgets.end(),qualityCompare)

把最好的20个元素放在widgets前端，但不用管它们的顺序：

nth\_element(widgets.begin(),widgets.begin()+19,widgets.end(),//qualityCompare);第二个参数widgets.begin()+19指向的元素位置在排序后一定正确，且之前的都小于等于之

partition算法，它重排区间中的元素以使所有满足某个标准的元素都在区间的开头，如把所有满足hasAcceptableQuality的widgets移动到widgets前端，并且返回一个指向第一个不满足的widget的迭代器：

vector<Widget>::iteratorgoodEnd=partition(widgets.begin(),widgets.end(),hasAcceptableQuality);

template< class RandomIt, class Compare >  
void sort( RandomIt first, RandomIt last, Compare comp );

std::sort能应用于string、vector、deque，不能用于list，因为list的迭代器不是随机访问迭代器

# 要求有序区间的STL算法：

binary\_search()、lower\_bound()、upper\_bound()、equal\_range()

set\_union()、set\_intersection()、set\_difference()、set\_ symmetric\_difference

merge()、inplace\_merge()

includes

要求有序区间的STL算法的最后一个参数是比较函数

要求有序区间的STL算法通过等价来判断两个值是否“相同”，就像标准关联容器

list<int> a = { 7,9,1,3,5,2,4 };

list<int> b = { 8,10,2,4,6,3,5 };

a.sort(); //1,2,3,4,5,7,9

b.sort(); //2,3,4,5,6,8,10

list<int> m;

//求并集，不改变原容器，存放合并结果的容器必须size足够

std::set\_union(a.begin(), a.end(), b.begin(), b.end(), std::back\_inserter(m)); //1 2 3 4 5 6 7 8 9 10

m.clear();

//合并，不改变原容器，存放合并结果的容器必须size足够

std::merge(a.begin(),a.end(),b.begin(),b.end(), std::back\_inserter(m)); //1 2 2 3 3 4 4 5 5 6 7 8 9 10

任何STL算法都不会删除原容器的元素，不会改变原容器的size

std::set\_intersection：求交集，不改变原容器，存放合并结果的容器必须size足够

std::set\_difference：集合相减，不改变原容器，存放合并结果的容器必须size足够//1 7 9

std::set\_symmetric\_difference：a-b并b-a，不改变原容器，存放合并结果的容器必须size足够//1 6 7 8 9 10

std::merge()：两个有序区间合并为一个有序区间，不改变原容器，存放合并结果的容器必须size足够

std::inplace\_merge(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last)，将[first,middle)和[middle,last)两个已经有序的序列合并成一个有序的序列。两个小区间应该已经有序，并且都是使用同一排序规则。first、middle、last都是在同一个容器里面的，只是被分开成了两个区间

std::includes()：集合a是否包含集合b；如果第二个集合是空的集合，也返回 true

a = { 1,2,3,7 };

b = { 1,3,4};

bool is1=std::includes(a.begin(), a.end(), b.begin(), b.end()); //0

bool is2 = std::includes(a.begin(), a.end(), b.begin(), --b.end()); //1

算法set\_union、set\_intersection、set\_difference和set\_symmetric\_difference四人组提供线性时间复杂度

算法binary\_search、lower\_bound、upper\_bound和equal\_range使用二分查找，因此当传递的是随机访问迭代器，则对数时间复杂度；如果给它们威力比较小的迭代器（比如双向迭代器）它们仍然进行对数次比较，但运行是线性时间的（缺乏 “迭代器算术”能力，需要花费线性时间来从一个地方移动到另一个地方）

算法merge和inplace\_merge：线性时间，单遍合并排序算法

算法includes()：线性时间

# std::unique\_copy:

unique\_copy相比unique，接收第三个迭代器实参，用于指定复制不重复元素的目标序列

unique\_copy先去除重复的，再copy

list<**int**> a = { 5 , 2 , 2 , 2 , 100 , 5 , 2 };

std::unique(a.begin(), a.end());  //5 2 100 5 2 5 2，类似std::remove，并未真正删除，只是拷贝赋值实现覆盖

a= { 5 , 2 , 2 , 2 , 100 , 5 , 2 };

list<**int**> lst;

std::unique\_copy(a.begin(), a.end(), std::back\_inserter(lst)); //容器a不变，lst存储5 2 100 5 2

# std::equal、std::mismatch

bool equal(first1, last1, first2)和std::pair<iterator,iterator> mismatch(first1, last1, first2)逐元素比较

这两个算法要求第二个容器比第一个区间长，否则运行崩溃

list<int> la = {1,3,5,7};

list<int> lb = { 1,3,5 };

//cout << std::equal(la.begin(), la.end(), lb.begin()) << endl; //崩溃

如果对于区间[first1,last1)内所有的first1+i,first1+i和first2+i所指元素都相等，则equal算法返回真，否则返回假。

mismatch算法的返回值是由两个迭代器first1+i和first2+i组成的一个pair，表示第1对不相等的元素的位置。如果没有找到不相等的元素，则返回last1和first2+(last1-first1)。

因此，语句equal(first1,last1,first2)和mismatch(first1,last1,first2).first==last1是等价的

list<int> la = {1,3,5,7};

list<int> lb = { 1,3,5,7,8 };

cout << std::equal(la.begin(), la.end(), lb.begin()) << endl; //1

std::mismatch和std::equal可提供第四个参数，第四个参数判断两个元素是否相等，返回true则相等，返回false不相等

# bool std::lexicographical\_compare(first1, last1, first2, last2, comp)：

#include<cctype>

std::tolower()，参数和返回值都是int，将大写字母转换为小写字母，非大写字母字符不做处理

std::toupper()，将小写字母转换为大写字母，非小写字母字符不做处理

STL中名字最长的算法std::set\_symmetric\_difference，

名字第二长的算法则std::lexicographical\_compare

template <class InputIterator1, class InputIterator2, class Compare>

bool lexicographical\_compare (InputIterator1 first1, InputIterator1 last1,

InputIterator2 first2, InputIterator2 last2,

Compare comp);

该函数是按照字典序测试[frist1,last1)是否小于[first2,last2)，该函数使用comp或默认的operator <进行比较，比较函数要求严格弱序，判断两个元素是否等价：!comp(a, b)&& !comp(b, a)

lexicographical\_compare基于comp寻找[frist1,last1)和[first2,last2)第一个不同的位置，对该位置的两个元素调用comp，comp的结果即整个lexicographical\_compare的结果

lexicographical\_compare对两个完全相等的区间返回false，因为[frist1,last1)不在[first2,last2)前

# 算法accumulate、for\_each：

#include<numeric>

list<double> lst = {3.7,2.8};

cout << "sum:" << std::accumulate(lst.begin(), lst.end(), 0.0) << endl; //不能是0，否则返回5

std::accumulate的返回值类型，由第三个参数决定，模板参数推导，第三个参数是0则返回int，是0.0则返回double

template<class \_InIt, class \_Ty> inline

\_Ty accumulate(\_InIt \_First, \_InIt \_Last, \_Ty \_Val)

std::accumulate()的另一种形式，带有一个初始和值与一个任意的统计函数

set<string> names {"zhang","zhou","li"};

int len = std::accumulate(names.begin(), names.end(), 0, [](int own, const string& cur, )->int {return own + (int)cur.size(); }); //11

自定义统计函数带有到目前为止区间的统计值和区间的下一个元素，它返回新的统计值

vector<double> incre{ 4.0,5.0,2.5};

double multi = std::accumulate(incre.begin(), incre.end(), 1.0, std::multiplies<double>()); //50, 计算乘积，使用仿函数std::multiplies<>

传给std::for\_each的函数可以有副作用，如函数对象可以有数据成员

传给std::accumulate的函数不可以有副作用

C++标准库、STL一般按值传递、按值返回函数指针、函数对象。

最好的证据是标准的for\_each声明。

但对for\_each，用户也可以显式给其按引用传递函数指针。但不建议这样做，对有些STL算法的实现会不能编译

std::for\_each声明：

template<class \_InIt, class \_Fn1>

inline \_Fn1 for\_each(\_InIt \_First, \_InIt \_Last, \_Fn1 \_Func)

显式地按引用给for\_each传递函数对象，让for\_each返回函数对象的引用：

class Do { public:void operator()(int) {} };

std::for\_each<vector<int>::iterator, Do&>(vec.begin(), vec.end(), Do());

# STL的判断式应该是纯函数：

const成员函数可以修改该类的multable数据成员、该类的非const静态数据成员、非const引用数据成员、命名空间域的非const变量和全局非const变量，const成员函数体内可以有非const局部静态变量。

因此const成员函数不一定是纯函数

class B

{

private:

int& age;

static string sex;

mutable double weight;

public:

B(int& age\_, double weight\_) :

age(age\_), weight(weight\_) {}

void show() const

{

static int callnum = 0;

cout << age << " " << sex << " " <<

weight<<" "<<callnum << endl;

age = 99;

sex = "boy";

weight = 45;

callnum++;

}

};

string B::sex = "girl";

int main()

{

int age = 26;

B obj(age, 78.0);

obj.show(); //26 girl 78 0

obj.show(); //99 boy 45 1

}

纯函数是返回值只依赖于参数的函数。如果f是一个纯函数，x和y是对象，f(x,y)的返回仅当x或y的值改变的时候才会改变

由于STL按值传递函数。传递给STL的的判断式（返回bool的函数），包括函数、函数对象、lambda表达式等，必须（最好）是纯函数，否则有时候运行结果不符合用户预期。如std::remove\_if()、std::copy\_if()、std::equal()、std::mismatch()等STL算法。

# std::binary\_search查找目标的类型与容器value\_type不一致：

struct comp

{

bool operator()(const std::pair<string, int>& p, const string& tar) { return p.first < tar; }

bool operator()(const string& tar, const std::pair<string, int>& p) { return tar < p.first; }

};

vector<std::pair<string, int>> vec{ std::make\_pair("li",21),std::make\_pair("zh",23) }; //按pair::first由小到大排序

bool exist = std::binary\_search(vec.begin(), vec.end(), "zh", comp()); //第四个参数是排序用的比较函数，而不是判断相同的函数

仿函数类可以有多个operator ()函数

std::binary\_search()第3个参数的类型（要查找的目标）可以不与容器的value\_type一致，如vector的元素类型为pair，查找参数类型为string

std::binary\_search()第3个参数的类型与容器的value\_type不一致时，必须提供的第四个参数。由于std::binary\_search(\_FwdIt \_First, \_FwdIt \_Last, const \_Ty& \_Val, \_Pr \_Pred)的实现中，即有\_Pred(\_Val, \*\_First)，又有\_Pred(\*\_Mid, \_Val)，因此提供的第四个参数必须包含两个比较函数，一个是bool (第3个参数的类型, 容器的value\_type)、一个是bool (容器的value\_type, 第3个参数的类型)，因此设计为一个仿函数，该仿函数有两个operator()函数

# STL函数适配器not1、not2、bind1st、bind2nd：

STL的四个标准函数适配器：not1、not2、bind1st和bind2nd

全局函数bool isInteresting(const Widget\* pw);

not1(isInteresting);//错误！不能编

not1(ptr\_fun(isInteresting));//没问题

std::not1将返回类型为bool、仅含一个参数的函数对象转换为反义函数对象

std::not2将返回类型为bool、含2个参数的函数对象转换为反义函数对象

std::not2的实现类似：

template<class Func> //Func为仿函数类

auto not2(Func x)

{

return [x](Func::first\_argument\_type fst, Func::second\_argument\_type sec)->bool {return !(x(fst, sec)); };

}

可见，传给not2()的函数对象要求有first\_argument\_type 、second\_argument\_type这两个typedef，因此，这时仿函数类一般继承std::binary\_function<其operator()的第一个参数的类型, 其operator()的第二个参数的类型, 其operator()的返回类型>

**struct** comp :**public** std::binary\_function<**int**, **int**, **bool**>

{

**bool** operator()(**int** a, **int** b) **const** { **return** a < b; }  //由小到大排序

};

**int** main()

{

    vector<**int**> vec{ 5,3,1,4,8,6};

    std::stable\_sort(vec.begin(), vec.end(), std::not2(comp()));   //由大到小排序，第三个参数可以替换为[](int a, int b) {return !comp()(a, b); }，则不需要:binary\_function、not2等

**for** (auto i : vec) cout << i << " "; cout << endl;  // 8 6 5 4 3 1

}

std::binary\_function类模板的定义为

**template**<**class** \_Arg1, **class** \_Arg2, **class** \_Result>

**struct** binary\_function

{   // base class for binary functions

**typedef** \_Arg1 first\_argument\_type;

**typedef** \_Arg2 second\_argument\_type;

**typedef** \_Result result\_type;

};

应用于std::not1适配器的仿函数类，应该public继承

std::unary\_function<operator()的参数类型, 其operator()的返回类型>

# 模板函数ptr\_fun、mem\_fun\_ref、mem\_fun：

vector<Widget> vec{ Widget(1,10),Widget(2,20),Widget(3,30),Widget(4,40)};

//以下两种方法等价

std::for\_each(vec.begin(), vec.end(), std::function<bool(const Widget&)>(&Widget::small));

std::for\_each(vec.begin(),vec.end(),std::mem\_fun\_ref(&Widget::small));

vector<Widget\*> ptr(vec.size());

for (int i = 0; i < ptr.size(); i++) ptr[i] = &vec[i];

std::for\_each(ptr.begin(), ptr.end(), std::mem\_fun(&Widget::small));

//ptr\_fun包装全局函数

//std::mem\_fun\_ref包装成员函数，返回函数对象的operator()的第一个参数是类对象

//std::mem\_fun\_包装成员函数，返回函数对象的第一个参数是类对象的指针

算法transform将区间的每个元素独立变换，然后放置到新的位置

//每一个元素，把它乘以20，然后将结果插入一个deque的前端

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

vector<double> b{6.0,7.0,8.0};

std::transform(a, a + 5, std::inserter(b, b.begin()), [](int x) {return x\*10.0; }); //10 20 30 40 50 6 7 8

把两个区间的成对的平均值写入一个流

vector<int> data{ 1,3,5,7,9 };

vector<int> res{ 4,8,12,16,20 };

std::transform(data.begin(),data.end(),res.begin(),

std::ostream\_iterator<int>(cout, " "),

[](int a, int b) {return (a + b) / 2; });//2 5 8 11 14

**C++ stl排列：**

#include <algorithm>

#include <cstdio>

using namespace std;

int main()

{

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

do

{

printf("%d %d %d %d\n", a[0], a[1], a[2], a[3]);

} while (std::prev\_permutation(a, a + 4));

printf("\nnext: %d %d %d %d\n", a[0], a[1], a[2], a[3]);

printf("\n------------------\n\n");

int b[] = { 2, 3, 1, 4 };

do

{

printf("%d %d %d %d\n", b[0], b[1], b[2], b[3]);

} while (std::next\_permutation(b, b + 4));

printf("\nnext: %d %d %d %d\n", b[0], b[1], b[2], b[3]);

}

Permutes the range [first, last) into the next permutation, where the set of all permutations is ordered lexicographically with respect to operator< or comp. Returns true if such a "next permutation" exists; otherwise transforms the range into the lexicographically first permutation (as if by std::sort(first, last, comp)) and returns false.

运行结果：

2 3 1 4

2 1 4 3

2 1 3 4

1 4 3 2

1 4 2 3

1 3 4 2

1 3 2 4

1 2 4 3

1 2 3 4

next: 4 3 2 1

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

2 3 1 4

2 3 4 1

2 4 1 3

2 4 3 1

3 1 2 4

3 1 4 2

3 2 1 4

3 2 4 1

3 4 1 2

3 4 2 1

4 1 2 3

4 1 3 2

4 2 1 3

4 2 3 1

4 3 1 2

4 3 2 1

next: 1 2 3 4

**C++11随机数：**

std::default\_random\_engine是平台实现相关的，可以定义为std::mt19937，也可以定义为std::minstd\_rand0;

如vs2017上typedef std::mt19937 std::default\_random\_engine;

随机数引擎实际上是个仿函数，即重载了()运算符，输出一个无符号整数，连续调用operator()输出的序列是均匀分布的

引擎内部维护一个状态，每次以operator()调用它，内部状态会改变，使得可以此后再产出一个新随机值

引擎类型及内部状态完全决定了引擎即将输出的随机数序列，如

std::mt19937 dre;

for (int i = 0; i < 4; i++) cout << dre() << " "; cout << endl;

std::stringstream eng\_state;

eng\_state << dre; //保存引擎状态

for (int i = 0; i < 4; i++) cout << dre() << " "; cout << endl;

eng\_state >> dre; //恢复引擎状态

for (int i = 0; i < 4; i++) cout << dre() << " "; cout << endl;

/\*运行结果：

3499211612 581869302 3890346734 3586334585

545404204 4161255391 3922919429 949333985

545404204 4161255391 3922919429 949333985

\*/

std::cout << std::endl;

std::mt19937 dre2;

for (int i = 0; i < 4; i++) cout << dre2() << " "; cout << endl;

std::stringstream state;

state << dre2; //保存引擎状态

std::mt19937 other;

state >> other;

for (int i = 0; i < 4; i++) cout << other() << " "; cout << endl;

/\*运行结果：

3499211612 581869302 3890346734 3586334585

545404204 4161255391 3922919429 949333985

545404204 4161255391 3922919429 949333985

\*/

与srand()给rand()设置随机数种子相似，引擎的初始状态即随机数种子，也可以设置

std::mt19937 dre; //不设置seed，则有默认固定值

std::mt19937 dre(unsigned(std::time(NULL))); //参数为seed

unsigned seed = std::time(NULL);

cout << seed << endl; //1619851230

std::mt19937 dre(seed);

std::stringstream state; state << dre; std::string str; state >> str;

cout << str << endl; //1619851230

虽然引擎的输出符合均匀分布，但不要用%n取余的方式，来生成小范围的随机数。就像不能用rand()%n一样，因为会导致随机性下降

使用rand()%n计算随机数，会因以下两个原因而失败：

1. 当商(quotient)为小整数时，许多伪(pseudo)随机数生成器制造出来的余数(remainder)并不很随机。例如rand()的连续执行结果非偶即奇（这并不十分罕见）这种情况下如果连续执行rand()%2，则输出一直是奇数或一直是偶数

cout << "RAND\_MAX=" << RAND\_MAX << endl;

srand((unsigned)std::time(NULL));

for (int i = 0; i < 100; i++) cout << rand() % 2<< " ";



1. 当商为大整数时，且被生成之最大值并非均匀地(evenly)可被除尽，那么某些余数的出现率会远高于其他。假设最大数是32767，而n是2000，则17个生成值(500,2500,…,30500,32500)将映射500,16个生成值(1500,3500,...,31500)映射至1500。这个情况大时愈糟糕。

使用良好的线性生成器(linear generator)把“引擎生成值”映射至“生成数区间”时，会考虑上述因素

分布的输出完全由引擎的输出决定

std::mt19937 dre;

std::uniform\_int\_distribution<int> di(10, 20);

for (int i = 0; i < 6; i++) cout << di(dre) << " "; cout << endl;

std::stringstream eng\_state;

eng\_state << dre; //保存引擎状态

for (int i = 0; i < 6; i++) cout << di(dre) << " "; cout << endl;

std::mt19937 dre2;

eng\_state >> dre2; //恢复引擎状态

for (int i = 0; i < 6; i++) cout << di(dre2) << " "; cout << endl;

/\*运行结果：

16 13 20 19 14 17

10 16 15 14 12 13

10 16 15 14 12 13

\*/

引擎的result\_type导出其类型，而static成员函数min()、max()获得可被生成之最小和最大值（两者都包含）

cout << typeid(std::mt19937::result\_type).name() << endl; //unsigned int

cout << std::mt19937::min() << " " << std::mt19937::max() << endl; // 0 4294967295（2^32-1）

cout << typeid(std::mt19937\_64::result\_type).name() << endl; // unsigned \_\_int64

cout << std::mt19937\_64::min() << " " << std::mt19937\_64::max() << endl; // 0 18446744073709551615（2^64-1）

cout << typeid(std::minstd\_rand::result\_type).name() << endl; // unsigned int

cout << std::minstd\_rand::min() << " " << std::minstd\_rand::max() << endl; // 1 2147483646

std::random\_device

真随机数生成器，不需要初始化seed，具体实现平台相关

std::random\_device rd; // 真随机数生成器（但有些编译器没有实现），用于生成伪随机数生成器的种子

std::mt19937 eng(rd()); // 使用梅森旋转法作为伪随机数生成器，随机程度较好

std::uniform\_int\_distribution<int> dis(10, 20); // 10到20之间的整数均匀分布

dis(eng)

为什么不直接使random\_device，因为生成真随机数需要消耗熵池里的资源，如果熵池空了，请求生成随机数的行为就会被阻塞，要等到收集足够多的环境噪声后才能继续产生真随机数。

**RAW String：**

传统字符串：std::string str=”aaa”

raw字符串：std::string str=”(aaa)”

即传统字符串用””界定，raw字符串用”()”界定

如果raw字符串中要打印的字符包含”)，则需要自定义界定符

界定符的长度以包含的字符集，都有明文规定：

The string delimiter can be any string up to 16 characters in length, including the empty string.

This string cannot contain spaces, control characters, '(', ')', or the '\' character.

想要输出换行符直接在代码里面换行即可

std::string str = R"(I am() raw \\"string".)";

cout << str << endl;

str= R"ZCJ(I am(" )" raw \\"string".)ZCJ";

cout << str << endl;

str = R"ZCJ(I am(" )" raw

\\"string".)ZCJ";

cout << str << endl;



**std::vector<bool>与auto：**

{

std::vector<bool> vec = { false,false };

auto x = vec[0]; //x类型为std::vector<bool>::reference

x = true;

cout << vec[0] << " " << vec[1] << endl; //1 0

}

{

std::vector<bool> vec = { false,false };

auto x = \*vec.begin();

x = true;

cout << vec[0] << " " << vec[1] << endl; //1 0

/\*

x是一个proxy class，而这个class在进行赋值操作符=时，是返回的引用，

于是对x操作时，会改变vector<bool>本身的元素

\*/

}

{

//易错用法

std::vector<bool> vec = { false,false };

for (auto i : vec)

{

i = true;

}

cout << vec[0] << " " << vec[1] << endl; // 1, 1

}

{

//无论是否含proxy class reference的range：

//当你想要修改range的元素时，使用for(auto && x : range)

//当你想要只读range的元素时，使用for(const auto & x : range)

}

# C++17的std::optional：

std::optional可表示的范围是是类型T 的所有值和一个单独的“什么都没有”状态

#include<optional>

int main()

{

std::optional<std::map<int, int>> opt = { {{1,2},{3,4}} };

if (opt.has\_value())

{

cout << opt.value()[1] << endl; //2

cout << opt.value()[3] << endl; //4

}

opt = std::optional<std::map<int, int>>();

if (!opt.has\_value())

{

cout << "empty" << endl; //empty

}

}

**decltype**

decltype作用于表达式时返回的是左值引用

int x = 10;

decltype(x) y1 = x; // y1 is int

decltype((x)) y2 = x; // y2 is int& !

y2 = 8;

cout << x << endl; //8

return 0;

如果x是变量名，那(x)就是表达式

**模板参数为bool类型**

1、

// "Choose<bool, A, B>":类型选择模板(使用模板来实现,根据参数不同匹配不同的模板)

template<bool C, typename A, typename B> struct Choose {};

// 模板特化

template<typename A, typename B> struct Choose<true, A, B> { typedef A type; }; // first place

template<typename A, typename B> struct Choose<false, A, B> { typedef B type; }; // second place

void testTemplate()

{

Choose<true, char, double>::type bb; // 直接匹配 first place 位置的特化模板

cout << " the sizeof(bb) is = " << sizeof(bb) << endl; //1

Choose<false, char, double>::type cc; // 直接匹配 second place 位置的特化模板

cout << " the sizeof(cc) is = " << sizeof(cc) << endl; //8

//使用这两个参数会报 参数不匹配 错误Choose<char, int>::type dd;

}

2、

template <bool isServer>

struct websocket

{

void show()

{

printf("%s\n", isServer ? "is a server" : "is a client");

}

};

int main()

{

websocket<true>().show(); //"is a server"

websocket<false>().show(); //"is a client"

websocket<1>().show(); //"is a server"

websocket<0>().show(); //"is a client"

}

# std::unordered\_multiset



int main()

{

srand((unsigned int)time(nullptr));

std::unordered\_multiset<int> unmset;

for (int i = 0; i < 100000; i++)

{

unmset.insert(rand());

}

cout << unmset.size() << endl; //100000

cout << unmset.bucket\_count() << endl; //131072

cout << unmset.load\_factor() << endl; //0.762939

cout << unmset.max\_size() << endl; //768614336404564650

cout << unmset.max\_bucket\_count() << endl; //1152921504606846975

cout << unmset.max\_load\_factor() << endl; //1

for (int i = 0; i < 10; i++)

{

cout << unmset.bucket\_size(i) << " "; //0 0 0 0 0 1 7 1 3 0

}

}

篮子的个数一定大于元素个数，篮子个数自动2倍增长

# 分配器：

new将内存分配和对象构造组合在了一起；

delete将对象析构和内存释放组合在了一起

std::allocator将内存分配和对象构造分离开来，定义在头文件memory中



还未构造对象的情况下就使用原始内存是错误的

当我们用完对象后，必须对每个构造的元素调用destroy来销毁它们

我们只能对真正构造了的元素进行destroy操作。

一旦元素被销毁后，就可以将这部分内存归还给系统，或者也可以用这部分内存来保存其它对象

标准库还为allocator类定义了两个伴随算法，可以在未初始化内存中创建对象。它们都定义在头文件memory中，如下：



int main()

{

std::allocator<A> alloc;

A\* p = alloc.allocate(2); //分配内存

alloc.construct(p); //构造 cons A()

alloc.construct(p+1, 1, 3.4); //构造 cons A(int, double)

alloc.destroy(p); //析构 decon

alloc.destroy(p+1); //析构 decon

alloc.deallocate(p, 2); //释放内存

}

# std::deque与std::stack、std::queue的关系：

std::deque::push\_front()不一定会重新导致内存分配，如果第一缓冲区尚有备用空间则直接添加，否则，增加一整块新的缓冲区再增加元素

std::deque不会像std::vector一样有数据的搬移

stack和queue都是借助deque实现的，deque覆盖了它们所需的功能

# 初始化列表{}初始化

{}不会对表达式做narrowing conversion收缩转换，如

double x =1, y=2, z = 3;

//int i{ x + y + z }; //报错，因为double转int是收缩转换

如果一个类有以std::initializer\_list为参数的构造函数，那么以{}构造该类对象时，要小心：

当你使用 {} 统一初始化的时候，会强烈优先匹配std::initializer\_list(强烈到即使参数的个数不一致）, 若不满足initializer\_list的类型，编译器会尝试转换出initializer\_list的类型，若是narrow 转化，如从double到int , bool等，转化失败就会报错，不再匹配。若是可以进行向上转化，则使用转化后的结果，即使其余的普通构造函数有更完美的匹配。那么，除非是参数与initializer\_list的类型不能进行隐式的转换，才会考虑普通函数

**class** **Widget**

{

**public:**

Widget(**int** i, **bool** b); *// #1*

Widget(**int** i, **double** d); *// #2*

Widget(std**::**initializer\_list**<long** **double>**); *// #3*

};

若使用Widget w {2, 3.4}， 这里的话2可以向上转化为long double，3.4也可以转化为long double ，于是选择的是3。 那么，若是我们这里变为Widget(std::initializer\_list<int>)，就会出现编译错误，编译器会阻止narrow转化。然而，若你是变为了Widget(std::initializer\_list<std::string>)，编译器发现没有办法隐式转换，于是就会考虑其余的普通构造函数。

# std::tuple

std::tuple中元素的创建区域是跟std::tuple创建的区域相同。如果std::tuple是在栈上创建的，则其中的元素也是在栈中创建的；如果std::tuple是在堆中创建的，则其中的元素也在堆中创建

{

cout << "------------------//(1)" << endl;

std::tuple<int, bool, string, Person> tp =

std::tuple<int, bool, string, Person>(

11,true,"ok",Person("cjz",21));

//Person构造、移动构造

}

{

cout << "---------------------//(2)" << endl;

std::tuple<int, bool, string, Person> tp{

11,true,"ok",Person("cjz",21) };

//Person构造、移动构造

}

{

cout << "----------------------//(4)" << endl;

bool b = true;

std::tuple<int, bool, string, Person> tp =

std::make\_tuple(

11, b, "ok", Person("cjz", 21));

//Person构造、移动构造、移动构造

}

{

cout << "----------------------//(5)" << endl;

int i = 11;

bool b = true;

string ok = "ok";

Person p("cjz", 21);

std::tuple<int&,bool&,string&,Person&> tp=

std::tie(i, b, ok, p);

//创建一个左值引用tuple

//Person构造

std::get<0>(tp)=23; //返回第一个元素的引用

cout << i << endl; //23

}

{

cout << "----------------------//(3)" << endl;

int i = 20;

std::tuple<int&, bool&&, string&&, Person&&> tp =

std::forward\_as\_tuple(

i, true, "ok", Person("cjz", 21));

//创建一个引用tuple,传入左值则左值引用，传入右值则右值引用

//Person构造

}

//遍历tuple

template<class Tuple, int Reverse = std::tuple\_size<Tuple>::value>

struct PrintTuple

{

static void print(const Tuple& tp)

{

constexpr int Index = std::tuple\_size<Tuple>::value

- Reverse;

using type = std::tuple\_element<Index, Tuple>::type;

type cur = std::get<Index>(tp);

cout << cur << " ";

PrintTuple<Tuple, Reverse-1>::print(tp);

}

};

template<class Tuple>

struct PrintTuple<Tuple, 0>

{

static void print(const Tuple& tp)

{

return;

}

};

int main()

{

cout << "----------------------//(3)" << endl;

std::tuple<int, bool, string> tp(

20, true, string("ok"));

//获取tuple元素个数

int len = std::tuple\_size<std::tuple

<int, bool, string>>::value;

int length = std::tuple\_size<decltype(tp)>::value;

//获取tuple元素类型

using type0 = std::tuple\_element<0, decltype(tp)>::type;

using type2 = std::tuple\_element<2, decltype(tp)>::type;

//访问某个元素

cout << "元素：" <<std::get<2>(tp) << endl;

//遍历

PrintTuple<decltype(tp)>::print(tp);

std::tuple<bool, int, double> tu;

cout << std::tuple\_size<decltype(tu)>::value << endl;//3

PrintTuple<decltype(tu)>::print(tu);//0 0 0

}

int a;

bool b;

std::string c;

std::tie(a, b, c) = f();

cout << a << b << c << endl;

# 传递包含任意参数的函数（类似thread构造函数）

**template**<**class** F>

auto my\_invoke(F&& f)

{

**return** std::forward<F>(f)();

}

**template**<**class** F, **class** FirstArg, **class**... RestArgs>

auto my\_invoke(F&& f, FirstArg&& first,RestArgs&&... rest)

{

**return** run<std::is\_member\_pointer<F>::value,F, FirstArg, RestArgs...>()(

        std::forward<F>(f),std::forward<FirstArg>(first),

        std::forward<RestArgs>(rest)...);

}

**template**<**bool** ismem, **class** F, **class** FirstArg, **class**... RestArgs>

**struct** run

{

    run() = **default**;

    auto operator ()(F&& f, FirstArg&& first, RestArgs&&... rest)

    {

**return** std::forward<F>(f)(std::forward<FirstArg>(first),

            std::forward<RestArgs>(rest)...);

    }

};

**template**<**class** F, **class** FirstArg, **class**... RestArgs>

**struct** run<**true**,F, FirstArg,RestArgs...>

{

    run() = **default**;

    auto operator ()(F&& f, FirstArg&& first, RestArgs&&... rest)

    {

**return** (std::forward<FirstArg>(first).\*std::forward<F>(f))(

        std::forward<RestArgs>(rest)...);

    }

};

**int** show(**int** a, **double** b)

{

    cout << **typeid**(a).name() << ":" << a << "  "

        << **typeid**(b).name() << ":" << b << endl;

**return** a;

}

**class** CS

{

**public**:

**double** show(**int** a, **double** b, **char** c)

    {

        cout << **typeid**(a).name() << ":" << a << "  "

            << **typeid**(b).name() << ":" << b << "  "

            << **typeid**(c).name() << ":" << c <<endl;

**return** b;

    }

};

**const** **char**\* gfunc()

{

**return** "hello";

}

**int** main()

{

    cout << my\_invoke(show, 1, 3.0) << endl;

    cout << my\_invoke(&CS::show, CS(), 1, 3.0, 'x') << endl;

    cout << my\_invoke(gfunc) << endl;

}

/\*

int:1  double:3

1

int:1  double:3  char:x

3

hello

请按任意键继续. . .

\*/

# std::tuple实现类型擦除（类似std::thread封装pthread\_create）

参考这篇回答，在其基础上修改使得支持可调用对象是类的成员函数，https://www.zhihu.com/question/455522367/answer/1845921797

*//LiteThread.h*

#include <tuple>

#include <utility>

#include <pthread.h>

class LiteThread

{

private:

    template<bool ismem, typename Callable, typename ... Args>

    struct Tmp

    {

        static void\* start\_rtn(void \* fun\_args\_pack) {

            typedef std::tuple<Callable, Args...> fun\_args\_pack\_t;

            fun\_args\_pack\_t \* fun\_args\_pack\_p = static\_cast<fun\_args\_pack\_t\*>(fun\_args\_pack);

            apply\_nonmem(\*fun\_args\_pack\_p, std::make\_index\_sequence<sizeof...(Args)>());

            delete fun\_args\_pack\_p;

            return NULL;

        }

    };

*//偏特化类模板，当可回调对象为成员函数时*

    template<typename Callable, typename ... Args>

    struct Tmp<true, Callable, Args...>

    {

        static void\* start\_rtn(void \* fun\_args\_pack) {

            typedef std::tuple<Callable, Args...> fun\_args\_pack\_t;

            fun\_args\_pack\_t \* fun\_args\_pack\_p = static\_cast<fun\_args\_pack\_t\*>(fun\_args\_pack);

            apply\_mem(\*fun\_args\_pack\_p, std::make\_index\_sequence<sizeof...(Args) - 1>());

            delete fun\_args\_pack\_p;

            return NULL;

        }

    };

    template <typename Callable, typename ... Args, size\_t ... I>

    static void apply\_nonmem(std::tuple<Callable, Args...> & pack, std::integer\_sequence<size\_t, I...>) {

        std::get<0>(pack)(std::get<I + 1>(pack)...);

    }

    template <typename Callable, typename ... Args, size\_t ... I>

    static void apply\_mem(std::tuple<Callable, Args...> & pack, std::integer\_sequence<size\_t, I...>) {

        (std::get<1>(pack)->\*(std::get<0>(pack)))(std::get<I + 2>(pack)...);

    }

public:

    template <typename Callable, typename ... Args>

    explicit LiteThread(Callable&& fun, Args&& ... args) {

        typedef std::tuple<Callable, Args...> fun\_args\_pack\_t;

        fun\_args\_pack\_t \* fun\_args\_pack = new fun\_args\_pack\_t(

            std::forward<Callable>(fun),

            std::forward<Args>(args)...);

        int err = pthread\_create(&m\_pthr,

            NULL,

            Tmp<std::is\_member\_pointer<Callable>::value, Callable, Args...>::start\_rtn,

            reinterpret\_cast<void\*>(fun\_args\_pack));

        if (err != 0) {

            delete fun\_args\_pack;

            throw - 1;

        }

    }

    void join() { pthread\_join(m\_pthr, nullptr); }

    void detach() { pthread\_detach(m\_pthr); }

private:

    pthread\_t m\_pthr;

};

测试代码：

*//main.cpp*

#include <iostream>

#include <unistd.h>

#include "LiteThread.h"

void dothing(int start, int stop){

    while (start <= stop) {

        std::cout << start << std::endl;

        start++;

        sleep(1);

    }

}

struct Counter {

    void work(int start, int stop){

        dothing(start,stop);

    }

};

int main() {

    Counter cnter;

    LiteThread thr1(&Counter::work, &cnter,1, 10);

    LiteThread thr2(dothing,20, 30);

    dothing(100,103);

    thr1.join();

    thr2.join();

}

*//g++ -o main main.cpp -lpthread -std=c++17*

*//可以看到，无论是普通函数还是类的成员函数，都能成功启动线程*

# C++的可变参数模板

C语言中printf("%s:%d\n","width",1920);

printf函数的参数的数量可变，类型可变

printf及类似函数的第一个参数常为格式化字符串，用于提供了两个关键信息给va\_\*宏：可变参数的个数（百分号的个数）、可变参数的类型（如%s、%d等）

一个非模板函数可以和一个同名的模板函数同时存在，而且这个模板函数还可以被实例化为这个非模板函数。

对于非模板函数和同名模板函数，如果其他条件都相同，在调用时重载解析过程会优先调用非模板函数。

C++可变参数模板示例：

#include <iostream>

void LitePrint(){std::cout<<std::endl;}

template<typename T, typename... Ts>

void LitePrint(T arg0, Ts... argLeft){

    std::cout<<arg0<<", ";

    LitePrint(argLeft...);

}

int main(int argc, char\*\* argv){

    LitePrint(2,3.5,"okay");

}

可变参数语法：

* typename... Ts，这是template parameter pack，表明这里有多种type；
* Ts... argLeft，这是function parameter pack，表明这里有多个参数；
* argLeft...，这是pack expansion，将参数名字展开为逗号分割的参数列表

模板展开过程：

1. main函数里调用了LitePrint(2, 3.5, "okay")，会导致LitePrint函数模板首先展开为：

void LitePrint(int, int, const char\*)

1. 在打印第1个参数2后，LitePrint递归调用自身，传递的参数为argLeft...，该参数会展开为[3.5,"okay"]，LitePrint第2次进行了展开：

void LitePrint(int, const char\*)

1. 在打印第1个参数3.5后，LitePrint递归调用自身，传递的参数为argLeft...，该参数会展开为["okay"]，LitePrint第3次进行了展开：

void LitePrint(const char\*)

1. 在打印第1个参数"okay"后，LitePrint递归调用自身，传递的参数为argLeft...，该参数会展开为[ ]，只能匹配到非模板重载版本：

void LitePrint(){std::cout<<std::endl;}

1. 递归停止

objdump -d main2 | c++filt查看反汇编，可以看到模板实例化结果：

000000000000128e <\_GLOBAL\_\_sub\_I\_\_Z9LitePrintv>

00000000000012a7 <void LitePrint<int, double, char const\*>(int, double, char const\*)>

0000000000012f7 <void LitePrint<double, char const\*>(double, char const\*)>

000000000000133f <void LitePrint<char const\*>(char const\*)>

上面的例子里有个LitePrint的“非模板重载”版本，用于递归能够最终退出。基于这个思路，可以设计出另一种重载版本：

#include <iostream>

template<typename T>

void LitePrint(T arg){std::cout<<arg<<std::endl;}

template<typename T, typename... Ts>

void LitePrint(T arg0, Ts... argLeft){

    std::cout<<arg0<<", ";

    LitePrint(argLeft...);

}

int main(int argc, char\*\* argv){

    LitePrint(2,3.5,"okay");

}

两个函数模板重载：当两个参数模板都适用某种情况时，优先使用没有“template parameter pack”的版本

sizeof...操作符，可以获取可变参数的个数（注意sizeof...的参数只能是parameter pack）

#include <iostream>

template<typename T, typename... Ts>

void LitePrint(T arg0, Ts... argLeft){

    std::cout<<"param num: "<<sizeof...(Ts)<<" | "<<sizeof...(argLeft)<<std::endl;

}

int main(int argc, char\*\* argv){

    LitePrint(2,3.5,"okay");

}

//运行结果：param num: 2 | 2

sizeof...实现递归终止，避免使用重载版本来终止递归：

#include <iostream>

template<typename T, typename... Ts>

void LitePrint(T arg0, Ts... argLeft){

    std::cout<<arg0;

    if (sizeof...(Ts)>0){ *//if constexpr(sizeof... (Ts)>0){*

        std::cout<<", ";

        LitePrint(argLeft...);

    }else{

        std::cout<<std::endl;

    }

}

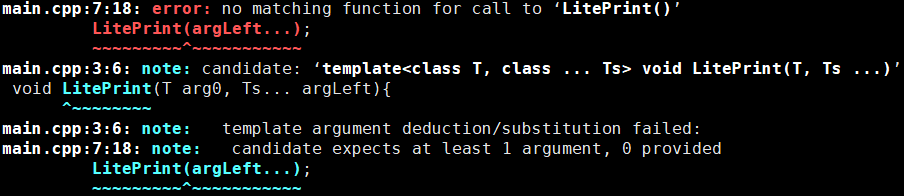
int main(int argc, char\*\* argv){

    LitePrint(2, 3.5, "okay");

}

*//运行结果：2, 3.5, okay*

上述程序编译时会报错：



这是因为if是运行期的，编译时所有分支都会被编译。即当sizeof...(Ts)为0时，下一句模板递归调用也会被编译。

C++17引入了编译期if表达式（if constexpr），完美解决了这个问题。

将上述程序中if (sizeof...(Ts)>0)修改为if constexpr(sizeof... (Ts)>0) 就可以编译通过

Variadic Expressions（可变参数表达式）：

对可变参数进行批量计算，即对parameter pack中的每个参数都独立进行同样的操作。

#include <iostream>

template<typename T, typename... Ts>

void LitePrint(T arg0, Ts... argLeft){

    std::cout<<arg0;

    if constexpr(sizeof...(Ts)>0){ *//if constexpr(sizeof... (Ts)>0){*

        std::cout<<", ";

        LitePrint(argLeft...);

    }else{

        std::cout<<std::endl;

    }

}

template<typename... T>

auto sum(const T&... s){

    LitePrint((s\*s+1)...);

}

int main(int argc, char\*\* argv){

    sum(2,2.5,3.0);

}

*//运行结果5, 7.25, 10*

批量处理可以扩展为一般函数process，如下：

template<typename T>

T process(const T& arg){

    std::cout<<"in="<<arg<<std::endl;

    return arg\*arg\*arg+arg\*arg+1;

}

template<typename... T>

auto sum(const T&... s){

    LitePrint((process(s))...);

}

*/\*运行结果：*

in=3

in=2.5

in=2

13, 22.875, 37

\*/

Variadic Indices（可变参数索引）：

索引操作也可以和可变参数语法结合起来：

#include <iostream>

#include <vector>

template<typename C, typename... Idx>

auto testVariadicIndices(const C& c, Idx... idx){

    LitePrint(c[idx]...);

}

int main(int argc, char\*\* argv)

{

    std::vector<std::string> vec{"today","is","a","Wed", "Apr 17"};

    testVariadicIndices(vec,0,3,4);

}

*//运行结果：today, Wed, Apr 17*

借助可变参数表达式也能实现可变参数索引的效果：

template<typename C, typename T>

auto process(const C& c, T idx){

    return c[idx];

}

template<typename C, typename... Idx>

auto testVariadicIndices(const C& c, Idx... idx){

    LitePrint((process(c,idx))...);

}

fold表达式（C++17）：

binary operator（双目操作符）和parameter pack结合起来后，可以自动循环执行计算

#include <iostream>

template<typename... T>

auto sum(T... s){

    return (... + s);

}

int main(int argc, char\*\* argv){

    std::cout<<sum(2, 0.5f, 4.8)<<std::endl;

}

//运行结果：7.3

上述示例中... + s即相当于((2 + 0.5) + 4.8)。

objdump -d main2 | c++filt查看，会发现只实例化出了一个模板函数：

0000000000400830 <auto sum<int, float, double>(int, float, double)>

fold表达式有四种形式：

* + (... op pack)，相当于((( pack1 op pack2 ) op pack3 ) ... op packN)
  + (pack op ... )，相当于(pack1 op ( ... ( packN-1 op packN)))
  + (init op ... op pack)，相当于((( init op pack1) op pack2) ... op packN)
  + (pack op ... op init),相当于(pack1 op ( ... (packN op init)))

这里的op几乎可以是任意的binary operator，包括加减乘除、指针操作以及<<等

folder表达式打印可变参数，避免使用递归：

*//简单、优雅！的方案*

#include <iostream>

template<typename... Ts>

void LitePrint(Ts... args){

    (std::cout<<...<<args)<<std::endl;

}

int main(int argc, char\*\* argv){

    LitePrint(2,3.5,"okay");

}

*//g++ -o main main.cpp -std=c++17*

*//运行结果：23.5okay*

# C++模板

模板报错信息冗长，如下示例：

#include <string>

#include <iostream>

struct A {};

int main(){

    std::cout << A{} << std::endl;

}

上述代码在gcc11.2.0编译器上，产生了数百行报错信息。重点信息被gcc标记了出来，如下所示：

no match **for** '**operator<<**' (operand types are 'std**::**ostream' {aka 'std**::**basic\_ostream**<char>**'} and 'A')

9 **|** std**::**cout **<<** A{} **<<** std**::**endl;

**|** **~~~~~~~~~** **^~** **~~~**

**|** **|** **|**

**|** **|** A

**|** std**::**ostream {aka std**::**basic\_ostream**<char>**}

意思是没有找到匹配的重载函数，也就是说我们需要为A重载operator<<。

那么剩下数百行报错在干嘛呢？其实关键就在于重载函数和隐式类型转换。观察其中一段信息如下：

note: **template** argument deduction**/**substitution failed:

note: cannot convert 'A()' (type 'A') to type '**const** **char\***'

9 **|** std**::**cout **<<** A{} **<<** std**::**endl;

意思是尝试用A类型匹配const char\*这个重载（通过隐式类型转换），结果失败了。标准库类实现了很多的重载函数，如operator<<就重载了int、bool、long、double 等，将近几十个函数。报错信息会把所有重载函数尝试失败的原因都列出来，于是轻松产生巨额日志信息。

模板参数推导失败和实例化失败

#include <string>

#include <iostream>

struct A {};

struct B {};

template<typename T>

void test(T a, T b){

    std::cout << a << b << std::endl;

}

int main(){

    test(A{}, B{}); *// 模板参数推导失败，短短几行报错*

    test(A{}, A{}); *// 模板参数推导成功，进入实例化阶段，但实例化失败，几百行报错*

}

如下程序使用requires语法把错误限制在类型推导阶段，而不去进行实例化，从而使得报错信息只有短短几行：

#include <string>

#include <iostream>

struct A {};

template<typename T>

requires requires(T x) { std::cout << x; } *// C++20加入的requires语法，意思就是要求std::cout << x`是合法的。*

void print(T x){

    std::cout << x << std::endl;

}

int main(){

    print(A{});

}

C++11前模板用于编译期计算：

*//计算阶乘*

#include <iostream>

#include <cassert>

template<int N>

struct Factorial{

    enum { value = N \* Factorial<N - 1>::value };

};

template<>

struct Factorial<0>{

    enum { value = 1 };

};

int main(){

    static\_assert(Factorial<5>::value == 120);

}

C++11后采用constexpr进行编译期计算：

#include <iostream>

#include <cassert>

constexpr int factorial(int n){

    if (n == 0){ *//不能是if constexpr，因为if constexpr要求其条件是一个编译时常量表达式，这里n并不是*

        return 1;

    }else{

        return n \* factorial(n - 1);

    }

}

int main(){

    static\_assert(factorial(5) == 120); *//这里factorial(5)可以在编译期计算出结果*

}

一个函数前面直接加上constexpr关键字修饰，表示这个函数既可以在运行期调用，也可以在编译期调用

在C++20之后几乎所有的标准库函数都是constexpr的了，比如编译期排序：

#include <cassert>

#include <array>

#include <algorithm>

constexpr auto sort(auto&& range){

    std::sort(std::begin(range), std::end(range));

    return range;

}

int main(){

    constexpr auto arr = sort(std::array{ 1,5,4,2,3 });

    static\_assert(arr[1] == 2);

*//下述写法无法编译通过，因为constexpr变量是只读的，不能被修改*

*//constexpr auto arr = std::array{ 1,3,4,2,3 };*

*//std::sort(std::begin(arr), std::end(arr));*

}

使用模板元编程是对类型做计算的唯一方法。C++中类型并非一等公民，只能作为模板参数参与计算。而且把类型当作值这样的改动对C++而言是完全不可接受unacceptable的。如下代码用来判断两个type\_list满不满足子序列关系：

#include <cassert>

#include <array>

#include <algorithm>

template<typename ...Ts>

struct type\_list {};

template<typename SubFirst, typename ...SubRest, typename SuperFirst, typename ...SuperRest>

constexpr auto is\_subsequence\_of\_impl(type\_list<SubFirst, SubRest...>, type\_list<SuperFirst, SuperRest...>) {

    if constexpr (std::is\_same\_v<SubFirst, SuperFirst>)

        if constexpr (sizeof...(SubRest) == 0)

            return true;

        else

            return is\_subsequence\_of(type\_list<SubRest...>{}, type\_list<SuperRest...>{});

    else

        if constexpr (sizeof...(SuperRest) == 0)

            return false;

        else

            return is\_subsequence\_of(type\_list<SubFirst, SubRest...>{}, type\_list<SuperRest...>{});

}

template<typename ...Sub, typename ...Super>

constexpr auto is\_subsequence\_of(type\_list<Sub...>, type\_list<Super...>)

{

    if constexpr (sizeof...(Sub) == 0)

        return true;

    else if constexpr (sizeof...(Super) == 0)

        return false;

    else

        return is\_subsequence\_of\_impl(type\_list<Sub...>{}, type\_list<Super...>{});

}

int main() {

    static\_assert(is\_subsequence\_of(type\_list<int, double>{}, type\_list<int, double, float>{}));

    static\_assert(!is\_subsequence\_of(type\_list<int, double>{}, type\_list<double, long, char, double>{}));

    static\_assert(is\_subsequence\_of(type\_list<>{}, type\_list<>{}));

    static\_assert(is\_subsequence\_of(type\_list<int, double>{}, type\_list<int, char, double, float>{}));

}

# C++模板编译期计算

C++标准通过不断放宽constexpr的限制，使得越来越多的函数可以在编译期执行。但具有外部链接（也就是被extern的函数）无论如何都无法在编译期执行的。如绝大部分从C继承过来的函数，例如memcpy、memmove等。

假设我写了一个constexpr的memcpy，虽然能在编译期用了，但是运行期效率肯定不如标准库的实现。如果能在编译期使用我的的实现，运行期使用外部链接的标准库函数就好了。

std::is\_constant\_evaluated()：判断当前函数是否在编译期执行

#include <iostream>

#include <cassert>

#include <cstring>

#include <array>

#include <type\_traits>

template <typename T>

constexpr void\* memcpy\_const(T\* dest, const T\* src, std::size\_t count) {

    for(std::size\_t i = 0; i < count; ++i) {

        dest[i] = src[i];

    }

    return dest;

}

template <typename T, std::size\_t N>

constexpr std::array<T, N> memcpy\_wrap(const std::array<T,N>& dest, const std::array<T,N>& src){

    std::array<T, N> result = dest;

    if(std::is\_constant\_evaluated()) {

        memcpy\_const(result.data(), src.data(), N);

    } else {

        ::memcpy(result.data(), src.data(), sizeof(T)\*N);

    }

    return result;

}

int main(){

    constexpr std::array<int,3> c\_src = {1, 3, 5};

    constexpr auto c\_dst = memcpy\_wrap(std::array{4,5,6}, c\_src);

    static\_assert(c\_dst[0]==1);

    std::array<int,3> v\_src = {2, 4, 6};

    std::array<int,3> v\_dst = memcpy\_wrap(std::array{4,5,6}, v\_src);

    assert(v\_dst[0]==2);

}

唯一的问题是，假设上面的memcpy\_wrap仅在运行期运行，但第一个分支仍然会被编译。虽然编译器最终应该会把if(false)这个分支优化掉，但是这个分支里面仍然会进行语法检查之类的工作，如果里面用到了模板，那么模板实例化仍然会被触发（甚至产生预料外的实例化导致编译错误），显然这不是我们想要的结果。

# C++模板类偏特化之间互相继承

题主正在写一个泛型容器，并且根据容器中存的类型的size不同采取不同的实现策略

template<typename T, bool = (sizeof(T) <= 8)>

class Container{

*// ... some functions*

};

template<typename T>

class Container<T, true>{

*// ... some functions*

};

由于实现策略不同，这两个不同的特化之间并没有太多可以共用的函数，所以就正常写就行了，没什么问题。但是现在题主想进一步特化T为char的情况，额外提供一些函数作为接口。如果直接写一个全特化

template<>

class Container<char, true>{

    // ... some functions

};

就得把先前那个特化实现中的函数都重新写一遍（也就是sizeof(T) <= 8）的情况，的确会有很多冗余代码。另一个思路呢，是直接新取一个名字

class CharContainer : Container<char, true>{

// ... some functions

};

但是呢，题主不想额外取一个名字了，就想在原来的类中进行修改。

那有没有办法做到呢？既能为一个新的特化提供特殊实现，还可以复用先前的特化代码？答案其实还是继承。不同的[模板特化](https://www.zhihu.com/search?q=%E6%A8%A1%E6%9D%BF%E7%89%B9%E5%8C%96&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A3416322222%7D)之间也是可以互相继承的，只要把原来的代码稍作修改

template<typename T, bool = (sizeof(T) <= 8), bool is\_char = std::is\_same\_v<T, char>>

struct Container{

    void f1() {}

};

template<typename T>

struct Container<T, true, false>{

    void f2() {}

};

template<>

struct Container<char, true, true> : public Container<char, true, false>{

    void f3() {}

};

int main(){

    Container<char> c;

    c.f2();

    c.f3();

    return 0;

}

这样就实现了题主希望的效果，只是额外加了一个bool模板参数用来区分T是不是char，现在，c既可以调用f2()也可以调用f3()。

补充：如果情况更加复杂的话，我们可以选择手动控制[偏序关系](https://www.zhihu.com/search?q=%E5%81%8F%E5%BA%8F%E5%85%B3%E7%B3%BB&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A3416322222%7D)，然后分发到枚举常量，如下：

enum class container\_tag{

    char\_,

    small,

    large

};

template <typename T>

constexpr container\_tag get\_container\_tag(){

    if (std::is\_same\_v<T, char>) {

        return container\_tag::char\_;

    }else if (sizeof(T) <= 8) {

        return container\_tag::small;

    }else {

        return container\_tag::large;

    }

}

template <typename T, container\_tag tag = get\_container\_tag<T>()>

struct Container;

template <typename T>

struct Container<T, container\_tag::large>{

    void f1() {}

};

template <typename T>

struct Container<T, container\_tag::small>{

    void f2() {}

};

template <typename T>

struct Container<T, container\_tag::char\_> : Container<T, container\_tag::small>{

    void f3() {}

};

# vs2015虚函数、虚继承的内存布局

class X {}; //sizeof(X)为1

class Y :public X {}; //sizeof(Y)为1

//虚函数表指针在对象首端

class X2

{

private: int a=10;

public:virtual void show() { cout << "X2::show()" << endl; }

};

cout << sizeof(X2) << endl; //8

X2 obj;

cout << \*(((int\*)&obj) + 0) << endl; //13565424

cout << \*(((int\*)&obj) + 1) << endl; //10

cout << "-----------------" << endl;

class Y3 :public X2{

private:int b = 20;

};

Y3 y3;

cout << sizeof(Y3) << endl; //12

cout << \*(((int\*)&y3) + 0) << endl; //13565424

cout << \*(((int\*)&y3) + 1) << endl; //10

cout << \*(((int\*)&y3) + 2) << endl; //20

cout << "-----------------" << endl;

class Y2 :virtual public X2{

private:int b = 20;

};

cout << sizeof(Y2) << endl; //16

Y2 y2;

cout << (int)(((int\*)&y2) + 0) << ": " << \*(((int\*)&y2) + 0) << endl; //13565484

cout << (int)(((int\*)&y2) + 1) << ": " << \*(((int\*)&y2) + 1) << endl; //20

cout << (int)(((int\*)&y2) + 2) << ": " << \*(((int\*)&y2) + 2) << endl; //13565484

cout << (int)(((int\*)&y2) + 3) << ": " << \*(((int\*)&y2) + 3) << endl; //10

# 向函数传递任意参数

欲设计一个转发函数，可以转发任意类型函数，可以设计接口为

void design(void\*(\*func)(void\*), void\* params)，即将接受的函数类型固定为void\*(\*)(void\*)，参数为params。使用时合理转化待转发函数为void\*(\*)(void\*)类型，再作为参数传递给design函数

void design(void\*(\*func)(void\*), void\* params)

{

//……

func(params);

//……

}

1、欲转发的函数仅一个double参数时：

double x = 3.0;

design(f, &x);

将函数设计为：

void\* f(void\* ptr)

{

cout << "double is: " << \*((double\*)(ptr)) << endl;

return nullptr;

}

2、欲转发的函数有多个参数时，如char、int两个参数，将参数封装成结构体

struct Params

{

char ch;

int i;

};

Params obj{ 'c',100 };

design(g, &obj);

将函数设计为：

void\* g(void\* ptr) {

Params\* par = (Params\*)ptr;

char ch = par->ch;

int i= par->i;

cout << "ch is:" << ch << ", i is:" << i << endl;

return nullptr;

}

3、欲转发的是一个成员函数时

class A;

struct Params

{

A\* obj;

char ch;

int i;

};

class A

{

public:

void\* f(char ch,int i) //要转发的函数

{

cout << "ch is:" << ch << ", i is:" << i << endl;

return nullptr;

}

static void\* medium(void\* ptr) //借助该函数实现对成员函数A::f的转发

{

Params\* p = (Params\*)ptr;

A\* obj = p->obj;

char ch = p->ch;

int i = p->i;

return obj->f(ch, i);

}

};

int main()

{

A a;

Params par{ &a,'c',99 };

design(&A::medium,&par);

}

# POD类型：

Plain Old Data

POD 类型是以下类型的统称：

* 标量类型（scalar type），即内置类型（包括std::nullptr\_t类型）和其cv限定版本
* POD 类（class/struct 和 union）
* 以上类型的数组和 cv 限定版本

POD=平凡的（trivial）+标准布局（standard layout）

std::is\_pod<>::value

但POD在C++20里已经被宣布弃用（deprecated），应该使用分解得更细的两个traits：

std::is\_trivial<>::value：平凡的

std::is\_standard\_layout<>::value：标准布局

平凡的：默认构造函数、析构函数、拷贝构造、移动构造、拷贝赋值、移动赋值是trivial

is\_trivial 可以拆分为：

1. is\_trivially\_default\_constructible：默认构造函数不需要实际执行任何动作
2. is\_trivially\_copyable：复制时可以简单复制内存块，无需特殊处理

is\_trivially\_copyable 还可以进一步拆分：

1. is\_trivially\_copy\_constructible：拷贝构造可以简单内存复制
2. is\_trivially\_copy\_assignable：拷贝赋值可以简单内存复制
3. is\_trivially\_move\_constructible：移动构造可以简单内存复制
4. is\_trivially\_move\_assignable：移动赋值可以简单内存复制
5. is\_trivially\_destructible：析构函数不需要实际执行任何动作

平凡的构造函数：

* 父类与member class object的构造函数是平凡的
* 无显示定义的任何构造函数或者出现了A()=default

class B{int x;B() = default;B(int x\_):x(x\_) {}}; B是trivial类型

* 没有虚函数
* 没有虚继承

平凡的拷贝构造函数：

* 父类与member class object的拷贝构造函数是平凡的
* 无显示定义的拷贝构造函数或者A(const A&)=default
* 没有虚函数
* 没有虚继承

标准布局：

* 所有非静态成员有相同的访问权限
* 当前类与父类们（单继承或多继承）只能其一有非静态数据成员
* 类中第一个非静态数据成员的类型不是父类类型
* 没有虚函数与虚基类

标准布局即对象内存标准：

* C++标准，同一访问权限下的数据按其声明顺序在内存排序，但不规定不同访问权限数据之间的顺序。

class B{private:int a; private:int c;}; B是标准布局

class A{private:int a; public:int c;}; A非标准布局

* C++标准不规定derived class members 和 base class(es)members 的排列顺序；多继承，C++ Standard 并未要求多个父类在内存有特定的排列顺序
* struct A : B { B b; };这里A非标准布局，因为第一个非静态成员变量b的类型跟A所继承的类型相同。而形如：struct C : B { int a; B b; }；则是一个标准布局的类型。

C++中允许优化不包含成员的基类。

class X {}; //sizeof(X)为1

class Y :public X {}; //sizeof(Y)为1

class Z :public X { int a; };//4

class Q :public X { X a; }; //1

如果子类的第一个成员仍然是基类，编译器仍然会为基类分配1字节的空间。分配为1字节空间是由于C++标准要求类型相同的对象必须地址不同

trivial是可以安全使用 memcpy 的充分非必要条件（POD更是可以安全使用 memcpy 的充分非必要条件）

其实只要类型是TriviallyCopyable的，那就能安全地使用memcpy去拷贝它。而POD是相比TriviallyCopyable更加严格的限制，没有必要。

trivial类型的要求中去掉对默认构造函数的约束即TriviallyCopyable

# 双击exe或者命令行exe回车：

Windows下双击exe或者命令行exe回车，exe就能运行。实质为explorer.exe调用CreateProcess创建进程，进程加载exe文件到进程内存，创建主线程执行代码。

# 多重继承、虚继承的内存布局：

多重继承：

class Point2d {

public:

//拥有 virtual 接口。所以 Point2d 对象之中会有 vptr

protected:

float \_x, \_y;

};

class Point3d : public Point2d {

protected:

float \_z;

};

class Vertex {

public:

//拥有 virtual 接口。所以 Vertex 对象之中会有 vptr)

protected:

Vertex \*next;

};

class Vertex3d : public Point3d, public Vertex {

protected:

float mumble;

};



虚继承：

class Point2d {

public:

//拥有 virtual 接口。所以 Point2d 对象之中会有 vptr

protected:

float \_x, \_y;

};

class Point3d : public Point2d {

protected:

float \_z;

};

class Vertex : public virtual Point2d {

protected:

Vertex \*next;

};

class Vertex3d : public Vertex, public Point3d{

protected:

float mumble;

};



如何能够存取 class 的共享部分（即虚基类subobject）呢

(1)如果我有三层虚拟派生，取得所有的 nested virtual base class 指针，放到 derived

class object 之中。仅一个虚基类时如下图：



(2)虚继承的子类有虚基类表，表中含所有指向虚基类的指针。对象有一个指针，指向该虚基类表

(3)在virtual fimction table 中放置virtual base class的offset(不是地址）



# std::string::copy()

size\_type string::copy( CharT\* dest, size\_type count, size\_type pos = 0) const;

将子串[pos, pos+count)拷贝到dest所指缓冲区。不会往dest字符传后填充null

如果count过大，导致要拷贝的子串超出了原string末尾，则只拷贝[pos,size())

如果pos=size()，则不拷贝不拷贝任何内容

如果pos>size，则抛出std::out\_of\_range异常

#define \_SCL\_SECURE\_NO\_WARNINGS

int main()

{

string addr = "lolwow";

constexpr unsigned int size = 1024;

char buf[size];

memset(buf,0,size);

addr.copy(buf, 5, 1); cout << buf << endl; //olwow

memset(buf, 0, size);

addr.copy(buf, 8); cout << buf << endl; //lolwow

memset(buf, 0, size);

addr.copy(buf, 2,6); //抛出std::out\_of\_range异常，因为addr.size()<最后一个参数

}

# cbegin()

iterator,const\_iterator作用：遍历容器内的元素，并访问这些元素的值。iterator可以改元素值,但const\_iterator不可改

auto i1 = Container.begin(); // i1 is Container<T>::iterator

auto i2 = Container.cbegin(); // i2 is Container<T>::const\_iterator

# std::unordered\_set：

定义：

template<

    class Key,  
    class Hash = [std::hash](http://en.cppreference.com/w/cpp/utility/hash)<Key>,  
    class KeyEqual = [std::equal\_to](http://en.cppreference.com/w/cpp/utility/functional/equal_to)<Key>,  
    class Allocator = [std::allocator](http://en.cppreference.com/w/cpp/memory/allocator)<Key>

> class unordered\_set;

C++11引入的关联容器，不对元素排序，但元素被组织成buckets

根据Hash计算值决定位于哪个bucket

查找、插入、移除具有平均O(1)复杂度

unordered\_set解决冲突采用链地址法，把具有同一hash值的数据组成一个链表

成员函数insert：

[std::pair](http://en.cppreference.com/w/cpp/utility/pair)<iterator,bool> insert( const value\_type& value );

Returns a pair consisting of an iterator to the inserted element (or to the element that prevented the insertion) and a bool denoting whether the insertion took place（发生）.

std::Hash

template<class Key >struct hash; 仿函数类，重载了()，返回size\_t，即

size\_t operator()(const \_Kty& \_Keyval) const

(1)若两个参数k1、k2相等，则std::hash<Key>()(k1)== std::hash<Key>()(k2).

(2)若两个不同的参数k1、k2不相等，则std::hash<Key>()(k1)== std::hash<Key>()(k2)成立的概率应非常小，接近1.0/std::numeric\_limits<size\_t>::max().

用途：

如判断vector中是否有重复元素

bool contain\_duplicates(const vector<int>& vec) {

std::unordered\_set<int> unset;

for (auto i : vec){

if (!unset.insert(i).second) return true;

}

return false;

}

# 使用宏实现enum和string互转：

#include <stdio.h>

#include <string.h>

#define MAKE\_ITEM(GEN) \

    GEN(RED) \

    GEN(BLUE) \

    GEN(GREEN)

enum Color {

    #define FOR\_ENUM(x) x,

    MAKE\_ITEM(FOR\_ENUM)

};

const char\* color\_to\_string(Color value)

{

    switch (value)

    {

        #define FOR\_NAME(x) case x: return #x;

        MAKE\_ITEM(FOR\_NAME)

    }

    return "";

}

Color string\_to\_color(const char\* value)

{

    #define FOR\_CMP(x) if (0 == strcmp(#x, value)) return x;

    MAKE\_ITEM(FOR\_CMP)

    return Color();

}

int main()

{

    printf("color[%d]=%s\n", RED, color\_to\_string(RED));

    printf("color[%d]=%s\n", BLUE, color\_to\_string(BLUE));

    printf("color[%d]=%s\n", GREEN, color\_to\_string(GREEN));

    printf("%s:%d\n", "RED", string\_to\_color("RED"));

    printf("%s:%d\n", "BLUE", string\_to\_color("BLUE"));

    printf("%s:%d\n", "GREEN", string\_to\_color("GREEN"));

    printf("%s:%d\n", "", string\_to\_color("")); *//0*

    return 0;

}

# enum class：

enum class，等价于enum struct，在C++11中引入，有enum相比：

* 作用域被限制在大括号内
* 与整形不能隐式转换，只能显示转换
* 默认的底层数据类型是int，用户可以通过（冒号+类型）的格式来指定任何整形（除了wchar\_t）作为底层数据类型

**enum** WindowModality {

    NonModal=2,

    WindowModal,

    ApplicationModal

};

**enum** DialogModality

{

    //NonModal,  //重定义错误，因为enum枚举成员的作用域在大括号之外

    appmodal

};

**enum** **class** Colorx { red, green, yellow };

**enum** **class** Colory { red, green = 100, yellow };

//默认的底层数据类型是int，用户可以通过：type（冒号+类型）来指定任何整形（除了wchar\_t）作为底层数据类型

**enum** **class** Year :int64\_t {a,b,c,d};

WindowModality modal = (WindowModality)3; //整型不能隐式转为enmu类型，必须显式转换；但enmu类型可以隐式转为整型

cout << WindowModality::NonModal << endl; //2

cout << WindowModal << endl;  //3, enum作用域在大括号之外，因此可以直接访问，不需要域运算符。

Colorx colx = (Colorx)2; //无论整型转enum class，还是enum class转整型，都不能隐式转换，必须显式

**int** val = (**int**)Colorx::green;  //enum class作用域限制在大括号内，必须使用域运算符访问。

enum class无法与int隐式互转换，但可以直接比较两个enum class变量的大小，因此可以作为set或map的key

//模板别名std::underlying\_type\_t<E>返回枚举类型E的底层类型

template <typename E>

using std::underlying\_type\_t<E>=typename std::underlying\_type<E>::type

enum class底层类型默认int，可以指定

enum class Color

{

red,green,blue

};

cout << typeid(std::underlying\_type\_t<Color>).name() << endl; //int

enum class Grade:std::int64\_t

{

good, medium, bad

};

cout << typeid(std::underlying\_type\_t<Grade>).name() << endl; //\_\_int64

# std::wstring、std::codecvt\_utf8：

Windows平台的std::wstring就是std::u16string, wchar\_t 就是char16\_t (utf-16编码)。window平台的终端编码一般是gbk

#include<string>   // std::wstring

#include <locale>         // std::locale, std::codecvt, std::use\_facet

#include <cwchar>         // std::mbstate\_t

#include<codecvt>

**int** main()

{

    std::string st = "殿军";

**const** **char** \* stch = st.c\_str();// 内存中为GB2312编码：b5ee befc，殿、军gb2312编码分别为0xb5ee、0xbefc

    std::wstring str = L"殿军";

**const** **wchar\_t**\* strch = str.c\_str();  //内存中为Unicode码点：bf6b 9b51，殿、军unicode码点分别为0x6bbf、519b

    std::wstring\_convert<std::codecvt\_utf8<**wchar\_t**> > conv;  //用于wstring（unicode码点）与utf-8之间的转换

    std::string utf8 = conv.to\_bytes(str);

**const** **char**\* utf8ch=utf8.c\_str();  //内存中为Unicode码点：e6aebf e5869b，殿、军utf-8编码分别为0xe6aebf、 e5869b

    std::wstring widestr = conv.from\_bytes(utf8);

**const** **wchar\_t**\* widestrch = widestr.c\_str();  //内存中为Unicode码点：bf6b 9b51

}

# nullptr不能转std::string

const char\* duration = nullptr;

//std::string stdstr = duration; 崩溃

//std::string stdstr(duration); 崩溃

\*p++=10; //\*p=10，然后p++

int tag=!!p; //将ppu规范化为0和1

cout输出16进制：

unsigned int p = 0xFF112233;

cout << p << endl; //4279312947

cout << std::hex << p << endl; //ff112233，以16进制输出，且之后都是16进制

cout << p << endl; //ff112233

cout << std::dec; //必须调用该语句才能恢复为10进制

cout << p << endl; //4279312947

# 文件间的编译依存关系：

类的声明式形如：**class** A;

类的定义式形如：

**class** A

{

**private**:**int** age\_;

**public**: **void** set::age();

};

#include一个类的头文件，如果头文件中是类的定义式，则A中私有数据成员改变，当前文件需要重新编译

#include一个类的头文件，如果头文件中是类的声明式，则A中私有数据成员改变，当前文件不需要重新编译。

定义一个class类型的变量时，需要该class的定义

定义一个指向class的指针或引用时，不需要class的定义，只要有class的声明式即可

当你**声明**一个函数，其参数或返回值类型为某个class并不需要该class的定义，只需要声明式即可；纵使函数以by value方式传递该类型的参数或返回值。如：

**class** Date; //class 声明式。

Date today( ); //没问题，这里并不需要class的定义式

**void** clearAppointments(Date d); //没问题，这里并不需要class的定义式

当前类形如：则包含类Date的声明式即可

**class** Date;

**class** Person

{

**public**:

    Person( **const** Date birthday);

**private**:

    Date\* theBirthDate;  //指针的大小编译器已知

};

当前类形如：则必须包含类Date的定义式

#include”data.h” //data.h中有data的定义式

**class** Person

{

**public**:

    Person( **const** Date birthday);

**private**:

    Date theBirthDate;   //编译器必须在编译器期间知道每个对象的大小，需要定义式

};

降低文件间编译依赖性的方法有两种：

1. pimpl，即pointer to implementation
2. 接口类

要发布PlayControl类

* 普通做法，文件编译依赖性高：

//widget.h

class Widget

{

private: int wight;int speed;

public:Widget(int w);void work();

};

//widget.cpp

#include<iostream>

#include"widget.h"

Widget::Widget(int w) :wight(w), speed(0) {}

void Widget::work() { std::cout << "Widget work" << std::endl; }

//playcontrol.h

#include "widget.h"

class PlayControl {

public:PlayControl(int w);void play();

private:Widget m\_wgt;

};

//playcontrol.cpp

#include"playcontrol.h"

PlayControl::PlayControl(int w) :m\_wgt(w) {}

void PlayControl::play() { m\_wgt.work(); }

这种方法，用户在使用PlayControl库时，除了playcontrol.h头文件还必须给其提供widget.h头文件，暴露了PlayControl库内部实现的一些信息，此外，一旦库发生改变如widget.h改变，用户代码也必须重新编译

* pimpl方法：

widget.h和widget.cpp文件不变

//"playcontrol.h

#include<memory>

class Imp;

class PlayControl

{

public:

PlayControl(int name);

void play();

private:

std::shared\_ptr<Imp> m\_imp;

};

//"playcontrol.cpp

#include"playcontrol.h"

#include"widget.h"

class Imp {

public:

Imp(int w) :m\_wgt(w){}

void play(){m\_wgt.work();}

private:

Widget m\_wgt;

};

PlayControl::PlayControl(int name){m\_imp = std::make\_shared<Imp>(name);}

void PlayControl::play(){m\_imp->play();}

* 借助接口类：

接口类析构函数为虚函数、其它函数都为纯虚函数、无数据成员

widget.h和widget.cpp文件不变

//playcontrol.h

#include<memory>

class PlayControl {

public:

virtual void play() = 0;

virtual ~PlayControl() {}

static std::shared\_ptr<PlayControl> createPlayControl(int name);

};

//"playcontrol.cpp

#include"playcontrol.h"

#include"widget.h"

class RealCtrl:public PlayControl

{

public:

RealCtrl(int w):m\_wgt(w){}

void play(){m\_wgt.work();}

~RealCtrl() {}

private:

Widget m\_wgt;

};

std::shared\_ptr<PlayControl> PlayControl::createPlayControl(int name)

{

return std::shared\_ptr<PlayControl>(new RealCtrl(name));

}

//main.cpp

#include"playcontrol.h"

int main()

{

std::shared\_ptr<PlayControl> playctl= PlayControl::createPlayControl(3);

playctl->play();

return 0;

}

# VS多线程调试：



调试-窗口-线程

调试-窗口-并行堆栈，可以直观地看每个线程的调用堆栈

线程窗口每列的含义：

* “优先级别”列，包含系统已向每个线程分配的优先级。
* “关联掩码”列，高级列，通常隐藏。 此列显示每个线程的处理器关联掩码。 在多处理器系统中，关联掩码确定线程可以在哪些处理器上运行。
* “已挂起”列，包含挂起项计数。 此计数确定线程是否可以运行

调试方法：

* 如果需要对某一个thread挂起，右键单击对应的thread，选择【冻结】即可；
* 如果需要对某一个thread重新调度，右键单击对应的thread，选择【恢复】即可；
* 如果需要查看特定thread的堆栈，双击或者右键->切换到线程，在调用堆栈窗口中查看；

或者在“位置”列中，单击线程位置旁边的倒三角形，此位置将展开以显示线程的调用堆栈

* 如果某个线程被挂住，那么此时所有的线程都挂住了，如果你step运行，所有的threads都会参与运行；
* 如果需要对某一个thread进行调试，那么需要对其他的thread进行【冻结】处理

**pthread\_mutex\_t互斥量：**

互斥变量使用特定的数据类型：pthread\_mutex\_t，使用互斥量前要先初始化，使用的函数如下：

#include <pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr);

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

简单的使用可以使用默认的属性初始化互斥量，函数的后一个参数设置为NULL即可。

pthread\_mutex\_t对互斥量加锁解锁的函数如下：

#include <pthread.h>

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthreadd\_mutex\_t \*mutex);

**pthread的条件变量：**

#include <pthread.h>

int pthread\_cond\_init(pthread\_cond\_t \*restrict cond,const pthread\_condattr\_t \*restrict attr);

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

pthread\_cond\_init(&cond, NULL);

pthread\_cond\_destroy(&cond);

pthread\_cond\_signal(&cond); //唤醒

pthread\_cond\_wait(&cond, & mutex); //等待

（1）int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex)

相当于C++11的std::condition\_variable:: wait (unique\_lock<mutex>& lck)

（2）int pthread\_cond\_signal(pthread\_cond\_t \* cond)

相当于C++11的std::condition\_variable:: notify\_one()，唤醒一个线程

（3）int pthread\_cond\_broadcast (pthread\_cond\_t \* cond)

相当于C++11的std::condition\_variable::notify\_all()，唤醒所有线程

（4）pthread中没有与C++11的std::condition\_variable::wait(unique\_lock<mutex>& lck, Predicate pred)类似的API，但可以实现为

while(!pred)

{

pthread\_cond\_wait(&cond, &mutex);

}

**pthread的信号量：**

#include <semaphore.h>

(1)初始化信号量：

int sem\_init(sem\_t \*sem, int pshared, unsigned int val);

该函数第一个参数为信号量指针，第二个参数为信号量类型（一般设置为0），第三个为信号量初始值。第二个参数pshared为0时，该进程内所有线程可用，不为0时不同进程间可用。

(2)销毁信号量：

int sem\_destory(sem\_t \*sem);

(3) 信号量减1：

int sem\_wait(sem\_t \*sem);

申请一个信号量，当前无可用信号量则等待，有可用信号量时占用一个信号量，对信号量的值减1。

(4) 信号量加1：

int sem\_post(sem\_t \*sem);

该函数释放一个信号量，信号量的值加1。

(5) sem\_trywait(&m\_semEnd)

如果信号量的值<=0，调用将立即返回错误(errno设置为EAGAIN)而不是阻塞

**pthread\_setname\_np:**

main中创建子线程thr，pthread\_setname\_np(thr.native\_handle(), "aa");

在thr的入口函数中创建子线程，不调用pthread\_setname\_np命名

编译为可执行程序main

./main执行，top -Hp显示有三个线程，线程名为main、aa、main

gdb ./main执行，top -Hp显示有三个线程，线程名为main、aa、aa

**XML**

XML，可扩展标记语言，E**X**tensible **M**arkup **L**anguage

在大多数web应用程序中，XML用于传输数据，而HTML用于显示数据。

等价于<?xml version="1.0"?><root><myNode1>newNode1 content</myNode1><myNode2>newNode2 content</myNode2><myNode3>newNode3 content</myNode3><myNode4 attribute="yes">NODE CONTENT</myNode4><son><grandson>This is a grandson node</grandson></son></root>

<?xml version="1.0" encoding="ISO-8859-1"?> <!-- 声明 -->

<note date="08/08/2008">

<body>Don't forget the meeting!</body>

</note>

等价于

（文本内容之间的空格不能删减，标签与标签之前的空格可删减）

第一行为声明：版本号+编码格式。声明，不算是XML元素

XML是必须包含根元素的树结构。

XML中，元素必须有关闭标签。对于<p>George</p>，<p>为开始标签、</p>为关闭标签。

所有元素均可拥有属性、文本内容、元素内容（即元素包含其他元素）。

XML中元数据（metadata，有关数据的数据）应当存储为属性，而数据本身应当存储为元素。XML的属性必须加双引号

对于<note date="08/08/2008">，note标签，date为属性，开始标签与结束标签之间为文本内容

如果你把预留字符字符 "<" 放在 XML 元素中，会发生错误，这是因为解析器会把它当作新元素的开始。

<message>if salary < 1000 then</message>错误。应该**用实体引用**来代替 "<" 字符，修改为<message>if salary &lt; 1000 then</message>

在 XML 中，有 5 个预定义的实体引用：

|  |  |  |
| --- | --- | --- |
| &lt; | < | 小于 |
| &gt; | > | 大于 |
| &amp; | & | 和号 |
| &apos; | ' | 单引号 |
| &quot; | " | 引号 |

在 XML 中，文档中的空格不会被删节。

HTML 会把多个连续的空格字符裁减（合并）为一个：

HTML: Hello my name is David.

输出: Hello my name is David.

XML 元素是可扩展的，添加新元素后，原有解析程序仍能正常工作

请看下面这个XML例子：

<note>

<to>George</to>

<from>John</from>

<body>Don't forget the meeting!</body>

</note>

让我们设想一下，我们创建了一个应用程序，可将 <to>、<from> 以及 <body> 元素提取出来，并产生以下的输出：

MESSAGE

To: George

From: John

Don't forget the meeting!

想象一下，之后这个 XML 文档作者又向这个文档添加了一些额外的信息：

<note>

<date>2008-08-08</date>

<to>George</to>

<from>John</from>

<heading>Reminder</heading>

<body>Don't forget the meeting!</body>

</note>

那么这个应用程序会中断或崩溃吗？

不会。这个应用程序仍然可以找到 XML 文档中的 <to>、<from> 以及 <body> 元素，并产生同样的输出。

XML 文档不会携带有关如何显示数据的信息。

由于XML标签由XML文档的作者自定义，浏览器无法确定像 <table> 这样一个标签究竟描述一个 HTML 表格还是一个餐桌。在没有任何有关如何显示数据的信息的情况下，大多数的浏览器都会仅仅把 XML 文档显示为源代码（有语法高亮，通过点击元素左侧的加号或减号，可以展开或收起元素的结构）。



使用CSS可以定义如何显示XML传输的数据

note.xml文件内容为：

<?xml version="1.0" encoding="utf-8"?> <!-- 声明 -->

<?xml-stylesheet type="text/css" href="note.css"?><!--将当前xml文件链接到CSS文件-->

<note>

    <to>George</to>

    <from>John</from>

    <heading>Reminder</heading>

    <body>Don't forget the meeting!</body>

</note>

note.ccs文件内容为：

note

{

*background-color*: #000000; /\*根元素的background-color字段决定整个页面的背景色\*/

}

to,from

{

*background-color*: #ffffff; /\*子体矩形区域的背景色，白色\*/

*color*: #ff0000; /\*字体颜色，红色\*/

*font-size*: 60pt;

*margin-left*: 60pt;

}

heading

{

*display*: block; /\*设置block，则该元素独占一行\*/

*color*: #00ff00;

*font-size*: 40pt;

*margin-left*: 40pt;

*margin-bottom*: 40pt;

}

body

{

*display*: block;

*color*: #0000ff; /\*蓝色字体\*/

*font-size*: 20pt; /\*字体大小\*/

*margin-left*: 20pt; /\*距离页面左边的缩进\*/

}

将两个文件置于同一目录下，然后浏览器打开note.xml，显示效果如下：





使用前缀来避免命名冲突





命名空间属性一般放置在元素的开始标记处，其使用语法如下所示:

xmlns:namespace-prefix="namespace"

在元素的开始标记处使用命名空间时，该元素的所有子元素也都将通过一个同前缀与同一个命名空间相互关联

用于标示命名空间的地址，如”https://www.baidu.com”，仅仅用于赋予命名空间一个惟一的名称。示例中使用URL，并不是说这个标识真的要到那个网址去读取什么信息，而是仅作为一种区别的标志而已。因此这个网络地址也可以是虚拟的，然而很多公司经常把这个网络地址指向一个真实的Web页面，这个地址包含了关于当前命名空间更详细的信息。

统一资源标识符（Uniform Resource Identifier (URI)）是一串可以标识因特网资源的字符。最常用的 URI 是用来标示因特网域名地址的统一资源定位器(URL)。另一个不那么常用的 URI 是统一资源命名(URN)



**libxml2库**

libxml2库依赖于iconv库和zlib库

typedef unsigned char xmlChar;

使用unsigned char作为内部字符格式是考虑到它能很好适应UTF-8编码，UTF-8编码正是libxml2的内部编码

xmlChar相关函数有：xmlMalloc是动态分配内存的函数；xmlFree是配套的释放内存函数；xmlStrcmp是字符串比较函数

因为总是要在xmlChar\*和char\*之间进行类型转换，所以定义了一个宏BAD\_CAST

#define BAD\_CAST (xmlChar \*)

文档指针xmlDocPtr等于xmlDoc\*，xmlNewDoc函数创建一个新的文档指针

xmlParseFile函数默认以UTF-8格式读入文档，并返回文档指针

xmlReadFile函数以指定的某种编码格式读入xml文档，并返回文档指针

xmlFreeDoc释放文档指针。调用xmlFreeDoc时，该文档所有包含的节点内存都被释放

xmlSaveFile将文档以默认方式存入一个文件

<?xml version="1.0" encoding="UTF-8"?> <!-- notetest.xml文件 -->

<son><grandson>This is a grandson node</grandson></son>

grandson是一个xmlNodePtr，没有content；grandson的children也是一个xmlNodePtr，类型为XML\_ELEMENT\_NODE，content为This is a grandson node

children字段：

const char\* doc\_name = "nodetest.xml";

xmlDocPtr doc = xmlReadFile(doc\_name, "UTF-8", XML\_PARSE\_RECOVER);

xmlNodePtr node = xmlDocGetRootElement(doc);

cout << "node->name: " << node->name << endl; //son

cout << "node->type: " << node->type << endl; //XML\_ELEMENT\_NODE

cout << "node->content: " << (void\*)node->content << endl; //nullptr

cout << "node->children->name: " << node->children->name << endl; // grandson

cout << "node->children->type: " << node->children->type << endl;//XML\_ELEMENT\_NODE

cout << "node->children->content: " << (void\*)node->children->content << endl; //nullptr

cout << node->children->children->name << endl; //text

cout << node->children->children->type << endl; //XML\_TEXT\_NODE

cout << node->children->children->content << endl; //This is a grandson node

next字段

<?xml version="1.0" encoding="UTF-8"?>

<relative><son><grandson>This is a grandson node</grandson></son><uncle>this is a uncle node</uncle></relative>

const char\* doc\_name = "nexttest.xml";

xmlDocPtr doc = xmlReadFile(doc\_name, "UTF-8", XML\_PARSE\_RECOVER);

xmlNodePtr root = xmlDocGetRootElement(doc);

cout << root->children->children->children->content << endl; //This is a grandson node

cout << root->children->next->children->content << endl; //this is a uncle node

cout <<(void\*) root->children->next->next<< endl; //nullptr

注意：如果将上边的nexttest.xml文件添加换行，修改为如下图，则以上程序不能解析该新的文件了。但两个文件用浏览器打开都能正确显示



xmlNodeGetContent(xmlNodePtr cur)函数：

以cur根节点，一直向下遍历，知道找到XML\_TEXT\_NODE结点，返回其content字段

cout << xmlNodeGetContent(root) << endl; //This is a grandson node

cout << xmlNodeGetContent(root->children) << endl; //This is a grandson node

cout << xmlNodeGetContent(root->children->children) << endl; //This is a grandson node

获取结点属性：

// 遍历属性列表

xmlAttr \* attr = node->properties;

while (attr)

{

printf("\t%s=%s\n", attr->name, attr->children->content);

attr = attr->next;

}

设置根节点：

xmlDocSetRootElement(doc, root\_node);将root\_node设置为文档指针doc所指xml文档的根结点

增删查改：

//删除结点

**if** (!xmlStrcmp(cur->name, BAD\_CAST("myNode1")))

{

    xmlNodePtr temp = cur->next;///使用一个临时变量来存储断链节点的后续节点

    xmlUnlinkNode(cur);//将当前节点从文档中断链（unlink），这样本文档就不会再包含这个子节点

    xmlFreeNode(cur);  //释放堆上内存

    cur = temp;

}

//修改结点的内容

**if** (!xmlStrcmp(cur->name, BAD\_CAST"myNode2"))

{

    xmlNodeSetContent(cur, BAD\_CAST "content changed");

}

//给结点增加属性

**if** (!xmlStrcmp(cur->name, BAD\_CAST"myNode3"))

{

    xmlNewProp(cur, BAD\_CAST "new\_attri", BAD\_CAST"unknown");

}

//修改结点的属性

**if** (!xmlStrcmp(cur->name, BAD\_CAST("myNode4")))

{

    xmlSetProp(cur, BAD\_CAST("attribute"), BAD\_CAST("no"));

}

//增加子结点

**if** (!xmlStrcmp(cur->name, BAD\_CAST("son")))

{

    xmlNodePtr add = xmlNewNode(nullptr, BAD\_CAST"granddaughter");

    xmlAddChild(cur, add);

    xmlAddChild(add, xmlNewText(BAD\_CAST"this is a granddaughter node"));

}

上述代码将



变为



**XML文档编码、.cpp源文件编码、程序中字符串编码**

xml文档自身编码格式任意，可以是UTF-8、GB2312等。对应于于libxml2中的以下语句：

xmlSaveFormatFileEnc(dst\_path, doc, "GB2312", 1);//第三个参数决定以何种编码格式保存xml文档的是什么，可以是UTF-8、GB2312等

xmlReadFile(dst\_path, "GB2312", XML\_PARSE\_RECOVER); //打开GB2312编码格式的xml文档

但UTF-8是libxml2的内部编码，因此利用libxml2写xml文档时，必须将字符串转为UTF-8编码格式、读xml文档时获取的xmlNode::content、xmlNode::name字段都是UTF-8编码的字符串

xmlDocPtr doc = xmlNewDoc(BAD\_CAST"1.0");

xmlNodePtr root = xmlNewNode(NULL, BAD\_CAST"root");

xmlDocSetRootElement(doc, root);

**char**\* utf8str = g2u("this is a  殿军 node");  //gb2312转为utf-8格式的字符串

xmlAddChild(root, xmlNewText(BAD\_CAST utf8str));

//利用libxml2往xml文档写内容时，必须将中文（可能是结点名、属性、内容）转为utf-8编码的字符串

free(utf8str);

**const** **char**\* dst\_path = "include\_chinese.xml";

**int** nRel = xmlSaveFormatFileEnc(dst\_path, doc, "GB2312", 1);

//第三个参数是决定保存的xml文档的编码格式是什么，可以是UTF-8、GB2312等

**if** (nRel != -1)

{

    cout << "一个xml文档" << dst\_path << "被创建,写入" << nRel << "个字节" << endl;

}

xmlFreeDoc(doc);

doc = xmlReadFile(dst\_path, "GB2312", XML\_PARSE\_RECOVER);

//第二个参数是待读取的xml文档的文本编码格式

**if** (!doc) {

    cout << "Failed to call xmlReadFile" << endl;

**return** -1;

}

xmlNodePtr r = doc->children;

cout << r->children->name << endl;

**char**\* gb2312str = u2g((**char**\*)r->children->content);  //utf-8转为gb2312格式的字符串

cout << gb2312str << endl; //cout不能显示utf-8格式的字符串，因此先转为gb2312格式

free(gb2312str);

xmlFreeDoc(doc);

VS的高级保存选项，只影响.CPP源文件自身的编码格式，不影响程序中字符串的编码格式（字符串以何种编码格式存在于内存中）



|  |  |  |  |
| --- | --- | --- | --- |
| 字符 | GB2312码 | UTF-8码 | Unicode码点 |
| a | 0x61 | 0x61 | U+0061 |
| 殿 | 0xB5 EE | 0xE6 AE BF | U+6BBF |
| 军 | 0xBE FC | 0xE5 86 9B | U+519B |
| b | 0x62 | 0x62 | U+0062 |

对C++程序

*char*\* *astr* = "a殿军b"; //内存中字节为：61 b5ee befc 62

*char*\* *bstr* = g2u("a殿军b");//内存中字节为：61 e6 ae bf e5 86 9b 62

无论高级保存选项是UTF-8带签名代码页（如上图所示），还是GB2312代码页，

字符串astr在内存中都是61 b5ee befc 62（16进制，由低地址到高地址）

字符串bstr在内存中都是61 e6 ae bf e5 86 9b 62

借助iconv库实现GB2312码和UTF8码的转换

#include<iconv.h>

#pragma comment(lib,"D:/iconv/lib/iconv.lib")

**int** code\_convert(**char**\* from\_charset, **char**\* to\_charset, **char**\* inbuf,   **int** inlen, **char**\* outbuf, **int** outlen)

{

    iconv\_t cd;

**char**\*\* pin = &inbuf;

**char**\*\* pout = &outbuf;

    cd = iconv\_open(to\_charset, from\_charset);

**if** (cd == 0) **return** -1;

    memset(outbuf, 0, outlen);

**if** (iconv(cd, (**const** **char**\*\*)pin, (unsigned **int** \*)&inlen, pout, (unsigned **int**\*)&outlen)== -1)

**return** -1;

    iconv\_close(cd);

**return** 0;

}

//UTF-8码转为GB2312码

//成功则返回一个动态分配的char\*变量，需要在使用完毕后手动free，失败返回NULL

**char**\* u2g(**char** \*inbuf)

{

**int** nOutLen = 2 \* strlen(inbuf) - 1;

**char**\* szOut = (**char**\*)malloc(nOutLen);

**if** (-1 == code\_convert("utf-8", "gb2312", inbuf, strlen(inbuf), szOut, nOutLen))

    {

        free(szOut);

        szOut = NULL;

    }

**return** szOut;

}

//GB2312码转为UTF-8码

//成功则返回一个动态分配的char\*变量，需要在使用完毕后手动free，失败返回NULL

**char**\* g2u(**char** \*inbuf)

{

**int** nOutLen = 2 \* strlen(inbuf) - 1;

**char**\* szOut = (**char**\*)malloc(nOutLen);

**if** (-1 == code\_convert("gb2312", "utf-8", inbuf, strlen(inbuf), szOut, nOutLen))

    {

        free(szOut);

        szOut = NULL;

    }

**return** szOut;

}

**UTF-8编码程序示例**

vs2017中：

源文件保存为utf-8编码：高级保存选项->utf-8无签名代码页65001

MSVC编译器启用utf-8，即以utf-8编译代码文件，字符串字面量在内存中将为utf-8编码格式：配置属性->C/C++->命令行->其它选项->增加一个/utf-8选项

经过上述的操作后，仍然不能保证输出到控制台能正常显示，这是因为Windows的控制台默认是使用本地代码页，比如在国内就使用的是中文代码页（GB码）

设置控制台代码页为utf-8：在程序中加一句system("chcp 65001");

程序示例：

#include <iostream>

#include <string>

int main()

{

system("chcp 65001");

std::string text = "a殿军b";

const char\* p = text.data();

std::cout << text.size() << " " << text.length() << "\n";

std::cout << text << "\n";

for (size\_t i = 0; i < text.size();)

{

int cplen = 1;

if ((text[i] & 0xf8) == 0xf0) cplen = 4;

else if ((text[i] & 0xf0) == 0xe0) cplen = 3;

else if ((text[i] & 0xe0) == 0xc0) cplen = 2;

std::cout << text.substr(i, cplen) << std::endl;

i += cplen;

}

return 0;

}

上面代码定义了一个4个字符的字符串，两个英文字母，两个中文汉字，打印size和length是8

为能正确遍历并挨个字符打印出来，对每个字符占用的字节长度进行了判断

一个中文字符占用的字节数不固定，但长度其实可以从utf-8编码后的首个字节的中读取出来



可以看到，通过Byte 1的前4个bit就可以区分出这个字符究竟占用几个字节，所以就有了上述程序中的遍历方式

p处内存中的值：



程序运行结果：

Active code page: 65001

8 8

a殿军b

a

殿

军

b

**字符编码**

charset是character set的简写，即字符集

encoding，编码，字符集的编码



* 一个字符集可能有多中encoding方案。unicode将最终的编码方案称为UTF(Unicode Transformation Format，Unicode转换格式)，包括UTF-8、UTF-16、UTF-32
* ASCII、GBK、GB2312，字符集与编码一对一，因此称呼 “GB2312编码”，“GB2312字符集”都可以。

ASCII、latin1字符集

ASCII字符集，共128个字符，0~127，编码为单字节

latin1字符集，也叫ISO 8859-1字符集，共256个字符，，编码为单字节，兼容ASCII字符集。，ASCII字符集基础上扩充了128个西欧常用字符(包括德法两国的字母)

Unicode码点

字符集中每个字符有一个唯一编号，如Unicode字符集的码点、GB（GB即国标）系列中的区位码

字符集🡪编号🡪编码



有些字符彼此间长得非常像，在有了编号后，则只要编号不同，就代表不同的字符（如unicode字符集中的全角与半角）

Unicode码点示例：



这里的U+[XX]XXXX是Unicode码点的表示形式，X代表一个十六制数字，可以有4-6位，不足4位前补0补足4位，超过则按是几位就是几位。具体范围是U+0000~U+10FFFF。

unicode码点个数（也即字符个数为）17\*65536=1114112（一百多万）

汉字在unicode中称为CJK统一汉字（CJK：Chinese, Japanese, and Korean，中日韩）

使用正则表达式[\u4e00-\u9fa5] 匹配中文（unicode码点范围，即U+4E00~U+9FA5，但这只是Unicode最主要的一段中文区域，并不能包含所有中文）



借助Word的插入符号功能，可以看到，U+4E00为汉字“一“的码点，小于该值的区域仍有汉字；大于U+9FA5的区域也有汉字

UTF-8编码



UTF-32编码：在码点前垫0至4字节

UTF-16编码：码点小于U+FFFF，UTF-16编码与码点一致；码点大于U+FFFF则编码长度变成了四字节，值也变得很不一样

UTF-8编码：

码长1~4个字节。采用高位固定位区分不同码长。如下图，彩色的表示是保留的固定位，X表示是有效编码位。



* 单字节最高位都是0
* 多字节，N字节模式，首字节以“N个1再加0”打头，后跟“N-1”个以“10”打头的字节。

哪些码点使用哪些变长？

* 一字节有效编码位有7位，2^7=128，码点U+0000~U+007F（0~127）使用一字节。一字节恰好为ASCII码字符，因此UTF-8兼容ASCII
* 二字节有效编码位只有5+6=11位， 2^11=2048，码点U+0080~U+07FF（128~2047）使用二字节。
* 三字节有效编码为4+6+6=16位，2^16=65536，码点U+0800~U+FFFF（2048~65535）使用三字节编码。常用的汉字都落在此区域，这就是我们常说的汉字在UTF-8里用三字节表示。但这么说并不严谨，如果所有10万多汉字都被收录进来的话，那些偏门的汉字自然只能被挤到四字节空间上去了。
* 四字节有效编码位是3+6+6+6=21位，码点在U+FFFF以上的字符都在这里来表示

码点具体如何转为UTF-8：



ANSI编码

Windows记事本保存选项里有个 ANSI 编码。在简体中文Windows操作系统中，ANSI 编码代表 GBK 编码，code page值为936；在繁体中文Windows操作系统中，ANSI编码代表Big5，code page值为950；在日文Windows操作系统中，ANSI 编码代表 Shift\_JIS 编码。

cmd窗口执行chcp命令，可查看ANSI对应的实际编码

执行：chcp 437，code page改为437，当前终端的默认编码就为ASCII编码了

上面的操作只在当前终端起作用，并不会影响系统默认的“ANSI编码”

Windows系统代码页由“控制面板” =>“时钟、语言和区域”=>“区域和语言”=>“管理”=>“更改系统区域设置...”决定，不同区域对应不同的代码页

# URL编解码



1、汉字替换为URL编码

将含中文的url中汉字替换为url编码输出，其余部分不变

输入"http://10.210.63.11:7776/videos/殿军.mp4"，

输出"http://10.210.63.11:7776/videos/%E6%AE%BF%E5%86%9B.mp4"

#include<iostream>

#include<stdio.h>

#include<windows.h>

#include<string>

**using** std::cout;

**using** std::endl;

**using** std::string;

string  UrlUTF8(**char** \* str)

{

    string tt;

    string dd;

    GB2312ToUTF\_8(tt, str, strlen(str));

**int** len = tt.length();

**for** (**int** i = 0; i<len; i++)

    {

        // isalnum判断字符是否是数或者英文

        // ispunct判断一个字符是否为标点符号或特殊字符

**if** (isalnum((**BYTE**)tt.at(i)) || ispunct((**BYTE**)tt.at(i)))

        {

**char** tempbuff[2] = { 0 };

            sprintf\_s(tempbuff, "%c", (**BYTE**)tt.at(i));

            dd.append(tempbuff);

        }

**else** **if** (isspace((**BYTE**)tt.at(i)))  {   dd.append("+");  }

**else**

        {

**char** tempbuff[4];

            sprintf\_s(tempbuff, "%%%X%X", ((**BYTE**)tt.at(i)) >> 4, ((**BYTE**)tt.at(i)) % 16);

            dd.append(tempbuff);

        }

    }

**return** dd;

}

**void** GB2312ToUTF\_8(string& out, **char** \*input, **int** len)

{

**char** buf[4];

    memset(buf, 0, 4);

    out.clear();

**int** i = 0;

**while** (i < len)

    {

        //如果是英文直接复制就可以

**if** (input[i] >= 0)

        {

**char** asciistr[2] = { 0 };

            asciistr[0] = (input[i++]);

            out.append(asciistr);

        }

**else**

        {

**WCHAR** pbuffer;

            Gb2312ToUnicode(&pbuffer, input + i);

            UnicodeToUTF\_8(buf, &pbuffer);

            out.append(buf);

            i += 2;

        }

    }

**return**;

}

//汉字由Gb2312编码（2字节）转为Unicode码点（2字节）

**void** Gb2312ToUnicode(**WCHAR**\* pOut, **char** \*gbBuffer)

{

    //gb2312每个汉字占2字节

    ::MultiByteToWideChar(CP\_ACP, MB\_PRECOMPOSED, gbBuffer, 2, pOut, 1);

**return**;

}

//汉字由UTF-8编码（三字节）转为unicode码点（2字节）

**void** UTF\_8ToUnicode(**WCHAR**\* out, **char** \*input)

{

    //input-8汉字占三个字节，扩展B区以后的汉字占四个字节

    //三字节input-8符号的编码格式为1110xxxx 10xxxxxx 10xxxxxx

    //注意 WCHAR高低字的顺序,低字节在前，高字节在后

**char**\* uchar = (**char** \*)out;

    uchar[1] = ((input[0] & 0x0F) << 4) + ((input[1] >> 2) & 0x0F);

    uchar[0] = ((input[1] & 0x03) << 6) + (input[2] & 0x3F);

**return**;

}

//汉字由unicode码点（2字节）转为UTF-8编码（三字节）

**void**  UnicodeToUTF\_8(**char**\* out, **WCHAR**\* input)

{

**char**\* pchar = (**char** \*)input;

    out[0] = (0xE0 | ((pchar[1] & 0xF0) >> 4));

    out[1] = (0x80 | ((pchar[1] & 0x0F) << 2)) + ((pchar[0] & 0xC0) >> 6);

    out[2] = (0x80 | (pchar[0] & 0x3F));

**return**;

}

**int** main()

{

**char** str[] = "http://10.210.63.11:7776/videos/殿军.mp4";

    string utf8Code = UrlUTF8(str);

    cout <<utf8Code << endl;

    //"http://10.210.63.11:7776/videos/%E6%AE%BF%E5%86%9B.mp4"

}

2、URL中路径转为UTF-8编码

输入"/videos/%E6%AE%BF%E5%86%9B.mp4"，或者"/videos/%u6BBF%u519B.mp4"

输出"/videos/"+ 0xE6 + 0xAE + 0xBF + 0xE5 + 0x86 + 0x9B + ".mp4"

|  |  |  |  |
| --- | --- | --- | --- |
| 字符 | GB2312码 | UTF-8码 | Unicode码点 |
| a | 0x61 | 0x61 | U+0061 |
| 殿 | 0xB5 EE | 0xE6 AE BF | U+6BBF |
| 军 | 0xBE FC | 0xE5 86 9B | U+519B |
| b | 0x62 | 0x62 | U+0062 |

在URL的路径中汉字的处理方法有两种，或者借助UTF-8码，或者借助Unicode码点

//参数c取值于（0~9或a~f或A~F），不在此范围则返回false

//将ASCII码转为10进制，结果存储于v中

**bool** is\_hex(**char** c, **int** &v) {

**if** (0x20 <= c && isdigit(c)) {

        v = c - '0';

**return** **true**;

    }

**else** **if** ('A' <= c && c <= 'F') {

        v = c - 'A' + 10;

**return** **true**;

    }

**else** **if** ('a' <= c && c <= 'f') {

        v = c - 'a' + 10;

**return** **true**;

    }

**return** **false**;

}

//将字符串s中长为cnt的子串s[i]~s[i+cnt-1]视作16进制表示，转为10进制数值

//结果存储于val中，如from\_hex\_to\_i("%E6%AE%BF",1,2,val), 则val为230

//如果出现s[i]~s[i+cnt-1]不存在或者非16进制等输入异常情况，则返回flase

**bool** from\_hex\_to\_i(**const** std::string &s, **size\_t** i, **size\_t** cnt,

**int** &val) {

**if** (i >= s.size()) { **return** **false**; }

    val = 0;

**for** (; cnt; i++, cnt--) {

**if** (!s[i]) { **return** **false**; }

**int** v = 0;

**if** (is\_hex(s[i], v)) {

            val = val \* 16 + v;

        }

**else** {

**return** **false**;

        }

    }

**return** **true**;

}

//code为Unicode码点的10进制数值 ，转为UTF-8编码，码长为返回值

**inline** **size\_t** to\_utf8(**int** code, **char** \*buff) {

**if** (code < 0x0080) {

        buff[0] = (code & 0x7F);

**return** 1;

    }

**else** **if** (code < 0x0800) {

        buff[0] = (0xC0 | ((code >> 6) & 0x1F));

        buff[1] = (0x80 | (code & 0x3F));

**return** 2;

    }

**else** **if** (code < 0xD800) {

        buff[0] = (0xE0 | ((code >> 12) & 0xF));

        buff[1] = (0x80 | ((code >> 6) & 0x3F));

        buff[2] = (0x80 | (code & 0x3F));

**return** 3;

    }

**else** **if** (code < 0xE000) { // D800 - DFFF is invalid...

**return** 0;

    }

**else** **if** (code < 0x10000) {

        buff[0] = (0xE0 | ((code >> 12) & 0xF));

        buff[1] = (0x80 | ((code >> 6) & 0x3F));

        buff[2] = (0x80 | (code & 0x3F));

**return** 3;

    }

**else** **if** (code < 0x110000) {

        buff[0] = (0xF0 | ((code >> 18) & 0x7));

        buff[1] = (0x80 | ((code >> 12) & 0x3F));

        buff[2] = (0x80 | ((code >> 6) & 0x3F));

        buff[3] = (0x80 | (code & 0x3F));

**return** 4;

    }

    // NOTREACHED

**return** 0;

}

// 输入s形如/videos/%E6%AE%BF%E5%86%9B.mp4，其中%E6%AE%BF%E5%86%9B

//为殿军的UTF-8编码，每个字节前加上%

//调用decode\_url(s)，则子串"%E6%AE%BF%E5%86%9B"被替换为字符串

//(char)0xE6+0Xae+0Xbf+0xE5+0x86+0x9B

std::string decode\_url(**const** std::string &s) {

    std::string result;

**for** (**size\_t** i = 0; i < s.size(); i++) {

**if** (s[i] == '%' && i + 1 < s.size()) {

**if** (s[i + 1] == 'u') {

**int** val = 0;

**if** (from\_hex\_to\_i(s, i + 2, 4, val)) {

                    // 4 digits Unicode codes

**char** buff[4];

**size\_t** len = to\_utf8(val, buff);

**if** (len > 0) { result.append(buff, len); }

                    i += 5; // 'u0000'

                }

**else** {

                    result += s[i];

                }

            }

**else** {

**int** val = 0;

**if** (from\_hex\_to\_i(s, i + 1, 2, val)) {

                    // 2 digits hex codes

                    result += val;

                    i += 2; // '00'

                }

**else** {

                    result += s[i];

                }

            }

        }

**else** **if** (s[i] == '+') {

            result += ' ';

        }

**else** {

            result += s[i];

        }

    }

**return** result;

}

**int** main()

{

**const** **char**\* path = "/videos/%E6%AE%BF%E5%86%9B.mp4";

    string res= decode\_url(path);

    cout << res << endl; //打印出/videos/娈垮啗.mp4，要正确打印"殿军"两字，需要转为gb2312编码，再cout

}

# 正则表达式：

| **代码** | **说明** |
| --- | --- |
| . | 匹配除换行符以外的任意字符 |
| \w | 匹配字母或数字或下划线或汉字 |
| \s | 匹配任意的空白符 |
| \d | 匹配数字 |
| \b | 匹配单词的开始或结束 |
| ^ | 匹配字符串的开始 |
| $ | 匹配字符串的结束 |

| **代码/语法** | **说明** |
| --- | --- |
| \* | 重复零次或更多次，但尽可能多重复 |
| + | 重复一次或更多次，但尽可能多重复 |
| ? | 重复零次或一次，但尽可能多重复 |
| {n} | 重复n次，但尽可能多重复 |
| {n,} | 重复n次或更多次，但尽可能多重复 |
| {n,m} | 重复n到m次，但尽可能多重复 |

| **代码/语法** | **说明** |
| --- | --- |
| \W | 匹配任意不是字母，数字，下划线，汉字的字符 |
| \S | 匹配任意不是空白符的字符 |
| \D | 匹配任意非数字的字符 |
| \B | 匹配不是单词开头或结束的位置 |
| [^x] | 匹配除了x以外的任意字符 |
| [^aeiou] | 匹配除了aeiou这几个字母以外的任意字符 |

<a[^>]\*>匹配用尖括号括起来的以a开头的字符串

匹配没有预定义元字符的字符集合(比如元音字母a,e,i,o,u)：

* 在方括号里列出它们就行，[aeiou]就匹配任何一个英文元音字母，[.?!]匹配标点符号.或?或!
* 指定一个字符范围，[0-9]含意同\d，[a-z0-9A-Z\_]匹配字母数字， [\u4e00-\u9fa5] 匹配中文（unicode码点范围，但这只是Unicode最主要的一段中文区域，并非所有）

**|**把不同的规则分隔开，满足其中任意一种规则都应该当成匹配。

匹配时左到右测试每个条件，如果满足了某个规则的话，就不会去再管其它的条件了，因此\d{5}-\d{4}|\d{5}与\d{5}|\d{5}-\d{4}含义不同

(0\d{2}\)[- ]?\d{8}|0\d{2}[- ]?\d{8}这个表达式匹配3位区号的电话号码，其中区号可以用小括号括起来，也可以不用，区号与本地号间可以用连字号或空格间隔，也可以没有间隔

^\w+匹配一行的第一个单词(或整个字符串的第一个单词，具体匹配哪个得看选项设置)

既能匹配192.168.1.111:8000@192.168.1.111，又能匹配192.168.1.111:8000的正则表达式：

((\d{1,3}\.){3}\d{1,3}):\d{1,5}(@\1)?

分析：

\d匹配一个数字，\d{1,3}数字重复1~3次

.为元字符，匹配除换行符以外的任意字符；要匹配元字符.本身，需要字符转义，即\.

(){3}小括号用来指定子表达式，然后对子表达式重复三次

?重复0次或1次

\1表示这里的文本与匹配第一个小括号里的规则的文本一致

(((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)):(6553[0-5]|655[0-2]\d|65[0-4]\d{2}|6[0-4]\d{3}|[1-5]\d{4}|\d{1,4})(@\1)?

匹配点分10进制IP地址：

((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)，

如3.111.111.111、03.111.111.111、003.111.111.111

匹配端口：

6553[0-5]|655[0-2]\d|65[0-4]\d{2}|6[0-4]\d{3}|[1-5]\d{4}|\d{1,4}|

0-65535，前边不能有0

后向引用：

使用小括号指定一个子表达式后，匹配这个子表达式的文本(也就是此分组捕获的内容)可以在表达式或其它程序中作进一步的处理。默认情况下，每个分组会自动拥有一个组号，规则是：从左向右，以分组的左括号为标志，第一个出现的分组的组号为1，第二个为2，以此类推。

后向引用用于重复搜索前面某个分组匹配的文本。例如，\1代表分组1匹配的文本

例如：\b(\w+)\b\s+\1\b可以用来匹配重复的单词，像go go, 或者kitty kitty。这个表达式首先是一个单词，也就是单词开始处和结束处之间的多于一个的字母或数字(\b(\w+)\b)，这个单词会被捕获到编号为1的分组中，然后是1个或几个空白符(\s+)，最后是分组1中捕获的内容（也就是前面匹配的那个单词）(\1)

你也可以自己指定子表达式的组名。要指定一个子表达式的组名，请使用这样的语法：(?<Word>\w+)(或者把尖括号换成'也行：(?'Word'\w+)),这样就把\w+的组名指定为Word了。要反向引用这个分组捕获的内容，你可以使用\k<Word>,所以上一个例子也可以写成这样：\b(?<Word>\w+)\b\s+\k<Word>\b。

零宽断言:

匹配宽度为零，满足一定的条件/断言

零宽断言用于查找在某些内容(但并不包括这些内容)之前或之后的东西，也就是说它们有像 \b、^、$ 这样的锚定作用，用于指定一个位置，这个位置应该满足一定的条件(即断言)

|  |  |  |
| --- | --- | --- |
| **分类** | **代码/语法** | **说明** |
| **捕获** | (exp) | 匹配exp,并捕获文本到自动命名的组里 |
| (?<name>exp) | 匹配exp,并捕获文本到名称为name的组里，也可以写成(?'name'exp) |
| (?:exp) | 匹配exp,不捕获匹配的文本，也不给此分组分配组号 |
| **零宽断言** | (?=exp) | 先行断言，匹配exp前面的位置 |
| (?<=exp) | 后发断言，匹配exp后面的位置 |
| (?!exp) | 匹配后面跟的不是exp的位置 |
| (?<!exp) | 匹配前面不是exp的位置 |

(?=exp)比如\b\w+(?=ing\b)，匹配以ing结尾的单词的前面部分(除了ing以外的部分)

(?<=exp)如(?<=\bre)\w+\b会匹配以re开头的单词的后半部分(除了re以外的部分)，例如在查找reading a book时，它匹配ading。

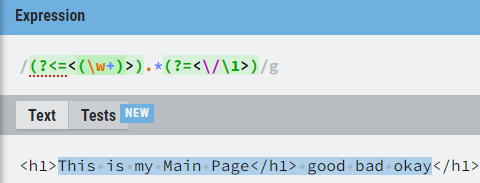
(?!exp)，如：\d{3}(?!\d)匹配三位数字，而且这三位数字的后面不能是数字；\b((?!abc)\w)+\b匹配不包含连续字符串abc的单词。

(?<!exp) ，如(?<![a-z])\d{7}匹配前面不是小写字母的七位数字

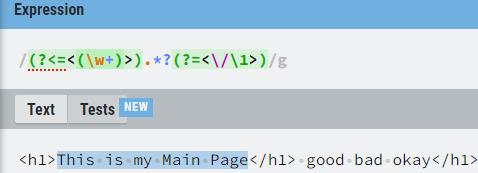
匹配不包含属性的简单HTML标签内里的内容：

如捕获<h1>This is my Main Page</h1>中的This is my Main Page，可以使用(?<=<(\w+)>).\*(?=<\/\1>)，其中使用了(?=exp)、(?<=exp)和后向引用

注意，上述正则可能因为贪婪匹配而匹配到额外内容，如



为了避免上图所述问题，正则应该设置为（中间的.\*变为.\*?）：



但是用grep中应用上述正则会报错：

[root@localhost home]$ echo '<h1>This is my Main Page</h1>' | grep -oP '(?<=<(\w+)>).\*(?=<\/\1>)'

grep: lookbehind assertion is not fixed length

这是因为grep等许多正则引擎不完备，后发断言中的pattern不支持非固定长度，如出现.\*等

修改为固定长度则正常：

[root@localhost home]$ echo '<h1>This is my Main Page</h1>' | grep -oP '(?<=<(h1)>).\*(?=<\/\1>)'

This is my Main Page

贪婪与懒惰：

当正则表达式中包含能接受重复的限定符时，通常的行为是（在使整个表达式能得到匹配的前提下）匹配尽可能多的字符。考虑这个表达式：a.\*b，它将会匹配最长的以a开始，以b结束的字符串。如果用它来搜索aabab的话，它会匹配整个字符串aabab。这被称为贪婪匹配。

懒惰匹配，也就是匹配尽可能少的字符。前面给出的限定符都可以被转化为懒惰匹配模式，只要在它后面加上一个问号?。这样.\*?就意味着匹配任意数量的重复，但是在能使整个匹配成功的前提下使用最少的重复。

a.\*?b应用于aabab的话，它会匹配aab（第一到第三个字符）和ab（第四到第五个字符）

| **代码/语法** | **说明** |
| --- | --- |
| \*? | 重复任意次，但尽可能少重复 |
| +? | 重复1次或更多次，但尽可能少重复 |
| ?? | 重复0次或1次，但尽可能少重复 |
| {n,m}? | 重复n到m次，但尽可能少重复 |
| {n,}? | 重复n次以上，但尽可能少重复 |

处理选项：

选项能用来改变处理正则表达式的方式。下面是.Net中常用的正则表达式选项。

| **名称** | **说明** |
| --- | --- |
| IgnoreCase(忽略大小写) | 匹配时不区分大小写。 |
| Multiline(多行模式) | 更改^和$的含义，使它们分别在任意一行的行首和行尾匹配，而不仅仅在整个字符串的开头和结尾匹配。(在此模式下,$的精确含意是:匹配\n之前的位置以及字符串结束前的位置.) |
| Singleline(单行模式) | 更改.的含义，使它与每一个字符匹配（包括换行符\n）。 |
| IgnorePatternWhitespace(忽略空白) | 忽略表达式中的非转义空白并启用由#标记的注释。 |
| RightToLeft(从右向左查找) | 匹配从右向左而不是从左向右进行。 |
| ExplicitCapture(显式捕获) | 仅捕获已被显式命名的组。 |
| ECMAScript(JavaScript兼容模式) | 使表达式的行为与它在JavaScript里的行为一致。 |

单行模式与多行模式可以共存

# 匹配url的正则表达式：

(https?|ftp|file)://[-A-Za-z0-9+&@#/%?=~\_|!:,.;]+[-A-Za-z0-9+&@#/%=~\_|]

IP地址、前后有汉字、带参数的，都是OK的



# C++标准正则表达式

|  |  |  |  |
| --- | --- | --- | --- |
| std::regex常用 | 类型 | 匹配次数 | 功能 |
| std::regex\_match | 函数模板 | 单次匹配 | * 正则表达式要与字符串完全匹配 * 可以获取子匹配的组 |
| std::regex\_search | 函数模板 | 单次匹配 | * 不要求整个字符序列完全匹配 * 单次匹配，搜索到与正则表达式匹配的子串则停止继续搜索 * 可以获取子匹配的组 |
| std::regex\_iterator | 类模板 | 多次匹配 | * 不要求整个字符序列完全匹配 * 多次匹配，找出所有匹配的子串 * 但不可以再获取子匹配的组 |
| std::regex\_replace | 函数模板 | 多次匹配 | * 多次匹配，找出所有匹配的子串(不考虑捕获组)，然后替换正则表达式匹配到的结果 |
| std::regex\_token\_iterator | 类模板 | 多次匹配 | * 找出所有匹配子串、前缀、子匹配的组 |

std::regex\_match 模板函数

bool parse\_request\_line(const char \*s) //const char\* s = "GET /books/?name=Professional%20Ajax HTTP/1.1";

{

static std::regex re("(GET|HEAD|POST|PUT|PATCH|DELETE|OPTIONS) "

"(([^?]+)(?:\\?(.+?))?) (HTTP/1\\.[01])");

//注意?:表示忽略该分组，因此下文m[4]为name=Professional%20Ajax，而不是?name=Professional%20Ajax

std::cmatch m;

if (std::regex\_match(s, m, re)) {

cout << "m[0]:" << std::string(m[0]) << endl; //GET /books/?name=Professional%20Ajax HTTP/1.1

cout << "m[1] is method:" << std::string(m[1]) << endl; //GET

cout << "m[2] is target:" << std::string(m[2]) << endl; //books/?name=Professional%20Ajax

cout << "m[3]:" << std::string(m[3]) << endl; // /books/

cout << "m[4]:" << std::string(m[4]) << endl; //name=Professional%20Ajax

cout << "m[5] is version:" << std::string(m[5]) << endl; //HTTP/1.1

return true;

}

return false;

}

//std::regex\_match: 正则表达式要与字符串完全匹配

//如果完全匹配，还可以获取子匹配的组

**void** match()

{

    std::string text = "Date:2017-10-10";

    //这里 "()" 用于捕获组, 捕获组的编号是按照 "(" 出现的顺序, 从左到右, 从1开始进行编号的

    std::string pattern = "Date.(\\d{4})-(\\d{2}-(\\d{2}))"; //Date.(\d{4})-(\d{2}-(\d{2}))

    std::regex express(pattern);

    std::cout.setf(std::ios\_base::boolalpha);

    //第0组一般是整个正则表达式匹配结果, 其他依次是捕获组的结果

    //捕获组的结果指第0组匹配结果中对应于正则表达式括号的文本

    std::match\_results<std::string::iterator> results1;

**if** (std::regex\_match(text.begin(), text.end(), results1, express))

    {

        std::match\_results<std::string::iterator>::const\_iterator iter;

**for** (iter = results1.begin(); iter != results1.end(); iter++)

{

            std::cout << iter->length() << ": " << iter->str() << std::endl;

        }

        //等价于

        std::cout << "或者" << std::endl;

**for** (**size\_t** index = 0; index <results1.size(); index++)

        {

            //sub\_match: 子匹配, match\_results里面都存放的是sub\_match

            //sub\_match 可以理解为 std::pair 的扩展, 它继承了 std::pair,

            //其中 first 实际上存放的是获取的字符串头指针地址, second 为尾指针地址

            auto sub = results1[index];

            std::cout << sub.length() << ": " << sub.str() << std::endl;

        }

    }

    std::cout << std::endl;

    /\*输出

    15: Date:2017-10-10 //0号分组为整个字符串

    4: 2017

    5: 10-10

    2: 10

    \*/

    //std::smatch定义为std::match\_results<std::string::const\_iterator>

    std::smatch s\_results1;

**if** (std::regex\_match(text.cbegin(), text.cend(), s\_results1, express))

    {

        std::match\_results<std::string::const\_iterator>::const\_iterator iter;

**for** (iter = s\_results1.begin(); iter != s\_results1.end(); iter++)

        {

            std::cout << iter->length() << ": " << iter->str() << std::endl;

        }

    }

    std::cout << std::endl;

    //std::cmatch定义为std::match\_results<const char\*>

    std::cmatch c\_results3;

**if** (std::regex\_match(text.c\_str(), c\_results3, express))

    {

**for** (auto iter = c\_results3.begin(); iter != c\_results3.end(); iter++)

        {

            std::cout << iter->length() << ": " << iter->str() << std::endl;

        }

    }

    std::cout << std::endl;

    //模板函数td::regex\_match，显示字符串是否符合正则表达式

    std::cout << std::regex\_match(text.begin(), text.end(), express) << std::endl;//true

    std::cout << std::regex\_match(text.c\_str(), express) << std::endl; //true

    std::cout << std::regex\_match(text, express) << std::endl;

    std::cout << std::endl;

}

std::regex\_search 模板函数

//std::regex\_search: 不要求整个字符序列完全匹配.

//单次匹配，搜索到与正则表达式匹配的子串则停止继续搜索

//此外, 它还可以获取子匹配的组

**void** search()

{

    std::string text = "Date:2017-10-10 ~ 2017-10-15";

    std::regex express("(\\d{4})-(\\d{2}-(\\d{2}))");

    std::cout.setf(std::ios\_base::boolalpha);

    //第0组一般是整个正则表达式匹配结果, 其他依次是捕获组的结果,

    //捕获组的结果指第0组匹配结果中对应于正则表达式括号的文本

    //不进行重复多次搜索

    std::match\_results<std::string::iterator> results1;

**if** (std::regex\_search(text.begin(), text.end(), results1, express))

    {

        //使用迭代器遍历, 这里的迭代器实际上是指向 std::sub\_match 的指针

        std::match\_results<std::string::iterator>::const\_iterator iter;

**for** (iter = results1.begin(); iter != results1.end(); iter++)

        {

            std::cout << iter->length() << ": " << iter->str() << std::endl;

        }

    }

    /\*输出

    10: 2017-10-10

    4: 2017

    5: 10-10

    2: 10

    \*/

    //类似regex\_match()函数，

    //regix\_search的匹配结果也可以是std::match\_results<std::string::iterator>、std::cmatch、std::smatch类型

   std::match\_results<const char\*> results1; //或std::cmath result1;

if (std::regex\_search(text.c\_str(), results1, express))

{

for (auto iter = results1.begin(); iter != results1.end(); iter++)

{

std::cout << iter->length() << ": " << iter->str() << std::endl;

}

}

    std::cout << std::regex\_search(text.begin(), text.end(), express) << std::endl;

    std::cout << std::regex\_search(text.c\_str(), express) << std::endl;

    std::cout << std::regex\_search(text, express) << std::endl;

    std::cout << std::endl;

}

std::regex\_iterator模板类

//std::regex\_iterator不要求整个字符序列完全匹配

//多次匹配，找出所有匹配的子串

//但不可以再获取子匹配的组

**void** iterator()

{

    std::string text = "Date:2017-10-10 ~ Date:2017-10-15";

    std::regex express("(\\d{4}-(\\d{2}-(\\d{2})))");

    std::regex\_iterator<std::string::const\_iterator> begin(text.cbegin(),

        text.cend(), express);

**for** (auto iter = begin; iter != std::sregex\_iterator(); iter++)

    {

        std::cout << iter->length() << ": " << iter->str() << std::endl;

    }

    std::cout << std::endl;

    /\*输出：

    10: 2017-10-10

    10: 2017-10-15

    \*/

}

或者：

std::regex\_iterator<const char\*> begin(text.c\_str(),text.c\_str()+text.size(),express);

for (auto iter = begin; iter != std::cregex\_iterator(); iter++)

{

std::cout << iter->length() << ": " << iter->str() << std::endl;

}

std::regex\_replace模板函数

//std::regex\_replace: 多次匹配，找出所有匹配的子串(不考虑捕获组)，然后

//替换正则表达式匹配到的结果

**void** replace()

{

    std::string text = "Date:2017-10-10~2017-10-15";

    std::regex express("(\\d{4})-(\\d{2}-(\\d{2}))");

    std::string result(256, '\0');

    std::string substitutes = "2019-10-12";

    //std::regex\_replace 模板函数返回值是新的字符串存入变量后尾部的指针位置

    // 置 0 是为了防止变量数据出错或乱码

    \*std::regex\_replace(result.begin(), text.begin(), text.end(), express, substitutes) = '\0';

    std::cout << result << std::endl;//Date:2019-10-12~2019-10-12

    std::cout << std::endl;

    result = std::regex\_replace(text,express,substitutes);

    std::cout << result << std::endl;//Date:2019-10-12~2019-10-12

    std::cout << std::endl;

}

std::regex\_token\_iterator模板类

//多次匹配，找出所有匹配子串、前缀、子匹配的组

**void** token\_iterator()

{

    std::string text = "Date:2016-10-10~2016-10-15";

    std::string pattern = "(\\d{4})-(\\d{2}-(\\d{2}))";

    std::regex express(pattern);

    //构造函数第四个参数取0，实现std::regex\_iterator功能

    //(多次匹配)显示正则表达式匹配

    std::regex\_token\_iterator<std::string::const\_iterator> begin(text.cbegin(),

        text.cend(), express); //第四个参数默认为0

**for** (auto iter = begin; iter != std::sregex\_token\_iterator(); iter++)

    {

        std::cout << iter->length() << ": " << iter->str() << std::endl;

    }

    std::cout << std::endl;

    /\*输出：

    10: 2016-10-10

    10: 2016-10-15

    \*/

    //构造函数第四个参数取-1，(多次匹配)显示正则表达式匹配到的文本的前缀

    std::regex\_token\_iterator<std::string::const\_iterator> begin2(text.cbegin(),

        text.cend(), express ,-1);

**for** (auto iter = begin2; iter != std::sregex\_token\_iterator(); iter++)

    {

        std::cout << iter->length() << ": " << iter->str() << std::endl;

    }

    std::cout << std::endl;

    /\*输出：

    5: Date:

    1:  ~

    \*/

    //(多次匹配)显示正则表达式子匹配, 3 表示第三组子匹配

    std::regex\_token\_iterator<std::string::const\_iterator> begin3(text.cbegin(),

        text.cend(), express, 3);

**for** (auto iter = begin3; iter != std::sregex\_token\_iterator(); iter++)

    {

        std::cout << iter->length() << ": " << iter->str() << std::endl;

    }

    std::cout << std::endl;

    /\*输出：

    2: 10

    2: 15

    \*/

    //(多次匹配)显示正则表达式匹配到的前缀和子匹配, -1 表示前缀, 2 表示第二个子匹配

    std::vector<**int**> vec{ -1,2 };

    std::regex\_token\_iterator<std::string::const\_iterator> begin4(text.cbegin(),

        text.cend(), express, vec);

**for** (auto iter = begin4; iter != std::sregex\_token\_iterator(); iter++)

    {

        std::cout << iter->length() << ": " << iter->str() << std::endl;

    }

    std::cout << std::endl;

    /\*输出：

    5: Date:

    5: 10-10

    1: ~

    5: 10-15

    \*/

    //(多次匹配)显示正则表达式匹配到的前缀、整个匹配、第一组子匹配、

    //第二组子匹配、第三组子匹配

    std::vector<**int**> v{ -1,0,1,2,3 };

    std::regex\_token\_iterator<std::string::const\_iterator> begin5(text.cbegin(),

        text.cend(), express, v);

**for** (auto iter = begin5; iter != std::sregex\_token\_iterator(); iter++)

    {

        std::cout << iter->length() << ": " << iter->str() << std::endl;

    }

    std::cout << std::endl;

    /\*输出：

    5: Date:

    10: 2016-10-10

    4: 2016

    5: 10-10

    2: 10

    1: ~

    10: 2016-10-15

    4: 2016

    5: 10-15

    2: 15

    \*/

}

# 动态库、静态库

DLL：动态链接库，Dynamic link library，简称动态库

动态库可以动态加载可以静态加载

LIB：静态库

VC++目录中的包含目录对应C/C++中的附加包含目录，VC++目录中的库目录对应链接器中的附加库目录，二者效果相同。

#include头文件的时候，<>表示搜索系统路径、VC++目录—包含目录中的文件，""表示先搜索当前源文件所处目录下的文件，当无法搜索到时再转而去搜索系统路径和VC++目录—包含目录

源码🡪预处理器（文本文件）🡪编译器（汇编语言文件）🡪汇编器（可重定位目标文件）🡪链接器（可执行目标文件）

编译单元：经预处理器后头文件被包含，只剩下.c或.cpp文件，一个.c或.cpp文件即一个编译单元。一个编译单元对应一个可重定位目标文件.obj文件

链接过程发生于：

编译阶段（静态库）

加载时（隐式链接静态加载动态库）

程序运行时（显示链接动态加载动态库）

Windows下静态库为.lib后缀，动态库为.dll后缀

在Linux下静态库为为.a后缀，动态库为.so后缀

C:\Program Files (x86)\Microsoft Visual Studio\2017\Professional\VC\Tools\MSVC\14.14.26428\bin\Hostx64\x64下有Dumpbin.exe的小程序，用它可以查看DLL文件中的函数结构

如C:\Program Files (x86)\Microsoft Visual Studio\2017\Professional\VC\Tools\MSVC\14.14.26428\bin\Hostx64\x64>dumpbin.exe /EXPORTS C:\Users\myth\E\VSC++Projects\LibCall\x64\Debug\DllCreate.dll



上图显示DllCreate.dll中有一个multi()函数

查看一个dll是32位还是64位的

dumpbin /headers C:\Users\myth\Desktop\msvcr100d.dll



**开源代码发布的常用方式：**

1. 预编译的开发包：包含一些.dll文件和一些.lib文件。这里的.lib是导入库，而不是静态库。但是引入方式和静态库一样，要在VC++目录—库目录添加这些.lib的路径、在链接器—附加库目录添加.lib文件名。而.dll则最好放到最后产生的应用程序exe执行文件所在目录。

2. 用户自己编译： 下载的是源代码，编译后生成的很可能也是.dll + .lib(导入库)文件

3. 如果只有dll，并且知道dll中函数的函数原型，那么显式链接。

**静态库与动态库的区别**

静态链接库与动态链接库都是共享代码的方式

* 采用静态链接库，则库中的机器码指令都被直接包含在最终生成的 EXE 文件中了
* 采用动态链接库，该DLL不被包含在最终EXE文件中，EXE文件加载到内存时，动态库被加载到内存（静态加载）或者EXE执行时动态地加载和卸载动态库（动态加载）。
* 静态链接库中不能再包含其他的动态链接库或者静态库，而在动态链接库中还可以再包含其他的动态或静态链接库
* 多进程能共享内存中的一份动态库





**Windows msvc生成/使用静态库：**

生成

VS2017控制台应用程序： Visual C++ ->Windows桌面向导->控制台应用程序、空项目

VS2017生成静态库: Visual C++ ->Windows桌面向导->静态库、空项目->编写程序->生成解决方案。项目中仅一个文件，内容如下：

#include<iostream>

void hello(const char\* name)

{

std::cout <<"hi, " <<name << ",how are you？" << std::endl;

}

使用：

(1)声明要使用的静态库中的函数void hello(const char\*);

或者通过包含一个头文件来声明，头文件采用绝对路径或者在项目属性->VC++目录->包含目录中配置头文件路径，只#include头文件名

(2)在项目属性->VC++目录->库目录中配置.lib的路径，且在项目属性->链接器->输入->附加依赖项中配置库名称，即.lib文件名

或者直接使用#pragma comment指令链接静态库

#pragma comment( lib, "../../MyStaticLib/Debug/MyStaticLib.lib" ) //指定与静态库一起连接

也可以使用绝对路径，即

#pragma comment( lib, "E:/VSProjects/MyStaticLib/Debug/MyStaticLib.lib" )

静态库是一个或者多个obj文件的打包。链接一个静态库，如果其中有错，它会准确的找到是哪个obj有错。

动态库一般会有对应的导入库（.lib），方便程序静态载入动态库，否则需要LoadLibrary()、FreeLibrary()和GetProcProcess()显式链接。有了导入库，你只需要链接导入库后按照头文件函数接口的声明调用函数就可以了。

静态库本身就包含了实际执行代码、符号表等等，而对于导入库而言，其实际的执行代码位于动态库中，导入库只包含了地址符号表等，确保程序找到对应函数的一些基本地址信息。

**Windows下静态库链接静态库**

静态库add2.lib中有add2(int,int)函数

静态库add3.lib链接静态库add2，有add3()函数

用户使用时，只要链接add3.lib，则即可使用add2(int,int)函数又可使用add3(int,int)函数

静态库不能链接动态库

**Windows msvc动态库生成：**

VS2017生成动态库: Visual C++ ->Windows桌面向导->动态库、空项目->编写程序->生成解决方案。项目中两个文件，内容如下：

//sum.h文件：

#ifdef SUM\_EXPORTS

#define SUMDLL\_API \_\_declspec(dllexport)

#else

#define SUMDLL\_API \_\_declspec(dllimport)

#endif

SUMDLL\_API **int** sum(**int** a, **int** b);

//sum.cpp文件：

#include"sum.h" //必须有此句，同时工程里定义预处理宏SUM\_EXPORTS，否则生成的dll没有导出sum函数，没有lib文件

**int** sum(**int** a, **int** b)

{

**return** a + b;

}

**Windows msvc动态库静态加载：**

在程序开始执行时加载DLL文件。

(1)声明要使用的动态库中的函数int sum(int, int);即可

或者通过包含之前的sum.h头文件来声明，头文件采用绝对路径或者在项目属性->VC++目录->包含目录中配置头文件路径，只#include头文件名

(2)在项目属性->VC++目录->库目录中配置.lib的路径，且在项目属性->链接器->输入->附加依赖项中配置库名称，即.lib文件名

(3)将.dll放于应用程序exe执行文件所在目录（这样更好，ctrl+F5可运行，点击exe文件也可运行）或者解决方案资源管理器—项目名—右键—在文件资源管理器中打开文件夹（这样ctrl+F5可运行，点击exe不能运行提示找不到dll）

前两步等价于

#include<iostream>

#include<E:/VSC++Projects/MyDynamicLib/MyDynamicLib/sum.h>

#pragma comment(lib,"E:/VSC++Projects/MyDynamicLib/Debug/MyDynamicLib.lib")

//\_declspec(dllimport)  int sum(int a, int b);

//不使用 \_\_declspec(dllimport) 也能正确编译代码，但使用 \_\_declspec(dllimport) 使

//编译器可以生成更好的代码。编译器之所以能够生成更好的代码，是因为它可以

//确定函数是否存在于 DLL 中，这使得编译器可以生成跳过间接寻址级别的代码，

//而这些代码通常会出现在跨 DLL 边界的函数调用中。但是，必须使用 \_\_declspec(dllimport)

//才能导入 DLL 中使用的变量

**int** main()

{

    std::cout << sum(4, 4) << ",how are you？" << std::endl;

**return** 0;

}

**Windows msvc动态库动态加载**

应用程序在执行过程中随时可以加载DLL文件，也可以随时卸载DLL文件，利用windows.h头文件中的LoadLibrary()、FreeLibrary()和GetProcProcess()函数。显式链接不需要.lib文件

dll工程仅一个cpp文件如

extern "C" \_declspec(dllexport) int multi(int x, int y)

{

return x \* y;

}

使用时，

#include<iostream>

#include <windows.h>

**int** main()

{

**HMODULE** hmod = LoadLibrary(L"E:/VSC++Projects/YourDynamicLib/Debug/YourDynamicLib.dll");//加载动态链接库

**if** (hmod == NULL)

    {

        std::cout << "Failed loading DLL.\n";

**return** 1;

    }

**int**(\*multi)(**int**, **int**) = (**int**(\*)(**int**, **int**))GetProcAddress(hmod, "multi");//获取dll中的函数的地址

**if** (multi == NULL)

    {

        std::cout << "not find function" << std::endl;

**return** 1;

    }

**int** x = multi(2, 3);

    FreeLibrary(hmod);  //释放动态链接库

    std::cout << "2 \* 3 = " << x << std::endl;

}

生成dll库的程序中extern "C"，表明函数使用c语言的风格编译函数，这样后面显式调用时，GetProcAddress第二个参数就是原来的函数名，否则如果使用C++编译的话支持函数重载，则第二个参数会发生变化

dll库源代码：

\_declspec(dllexport) int multi(int x, int y)

{

return x \* y;

}

查看生成的dll文件，



函数名变成了?multi@@YAHHH@Z，因此，应用程序中

int (\*multi)(int,int) = (int(\*)(int,int))GetProcAddress(hmod, "?multi@@YAHHH@Z");

严重性使用GetProcAddress()函数时，可以利用MAKEINTRESOURCE()函数直接使用DLL中函数出现的顺序号，代替复制dll生成的函数的一大串字符串名，但是必须知道函数生成的顺序ordinal，所以也必须打开dll查看

int (\*multi)(int,int) = (int(\*)(int,int))GetProcAddress(hmod, "?multi@@YAHHH@Z");可以替换为int(\*multi)(int, int) = (int(\*)(int, int))GetProcAddress(hmod, MAKEINTRESOURCE(1))

**Windows动态库链接动态库**

add.dll和导出库add.lib

//add2.cpp

\_declspec(dllexport) int add2(int x, int y)

{

return x+y;

}

multi.dll和导出库multi.lib，multi.dll依赖于add.dll（生成multi动态库时隐式链接动态库add.dll）

int add2(int, int);

#pragma comment(lib,"C:/Users/myth/Desktop/add2.lib");//必须有该预处理指令。但此时不需要add2.dll文件即可生成解决方案

\_declspec(dllexport) int add3(int x, int y,int z)

{

return add2(add2(x, y), z);

}

用户exe使用multi动态库时，隐式链接multi动态库（使用multi.lib），

#include<iostream>

extern "C" int multi(int, int);

#pragma comment(lib,"C:/Users/myth/Desktop/multi.lib")

int main()

{

std::cout << multi(-4,-7)<<std::endl;

}

此时，即可以生成解决方案。但要运行，则必须multi.dll和add.dll都能被应用程序搜索到

**Windows msvc动态库的模块定义文件.def**

\_declspec(dllexport)语句声明函数int multi(int x, int y)为DLL的导出函数。DLL内的函数分为两种：

(1)DLL导出函数，可供应用程序调用；

(2)DLL内部函数，只能在DLL程序使用，应用程序无法调用它们。

添加.def文件后，需要工程属性-链接器-输入-模块定义文件中加入自定义的.def文件名



dll库源文件函数声明为

\_declspec(dllexport) int mysubtract(int x, int y)

\_declspec(dllexport) float mysubtract(float x, float y)，生成dll文件后用dumpbin.exe文件查看，如下：



?mysubtract@@YAHHH@Z和?mysubtract@@YAMMM@Z是编译器采用默认命名方式命名

应用程序中Func mysubtract = (Func)GetProcAddress(hdll, "?mysubtract@@YAHHH@Z");

采用自定义.def文件重命名，

LIBRARY DynamicLib

EXPORTS

subtract\_int=?mysubtract@@YAHHH@Z

subtract\_float=?mysubtract@@YAMMM@Z

再生成dll文件，用dumpbin查看，内容如下：



应用程序中Func mysubtract = (Func)GetProcAddress(hdll, "subtract\_int");

**Windows搜索动态库顺序**

一个64位应用程序，所有与这个应用程序相关的东西，包括DLL文件，必须是64位的

32位应用程序相关的所有东西必须都是32位的

WOW64” 的意思是 “Windows On Windows64”， 所以说在SysWOW64里并不是运行64-bit的程序，而是32-bit的程序

32位Windows操作系统：

\Windows\System文件夹存放16位的DLL文件(以及其它的支持文件)

\Windows\System32文件夹存放32位DLL文件(以及其它的支持文件)

64位Windows操作系统：

\Windows\System32文件夹存放64位DLL文件(以及其它的支持文件)

\Windows\SysWOW64文件夹存放32位DLL文件(以及其它的支持文件)

Windows目录，"C:\Windows"，用GetWindowsDirectory()函数，cmd下可用echo %WINDIR%查看

Windows 系统目录，用GetSystemDirectory()函数获得，“C:\Windows\system32”

无论x86还是x64编译选项，GetSystemDirectory()函数返回都是“C:\Windows\system32”。

但对于32位应用程序，依赖的dll置于C:\Windows\SysWOW64，对于64位应用程序，依赖的dll应置于C:\Windows\SysWOW64

Windows系统按如下顺序搜索应用程序依赖的DLL：

* 包含exe文件的目录，exe在VSC++Projects/UsingLib/Debug/，则dll置于此目录
* 进程的当前工作目录，cd到e/VSC++Projects/UsingLib目录下，执行$Debug/UsingLib.exe，则进程的当前工作目录为e/VSC++Projects/UsingLib /
* Windows系统目录，对32位应用程序在 C:\Windows\SysWOW64搜索所需的dll，对64位应用程序在C:\Windows\system32搜索dll
* Windows目录，即C:\Windows
* 列在Path环境变量中的一系列目录，如D:/Git/cmd

因此应用程序exe静态加载dll或动态加载dll时，dll可以置于这些搜索路径下

**软件缺少C++运行时库报错**

一个在VS2010下开发的播放器接口，编译选项为x64 Debug、C/C++代码生成运行库选择多线程调试DLL(/MDd)，最终生成包括一些动态库、静态库、定义接口的头文件

用户使用该播放器接口，用于自己的QT工程，VS2017、x64、Debug，成功生成了解决方案，exe文件，但运行时提示缺少msvcp100d.dll、msvcr100d.dll。

msvcp100d.dll、msvcr100d.dll是VS2010开发C++时的运行时库

从微软官网下载

Microsoft Visual C++ 2010 Redistributable Package (x64)，

，安装后依然无效，因为可再发行组件包中只有Release版本的运行时库，即msvcp100.dll、msvcr100.dll这些，没有调试版本的

因此，在网上下载VC2010、Debug版的64位/32位的msvcp100d.dll、msvcr100d.dll文件

同四个如下，将这四个置于系统可搜索到的地方即可



**动态库位置无关代码**

动态库被不同进程共享

进程共享的是动态库的代码段，动态库的数据段不可能被不同进程共享

对不同进程，动态库会被加载到进程的虚拟内存空间的不同位置

考虑如下dll库工程，仅一个cpp文件：

//circle.cpp

\_declspec(**dllexport**) **double** PI = 3.14159265;

\_declspec(**dllexport**) **double** area(**double** radius)

{

**return** 2 \* PI\*radius;

}

生成circle.dll和对应的导出库circle.lib文件

用户应用程序，如下

//main.cpp

#include<iostream>

**using** **namespace** std;

\_declspec(**dllimport**) **double** PI;

**double** area(**double**);

#pragma comment(lib,"E:/vsprojects/MyDynamicLib/x64/Debug/circle.lib")

**int** main()

{

    cout << PI << endl;

    cout << area(2.0) << endl;

}

对不同进程，动态库会被加载到进程的虚拟内存空间的不同位置，因此库中定义的全局变量PI的内存地址对不同进程不同，

如果"2\*PI\*radius"的汇编代码中，直接使用PI的绝对地址，则对不同进程，该地址值不同，导致dll库的代码段应该不同，与进程共享动态库的代码段矛盾。

因此由circle.cpp生成dll的过程中，汇编出的是位置无关代码(PIC,position independent code)

位置无关代码对导出变量的处理方式为：

GOT，全局偏移量表(global offset table)。对PIC的数据引用而言，里面存放的就是全局变量的地址。

单独编译动态库的.o时，可以将text段和data段紧密排列，比如将data放在text之后，这样data和text之间的偏移是常数。 然后我们将GOT放在data的固定位置比如头部。一旦模块载入的话，那么动态链接器就会解析GOT里面所有的条目， 并且填写上对应的地址。

此时"2\*PI\*radius"的汇编代码中，获取PI值的方式为：当前指令地址减去固定值得到一个新地址，读取该新地址里存放的值，该值才是变量PI的绝对地址，从而获得PI的值

这样即使动态库被加载到不同进程的进程虚拟内存空间的不同位置，也不需要修改库的代码段。库的数据段与main.cpp生成的可重定位目标文件的数据段合并，只需要修改合并后数据段的GOT表即可。

**g++编译选项**

gcc最开始的时候是GNU C Compiler，就是一个c编译器。但是后来因为这个项目里边集成了更多其他不同语言的编译器，GCC就代表 the GNU Compiler Collection，所以表示一堆编译器的合集。 g++则是GCC的c++编译器

示例：

main.cpp是包含main函数的helloworld文件，

在所在目录下，$ g++ -o main.o main.cpp，此时目录下多出main.o可执行文件

运行，$ ./add.o 3 6

-o：

g++ -o hello.exe a.cpp b.cpp

将两个源码文件编译链接成一个单一的可执行程序，名为hello.exe

如果没有-o，编译器采用默认的a.out

C++11编译：g++ -std=c++11 -o main.exe main.cpp

-c：

　　激活预处理、编译和汇编，不执行链接，生成可重定位目标文件

gcc -c hello.c 生成hello.o的obj文件

g++ -c st1.cpp st2.cpp 生成st1.o和st2.o两个可重定位目标文件

-S：

　　激活预处理和编译，生成汇编代码。

　　gcc -S hello.c

　　生成hello.s的汇编代码，也可手动指定输出，如gcc –S hello.c –o hello.s

-E：

　　只激活预处理, 生成 \*.ii(.c文件生成的是\*.i) 文件

　　面.

　　gcc -E hello.c –o hello.ii

-g：

产生Debug版本的可执行程序

[root@guest: zcj]$ g++ -g -o a.out a.cpp b.cpp

[root@guest: zcj]$ gdb a.out

(gdb) set args 5 6

(gdb) b 'main(int,char\*\*)'

**gdb调试**

gcc -g -o test test.c

gdb ./test ：进入后提示符变为(gdb)

(gdb) set args hello 6 ：给main函数传递两个参数第一个是hello，第二个是6

(gdb) b 6 ：在第6行打断点，也可b X264Encoder.cpp:6

(gdb) b func if target == 5 ：条件断点，if和其后内容用空格隔开，无论if后的内容是否有括号都可

(gdb) ignore <断点编号> 3 ：忽略前3次断点命中

(gdb) info b：查看断点情况

(gdb) r：开始运行程序

(gdb) p a：显示变量a的值

(gdb) info source：查看源文件的完整路径（list只显示文件名）

(gdb) info local：查看查看当前函数中所有局部变量

(gdb) p malloc\_stats()，查看ptmalloc内存分配统计信息

(gdb) n ：单步运行

(gdb) c ：程序继续运行

(gdb) q：退出GDB

(gdb) step：单步调试如果有函数调用，则进入函数；与命令n不同，n是不进入调用的函数的

(gdb) finish：跳出函数

(gdb) bt：看堆栈，继续f 3可看堆栈第3条记录（由第0开始）。调用栈被分割成了一些连续的被称为stack frames的块

(gdb) list：罗列源码，命令一次只能显示十行，若想查看后面的源码可一直按回车键，直到所有源码显示完

(gdb) set listsize <count>：设置list一次显示源代码的行数。

(gdb) show listsize：查看当前listsize的设置。

多线程相关调试：

(gdb) info inferiors：查看进程

(gdb) info threads：查看线程

(gdb) thread n：切换线程（n代表线程号，由GDB分配，info threads时会显示出来）

(gdb) thread apply all <command>：让所有被调试线程执行命令，如thread apply all bt查看所有线程的栈结构

(gdb) thread apply ID1 ID2 command：让一个或者多个线程执行GDB命令command

实时修改变量值：set var a=3 将变量a的值修改为3，对后续代码会生效

查看内存映射：info proc mappings

导出内存数据：dump binary memory ret.bin 0x7ffda00672c0 0x7ffda0120000（后两个参数为起始地址、结束地址）

检测某个地址上的内容是否变化：watch \*(size\_t\*)(0x7ffda40672c0)

watch和if配合使用：watch \*(int\*)0x7fffa9abdbc0 if \*(int\*)0x7fffa9abdbc0==13

捕获异常：catch throw

动态打印：dprintf是一种特殊的断点，因此目标程序并不是持续执行的，而会因为断点而暂停执行，等dprintf打印消息后再继续，这个过程非常短暂。

dprintf main.cpp:30,"sample rate=%d, channels=%d\n",rate,num

或dprintf Func,"start call function\n"

导出断点设置：save br br.txt

加载断点设置：gdb ./Media -ex "set breakpoint pending on" -x br.txt

gdb执行程序，直接设置main函数参数，并自动加载断点：

gdb -ex "set breakpoint pending on" -x br.txt -ex run --args ./build/app/RxTxApp --config\_file config/test\_tx\_1port\_1v.json

在断点命中后自动执行若干命令：

(gdb) b 1701

(gdb) commands

>p codec->sl\_hdr\_data\_size

>p codec->cuva\_hdr\_data\_size

>end

创建自定义命令：（实现断点后自动打印函数参数）

(gdb) define myshow

>b $arg0

>commands

>info args

>end

>end

(gdb) myshow nvenc\_encode\_frame

编写脚本：

#文件mymv.txt

define mv

if $argc == 2

delete $arg0

break $arg1

else

print "输入参数数目不对"

end

end

加载脚本：source mymv.txt

gdb打印数组16进制：

char\* buf, int len，其中len为36，查看buf中16进制值：p /x (char[36])\*buf或者p /x \*buf@16

int arr[] = {1264, 1234, 1225, 1212, 1222, 1211, 1290}，以16进制查看arr中元素：p /x arr

gdb查看数组元素：

p \*array@length // length是想查看的长度

@的左边是数组的首地址的值，如数组名

如果是静态数组的话，可以直接用print数组名，就可以显示数组中所有值

示例程序：

int arr1[3]={0,1,2};

double\* arr2=new double[3];arr2[0]=1.1;arr2[1]=2.2;arr2[2]=3.3;

delete[] arr2;

gdb$ p arr1

$1 = {0x0, 0x1, 0x2}

gdb$ p arr2

$2 = (double \*) 0x55555556b2c0

gdb$ p \*arr2@2

$3 = {1.1000000000000001, 2.2000000000000002}

gdb$ p \*arr2@3

$4 = {1.1000000000000001, 2.2000000000000002, 3.2999999999999998}

gdb$ p \*(arr2+1)@2

$5 = {2.2000000000000002, 3.2999999999999998}

**gdb中SIGINT信号：**

Ctrl+C 发送的是 SIGINT 信号

gdb中SIGINT信号默认被gdb接管，并不会传递给引用程序，如何避免

方法1，在GDB中按Ctrl+C后，输入：

(gdb) signal SIGINT

方法2，更改GDB的信号处理设置

# 查看当前信号处理方式

(gdb) info signals

会显示

Signal        Stop      Print   Pass to program Description

SIGINT        Yes       Yes     No              Interrupt

# 让GDB将SIGINT直接传递给程序

(gdb) handle SIGINT nostop noprint pass

# 恢复默认设置

(gdb) handle SIGINT stop print nopass

从另一个终端如何发送SIGINT信号： kill -2 <PID>

**Linux g++生成/使用静态库：**

//st1.cpp

#include <iostream>

using namespace std;

void display1()

{

cout<<"This is my first static library!!!"<<endl;

}

//st2.cpp

#include <iostream>

using namespace std;

void display2()

{

cout<<"This is my second static library"<<endl;

}

//main.cpp

void display1(); void display2();

int main()

{

display1();

display2();

return 0;

}



ar –rsv 也可ar rsv，也可ar rcs，也可ar –rcs，也可ar rcsv，也可ar –rcsv，也可ar –rs，也可ar rs

符号“-”可有可无

r 将文件插入备存文件中

c 建立备存文件，可省略

s将目标文件索引写入存档

v 程序执行时显示详细的信息，可省略

链接静态库也可以：g++ -o main.exe main.cpp -L./ -lmytest

-l (L的小写)：指定需要链接的库的名字(注意：-l后面直接添加库名省略“lib”前缀和“.so”或“.a”后缀 ），如链接静态库libc.a则：-lc、链接动态库libc.so则: -lc

在linux环境下，静态库（\*.a）与动态库（\*.so）的库文件名命名规则为lib+<库名>+.so、lib+<库名>+.a，如果有一个 protobuf 库, 那么相应的库文件为 libprotobuf.so、libprotobuf.a

**Linux g++生成/使用动态库**

生成动态库：gcc -fpic -shared -o libsum.so sum.c

编译时链接动态库的三种方式：

* 1. 以全路径方式指定要链接的库

以全路径方式指定要链接的库，gcc自动识别是静态库还是动态库

gcc -o main2.exe main2.c libsum.a则链接静态库libsum.a

gcc -o main2.exe main2.c libsum.so则链接动态库libsum.so

* 1. 以-L、-l<库名>方式指定要链接的库

gcc -o main2.exe main2.c -L./ -lsum

既有动态库libsum.so，又有静态库libsum.a，则gcc链接的是动态库

只有静态库libsum.a，则gcc链接的是静态库libsum.a

只有动态库libsum.so，则gcc链接的是动态库libsum.so

* 1. 以-L、-l:<文件名>方式指定要链接的库

gcc -o main2.exe main2.c -L./ -l:libsum.a则链接静态库libsum.a

gcc -o main2.exe main2.c -L./ -l:libsum.so则链接动态库libsum.so

如果用-l:filename格式指定一个文件名，则链接程序直接去找这个文件名了，不会再像使用-lname时一样将name扩展成lib<name>.a或lib<name>.so格式的库文件名.

动态链接时，全路径方式和-l方式不完全等价，存在区别

*//动态库./lib/sum.cpp*

#include<cstdio>

void sum(int a, int b) { printf("sum execute\n");}

*//可执行程序main.cpp*

#include<stdio.h>

void sum(int,int);

int main()

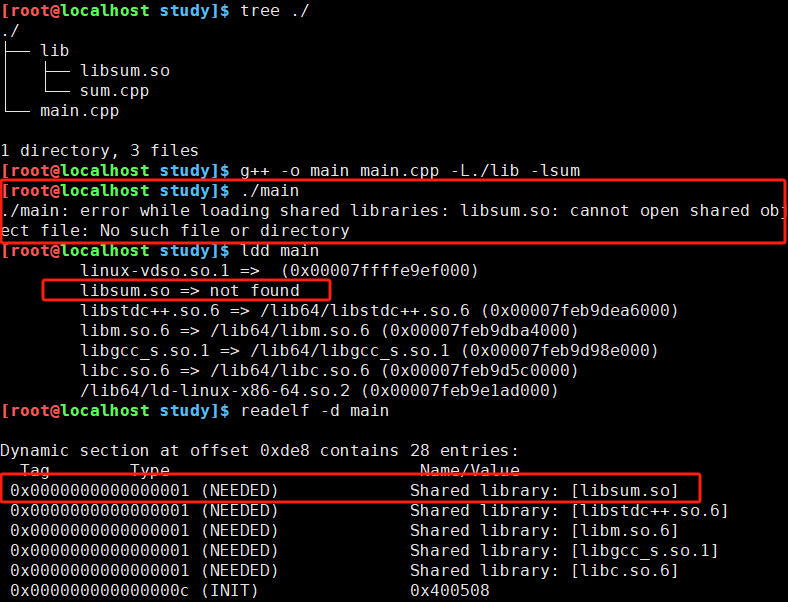
{

   printf("main execute\n");

   sum(3,4);

}

g++ -fpic -shared -o libsum.so sum.cpp

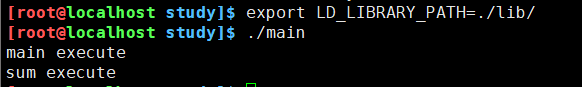


如上图，-l<库名>方式指定链接库，编译链接成功，但运行时报错找不到依赖库libsum.so

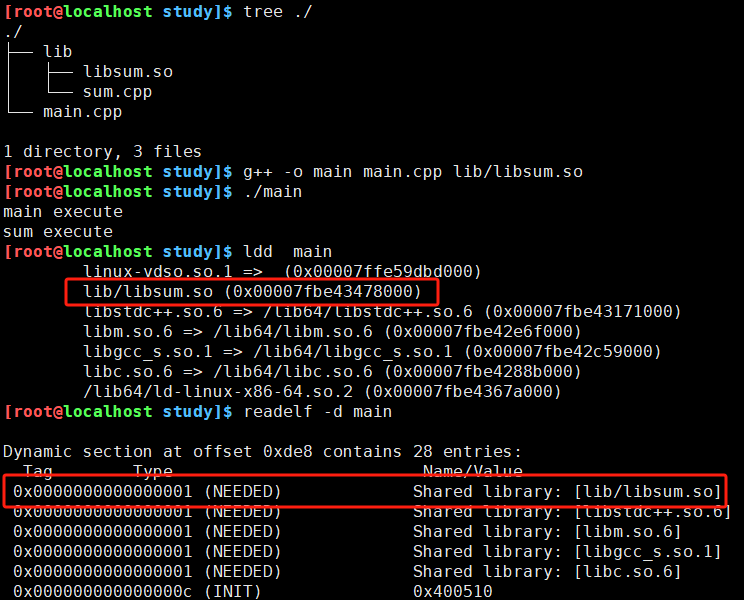
ldd验证，发现确实运行时会出现libsum.so将not found

readelf –d查看可执行文件内容，发现包含链接信息Shared library: [libsum.so]

要使运行正常，export LD\_LIBRARY\_PATH=./lib/指定运行时搜索依赖库的路径即可，如下



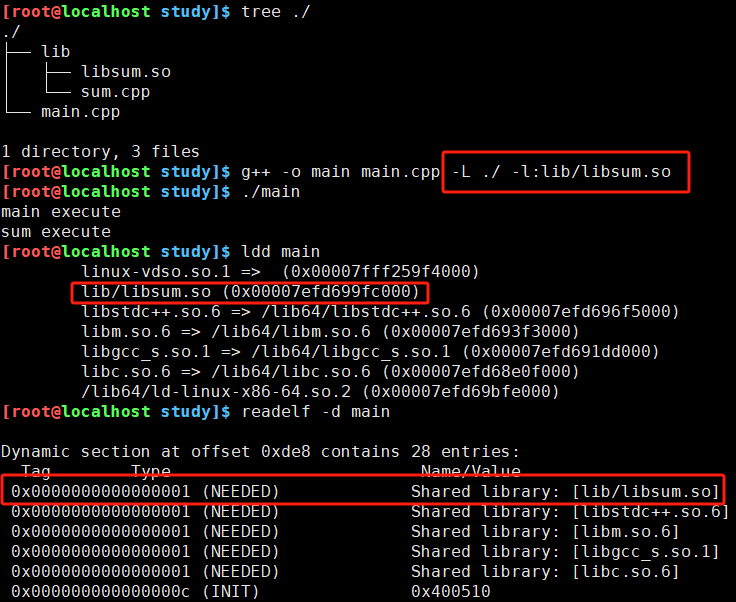
如果使用全路径方式指定要链接的库，则



如上图，使用全路径方式指定要链接的库，运行时无需export LD\_LIBRARY\_PATH指定依赖库路径

readelf –d查看，发现生成的可执行文件中，Shared library: [lib/libsum.so]，已经包含了库路径以及库文件名

如果使用-l:<库文件路径及名字>指定要链接的库，则



**Linux g++静态库/动态库依赖动态库**

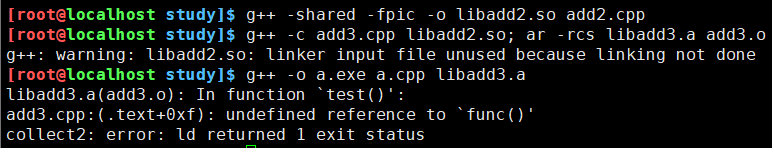
静态库依赖动态库（add3.cpp调用了add2.cpp的函数），然后exe使用静态库，则生成exe时必须链接动态库：

g++ -shared -fpic -o libadd2.so add2.cpp //生成动态库

g++ -c add3.cpp; ar -rcs libadd3.a add3.o //生成静态库，这里不链接libadd2.so

g++ -o a.exe a.cpp libadd3.a libadd2.so //注意这里必须链接libadd2.so

注意：生成静态库时，无法链接其依赖的动态库，如下图：



动态库依赖动态库（main.cpp-->test.cpp-->func.cpp）：

g++ -fpic -shared -o libfunc.so func.cpp

g++ -shared -fpic -o libtest.so test.cpp这里不链接libfunc.so

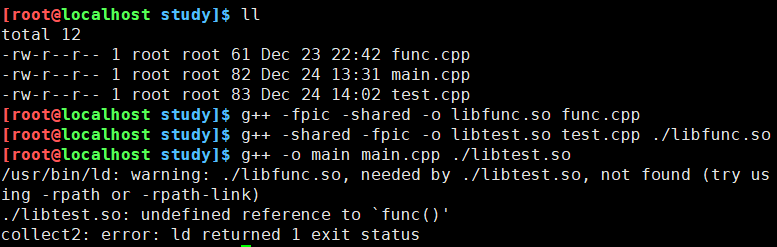
g++ -o main main.cpp ./libtest.so ./libfunc.so 这里必须既有libtestso又有libfunc.so

或者

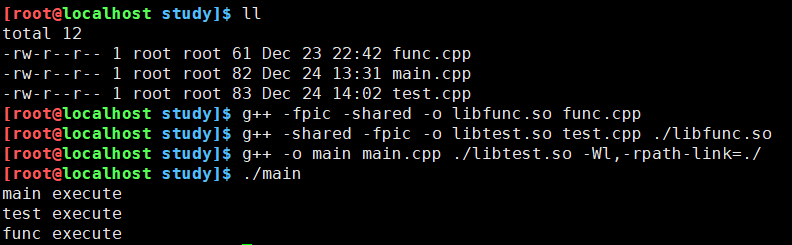
g++ -fpic -shared -o libfunc.so func.cpp

g++ -shared -fpic -o libtest.so test.cpp ./libfunc.so这里链接了libfunc.so

g++ -o main main.cpp ./libtest.so -Wl,-rpath-link=./ 这里只链接libtest.so，无需再链接libfunc.so，但必须用-Wl,-rpath-link参数指明间接依赖的libfunc.so库的路径



如上图，编译生成可执行文件时，发现可执行文件链接了./libtest.so，libtest.so链接了libfunc.so，但是无法找到libfunc.so



如上图，-rpath-link参数指定了间接依赖的库的查找目录，在该目录下成功找到了libfunc.so

[root@localhost study]$ ll

func.cpp

main.cpp

test.cpp

[root@localhost study]$ g++ -fpic -shared -o libfunc.so func.cpp

[root@localhost study]$ g++ -shared -fpic -o libtest.so test.cpp -L./ -lfunc

[root@localhost study]$ g++ -o main main.cpp -L./ -ltest

/usr/bin/ld: warning: libfunc.so, needed by .*//libtest.so, not found (try using -rpath or -rpath-link)*

.*//libtest.so: undefined reference to `func()'*

collect2: error: ld returned 1 exit status

[root@localhost study]$ g++ -o main main.cpp -L./ -ltest -Wl,-rpath-link=./

[root@localhost study]$ ./main

./main: error while loading shared libraries: libtest.so: cannot open sharedobject file: No such file or directory

[root@localhost study]$ ldd libtest.so

        linux-vdso.so.1 =>  (0x00007ffc0faa8000)

        libfunc.so => not found

[root@localhost study]$ ldd main

        linux-vdso.so.1 =>  (0x00007fffdd3c0000)

        libtest.so => not found

[root@localhost study]$ export LD\_LIBRARY\_PATH=./

[root@localhost study]$ ./main

main execute

test execute

func execute

[root@localhost study]$ ldd libtest.so

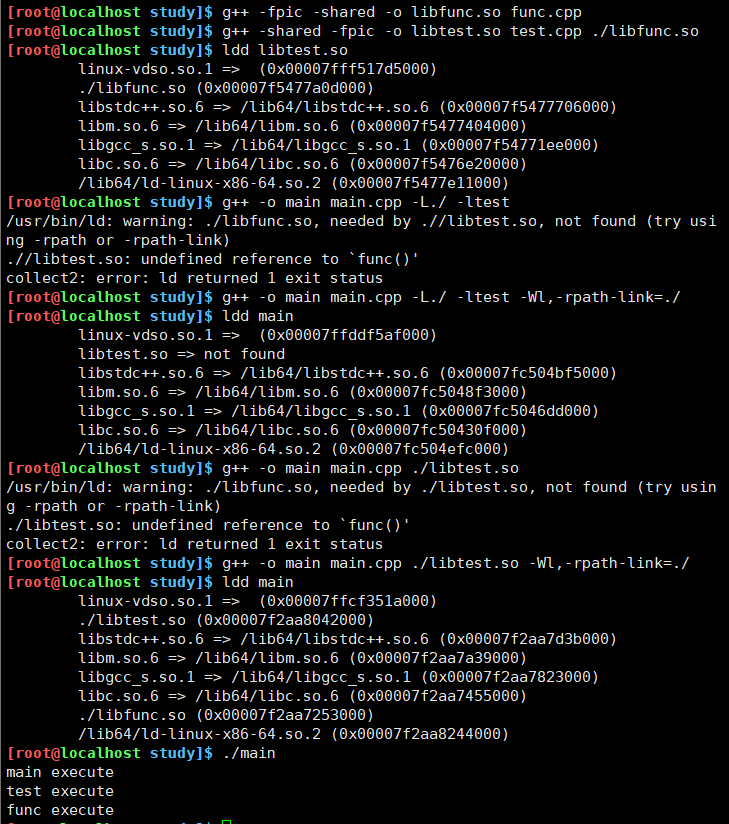
        linux-vdso.so.1 =>  (0x00007ffc9c5f2000)

        libfunc.so => ./libfunc.so (0x00007f70a58d6000)

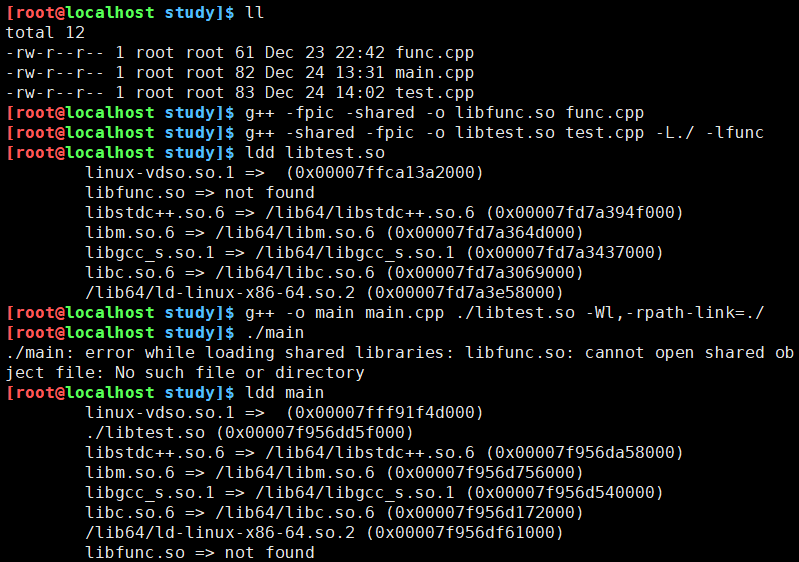
[root@localhost study]$ ldd main

        linux-vdso.so.1 =>  (0x00007ffdb93ab000)

        libtest.so => ./libtest.so (0x00007f9fdaf33000)



如上图，编译生成main时，如果-L./ -ltest链接，则运行时找不到libtest.so，需要额外 export LD\_LIBRARY\_PATH=./才可以，如果./libtest.so链接，则运行时即可直接找到libtest.so



如上图，编译生成正常，运行时main依赖的libtest.so可以找到，但是libtest.so依赖的libfunc.so找不到

**Linux静态库依赖动态库时的链接顺序：**

//动态库func.cpp

#include <cstdio>

void func() { printf("func execute\n"); }

//静态库test.cpp

#include <cstdio>

void func();

void test(){

printf("test execute\n");

func();

}

//main.cpp

#include <cstdio>

void test();

int main(){

printf("main execute\n");

test();

}

g++ -fpic -shared -o libfunc.so func.cpp

g++ -c test.cpp; ar -rsv libtest.a test.o

export LD\_LIBRARY\_PATH=./

在环境centos7.7.1908+gcc 4.8.5上：

g++ -o main main.cpp -L./ -ltest -lfunc，生成的main运行正常；

g++ -o main main.cpp -L./ -lfunc -ltest，生成的main运行正常

在环境ubuntu22.04+gcc11.2.0上：

g++ -o main main.cpp -L./ -ltest -lfunc，生成的main运行正常；

g++ -o main main.cpp -L./ -lfunc -ltest，生成时链接器报错，undefined reference to `func()'

**Linux g++强制动态\静态链接**

gcc -static -o main main.c libsum.a

注意：如果报错找不到libm、libc等库，则yum install glibc-static，yum install libstdc++-static安装静态库

指定了-static这个选项，gcc在链接接时对项目所有的依赖库都尝试去搜索名为lib<name>.a的静态库文件，完成静态链接，如果找不到就报错了。这里指的所有是不仅指显式指出的第三方库如libsum.a，还包括gcc编译器自带的库libgcc、libstdc++、libc、libm等

gcc默认对libgcc、libstdc++等库采用动态链接，使用的是动态库，如下图：

[root@localhost study]$ cat main.cpp

#include <cstdio>

void test();

int main(){

        printf("main execute\n");

}

[root@localhost study]$ g++ -o main main.cpp

[root@localhost study]$ ./main

main execute

[root@localhost study]$ ldd main

        linux-vdso.so.1 =>  (0x00007ffcf97b7000)

        libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007f66a4f6f000)

        libm.so.6 => /lib64/libm.so.6 (0x00007f66a4c6d000)

        libgcc\_s.so.1 => /lib64/libgcc\_s.so.1 (0x00007f66a4a57000)

        libc.so.6 => /lib64/libc.so.6 (0x00007f66a4689000)

        /lib64/ld-linux-x86-64.so.2 (0x00007f66a5276000)

[root@localhost study]$

gcc使用-Wl传递链接器参数，ld使用-Bdynamic强制链接动态库，-Bstatic强制链接静态库。

因此既有静态链接又有动态链接时，语法格式为：gcc ... -Wl,-Bstatic -l<static-lib> -Wl,-Bdynamic -l<dynamic-lib> ...

如g++ -o main main.cpp -L./ -Wl,-Bstatic -lA -Wl,-Bdynamic –lB -lC

注意，-Wl,-Bstatic或者-Wl,-Bdynamic标记之后的链接库都将按照前面最近的一个标记进行链接，所以上述语句中libC将会被动态链接

如果编译参数里面没强制指定链接方式，那么gcc将按照默认优先级去链接，即优先动态链接

由于-B链接标记会改变默认链接方式，所以Makefile里面如果这么写：LIBS += -Wl,-Bstatic -lC

那么它后面的LIBS+=的库就都只能以静态方式链接了，正确做法是:

LIBS += -l<auto-link-lib>

STATIC\_LIBS += -l<static-lib>

DYN\_LIBS += -l<dynamic-lib>

LDFLAGS := ${LIBS} -Wl,-Bstatic ${STATIC\_LIBS} -Wl,-Bdynamic ${DYN\_LIBS}

**Linux动态库三种名称：**

在Linux下一个动态库有三种名称：

* + 1. real name真名 ： lib + 链接库名字 + .so + .版本号/次版本号/发行号等。创建一个动态库时使用-o参数指定真名，即生成文件名，如g++ -fpic -shared -o libtest.so.0.0.1 test.cpp
    2. linker name 链接名：lib + 链接库名字 + .so。当使用命令g++ -o main main.cpp -L./ -lpthread在进行链接时, 链接器ld会去寻找libpthread.so和libpthread.a
    3. soname别名：别名由-Wl,-soname参数指定，如g++ -fpic -shared -Wl,-soname,libtest.so -o libtest.so.0.0.1 test.cpp

-Wl代表后面的这个参数是一个链接器参数，而不是g++参数

-soname指定了别名, 那么生成的动态库文件中Dynamic section中的SONAME将会被填入输入的别名. 如果没有-soname参数, 则Dynamic section中将不会有SONAME字段生成.

readelf –d查看动态库SONAME字段



ldconfig -vn ./ 命令将为当前目录下的动态库生成软链接，软链接名为soname

-v：显示详细信息

-n：仅对指定目录下的lib\*.so.\*文件起作用，加上-n参数后不会更改刷新缓存文件/etc/ld.so.cache



执行ldconfig命令后，当前目录下生成了一个新的文件libtest.so，是一个软链接文件（这是-soname,libtest.so选项导致的，如果没有该选项，那么ldconfig执行后不会生成libhello.so。

**Linux编译时搜索动态库顺序：**

g++ -o main main.cpp -L./ -lpthread编译时搜索动态库的顺序：

1. 编译命令里指定顺序的目录，如-L
2. 系统的环境变量LIBRARY\_PATH指定的动态库搜索路径
3. 内定目录 /lib、/usr/lib、/usr/local/lib 这是当初编译gcc时写在程序内的

**Linux编译时搜索头文件顺序：**

对#include"file.h"，顺序如下：

1. 搜索当前目录
2. 搜索-I指定的目录
3. 搜索gcc的环境变量CPLUS\_INCLUDE\_PATH（C++编译）、C\_INCLUDE\_PATH（C编译）指定的路径

如export C\_INCLUDE\_PATH=XXXX:$C\_INCLUDE\_PATH

如export CPLUS\_INCLUDE\_PATH=XXX:$CPLUS\_INCLUDE\_PATH

1. 搜索gcc的头文件默认搜索目录/usr/include、/usr/local/include、/usr/lib/gcc/x86\_64-redhat-linux/4.1.1/include。即`gcc -print-prog-name=cc1plus` -v或`g++ -print-prog-name=cc1plus` -v显示的目录

对于#include<file.h>，除了不在当前目录下搜索外，其余路径顺序与#include"file.h"一致



**Linux可执行程序搜索动态库顺序：**

可执行文件运行时，linux系统按如下顺序搜索依赖的so：

1. 编译目标代码时指定的动态库搜索路径（-rpath）；
2. 环境变量LD\_LIBRARY\_PATH指定的动态库搜索路径；
3. 配置文件/etc/ld.so.conf中指定的动态库搜索路径；
4. 默认的动态库搜索路径/lib、/usr/lib

详细：

（1）-rpath指定的路径会被记录在生成的可执行程序中

g++ -o main main.cpp libtest.so -Wl,-rpath=/home/c++可以

g++ -o main main.cpp -L./ -ltest -Wl,-rpath=/home/c++可以编译成功，但实际运行时还是找不到

（2）修改环境变量LD\_LIBRARY\_PATH

编辑.bashrc文件，在末尾增加export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/usr/local/lib

执行source .bashrc

如果只是临时使用，则直接在bash输入export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/usr/local/lib回车即可

注意：共享库搜索路径检查环境变量LD\_LIBRARY\_PATH，而不是变量LD\_LIBRARY\_PATH

如果在bash输入LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/usr/local/lib，不带export，如果此时环境变量里有LD\_LIBRARY\_PATH，则追加有效；如果LD\_LIBRARY\_PATH是变量但不是环境变量或既不是环境变量也不是变量，则执行后LD\_LIBRARY\_PATH仍不是环境变量，仍不会在此路径搜素动态库

（3）

/etc/ld.so.cache是ldconfig程序读取动态库配置文件/etc/ld.so.conf文件生成的

ldconfig命令的用途, 主要是在默认搜寻目录(/lib和/usr/lib)以及动态库配置文件/etc/ld.so.conf所指定的目录下,

搜索动态链接库(格式如lib\*.so\*), 进而创建出动态装入程序(ld.so)所需的软链接和缓存文件.

缓存文件默认为/etc/ld.so.cache, 此文件保存已排好序的动态链接库名字列表.

g++ -fpic -shared -Wl,-soname,libtest.so -o libtest.so.0.0.1 test.cpp生成动态库libtest.so.0.0.1之后：



然后可执行文件main无论拷贝到系统任何地方，都可以执行，因为ldconfig将libtest.so的位置写入了/etc/ld.so.cache

**coredump file：**

ulimit -a：列出系统所有资源限制的值

ulimit -c：查看core file的最大大小

ulimit -c unlimited：设置core file大小可无限大

//程序中设置当前进程的core file限制：

struct rlimit core\_limits;

core\_limits.rlim\_cur = RLIM\_INFINITY;

core\_limits.rlim\_max = RLIM\_INFINITY;

setrlimit(RLIMIT\_CORE,&core\_limits);

**符号表与可执行程序（或DSO）分离**

g++ -o test\_release test.c然后查看所有节的统计信息（readelf -S -W test\_release），如下图：



由上图可知符号表还在，nm test\_release查看符号表，如下：



ulimit -c unlimited，然后./test\_release产生coredump文件，然后gdb -c core ./test\_release调试coredump文件，发现堆栈还在，但无行号、源码、局部变量符号信息，如下图：



g++ -g -o test\_debug test.c，然后readelf -S -W test\_debug，发现多了.debug\_info、.debug\_line、.debug\_str等五个debug相关的section，如下图：



ulimit -c unlimited，然后./test\_debug产生coredump文件，然后gdb -c core ./test\_debug调试coredump文件，堆栈、行号、源码、局部变量符号等信息都正常，如下图：



移除debug相关的section，strip --strip-debug test\_debug -o test\_debug\_stripdebug然后readelf -S -W test\_release查看所有节统计信息，发现符号表仍在，输出同不加-g参数的编译结果，如下图：



移除debug相关section和符号表，strip test\_debug -o test\_debug\_stripall然后readelf -S -W test\_debug\_stripall查看所有节统计信息，输出如下图：



nm test\_debug\_stripall查看符号表，显示no symbols

ulimit -c unlimited，然后./test\_debug\_stripall产生coredump文件，然后gdb -c core ./test\_debug\_stripall调试coredump文件，堆栈都不能显示，如下图：



使用eu-strip分离出符号表，调试coredump文件时再加载

eu-strip test\_debug -f test\_debug.sym -o test\_debug\_eustrip

生成的test\_debug.sym中有符号表

查看test\_debug\_eustrip的所有节统计信息，发现没有debug相关section和符号表，但多了.gnu\_debuglink这个section，gnu\_debuglink中记录的是.sym的位置

ulimit -c unlimited，然后./test\_debug\_eustrip产生coredump文件，然后gdb -c core ./test\_debug\_eustrip调试coredump文件，堆栈都不能显示，如下图：



gdb中用symbol-file <sym文件完整路径>加载符号表

或者直接将符号表置于默认查找位置，

strace gdb -c core.19179 ./test\_debug\_eustrip，显示有如下几行，即为符号表的默认查找位置，包括当前目录、./.debug/目录、/usr/lib/debug/xxx目录



# 创建可执行的共享库:

1. 普通共享库生成：

g++ -o -shared -fpic -o libhello.so hello.cpp

1. gcc支持生成可执行的共享库

[root@localhost study]$ cat hello.cpp

#include <stdio.h>

void hello() { printf("execute hello\n");}

int main(void)

{

    printf("version: 1.0.0\n");

}

[root@localhost study]$ g++ -pie -fpie -rdynamic -o libhello.so hello.cpp

[root@localhost study]$ ./libhello.so

version: 1.0.0

[root@localhost study]$ cat main.cpp

#include<cstdio>

void hello();

int main()

{

    printf("execute main\n");

    hello();

}

[root@localhost study]$ g++ -o main main.cpp -L./ -lhello

[root@localhost study]$ ./main

execute main

execute hello

上述编译选项简介：

-rdynamic Pass the flag -export-dynamic to the ELF linker, on targets that support it. This instructs the linker to add all symbols, not only used ones, to the dynamic symbol table.

-pie Produce a position independent executable on targets that support it.

-fpie These options are similar to -fpic and -fPIC, but generated position independent code can be only linked into executables.

如果不加-rdynamic选项，则由于hello.cpp中hello()函数未被用到，所以hello()未被添加到动态符号表，后文g++ -o main main.cpp -L./ -lhello时会报错undefined reference to `hello()'

[root@localhost study]$ g++ -pie -fpie -rdynamic -o libhello.so hello.cpp

[root@localhost study]$ nm -D libhello.so

0000000000000930 T \_\_libc\_csu\_init

                 U \_\_libc\_start\_main

0000000000000917 T main

                 U puts

00000000000007f0 T \_start

0000000000000905 T \_Z5hellov

[root@localhost study]$ g++ -pie -fpie -o libhello.so hello.cpp

[root@localhost study]$ nm -D libhello.so

                 U \_\_libc\_start\_main

                 U puts

1. 其它实现

[root@localhost study]$ cat hello2.cpp

#include <stdio.h>

#include <unistd.h>

int main(void)

{

printf("hello\n");

\_exit(0);

}

[root@localhost study]$ g++ -shared -fpic -o libhello2.so hello2.cpp -Wl,-emain

[root@localhost study]$ /lib64/ld-linux-x86-64.so.2 ./libhello2.so

hello

[root@localhost study]$ g++ -o hello2.noc -L./ libhello2.so

[root@localhost study]$ ./hello2.noc

hello

上述编译选项简介：

-Wl,-emain，指定入口函数为main

/lib64/ld-linux-x86-64.so.2是链接器ld的运行时组件，借助ldd可以查看ld依赖了ld-linux-x86-64.so.2

[root@localhost study]$ which ld

/usr/bin/ld

[root@localhost study]$ ldd /usr/bin/ld

linux-vdso.so.1 => (0x00007ffee7931000)

libbfd-2.27-41.base.el7.so => /usr/lib64/libbfd-2.27-41.base.el7.so (0x00007ff537295000)

libdl.so.2 => /usr/lib64/libdl.so.2 (0x00007ff537091000)

libc.so.6 => /usr/lib64/libc.so.6 (0x00007ff536cc3000)

/lib64/ld-linux-x86-64.so.2 (0x00007ff5375dc000)

ld是可执行文件：

[root@localhost study]$ file /usr/bin/ld

/usr/bin/ld: symbolic link to `/etc/alternatives/ld'

[root@localhost study]$ file /etc/alternatives/ld

/etc/alternatives/ld: symbolic link to `/usr/bin/ld.bfd'

[root@localhost study]$ file /usr/bin/ld.bfd

/usr/bin/ld.bfd: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.32, BuildID[sha1]=f07d5729233750f599de81f8699d469121edd4ce, stripped

ldd是可执行脚本

[root@localhost study]$ file /usr/bin/ldd

/usr/bin/ldd: Bourne-Again shell script, ASCII text executable

**全局符号介入：**

elf文件的.dynamic段中的NEEDED域指出的是当前可执行文件（或动态库）所依赖的共亨对象

readelf –d <file>查看.dynamic段的内容

示例：

*//a1.c*

#include<stdio.h>

void a()

{

        printf("a1.c\n");

}

*//b1.c*

void a();

void b1()

{

        a();

}

*//a2.c*

#include<stdio.h>

void a()

{

        printf("a2.c\n");

}

*//b2.c*

void a();

void b2()

{

        a();

}

*//main.c*

void b1();

void b2();

int main()

{

        b1();

        b2();

}

编译指令：

gcc -fpic -shared -o a1.so a1.c

gcc -fpic -shared -o a2.so a2.c

gcc -fpic -shared -o b1.so b1.c ./a1.so

gcc -fpic -shared -o b2.so b2.c ./a2.so

gcc -o main main.c ./b1.so ./b2.so

export LD\_LIBRARY\_PATH=.

运行结果：

[root]$ readelf -d b1.so

Dynamic section at offset 0x2e50 contains 21 entries:

  Tag        Type                         Name/Value

 0x0000000000000001 (NEEDED)             Shared library: [./a1.so]

 0x000000000000000c (INIT)               0x1000

 0x000000000000000d (FINI)               0x1130

[root]$ readelf -d main

Dynamic section at offset 0x2da0 contains 29 entries:

  Tag        Type                         Name/Value

 0x0000000000000001 (NEEDED)             Shared library: [./b1.so]

 0x0000000000000001 (NEEDED)             Shared library: [./b2.so]

 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]

 0x000000000000000c (INIT)               0x1000

 0x000000000000000d (FINI)               0x1208

[root]$ ./main

a1.c

a1.c

程序文件装载时，链接器根据.dynamic段的NEEDED域列出可执行文件所直接依赖的所有共享对象，然后依次找到相应的文件，并装载到进程的内存空间。

如果装载的ELF共享对象还直接依赖于其他共亨对象，那么继续将其所依赖的共享对象装载。如此循环直到所有依赖的共享对象都被装载进来为止。

当然，链接器可以有不冋的装载顺序，装载过程实际就是一个图的遍历过程，链接器常见的算法是广度优先

当一个新的共享对象被装载进来的时候，它的符号表会被合并到全局符号表中

当一个符号需要被加入全局符号表时，如果相同的符号名已经存在，则后加入的符号会被忽略，这被称为全局符号介入。

上述示例中，模块装载顺序为main->b1->b2->a1->a2，装载a2时，发现函数a()已存在于全局符号表中，因此忽略a2中的函数a()，因此程序打印结果为a1.c、a1.c

LD\_DEBUG=bindings查看符号绑定，如下：

[root@DESKTOP-UN1P95E 程序员的自我修养]$ LD\_DEBUG=bindings ./main

507:     binding file ./main [0] to ./b2.so [0]: normal symbol `b2'

507:     binding file ./main [0] to ./b1.so [0]: normal symbol `b1'

507:     transferring control: ./main

507:     binding file ./b1.so [0] to ./a1.so [0]: normal symbol `a'

507:     binding file ./b2.so [0] to ./a1.so [0]: normal symbol `a'

……

符号b2、b1是在应用程序获取控制权之前绑定

前两行表明 main 程序绑定到了 b2.so 中的 b2符号、b1.so中的b1符号

"transferring control: ./main"表示动态链接器 (ld.so) 完成了所有必要的符号绑定和初始化工作后，将程序的控制权转交给主程序

后两行表明b1.so、b2.so都绑定到了 a1.so 中的 a符号，是延迟绑定，在应用程序获得控制权后第一次调用函数的时绑定

要避免全局符号介入，可使用static定义编译单元私有函数，如下：

*//c1.c*

#include <stdio.h>

static void help() { printf("c1 help\n"); }

void ok() { printf("c1 ok\n"); }

void c1() { printf("c1 call help:");  help(); ok(); }

*//cat c2.c*

#include <stdio.h>

static void help() { printf("c2 help\n"); }

void ok() { printf("c2 ok"); }

void c2() { printf("c2 call help:"); help(); ok(); }

*//c\_main.c*

#include <stdio.h>

void c1();

void c2();

void ok() { printf("main ok\n"); }

int main() {

    c1();

    c2();

}

*//编译指令*

gcc -fpic -shared -o c1.so c1.c

gcc -fpic -shared -o c2.so c2.c

gcc -o c\_main c\_main.c ./c1.so ./c2.so

*//运行结果：*

c1 call help:c1 help

main ok

c2 call help:c2 help

main ok

**多个链接库存在同名函数：**

1. 可重定位.o目标文件

[root@localhost study]$ cat test1.cpp

*// test1.cpp*

#include <stdio.h>

void test() {

    printf("call from test1.cpp\n");

}

[root@localhost study]$ cat test2.cpp

*// test2.cpp*

#include <stdio.h>

void test() {

    printf("call from test2.cpp\n");

}

[root@localhost study]$ cat main.cpp

*// main.c*

extern void test();

int main() {

    test();

}

[root@localhost study]$ g++ -o main main.cpp test1.cpp test2.cpp

/tmp/ccTJ9H6t.o: In function `test()':

test2.cpp:(.text+0x0): multiple definition of `test()'

/tmp/cczhnAYn.o:test1.cpp:(.text+0x0): first defined here

collect2: error: ld returned 1 exit status

1. 静态库

[root@localhost study]$ g++ -c test1.cpp

[root@localhost study]$ g++ -c test2.cpp

[root@localhost study]$ ar -rcs libtest1.a test1.o

[root@localhost study]$ ar -rcs libtest2.a test2.o

[root@localhost study]$ g++ -o main\_static main.cpp -L./ -ltest1 -ltest2

[root@localhost study]$ ./main\_static

call from test1.c

[root@localhost study]$ g++ -o main\_static main.cpp -L./ -ltest2 -ltest1

[root@localhost study]$ ./main\_static

call from test2.c

静态库链接，由前向后扫描进行符号解析，当前静态库提供了某个未解析符号的定义，才会被链接

1. 动态库

[root]$ g++ -shared -fpic -o libtest1.so test1.cpp

[root]$ g++ -shared -fpic -o libtest2.so test2.cpp

[root]$ g++ -o main\_dyna main.cpp ./libtest1.so ./libtest2.so

[root]$ ./main\_dyna

call from test1.c

LD\_DEBUG=bindings ./main\_dyna会打印出

binding file ./main\_dyna [0] to ./libtest1.so [0]: normal symbol `\_Z4testv'

[root]$ g++ -o main\_dyna main.cpp ./libtest2.so ./libtest1.so

[root]$ ./main\_dyna

call from test2.c

LD\_DEBUG=bindings ./main\_dyna会打印出

binding file ./main\_dyna [0] to ./libtest2.so [0]: normal symbol `\_Z4testv'

**违反ODR原则导致stack buffer overflow**

在 C/C++ 中当存在两个同名函数时， 如果它们都是外部编译单元可见的（external linkage）、但函数实现的内容不一样， 就有可能被链接器错误的链接， 导致程序运行阶段结果错误或者非预期停止（发生“crash”）。

对于同名但取值不同的全局变量也有类似问题。

ODR 原则是说要有唯一定义, one definition rule。

*//main.cpp*

#include <stdio.h>

#include <string.h>

struct Rect{

    Rect() : score(1), x(0), y(0), w(0), h(0) {

*//printf("main rect construct\n");*

    }

    int score, x, y, w, h;

};

extern void cpp\_func();

int main(void){

    Rect r;

*//printf( "test, %d.\n", r.x );*

    cpp\_func();

    return 0;

}

*//other.cpp*

#include <stdio.h>

struct Rect{

    Rect() : x(233), y(0), w(0), h(0) {

*//printf("other rect construct\n");*

    }

    int x, y, w, h;

};

void cpp\_func(){

    Rect r;

*//printf( "cpp\_func: %d.\n", r.x );*

}

g++ -g -o main main.cpp other.cpp

./main，则运行崩溃

分析：

将上述编译指令拆分为两条，

g++ -g -o other.o -c other.cpp

g++ -g -o main main.cpp other.o

gdb运行，查看生成的可执行文件的反汇编：

[root@localhost test]$ gdb ./main

(gdb) b main

(gdb) r

(gdb) set disassembly-flavor intel

(gdb) disas main

Dump of assembler code for function main():

   0x000000000040050d <+0>:     push   rbp

   0x000000000040050e <+1>:     mov    rbp,rsp

   0x0000000000400511 <+4>:     sub    rsp,0x20

=> 0x0000000000400515 <+8>:     lea    rax,[rbp-0x20]

   0x0000000000400519 <+12>:    mov    rdi,rax

   0x000000000040051c <+15>:    call   0x40052e <Rect::Rect()>

   0x0000000000400521 <+20>:    call   0x40056e <cpp\_func()>

   0x0000000000400526 <+25>:    mov    eax,0x0

   0x000000000040052b <+30>:    leave

   0x000000000040052c <+31>:    ret

End of assembler dump.

(gdb) disas cpp\_func

Dump of assembler code for function cpp\_func():

   0x000000000040056e <+0>:     push   rbp

   0x000000000040056f <+1>:     mov    rbp,rsp

   0x0000000000400572 <+4>:     sub    rsp,0x10

   0x0000000000400576 <+8>:     lea    rax,[rbp-0x10]

   0x000000000040057a <+12>:    mov    rdi,rax

   0x000000000040057d <+15>:    call   0x40052e <Rect::Rect()>

   0x0000000000400582 <+20>:    leave

   0x0000000000400583 <+21>:    ret

End of assembler dump.

(gdb) disas Rect::Rect

Dump of assembler code for function Rect::Rect():

   0x000000000040052e <+0>:     push   rbp

   0x000000000040052f <+1>:     mov    rbp,rsp

   0x0000000000400532 <+4>:     mov    QWORD PTR [rbp-0x8],rdi

   0x0000000000400536 <+8>:     mov    rax,QWORD PTR [rbp-0x8]

   0x000000000040053a <+12>:    mov    DWORD PTR [rax],0x1

   0x0000000000400540 <+18>:    mov    rax,QWORD PTR [rbp-0x8]

   0x0000000000400544 <+22>:    mov    DWORD PTR [rax+0x4],0x0

   0x000000000040054b <+29>:    mov    rax,QWORD PTR [rbp-0x8]

   0x000000000040054f <+33>:    mov    DWORD PTR [rax+0x8],0x0

   0x0000000000400556 <+40>:    mov    rax,QWORD PTR [rbp-0x8]

   0x000000000040055a <+44>:    mov    DWORD PTR [rax+0xc],0x0

   0x0000000000400561 <+51>:    mov    rax,QWORD PTR [rbp-0x8]

   0x0000000000400565 <+55>:    mov    DWORD PTR [rax+0x10],0x0

   0x000000000040056c <+62>:    pop    rbp

   0x000000000040056d <+63>:    ret

End of assembler dump.

(gdb)

查看other.cpp编译生成的可重定位目标文件的代码段：

[root@localhost test]$ objdump -M intel -d other.o

Disassembly of section .text:

0000000000000000 <\_Z8cpp\_funcv>:

   0:   55                      push   rbp

   1:   48 89 e5                mov    rbp,rsp

   4:   48 83 ec 10             sub    rsp,0x10

   8:   48 8d 45 f0             lea    rax,[rbp-0x10]

   c:   48 89 c7                mov    rdi,rax

   f:   e8 00 00 00 00          call   14 <\_Z8cpp\_funcv+0x14>

  14:   c9                      leave

  15:   c3                      ret

Disassembly of section .text.\_ZN4RectC2Ev:

0000000000000000 <\_ZN4RectC1Ev>:

   0:   55                      push   rbp

   1:   48 89 e5                mov    rbp,rsp

   4:   48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi

   8:   48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]

   c:   c7 00 e9 00 00 00       mov    DWORD PTR [rax],0xe9

  12:   48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]

  16:   c7 40 04 00 00 00 00    mov    DWORD PTR [rax+0x4],0x0

  1d:   48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]

  21:   c7 40 08 00 00 00 00    mov    DWORD PTR [rax+0x8],0x0

  28:   48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]

  2c:   c7 40 0c 00 00 00 00    mov    DWORD PTR [rax+0xc],0x0

  33:   5d                      pop    rbp

  34:   c3                      ret

C++语句Rect r;转为汇编相当于以下两句，

1. 在栈上开辟空间sizeof(Rect)
2. 调用函数，即Rect的构造函数，给分配的空间中各个地址处赋值，即给结构体的各个成员函数赋值

编译阶段，生成other.o时，编译器根据other.cpp中的struct Rect定义得知需要在栈上分配多大空间

链接阶段，对于other.cpp和main.cpp中两个同名的Rect符号，链接器选择其中后者，这时候生成的可执行文件里，cpp\_func()函数对应的汇编语句变为，

1. 根据other.cpp中的Rect定义在栈上开辟一定大小的空间
2. 根据main.cpp中的Rect构造函数为这块空间赋值

因而出现stack buffer overflow问题。具体的，看gdb反汇编可以发现，cpp\_func的栈基址和栈顶指针之间的空间总共才16字节，但是调用main.cpp中的Rect构造函数时，该函数中向20字节的空间写数据，明显的栈溢出。

**ldd、nm、objdump、readelf：**

nm -D input.apf查看符号表，-D显示动态符号适用于动态库，对可执行文件则nm main.exe

ldd input.apf查看依赖库

pldd <pid>：显示进程运行时加载的所有动态库

查看tls类库依赖情况：pldd $PID | grep -E "tls|ssl|nspr|nss"

objdump -d <file>：显示executable section的反汇编代码，包括.text段、.plt段等

objdump -j <name> <file>：仅显示名为name的段的信息

objdump -d -j .plt <file>：显示.plt段的反汇编代码，即plt表的内容

objdump -R <file>：显示dynamic relocation entries，即got表

objdump -t libtest.a 查看可重定位目标文件或静态库或动态库的符号表信息

readelf -h main.o查看ELF文件的elf头信息

readelf -S -W main.o查看ELF文件的节头部表信息，即所有节的统计信息，如节偏移、size等

**objdump -t查看ELF文件符号表：**

可重定位目标文件即windows下的obj文件、linux下gcc –c生成的.o文件

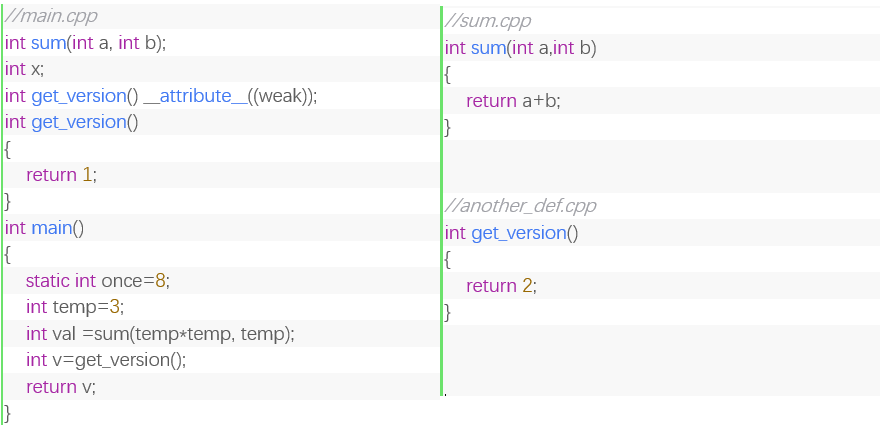
gcc -c main.c，则生成gcc.o

查看符号表信息： objdump -t main.o

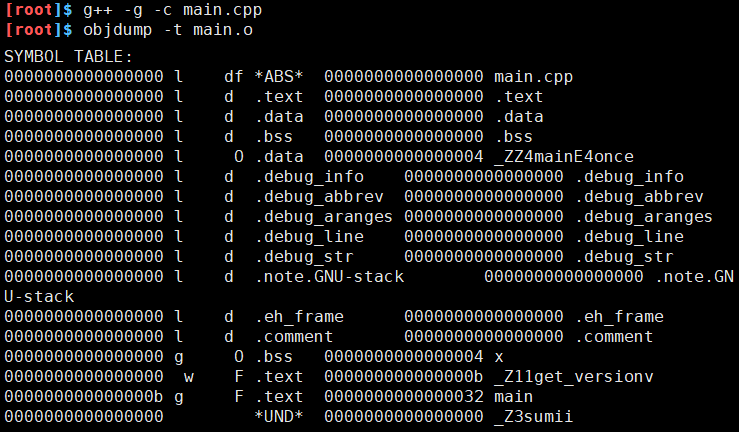
可重定位目标文件的符号表记录以下3种符号：

1. 本模块定义的全局变量/全局函数，这些也可供外部模块使用
2. 本模块引用的外部符号（由其它模块定义）
3. static全局变量/ static全局变量函数/static局部变量，本模块定义，仅本模块可用

可重定位目标文件的符号表不记录局部非静态变量，链接器对这些符号不感兴趣，在运行时在栈中被管理



objdump -t查看符号表：



符号表中记录的符号的信息包括但不限于以下：

1. 符号名，上图中最后一列，注意局部static变量的符号名由编译器分配，与用户使用的变量名不一致
2. g或l或u或!或空格，g表示符号是全局的，可供其它模块使用，l表示符号是本地的，如static全局变量/static全局函数/static局部变量
3. w或空格，w表示弱符号，空格表示强符号，由于弱符号的机制，上述程序运行后main函数返回值为2而不是1
4. d或D或空格，d表示debug符号，D表示动态符号
5. f或F或O或空格，f表示文件，F表示函数，O表示变量
6. 符号所在节，如果是引用的外部符号，则为\*UND\*
7. 符号在所属节中的偏移，上图中倒数第二列

**静态链接符号解析、重定位分析：**

链接器符号解析，将可重定位文件中每一个符号引用与一个符号表表项对应起来

链接器将所有可重定位目标文件同类型的节进行合并。例如，来自所有输人模块的.data节被全部合并成一个节，这个节成为输出的可执行目标文件的.data节。

节合并后，所有符号的内存地址则已知

* + 1. 对于内部符号引用，节合并后，引用地址更新简单

我们假设链接器需要链接三个目标文件：

* 目标文件一：该文件数据段定义了两个变量apple和banana，apple的长度为2字节，banana的长度4字节，因此目标文件一的数据段长度为6字节。从图中也可以看出apple的内存地址为0，也就是相对地址，即apple这个变量在目标文件一的地址是0，banana的地址为2。
* 目标文件二：该文件的数据段比较简单，只定义了一个变量orange，其长度为2，因此该目标文件的数据段长度为2。
* 目标文件三：该文件的数据段定义了三个变量grape、mango以及limo，其长度分别为4字节、2字节以及2字节，因此该目标文件的数据段长度为8字节。

链接器在链接三个目标文件时其顺序是依次链接的，链接完成后：

* 目标文件一：该数据段的起始地址为0，因此该数据段中的变量的最终地址不变。
* 目标文件二：由于目标文件一的数据段长度为6，因此链接完成后该数据段的起始地址为6(这里的起始地址其实就是偏移offset)，相应的orange的最终内存地址为0+offset即6。
* 目标文件三：由于前两个数据段的长度为8，因此该数据段的起始地址为8(即offset为8)，因此所有该数据段中的变量其地址都要加上该offset，即grape的最终地址为8，即0+offset，mango的最终地址为4+offset即12，limo的最终地址为6+offset即14。
  + 1. 对于外部符号引用，编译器将机器指令中的引用地址设置为空(比如call 0x000000)，并将该信息记录在了可重定位目标文件的.rel.text或.rel.data节中。链接器节合并后，遍历所有rel.text以及.rel.data段，并找到对应符号的最终内存地址，将机器指令中的0x000000修正为所引用符号的最终地址

如函数调用，即call指令，参数为地址，如果是调用的外部函数，则在rel.text段中对应一个表项

如语句void\* ptr=&func，ptr是全局变量，初始化值func是外部函数，则对应rel.data段中对应一个表项

如语句printf(“%p”,ptr)=0，ptr是指针类型的外部变量，则在rel.text段中对应一个表项

可执行文件中代码以及数据的运行时内存地址是链接器指定的

.rel.text: —个.text节中位置的列表，当链接器把这个目标文件和其他文件链接时，需要修改这些位置。一般而言，任何调用外部函数或者引用外部变量的指令都需要修改。另一方面，调用本地函数的指令则不需要修改。注意 ，可执行目标文件中并不需要重定位信息，因此通常省略，除非用户显式地指示链接器包含这些信息。

.rel.data: —个.data节中位置的列表。一般而言，任何已初始化的全局变量，如果它的初始值是一个全局变量地址或者外部定义函数的地址，都需要被修改

[root@localhost test]$ cat main.c

*//main.c*

int sum(int,int,int);

int aaa=5;

static int bbb=6;

int dosth()

{

    static int ccc=7;

    aaa+=1;

    bbb+=2;

    ccc+=3;

    int val=sum(aaa,bbb,ccc);

    return val;

}

[root@localhost test]$ g++ -o main.o -c main.c

[root@localhost test]$ objdump -d main.o

Disassembly of section .text:

0000000000000000 <\_Z5dosthv>:

   0:   55                      push   %rbp

   1:   48 89 e5                mov    %rsp,%rbp

   4:   48 83 ec 10             sub    $0x10,%rsp

   8:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # e <\_Z5dosthv+0xe>

   e:   83 c0 01                add    $0x1,%eax

  11:   89 05 00 00 00 00       mov    %eax,0x0(%rip)        # 17 <\_Z5dosthv+0x17>

  17:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 1d <\_Z5dosthv+0x1d>

  1d:   83 c0 02                add    $0x2,%eax

  20:   89 05 00 00 00 00       mov    %eax,0x0(%rip)        # 26 <\_Z5dosthv+0x26>

  26:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 2c <\_Z5dosthv+0x2c>

  2c:   83 c0 03                add    $0x3,%eax

  2f:   89 05 00 00 00 00       mov    %eax,0x0(%rip)        # 35 <\_Z5dosthv+0x35>

  35:   8b 15 00 00 00 00       mov    0x0(%rip),%edx        # 3b <\_Z5dosthv+0x3b>

  3b:   8b 0d 00 00 00 00       mov    0x0(%rip),%ecx        # 41 <\_Z5dosthv+0x41>

  41:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 47 <\_Z5dosthv+0x47>

  47:   89 ce                   mov    %ecx,%esi

  49:   89 c7                   mov    %eax,%edi

  4b:   e8 00 00 00 00          callq  50 <\_Z5dosthv+0x50>

  50:   89 45 fc                mov    %eax,-0x4(%rbp)

  53:   8b 45 fc                mov    -0x4(%rbp),%eax

  56:   c9                      leaveq

  57:   c3                      retq

[root@localhost test]$ objdump -r main.o

RELOCATION RECORDS FOR [.text]:

OFFSET           TYPE              VALUE

000000000000000a R\_X86\_64\_PC32     aaa-0x0000000000000004

0000000000000013 R\_X86\_64\_PC32     aaa-0x0000000000000004

0000000000000019 R\_X86\_64\_PC32     .data

0000000000000022 R\_X86\_64\_PC32     .data

0000000000000028 R\_X86\_64\_PC32     .data+0x0000000000000004

0000000000000031 R\_X86\_64\_PC32     .data+0x0000000000000004

0000000000000037 R\_X86\_64\_PC32     .data+0x0000000000000004

000000000000003d R\_X86\_64\_PC32     .data

0000000000000043 R\_X86\_64\_PC32     aaa-0x0000000000000004

000000000000004c R\_X86\_64\_PC32     \_Z3sumiii-0x0000000000000004

RELOCATION RECORDS FOR [.eh\_frame]:

OFFSET           TYPE              VALUE

0000000000000020 R\_X86\_64\_PC32     .text

[root@localhost test]$ objdump -t main.o

SYMBOL TABLE:

0000000000000000 l    df \*ABS\*  0000000000000000 main.c

0000000000000000 l    d  .text  0000000000000000 .text

0000000000000000 l    d  .data  0000000000000000 .data

0000000000000000 l    d  .bss   0000000000000000 .bss

0000000000000004 l     O .data  0000000000000004 \_ZL3bbb

0000000000000008 l     O .data  0000000000000004 \_ZZ5dosthvE3ccc

0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack

0000000000000000 l    d  .eh\_frame      0000000000000000 .eh\_frame

0000000000000000 l    d  .comment       0000000000000000 .comment

0000000000000000 g     O .data  0000000000000004 aaa

0000000000000000 g     F .text  0000000000000058 \_Z5dosthv

0000000000000000         \*UND\*  0000000000000000 \_Z3sumiii

**Linux动态链接PLT和GOT表**

程序示例：

#include <stdio.h>

**void** print\_banner()

{

    printf("Welcome to World of PLT and GOT\n");

}

**int** main(**void**)

{

    print\_banner();

**return** 0;

}

编译：gcc -Wall -g -o test.o -c test.c -m32

链接：gcc -o test test.o -m32

为方便分析汇编，这里使用-m32选项生成i386架构指令而非x86\_64架构指令。

调用printf函数的汇编语句形如：call \*\*<printf函数的地址>\*\*

由于printf函数位于glibc动态库内，编译链接阶段无法知道进程运行起来之后printf函数的真正加载地址，所以上述的\*\*<printf函数地址>\*\*一项是无法填充的，只有在进程运行后，printf函数的地址才能确定。

但如果在进程运行起来之后直接将上述call指令的操作数\*\*<printf函数地址>\*\*替换为printf函数的真正地址，会存在两个问题：

1. 现代操作系统不允许修改代码段，只能修改数据段
2. 如果print\_banner函数是在一个动态库（.so对象）内，修改了代码段，那么它就无法做到系统内所有进程共享同一个动态库的text段

因此，运行时重定位后printf函数的地址只能保存到数据段内，而不能保存到代码段上。

print\_banner函数的汇编指令如下（使用objdump -d test.o查看）：

00000000 <print\_banner>:

  0:  55                   push %ebp

  1:  89 e5                mov %esp, %ebp

  3:  83 ec 08             sub   $0x8, %esp

  6:  c7 04 24 00 00 00 00 movl  $0x0, (%esp)

  d:  e8 fc ff ff ff       call  e <print\_banner+0xe>

 12:  c9                   leave

 13:  c3                   ret

可以看到，call指令的操作数是fc ff ff ff，即16进制数0xfffffffc（x86架构是小端的字节序），看成有符号数是-4。这里应该存放printf函数的地址，但由于编译阶段无法知道printf函数的地址，所以预先放一个-4在这里，然后用重定位项来描述：这个地址在链接时要修正，修正为printf符号的地址，按相对引用方式。链接阶段主要完成以下事情：

1. 各个可重定位文件的同名section合并
2. 对代码段，数据段以及各符号进行地址分配
3. 链接时重定位，重定位只能是修改指令的操作数，而无法修改指令自身，如e8表示call指令

编译器只能老老实实地生成调用printf的汇编指令，printf是在glibc动态库定义或者是在其它.o定义这两种情况下，它都能工作。链接阶段是知道printf在哪定义的，如果是在其它.o中定义的，那么在链接阶段，printf地址可以确定，直接重定位；如果printf是在动态库内定义的，那么链接阶段无法完成重定位，只能运行时重定位。

根据前文，运行时重定位无法修改代码段，只能将printf重定位到数据段。那在编译阶段就已生成好的call指令，怎么感知这个已重定位好的数据段内容呢？现代Linux系统的做法是链接器生成一小段额外的代码片段，通过这段代码获取printf函数地址，并调用printf函数。伪代码示意如下：

// 调用printf的call指令

call printf\_stub

//plt表

printf\_stub:

    mov rax, [printf函数的储存地址] // 获取printf重定位之后的地址

    jmp rax // 跳过去执行printf函数

//got表

printf函数的储存地址：

　　这里储存printf函数重定位后的地址

链接阶段发现printf定义在动态库时，链接器生成一段小代码printf\_stub，然后call语句中printf\_stub地址取代原来的printf。因此转化为链接阶段对printf\_stub做链接时重定位，而运行时才对printf做运行时重定位。

存放外部函数地址的数据段，即全局偏移表（GOT, Global Offset Table）的表项

用于获取外部函数地址并调用的一小段额外代码，即程序链接表（PLT，Procedure Link Table）的表项

描述PLT/GOT是如何运行的示意图如下：



**动态链接GOT+PLT实现延迟绑定：**

objdump -d <file>：显示executable section的反汇编代码，包括.text段、.plt段等

objdump -j <name> <file>：仅显示名为name的段的信息

objdump -d -j .plt <file>：显示.plt段的反汇编代码，即plt表的内容

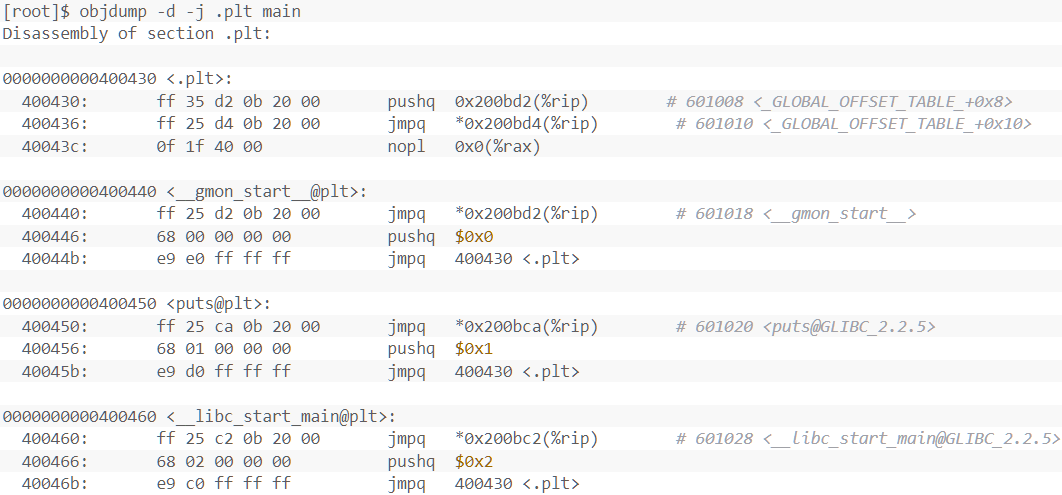
objdump -R <file>：显示dynamic relocation entries，即got表

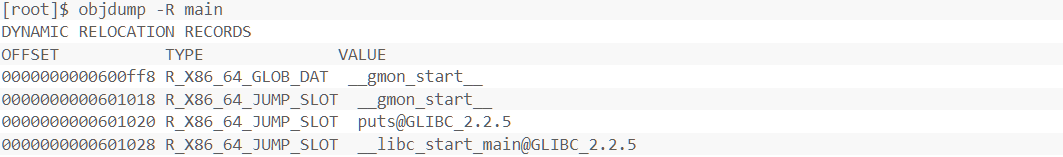
gdb中disassemble <函数名或内存地址>：显示函数或内存地址处的反汇编代码

gdb中x /nfu addr：查看内存地址addr处存放的值。其中，n表示列举出从addr开始的n个单位的内容；f表示输出格式(比如x表示将结果显示成十六进制)；u表示单位(w表示四字节，g表示八字节)

汇编语句jmpq \*0x200bca(%rip)：内存位置0x200bca + (%rip)处存储着一个内存地址值，跳转到该内存地址处执行指令，其中%rip代表指令指针寄存器的值

汇编语句jmpq 0x400430：跳转到0x400430地址处执行指令





[root]$ gdb ./main

(gdb) disas main

Dump of assembler code for function main():

   0x000000000040055d <+0>:     push   %rbp

   0x000000000040055e <+1>:     mov    %rsp,%rbp

   0x0000000000400561 <+4>:     mov    $0x400610,%edi

   0x0000000000400566 <+9>:     callq  0x400450 <puts@plt>

   0x000000000040056b <+14>:    mov    $0x0,%eax

   0x0000000000400570 <+19>:    pop    %rbp

   0x0000000000400571 <+20>:    retq

为了实现“动态”的效果，可执行文件访问外部库函数实际上会通过plt这个跳板代码访问，整个过程采用延迟绑定（懒加载）的方式，第一次执行时，plt代码在got表的指令实际是执行下一行命令（类似于一个查缓存的操作，如果缓存不存在，就继续往下执行写缓存），后续的代码会解析外部库函数的实际地址，然后填充到got表中，这样外部函数地址便有了一个缓存，后续再执行时，plt就会直接获取外部函数库的地址进行跳转，不用再次解析。

C代码中printf()函数调用对应的汇编语句为callq  0x400450，表示跳转到0x400450处执行，查看plt表可知，0x400450即为puts符号对应的plt表项的地址，表项中第一条指令为jmpq \*0x200bca(%rip)，下一条指令的地址为0x400456，即为指令指针寄存器的值，0x200bca+(%rip)= 0x200bca+0x400456=0x601020，

0x601020其实为puts符号对应的got表项的地址，从该地址处读取8字节的值，为0x400456，跳转到该值所描述的地址处。

查看plt表可知，0x400456即puts符号对应的plt表项中的第二条指令的地址，往下执行，跳转到plt[0]，plt[0]中会执行jmpq   \*0x200bd4(%rip)，0x200bd4+(%rip)= 0x200bd4+0x40043c=0x601010

从0x601010处读取8字节的地址值，发现为0x00。这是因为动态链接器初始化工作还未完成。

执行到main函数时，断点，再查看0x601010处的值，变为一个有效地址值，反汇编查看该地址处的函数为dl\_runtime\_resolve\_xsavec，该函数用于实现符号解析和重定位工作，会将puts的真实地址填入对应的got表项中，执行完printf语句后再查看puts对应的got表项x/1xg 0x601010，发现已经改变，指向实际的puts函数指令处

(gdb) x/1xg 0x601020

0x601020:       0x0000000000400456

(gdb) x/1xg 0x601010

0x601010:       0x0000000000000000

(gdb) b main

Breakpoint 1 at 0x400561: file main.cc, line 4.

(gdb) r

Breakpoint 1, main () at main.cc:4

4               printf("hello, world\n");

(gdb) x/1xg 0x601010

0x601010:       0x00007ffff7df1a30

(gdb) disas 0x00007ffff7df1a30

Dump of assembler code for function \_dl\_runtime\_resolve\_xsavec:

   0x00007ffff7df1a30 <+0>:     push   %rbx

   0x00007ffff7df1a31 <+1>:     mov    %rsp,%rbx

   ……

   0x00007ffff7df1a9d <+109>:   mov    0x10(%rbx),%rsi

   0x00007ffff7df1aa1 <+113>:   mov    0x8(%rbx),%rdi

   0x00007ffff7df1aa5 <+117>:   callq  0x7ffff7de9d20 <\_dl\_fixup>

   ……

(gdb) b \*0x0000000000400570

Breakpoint 2 at 0x400570: file main.cc, line 6.

(gdb) c

hello, world

Breakpoint 2, main () at main.cc:6

6       }

(gdb) x/1xg 0x601020

0x601020:       0x00007ffff725d740

(gdb) disas 0x00007ffff725d740

Dump of assembler code for function puts:

   0x00007ffff725d740 <+0>:     push   %r13

   0x00007ffff725d742 <+2>:     push   %r12

    ……

   0x00007ffff725d74d <+13>:    callq  0x7ffff7279e60 <\_\_strlen\_sse2>

   ……

# strip

可执行文件strip

strip –s main裁剪后，objdump -t main查看没有任何符号，运行不受影响

静态库strip

//add.cpp

const char\* VERSION = "4.4.1";

extern "C" int add(int lhs, int rhs);

static int a = 10;

int add(int lhs, int rhs)

{

static int b = 20;

return lhs + rhs;

}

g++ -c add.cpp

ar -rcs libadd.a add.o

objdump -t libadd.a查看有符号a、b、add



strip –s libadd.a后查看则无任何符号，不能用于链接

strip --strip-unneeded libadd.a后查看有符号add，没了符号a、b，清除了不能提供给外部的符号



动态库strip

g++ -shared -fpic -o libadd.so add.cpp

readelf -S libadd.so查看有26个section

strip –s libadd.so裁剪后，剩24个section，失去了.symtab和.strtab节，依旧可以链接

strip 动态库，只清除普通符号表，会保留动态符号表，即dynsym、dynstr段，而动态链接依靠的就是动态符号表

**Union中有non-trival 成员**

union U {

U() {}

~U() {}

std::shared\_ptr<A> a;

int b;

};

int main()

{

U u;

u.a = std::make\_shared<A>();

return 0;

/\*输出：A()\*/

}

发生了内存泄漏

只要Union中有non-trival成员，则需要复杂的处理才能避免内存处理，因此Union中最好不要放置non-trival成员

**union** S

{

    std::string str;

    std::vector<**int**> vec;

    ~S() {} // needs to know which member is active, only possible in union-like class

};          // the whole union occupies max(sizeof(string), sizeof(vector<int>))

**int** main()

{

    S s = { "Hello, world" };

    // at this point, reading from s.vec is undefined behavior

    std::cout << "s.str = " << s.str << '\n'; //s.str = Hello, world

    s.str.~basic\_string();

**new** (&s.vec) std::vector<**int**>;

    // now, s.vec is the active member of the union

    s.vec.push\_back(10);

    std::cout << s.vec.size() << '\n';  //1

    s.vec.~vector();

}

**unordered容器：**

无序容器的核心是hash

unordered\_set\map不允许元素重复，而unordered\_multiset\multimap允许

template <typename T,

typename Hash = std::hash<T>,

typename EqPred = std::equal\_to<T>,

typename Allocator = std::allocator<T> >

class unordered\_set;

向unordered\_set中插入元素时，先比较hash值，如果hash值不同，则一定能插入成功；如果hash值相同，则继续operator ==比较元素自身，如果返回true则插入失败，否则插入成功。

注意：即使元素自身”相等”，但hash值不同，依然能插入成功

C++标准规定：

1. unordered容器的hash table使用chaining做法，于是hash code将被关联至linked list（具体是单or双链表则取决于实现）
2. rehashing(重新散列）只可能发生在以下调用之后：insert()、rehash()、reserve()、clear()。这是以下保证的自然结果：erase()绝不会造成指向元素的iterator、reference、pointer失效。因此，如果你删除数百个元素，bucket的大小并不会改变。但如果你在那之后安插个元素，bucket的大小就有可能缩小。
3. rehashing后，迭代器可以失效，但是对元素的指针/引用不能失效。考虑map[key1] = map[key2]，key2不存在从而引发rehash，如果rehash后对元素的引用失效，则map[key1]成为悬挂指针，有几率会core掉。

c.bucket\_count () //返回当前的 bucket 个数

c.max\_bucket\_count () //返回 bucket 的最大可能数量

c.load\_factor() //返回当前的负载系数

c. max\_load\_factor() //返回当前的最大负载系数

c.max\_load\_factor (val) //设定最大负载系数

c.rehash(bnum) //将容器 rehash, 使其 bucket 个数至少为bnum

c.reserve (num) //将容器 rehash, 使其空间至少可拥有num 个元素

负载系数，即load\_factor，unordered\_set中的元素个数与bucket个数的比值。当负载系数超过某个值时，hash表性能会明显下降，因此这时候会进行rehash，增加表的大小。

最初TRI提供rehash()，它要求为hash table提供bucket的大小至少是”传入的参数大小＂。问题是，以此接口，你仍然必须考虑最大负载系数。如果最大负载系数是0.7而你打算准备100个元素，你必须将100除以0.7，以此计算“只要没超过100个元素就一定不会引发rehashing"的bucket数目。也就是说，你必须传入rehash(143)以避免在最高达到100个元素的情况下出现出现rehashing。如果使用reserve()，计算是内部进行的，所以你可以仅仅传递hash table应该筹备的元素个数：

coll.rehash(100);//prepare for 100/mx\_load\_factor() elements

coll.reserve(100);//prepare for 100 elements

由于rehash的存在，对于同一个unordered容器，一番增删操作后，即使操作前后的元素完全相同，但遍历时元素顺序不一定相同。

Hash函数必须是个函数，或function object，它接受的参数为元素类型，并返回std::size\_t 类型。因此，bucket的当前数量并未被考虑。将Hash函数的返回值映射至合法的bucket索引范围内，是由容器内部完成。

struct A

{

A(int x) :m\_x(x) {}

bool operator == (const A& rhs) const { return m\_x == rhs.m\_x; }

int m\_x;

};

struct HashA {

size\_t operator()(const A& a)const {

return (size\_t)(a.m\_x);

}

};

std::unordered\_set<A, HashA> us1;

auto HashAFunc = [](const A& a) {return (size\_t)(a.m\_x); };

std::unordered\_set<A, size\_t(\*)(const A&)> us2(8, HashAFunc); //第一个参数为bucket数目，默认是8

//或者std::unordered\_set<A, decltype(HashAFunc)> us2(8, HashAFunc);