Modern C++

[C++11/14/17]

Second Edition

A new language ,more concise,more effective!

Auth : 王桂林

Mail : guilin_wang@163.com

Org : 能众软件

Web : http://edu.nzhsoft.cn

版本信息：

版本	修订人	审阅人	时间	组织
v1.0	王桂林		2016.04.02	
v2.0	王桂林		2017.06.01	山东能众软件科技有限公司

更多学习：

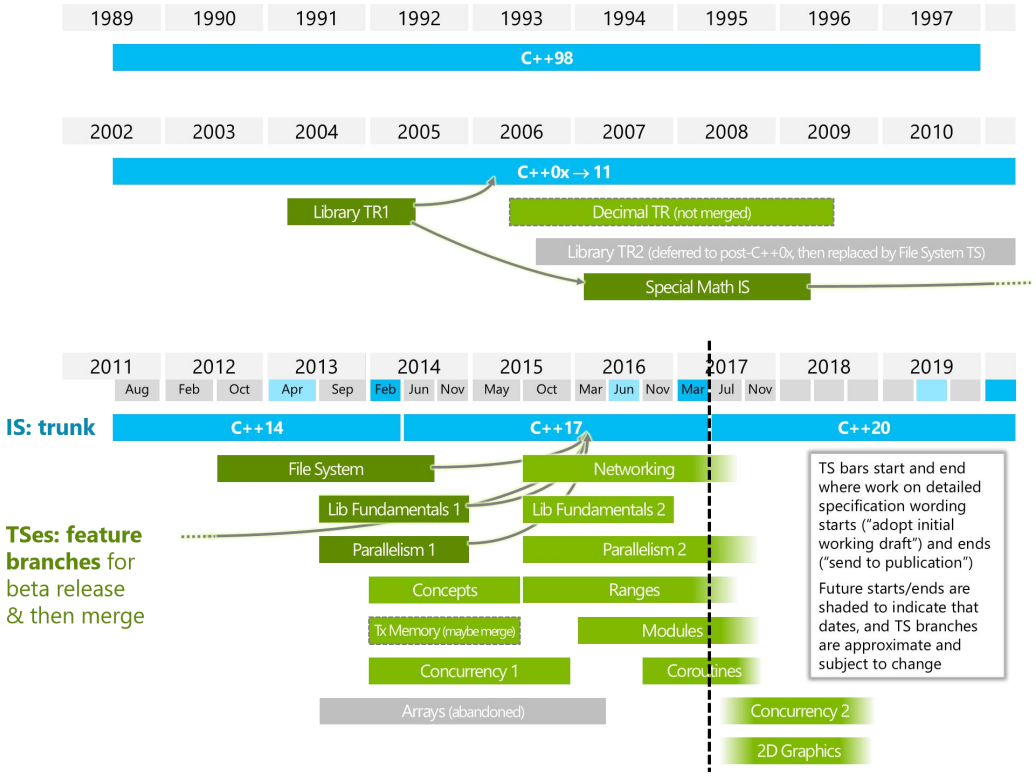
1.新语言 C++ 11

1.1.C++ History

1979 年，Bjame Sgoustrup 到了 Bell 实验室，开始从事将 C 改良为带类的 C（C with classes）的工作。1983 年该语言被正式命名为 C++。自从 C++被发明以来，它经历了 3 次主要的修订，每一次修订都为 C++增加了新的特征并作了一些修改。第一次修订是在 1985 年，第二次修订是在 1990 年，而第三次修订发生在 c++的标准化过程中。在 20 世纪 90 年代早期，人们开始为 C++建立一个标准，并成立了一个 ANSI 和 ISO（Intemational Standards Organization）国际标准化组织的联合标准化委员会。该委员会在 1994 年 1 月 25 日提出了第一个标准化草案。在这个草案中，委员会在保持 Stroustrup 最初定义的所有特征的同时，还增加了一些新的特征。

在完成 C++标准化的第一个草案后不久，发生了一件事情使得 C++标准被极大地扩展了：Alexander stepanov 创建了标准模板库（Standard Template Library，STL）。STL 不仅功能强大，同时非常优雅，然而，它也是非常庞大的。在通过了第一个草案之后，委员会投票并通过了将 STL 包含到 C++标准中的提议。STL 对 C++的扩展超出了 C++的最初定义范围。虽然在标准中增加 STL 是个很重要的决定，但也因此延缓了 C++标准化的进程。

委员会于 1997 年 11 月 14 日通过了该标准的最终草案，**1998 年，C++的 ANSI/IS0 标准被投入使用。通常，这个版本的 C++被认为是标准 C++。**所有的主流 C++编译器都支持这个版本的 C++，包括微软的 Visual C++和 Borland 公司的 C++Builder。



1.2.Modern C++

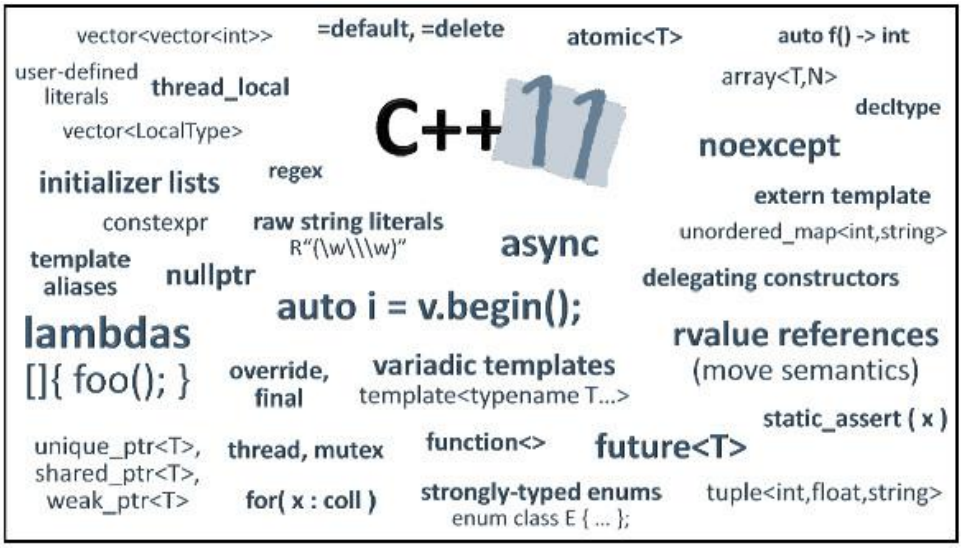
C++0x 是 C++最新标准标准化过程中的曾用名。在 2011 年 9 月份，C++0x 正式由官方发布并命名 C++11，现在很多编译器已经支持了部分 C++11 特性。

C++11 包括大量的新特性：主要特征像 lambda 表达式和移动语义，实用的类型推导关键字 auto，更简单的容器遍历方法，和大量使模板更容易使用的改进。

过去	C++98	C++03
现在	C++0x/C++11	C++14
未来	C++17	C++20

1.3.功能一览

1.3.1.图示



1.3.2.更多

```
https://en.wikipedia.org/wiki/C++11#Regular_expressions
```

1.4.Why ModernC++

效率重要，且同时需要好的抽象机制的领域。保持性能的同时提高代码的安全性和生产力。

Modern C++增加了更简洁，更高效的机制，并且引入了内存管理和线程框架。cocos2dx3.16.1 和 crossapp2.0 都已经全面支持 C++11 了。

2.简洁 More Concise

2.1.nullptr

nullptr 是用于解决 NULL 和 0 的有疑义关系的。NULL 通常被义为(void*)0。在如下应用中会引发歧义。

2.1.1.入参

```
#include <iostream>
using namespace std;


void f(int){}
void f(bool){}
void f(void*){}
int main()
{
    f(0);      // calls f(int), not f(void*)
    f(NULL);    // might not compile, but typically calls f(int). Never calls f(void*)
    f(nullptr); // calls f(void*) overload
}
```

1) C++ 视 0 首先为 int 型，因此，调用 f(0) 即调用 f(int)

2) NULL 的情况复杂些，C++ 首先视其为广义整型。假如 NULL 被定义为普通的 0，则调用 f(int);

如果 NULL 被定义成 0L，则 long -> int, long -> bool, 0L -> void*, 这三种情况都是合法的，此时，编译器会报错

3) 使用 nullptr，则不会有重载函数调用模糊的问题

- nullptr 不属于广义整型，也不是普通意义上的指针。

- nullptr 的实际类型是 std::nullptr_t，它能够隐式的转换成所有的原始指针类型，故可将其视为一个可指向所有类型的指针。

2.1.2.返值

使用 0 与 result 作比较，则一时不能确定 findRecord 的返回值类型 (可能是广义整型，也可能是指针类型)；使用 nullptr，可以清楚地知道 findRecord 的返回值，必定是一个指针类型。

```
auto result = findRecord( /* arguments */ );
if (result == 0)
{
    ...
}

auto result = findRecord( /* arguments */ );
if (result == nullptr)
```

```
{
    ...
}
```

2.2.override

2.2.1.语义

覆写父类的虚函数时候，好的 IDE 一定会给出斜体等的提示，表示此函数覆写自父类。

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    void mouseMoveEvent(QMouseEvent *event) ;
    ~MainWindow();
};
```

若是在文本编辑器中，写错了，参数类型不对或个数不对，但是编译没问题，运行时候却和你设计的不一样不被调用，override 就是辅助你检查是否继承了想要虚继承的函数。

关键字 override 则指明，此种覆写关系，若此关系不成立，则以报错的形式提示给用户。

把可能粗心的事，交给编译器，是对的。

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    void mouseMovEvent(QMouseEvent *event) override;
    ~MainWindow();
};

#endif // MAINWINDOW_H
```

2.2.2.示例

```
class  G
{
public:
    virtual void func(int);
};
```

```
class H: G
{
public:
   virtual void func(double);
};
```

```
H *p=new H; p->func(5);   //calls G::f
p->func(5.0);            // calls H::f
```

2.3.final

2.3.1.语义

关键字 final 有两个用途。第一，它阻止了从类继承；第二，阻止一个虚函数的覆写。

2.3.2.示例

```
class A           //final
{
public:
   virtual void func() const;
};
class  B: A
{
public:
   void func() const override final; //OK
};
class C: B
{
public:
   void func()const; //error, B::func is final
};
```

2.3.3.意义

阻止了类的无限扩展。

2.4.=default =delete

2.4.1.default

C++ 的类有四类特殊成员函数，它们分别是：默认构造函数、析构函数、拷贝构造函数以及拷贝赋值运算符。

这些类的特殊成员函数负责创建、初始化、销毁，或者拷贝类的对象。

如果程序员没有显式地为一个类定义某个特殊成员函数，而又需要用到该特殊成员函数时，则编译器会隐式的为这个类生成一个默认的特殊成员函数。

```
#include <iostream>

using namespace std;

class A
{
public:
    A() = default;
    A(int x ):_x(x)
    {}
private:
    int _x;
};

int main()
{
    A a;
    return 0;
}
```

2.4.2.delete

为了能够让程序员显式的禁用某个函数，C++11 标准引入了一个新特性："=delete"函数。程序员只需在函数声明后上“=delete;”，就可将该函数禁用。

```
class Singleton
{
public:
    static Singleton* getInstance()
    {
        if(_ins == nullptr)
            _ins = new Singleton;
        return _ins;
    }
    Singleton(Singleton &) = delete;
    Singleton& operator=(Singleton &) = delete;
private:
    Singleton(){}
    static Singleton* _ins;
};
Singleton* Singleton::_ins = nullptr;
int main()
{
    Singleton *ps = Singleton::getInstance();
    Singleton ps(*ps);
    *ps = Singleton::getInstance();
    return 0;
}
```

2.5.>>right angle brackets

```
r<vector<int> > vector_of_int_vectors;
vector<vector<int>> vector_of_int_vectors;
```

2.6.委托构造 construct reuse

一个构造函数，使用自己的参数，传递给其他构造函数去构造，作为自己的构造函数实现。

```
struct A {
    bool a_;
    char b_;
    int c_;
    float d_;
    double e_;
    A(bool a, char b, int c, float d, double e)
        : a_(a), b_(b), c_(c), d_(d), e_(e) {}
    //construct reuse
    A (int c): A(true, 'b', c, 1.1, 1000.1) {}
    A (double e): A (false, 'a', 0, 0.1, e) {}
};
int main(){

    A o1(10);
    cout <<"a:"<<o1.a_
         <<", b:" <<o1.b_
         <<", c: " <<o1.c_
         <<", d: "<< o1.d_
         <<", e: "<< o1.e_;
    A o2(5.5);
    cout <<"a:"<<o2.a_
         <<", b:" <<o2.b_
         <<", c: " <<o2.c_
         <<", d: "<< o2.d_
         <<", e: "<< o2.e_;
    return 0;
}
```

2.7.Raw String Literals

C/C++中提供了字符串，字符串的转义序列，给输出带来了很多不变，如果需要原生义的时候，需要反转义，比较麻烦。

C++提供了 *R"()"*，原生字符串，即字符串中无转义，亦无需再反义。**但是注意()中的)"会导至提前结束。**

详细规则见带码：

```
#include <iostream>

using namespace std;

string path = "C:\Program Files (x86)\alipay\aliedit\5.1.0.3754";
string path2= "C:\\Program Files (x86)\\alipay\\aliedit\\5.1.0.3754";
string path3= R"(C:\Program Files (x86)\alipay\aliedit\5.1.0.3754)";
string path4= R"(C:\Program "Files" (x86)\\alipay\aliedit\5.1.0.3754)";

int main(int argc, char *argv[])
{
    cout<<path<<endl;
    cout<<path2<<endl;

    cout<<path3<<endl;
    cout<<path4<<endl;
    return 0;
}
```

2.8. 区间迭代 Range-Based for

2.8.1.Normal for

```
#include <iostream>

using namespace std;

int main()
{

    char *p = "aksdfjlaskdf";
    for(auto &i: p)
    {
        cout<<i<<endl;
    }

    int arr[10] = {1,2,3,4,5,6,7};
    for(auto &i: arr)
    {
        cout<<i<<endl;
    }

    string str = "china";
    for(auto& ch: str)
    {
        cout<<ch<<endl;
    }

    return 0;
}
```

2.8.2.Range for

2.8.2.1.语义

C++为 for 提供 for range 的用法。对于 STL(vector /list /map)的遍历带来了极大的书写便利。

（range for）语句遍历给定序列中的每个元素并对序列中的每个值执行某种操作，其语法形式是：

```
for (declaration : expression)
     statement</code>
```

expression 部分是一个对象，必须是一个序列，比方说用花括号括起来的初始值列表、**数组**或者 vector 或 string 等类型的对象。**这些类型的共同特点是拥有能返回迭代器的 begin 和 end 成员。**

declaration 部分负责定义一个变量，该变量将被用于访问序列中的基础元素。每次迭代，declaration 部分的变量会被初始化为 expression 部分的下一个元素值。确保类型相容最简单的办法是使用 auto 类型说明符。

2.8.2.2.for 实战 STL

```
#include <iostream>
#include <vector>
#include <map>

using namespace std;

int main()
{

//    vector<string> vs = {"abc","xyz","mnq"};

//    vector<string>::iterator itr = vs.begin();
//    for(; itr != vs.end(); itr++)
//    {
//        cout<<*itr<<endl;
//    }

//    for(auto &s : vs)
//    {
//        cout<<s<<endl;
//    }


   map<int,string> mis={
                {1,"c++"},
                {2,"java"},
                {3,"python"}
                };
```

```
    map<int,string>::iterator itr = mis.begin();
    for(; itr != mis.end(); ++itr)
    {
        cout<<(*itr).first<<"\t"<<itr->second<<endl;
    }

    for(auto &pair: mis)
    {
        cout<<pair.first<<"\t"<<pair.second<<endl;
    }

    return 0;
}
```

2.9.初始化列表 initializar list {}

2.9.1.常规初始化 normal init

以 vector/map 的几种初始化方式给大家介绍，所谓的常规初始化，即调用类的构造器。

比如：vector<int> vi;　vector<int> vi(10,10);　vector<int> vi(arr,arr+10);

```
#include <iostream>
#include <list>
#include <vector>
#include <map>
using namespace std;

int main()
{
    vector<int> vi;
    vector<int> vi2(10);
    vector<int> vi3(10,1);
    int arr[10];
    vector<int> vi4(arr,arr+10);


    list<int> li(10);
    list<int> li2(10);
    list<int> li3(10,1);
    int arr2[10] = {};
    list<int> li4(arr2,arr2+10);


    map<int,string> mis;
    mis[0] = "first";
    mis[1] = "second";
    mis[2] = "third";
    map<int,string> mis2(mis.begin(),mis.end());
```

```
    mis.insert(pair<int,string>(3,"fourth"));
    mis.insert(map<int,string>::value_type(3,"fourth"));


    for(auto& pair:mis)
        cout<<pair.first<<":"<<pair.second<<endl;

    return 0;
}
```

2.9.2.initialization List {}

以初始化列表的方式来进行实始化，打破了，原有的初始化格局，令实始化更直观化，人性化。

```
#include <iostream>
#include <list>
#include <vector>
#include <map>
using namespace std;


int main(int argc, char *argv[])
{
    vector<int> vi = {1,2,3,4,5};

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

    map<int,string> mis =
    {
        {1,"c"},   {2,"c++"},
        {3,"java"},{4,"scala"},{5,"python"}
    };
    mis.insert({6,"ruby"});

    for(auto &is: mis)
    {
        cout<<is.first<<is.second<<endl;
    }

    return 0;
}
```

对比没有 C++11 支持的情况

```
[root@localhost opt]# g++ aa.cpp
aa.cpp: In function 'int main()':
aa.cpp:6: error: in C++98 'vi' must be initialized by constructor, not by '{...}
'
aa.cpp:6: warning: extended initializer lists only available with -std=c++0x or
-std=gnu++0x
aa.cpp:6: error: deducing from brace-enclosed initializer list requires #include
 <initializer_list>
aa.cpp:6: error: no matching function for call to 'std::vector<int, std::allocat
or<int> >::vector(<brace-enclosed initializer list>)'
/usr/lib/gcc/i686-redhat-linux/4.4.7/../../../../include/c++/4.4.7/bits/stl_vect
or.h:241: note: candidates are: std::vector<_Tp, _Alloc>::vector(const std::vect
or<_Tp, _Alloc>&) [with _Tp = int, _Alloc = std::allocator<int>]
/usr/lib/gcc/i686-redhat-linux/4.4.7/../../../../include/c++/4.4.7/bits/stl_vect
or.h:207: note:                 std::vector<_Tp, _Alloc>::vector() [with _Tp = i
nt, _Alloc = std::allocator<int>]
aa.cpp:7: error: expected initializer before ':' token
aa.cpp:9: error: expected primary-expression before 'return'
aa.cpp:9: error: expected ';' before 'return'
aa.cpp:9: error: expected primary-expression before 'return'
aa.cpp:9: error: expected ')' before 'return'
```

2.9.3.initializer_list

在 C++11 可以使用 {} 而不是 () 来调用类的构造函数。为什么呢?

template< class T > class initializer_list; C++11 中提供了新的模板类型 initializer_list。

initializer_list 对象只能用大括号{}初始化,其内有元素都是 const 的。常用于构造器和普通函数参数。

2.9.3.1.引入

```
#include <iostream>
using namespace std;
int main()
{
    auto  ili = {1,2,3,4,5};
    initializer_list<int> ili2 = {1,2,3,4,5};
    return 0;
}
```

2.9.3.2.普通函数参数

```
#include <iostream>

using namespace std;

double sum(const initializer_list<double> &il);
double average(const initializer_list<double> &ril);

int main()
{
    auto il = {1,2,3,4,5};

    double s  = sum({1,2,3,4,5});
```

```
    cout<<"sum = "<<s<<endl;
    double avg  = average({1,2,3,4,5});
    cout<<"average = "<<avg<<endl;

    return 0;
}
double sum(const initializer_list<double> &il)
{
    double s = 0;
    for(auto d:il)
        s += d;
    return s;
}

double average(const initializer_list<double> &ril)
{
    double s = 0;
    for(auto d:ril)
        s += d;
    double avg = s/ril.size();
    return avg;
}
```

2.9.3.3.构造器参数（vector<int> vi ={}）

```
#include <iostream>
#include <list>
#include <vector>
#include <map>
using namespace std;


template <typename T>
class myarr
{
private:
    vector<T> _arr;

public:
    myarr() {
        cout<<"called myarr()"<<endl;
    }

    myarr(const initializer_list<T>& il)
    {
        cout<<"called  myarr(const initializer_list<T>& il)"<<endl;
        for (auto x : il)
            _arr.push_back(x);
    }
```

```
};

int main()
{
    myarr<int> ma;
    myarr<int> ma2 = {1,2,3,4,5};
    return 0;
}
```

2.9.4.Uniform Initialization 统一初始化风格

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int i = 3;
    int ii({3});
    int ii{3};
    int arr[] = {1,2,3};
    int arr2[] ({1,2,3});
    int arr2[] {1,2,3};
    vector<int> vi = {1,2,3};
    vector<int> vi2({1,2,3});
    vector<int> vi2{1,2,3};
    return 0;
}
```

2.10.自动类型推导 auto

2.10.1.引入

```
#include <iostream>
using namespace std;

int func()
{
    return 8;
}

int main(int argc, char *argv[])
{
    auto i = 5;
    auto &ri = i;
    auto rf = func();

    const auto *p = &ri;
```

```
    static auto si = 100;
    return 0;
}
```

2.10.2.auto 语义

2.10.2.1.占位

auto 能够实现类型的**自我推导**，并不代表一个实际的类型声明。auto 只是一个类型声明的**占位符**。

auto 声明的变量，必须马上初始化，以让编译器推断出它的实际类型，并在编译时将 auto 占位符**替换为**真正的类型。

注意:auto 不能用于函数参数（C++14 可以通过）

```
void func(auto v){

}
int main(int argc, char *argv[])
{
    return 0;
}
```

2.10.2.2.推导不出修饰

```
#include <iostream>

using namespace std;

void foo(){
    static int i= 1;
    auto b = i;
    cout<<++b<<endl;

}

int main()
{
    const int a = 100;
    auto  b = a;
    b = 300;

    foo();
    foo();

    return 0;
}
```

2.10.3.应用

2.10.3.1.推导

如果对字符串"china" 或是数组 int arr[10]进行推导，会推导出什么类型呢？可以采用运行时类型信息打印类型，typeid().name()。

```
#include <iostream>
#include <typeinfo>
using namespace std;

void func(int ,int){

}


int main(int argc, char *argv[])
{
    auto p = "china";
    cout<<sizeof(p)<<" "<<typeid(p).name()<<endl;
    int arr[] = {1,2,3,4,5,6,7,8};
    auto parr = arr;
    cout<<sizeof(parr)<<" "<<typeid(parr).name()<<endl;

    auto pfunc = func;
    cout<<sizeof(pfunc)<<" "<<typeid(pfunc).name()<<endl;
    return 0;
}
```

2.10.3.2.STL 之 iterator

```
auto itr = vi.begin();
```

2.11.类型推导 decltype

2.11.1.获取表达式类型

auto 类型，作为占位符的存在来修饰变量，必须初始化，编译器通过初始化来确定 auto 所代表的类型,即必须定义变量。

如果，我**仅**希望**得到类型**，而不是具体的变量产生关系，该如何作到呢？

decltype(expr); expr 代表被推导的表达式，

```
#include <iostream>

using namespace std;

int func()
```

```
{
    cout<<"int func()"<<endl;
    return 8;
}


typedef int (*FUNC)();

int main()
{
    double a = 10;
    cout<<sizeof(a)<<endl;
    decltype(a) b;
    cout<<sizeof(b)<<endl;
    cout<<sizeof(decltype(a))<<endl;


     decltype("abcde") ds = "china";   //少于 5 个字符会报错
     //"abc" "abcde" string("abc")
    cout<<ds<<endl;

    char *p = "abc";
    decltype(p) pp = "12345";
    cout<<sizeof(pp)<<endl;
    cout<<pp<<endl;


    decltype(func())  fa;             //返值类型推导
    cout<<sizeof(fa)<<endl;

    decltype(&func)  fp = func;       //? ? ? ?
    fp();

    FUNC pf = func;
    pf();
    decltype(pf)  pf2 = func;         //推导的是变量
    pf2();

    return 0;
}
```

上例中，关于函数的类型推导，不可以直接使用函数名字，可以使用函数的类型生成的对象推导。

2.11.2.推导规则

decltype(expr); 所推导出来的类型，完全与 expr 类型一致。同 auto 一样，在**编译期间**完成，并**不会真正计算表达式的值**，即上面的推导数并不会导致函数打印的。

decltype 不可对类型推导。

2.11.3.应用

2.11.3.1.推导类型(decltype) 重定义 typedef

对于推导出来的类型进行重定义，是一种不错的选择。比宏更准确，因为发生在编译阶段。

```
#include <iostream>
#include <vector>
#include <map>


using namespace std;

int main(int argc, char *argv[])
{
    vector<int> vi = {1,2,3,4,5,0};

    typedef decltype(vi.begin()) Itr;

    for(Itr itr = vi.begin(); itr != vi.end(); ++itr)
    {
        cout<<*itr<<endl;
    }

    map<int,string> mis;

    mis.insert(map<int,string>::value_type(1,"abc"));
    mis.insert(decltype(mis)::value_type(2,"java"));

    typedef decltype(map<int,string>::value_type()) Int2String;

    mis.insert(Int2String(3,"c++"));

    for(auto& is:mis)
    {
        cout<<is.first<<is.second<<endl;
    }
    return 0;
}
```

2.11.3.2.返回类型推导 auto -> decltype

->decltype()放在函数的结尾，称为**尾推导**。返回值类型用 auto 来配合使用，是一种绝佳组合。

```
#include <iostream>

using namespace std;

template<typename T1, typename T2,typename T3>
```

```
T1 add(T2 a, T3 b)
{
    return a+b;
}

template<typename T1, typename T2>
auto add2(T1 a, T2 b)->decltype(a+b)
{
    return a+b;
}
int main(int argc, char *argv[])
{
    int a = 1;
    float b = 1.1;
    auto ret = add<decltype(a+b),int,float>(a,b);
    cout<<ret<<endl;

    auto ret2 = add2<int,float>(a,b);
    cout<<ret2<<endl;

    return 0;
}
```

2.12.仿函数 functor

2.12.1.operator()

重载了 operator()的类的对象，在使用中，语法类型于函数。故称其为仿函数。此种用法优于常见的函数回调。

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class Compare
{
public:
    int operator()(int x, int y)
    {
        return x>y;
    }
};

int main()
{
    vector<int> vi = {1,3,5,7,9};
    sort(vi.begin(),vi.end(),Compare());
```

```
    for(auto &i:vi)
        cout<<i<<endl;

}
```

2.12.2.带状态的 operator()

相对于函数，仿函数，可以拥用初始状态，一般通过 class 定义私有成员，并在声明对象的时候，进行初始化。

上例继续修改：

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class Compare
{
public:
    Compare(bool f=true):flag(f){}
    int operator()(int x, int y)
    {
        if(flag)
            return x > y;
        else
            return x < y;
    }
protected:
    bool flag;
};

int main()
{
    vector<int> vi = { 1, 3, 5, 7, 9};
//    sort(vi.begin(),vi.end(),Compare(false));
    sort(vi.back(),vi.end(), Compare(true));

    for(auto &i:vi)
        cout<<i<<endl;

}
```

私有成员的状态，就成了仿函数的初始状态。而由于声明一个仿函数对象可以拥有多个不同初始状态的实例。

```
#include <iostream>
using namespace std;

class Tax
```

```
{
public:
    Tax(float r, float b):_rate(r),_base(b){}
    float operator()(float money)
    {
        return (money-_base)*_rate;
    }
private:
    float _rate;
    float _base;
};


int main()
{
    Tax high(0.3,30000);
    Tax middle(0.2,15000);
    Tax low(0.1,3500);

    cout<<"请输入你的收入:";
    double salary;
    cin>>salary;
    if(salary>30000)
        cout<<high(salary)<<endl;
    else if(salary >15000)
        cout<<middle(salary)<<endl;
    else
        cout<<low(salary)<<endl;

    return 0;
}
```

2.13.Lambda

2.13.1.匿名函数

简短函数，就地书写，调用，即 Lambda 存在的意义，常用于取代作回调用的简短函数指针与仿函数。

就地书写，因只有函数体，即无函数名，也称匿名函数。

2.13.2.Gammar 格式

2.13.2.1.格式

[capturelist] *(parameterlist) mutable ->return type* { *function body* }

2.13.2.2.解析

- **capturelist**

捕获列表。捕获列表，总是出现在 lambda 函数的开始处。事实上[]是 lambda 的引用符。换句话说，编译器根据引出符判断接下来的代码是否是 lamba 函数。

- **parameterlist**

参数列表。与普通函数的参数列表一致。如果不需要传递参数，可以连同()一起省略。

- **mutable**

默认情况下，lambda 函数总是一个 **const 函数**，mutable 可以取消其常量性,所谓的常量性是指不可修改 capturelist 中的东西。在使用该修饰符时，参数列表不可以省略（即使参数为空）。

- **->return type**

返回类型。用于追踪返回类型形式声明函数的返回类型。出于方便，不需要返回值的时候可以连同->一起省略。此外返回类型明确的情况下，也可以省略该部分。编译器可以自行推导。

- **{ function body }**

函数体。内容与普通函数一样，不过除了可以使用参数之外，还可以使用所有捕获的变量。

2.13.2.3.Smallest Lambda:[]{}

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    auto foo = []{ return 1 +2;};
    cout<<foo();
    cout<<[]{ return 1 +2;}()<<endl;
    return 0;
}
```

2.13.2.4.Smaller Lambda:[](){}

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    auto foo = [](int x = 1,int y = 2){ return x+y;};
    cout<<foo();
    cout<<[](int x,int y){ return x +y;}(1,2)<<endl;
    return 0;
}
```

2.13.2.5.Small Lambda:[]()->{}

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
```

```
{
    auto foo = [](int x = 1,int y = 2)->int{ return x+y;};
    cout<<foo();
    cout<<[](int x,int y)->int{ return x +y;}(1,2)<<endl;
    return 0;
}
```

2.13.3.闭包[]与 mutable

2.13.3.1.闭包

closure 是一个函数和它所引用的非本地变量的上下文环境的集合。从定义我们可以得知，closure 可以访问在它定义范围之外的变量，也即 non-local vriables（非局部变量）。关于 closure 的最重要的应用就是回调函数

lambda 函数能够捕获 lambda 函数外的具有自动存储时期的变量。函数体与这些变量的集合合起来叫闭包。

闭包的概念在 lambda 中通过[]来体现出来。

2.13.3.2.[] mutable

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int x = 10; int y = 100;
    cout<<"main:"<<x<<y<<endl;
    auto foo = [&]() mutable {
        x = 20;
        y = 200;
        cout<<"lambda:"<<x<<y<<endl;
    };
    foo();
    cout<<"main:"<<x<<y<<endl;
    return 0;
}
```

2.13.3.3.[]捕获规则

形式	语意
[]	不截取任何变量。
[bar]	仅对外部变量 bar 值传递在函数体中使用。
[&bar]	仅对外部变量 bar 引用传递在函数体中使用。
[x, &y]	仅 x 按值传递，y 按引用传递在函数体中使用。
[&]	截取外部作用域中所有变量，并作为引用传递在函数体中使用。
[=]	截取外部作用域中所有变量，并按值传递在函数体中使用。
[=, &foo]	截取外部作用域中所有变量，并值传递在函数体中使用，但是对 foo 变量使用引用传递。

[&, =foo]	截取外部作用域中所有变量，在函数体中作引用传递使用，但是对 foo 变量作值传递。=foo -> foo

上述，中涉及到值传递要发生拷贝行为，而引用传递则不会发生拷贝行为。捕获列表中不允许重复。比如：[=, a] [&,&this]。

2.13.3.4.闭包及 Lambda 本质

lambda 即匿名对象，闭包的本质，初始化 lambda 对象。

2.13.3.5.返回闭包

下面的程序实现了，返回一个累加器和一个 fibonacci 的生成器的闭包。对于需要携带状态函数，不再需要 static 亦可以实现。

```
#include <iostream>

using namespace std;

auto adder(){
    auto sum= 0;
    return[=](int value)mutable{
        sum += value;
        return sum;
    };
}
// a0b1
// 0 1  1  2  3  5  8

auto fibonacii(){
    auto a = 0,b = 1,t=0;
    return [=]()mutable{
        t = a;
        a = b;
        b = t +b;
        return a;
    };
}

int main()
{
    auto f = adder();
    for(int i=0; i<101; i++){
        cout<<f(i)<<endl;
    }

    auto f2 = fibonacii();

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

```
        cout<<f2()<<endl;
    }
    return 0;
}
```

2.13.4.应用

2.13.4.1.STL::for_each

```
#include <iostream>
#include <time.h>
#include <list>
#include <algorithm>
using namespace std;

void func(int i)
{
    cout<<i<<endl;
}

int main(int argc, char *argv[])
{
    list<int> li;
    srand(time(NULL));
    for(int i=0;i <10; i++)
    {
        li.push_back(rand()%100);
    }

    for_each(li.begin(),li.end(),func);
    for_each(li.begin(),li.end(),[](int i){
        cout<<i<<endl;
    });
    return 0;
}
```

2.13.4.2.sort()/list::sort

```
#include <iostream>
#include <vector>
#include <list>
#include <stdlib.h>
#include <time.h>
#include <algorithm>
#include <functional>
using namespace std;

bool Compare(int i, int j)
```

```
{
    return i<j;
}

int main(int argc, char *argv[])
{
    vector<int> vi;
    srand(time(NULL));
    for(int i=0;i <10; i++)
    {
        vi.push_back(rand()%100);
    }

//    sort(vi.begin(),vi.end(),Compare);
    sort(vi.begin(),vi.end(),[](int x, int y){
        return x<y;
    });

    for_each(vi.begin(),vi.end(),[](int i){
        cout<<i<<endl;
    });
    return 0;
}
```

2.13.4.3. QT 中 signal 和 slot 机制

QMetaObject::Connection QObject::connect(

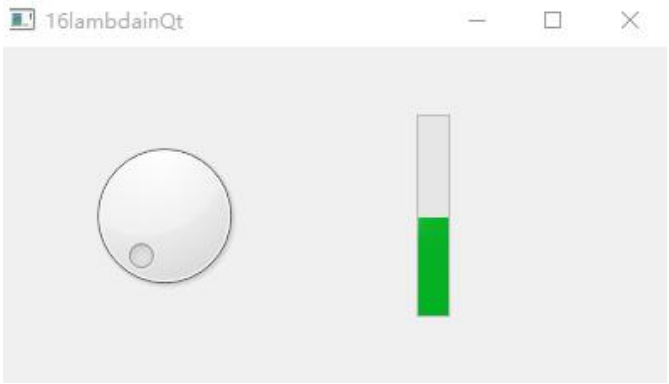
const QObject *sender,

PointerToMemberFunction signal,

Functor functor

)

图示：



代码：

```
#include "mainwindow.h"
```

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    this->setGeometry(600,400,400,200);
    _dial = new QDial(this);
    _dial->setRange(0,100);
    _dial->setGeometry(50,50,100,100);
    _dial->setAutoFillBackground(true);


    _pBar = new QProgressBar(this);
    _pBar->setGeometry(250,40,20,120);
    _pBar->setMinimum(0);
    _pBar->setMaximum(100);
    _pBar->setOrientation(Qt::Vertical);

    //connect(_dial,SIGNAL(valueChanged(int)),
             _pBar,SLOT(setValue(int)));


    //QObject::connect(const QObject *sender,
                  PointerToMemberFunction signal,
                  Functor functor)

     connect(_dial,
           &QDial::valueChanged,
           [&](int v){
              _pBar->setValue(v);
           });
}

MainWindow::~MainWindow()
{
}
```

2.13.4.4.lambda 与仿函数

lambda 本质就是**匿名的仿函数**。

```
#include <iostream>

using namespace std;


int main()
{
    double highrate = 0.3;
    double highbase = 30000;

    auto high = [highrate,highbase](double money){
        return (money-highbase)*highrate;
```

```
    };
    double midrate = 0.3;
    double midbase = 30000;
    auto middle = [midrate,midbase](double money){
        return (money-midbase)*midrate;
    };

    double lowrate = 0.1;
    double lowbase = 3500;
    auto low = [lowrate,lowbase](double money){
        return (money-lowbase)*lowrate;
    };

    cout<<"请输入你的收入:";
    double salary;
    cin>>salary;
    if(salary>30000)
        cout<<high(salary)<<endl;
    else if(salary >15000)
        cout<<middle(salary)<<endl;
    else
        cout<<low(salary)<<endl;

}
```

2.14.enum class(struct)

2.14.1.C -> C++

C 语言中的 enum 对象，是可以被其它非枚举值，赋值，而 C++中的枚举变量，则只能用枚举值来赋值。

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;


enum  Season
{
    Spring, Summer, Autumn, Winter
};

int main()
{
    Season s;
    s = Spring;
```

```
//    s = 100;
    return 0;
}
```

2.14.2.C++->C++11

枚举体的声明和定义使用 enum class 或是 enum struct，二者是等价的。使用 enum class\enum struct 不会与现存的 enum 关键词冲突。而且 enum class\enum struct 具有更好的类型安全和类似封装的特性（scoped nature）。

问题	描述
1	向整形的隐式转换(Implicit conversion to an integer)
2	无法指定底层所使用的数据类型(Inability to specify underlying type)
3	enum 的作用域(Scope)
4	不同编译器解决该问题的方法不统一

2.14.2.1.作用域

类内的枚举

```
#include <iostream>
using namespace std;

//enum  Color { red, green, yellow};

class Painter
{
public:
   enum  Color { red, green, yellow};
public:
    Painter()
    {
    }
    void dis(Color c)
    {
        cout<<c<<endl;
    }
};

int main()
{
//    Painter p;
//    p.dis(red);

    Painter  p;
```

```
    p.dis(Painter::red);
    return 0;
}
```

带作用域的枚举

```
#include <iostream>
using namespace std;

enum class Color { red, green, yellow};

class Painter
{
public:
    Painter()
    {
    }
    void dis(Color c)
    {
        cout<<static_cast<int>(c)<<endl;
    }
};

int main()
{
    Painter p;
//    p.dis(red);
    p.dis(Color::red);
    return 0;
}
```

2.14.2.2.指定类型

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。

```
#include <iostream>
using namespace std;

#include <iostream>

enum class color:int   { red, green, yellow};
enum class colorX:char { red, green, yellow };

int main()
{
    //使用域运算符访问枚举体成员，强转后打印
    std::cout << static_cast<int>(color::red) << std::endl;
    std::cout << static_cast<int>(colorX::red) << std::endl;
    return 0;
```

```
}
```

3.效率 More Effective

前面的章节，近乎 c++到 c++11 的语法糖，但是从章节来看，解决的就是效率的问题，效率是关乎语言生命。

3.1.assert/static_assert

3.1.1.assert

assert 是运行期断言，它用来发现运行期间的错误，**不能提前**到**编译期**发现错误，也不具有强制性，也谈不上改善编译信息的可读性，既然是运行期检查，对性能当然是有影响的，所以经常在发行版本中，assert 都会被关掉。

assert 的关键在于判断 expression 的逻辑真假，如果为 false，就会在 stderr 上面打印一条包含"表达式，文件名，行号"的错误信息，然后调用 abort 结束整个程序。

示例 assert

```
#include <iostream>
#include <assert.h>

using namespace std;

char * myStrcpy(char *dest, const char *src)
{
   assert(dest); assert(src);

   while(*dest++ = *src++);
}

int main(int argc, char *argv[])
{
//    char buf[1024]; char * p = NULL;
//    myStrcpy(buf,p);
//    cout<<buf<<endl;

   FILE *fp = fopen("aa.c","w");
   assert(fp);

   return 0;
}
```

3.1.2.static_assert（提前判误）

static_assert 这个关键字，用来做**编译期间的断言**，因此叫做静态断言。其语法很简单：static_assert(常量表达式，提示字符串)。

如果第一个参数常量表达式的值为真(true 或者非零值)，那么 static_assert 不做任何事情，就像它不存在一样，否则会产生一条编译错误，错误位置就是该 static_assert 语句所在行，错误提示就是第二个参数提示字符串。

使用 static_assert，我们可以在**编译期间发现更多的错误**，用编译器来强制保证一些契约，并帮助我们改善编译信息的可读性，尤其是用于模板的时候。

static_assert 可以用在**全局作用域中，命名空间中**，类作用域中，函数作用域中，几乎可以不受限制的使用。

编译器在遇到一个 static_assert 语句时，通常立刻将其第一个参数作为常量表达式进行演算，但如果该常量表达式依赖于某些模板参数，则延迟到模板实例化时再进行演算，这就让检查模板参数成为了可能。

```
#include <iostream>
using namespace std;

static_assert(
        sizeof(void *) !=4,
       "64-bit code generation is not supported."
       );

/*
该 static_assert 用来确保编译仅在 32 位的平台上进行，不支持 64 位的平台，该语句可以放/
在文件的开头处，这样可以尽早检查，以节省失败情况下的编译时间。
*/


template<typename T, typename U>
int my_bit_copy(T& a, U& b)
{
   static_assert(
             sizeof(a) == sizeof(b),
             "parameters must have same width"
             );
}

int main(int argc, char *argv[])
{
   int a; float b;
   my_bit_copy(a,b);

   char c;
   my_bit_copy(a,c);
   return 0;
}
```

3.2.右值引用(r-value ref) &&

3.2.1.右值定义

通俗来讲，赋值号左边的就是左值，赋值号右边的就是右值。可以取地址是左值，不可以取地址的是右值。C++11，之前没有明确提出右值的概念，所以 C++11 以前这些说活都是正确的。

C++11 中的左值，仍然等同于 C++98 左值。C++11 中的右值，除了 C++98 中的右值以外，增加了将亡值。

对比图如下：

C++98		C++11	
左值	右值	左值	右值
	1-临时对象（函数返回值、运算表达式结果） 2-字面值常量（2、'c'、true）		1-纯右值 == C++98 右值 2-将亡值：C++11 新增的，跟右值引用相关的表达式。 通常是**将要被移动的对象。**

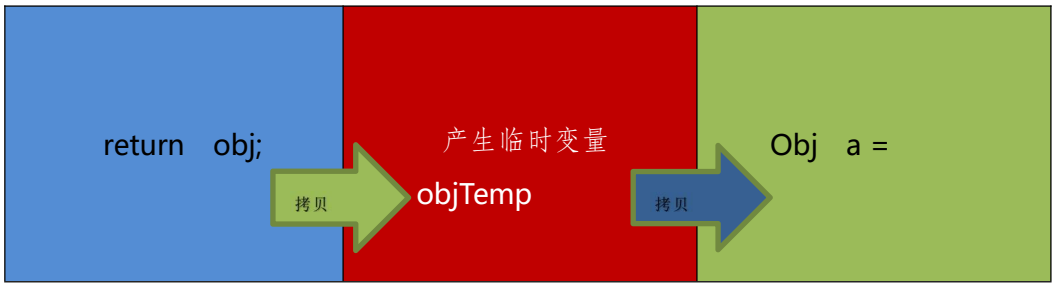
3.2.2.右值引用解决了什么问题

为什么要引入右值引用这个概念，其实就是为了**解决临时对象带来的效率**问题。

比如，我们返回一个临时对象。*在 C++课中，返回栈对象一章中，我们讲过，栈对象是可以返回的，栈对象的引用却不可以返回。*

栈对象返回，如果在没有优化的情况下**-fno-elide-constructors**，会产生临时对象。

过程如下：



配置如下：

```
TEMPLATE = app
CONFIG += console c++11
CONFIG -= app_bundle
CONFIG -= qt

SOURCES += main.cpp

QMAKE_CXXFLAGS += -fno-elide-constructors
```

示例如下：

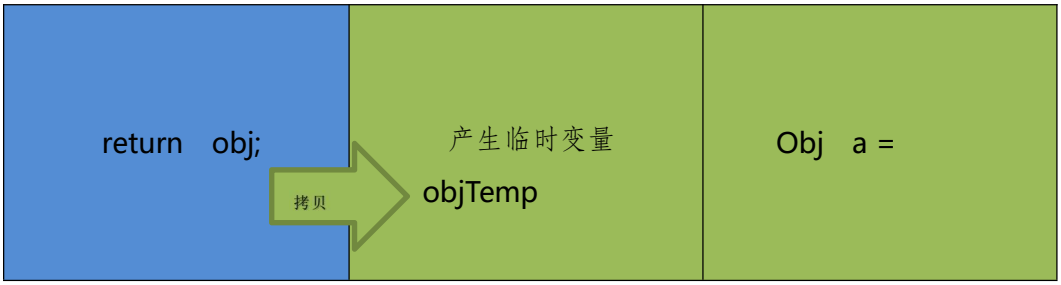
```
#include <iostream>
using namespace std;
class A
{
public:
    A(){
        cout<<"A() "<<this<<endl;
    }
    ~A(){
        cout<<"~A() "<<this<<endl;
    }
    A(const A &another)
    {
        cout<<"A(const A&)"<<&another<<"->"<<this<<endl;
    }
    void dis()
    {
        cout<<"xxxxooooooooooo"<<endl;
    }
};
A getObjectA()
{
    return A();
}

int main(int argc, char *argv[])
{
    A a = getObjectA();
    return 0;
}
```

在没有优化的情况下，要经历两次复制，优后以后，各个平台大不相同(有构造一次的，也有拷贝一次的)。

对此，c++也是志在解决平台不一致的问题。

```
int main(int argc, char *argv[])
{
    A&& ra =getObjectA();
    return 0;
}
```

此时可以明确的右值引用的意义了。



3.2.3.const T & 万能常引用

在没有出现右值引用之前，也有一个种方案，可与右值引用相媲美。那就是 const T & 方案。此方案完美吗？ 为什么会有 const 重载？

3.2.3.1.万能乎?

const 可以实现**常量引用，不同类型之引用**（非基本数据类型也适用，但需要转化构造函数）。

其本质也是产生了临时对象，并且该临时对象是 const 的。

```
int main(int argc, char *argv[])
{
    int& ri = 5;
    const & cri  = 5;

    float f = 34.5;
    int & irf = f;
    const int & cirf = f;

    A objA;
    int& irA  = objA;
    const int&  cirA = objA;
    cout<<cirA<<endl;

    const A & cra = getObjectA();
    cra.dis();
    return 0;
}
```

3.2.3.2.必要性

如果一个函数的参数是，非 const 类型，此时传递临时对象过来，则会编译不过。要想编译能过，则必须加 const 修饰 。

3.2.3.3.从 const T & 到 &&

测试如下案例，第一个拷贝两次，第二个，拷贝一次,且 const 不可去掉，但有缺陷，第三个，则近乎完美。

```
#include <iostream>
using namespace std;

class Copyable
{
public:
    Copyable(){}
```

```
    Copyable(const Copyable & another)
    {
        cout<<this<<" Copy from "<<&another<<endl;
    }
};

Copyable getCopyable()
{
    return Copyable();
}

void acceptValue(Copyable o)
{
}
void acceptRef(const Copyable & o)
{
}
void accetpRRef(Copyable && o)
{
}

int main()
{
//    acceptValue(getCopyable());
//    acceptRef(getCopyable());
    accetpRRef(getCopyable());
    return 0;
}
```

3.2.4.缺陷

此种举措，也可以在返回中避免产生临时对象，再次拷贝和销毁。但时临时对象的性质是 const 的，也会给后续的使用带来不便。

const 函数重载也是在这种反值 const 类型的情型下诞生的，const 对象，只能调用 const 成员函数。

```
int main(int argc, char *argv[])
{
    const A & cra = getObjectA();
    cra.dis();
    return 0;
}
```

3.2.5.与左值引用的对比

- 都属于引用类型。
- 都必须初始化。左值引用是**具名变量**值的别名，右值引用是**匿名变量**的别名。
- 左值引用，用于传参，好处在于，**扩展**对象的作用域。则右值引用的作用就在于

延长了临时对象的生命周期。

- 避免"先拷贝再废弃"带来的性能浪费。
- 对将亡值进行引用的类型；它存在的目的就是为了实现移动语义。

3.3.移动构造 move constuctor

3.3.1.拷贝语意

对于类中，含有指针的情况,即含有堆内存资源的情况，要自实现其拷贝构造和拷贝赋值。也就是所谓的深拷贝和深赋值。我想这已经成为一种共识了。

比如，如下类：

```
#include <iostream>
using namespace std;
class HasPtrMem
{
public:
    HasPtrMem():_d(new int(100)){
        cout<<"HasPtrMem()"<<this<<endl;
    }
    HasPtrMem(const HasPtrMem& another)
        :_d(new int(*another._d)){
        cout<<"HasPtrMem(const HasPtrMem& another)"<<
             this<<"->"<<&another<<endl;
    }
    ~HasPtrMem(){
        delete _d;
        cout<<"~HasPtrMem()"<<this<<endl;
    }

    int * _d;
};

HasPtrMem getTemp()
{
    return HasPtrMem();
}

int main(int argc, char *argv[])
{
//    HasPtrMem a;
//    HasPtrMem b(a);
//    cout<<*a._d<<endl;
//    cout<<*b._d<<endl;

     HasPtrMem&& ret = getTemp();

    return 0;
}
```

上面的过程，我们已经知晓，ret 作为右值引用，引用了临时对象，由于临时对象是待返回对象的复本，所以表面上看起来是，**待返回对象的作用域扩展了，生命周期也延长了**。

3.3.2.移动语义

前面我们建立起来了一个概念，就是右值引用。用右值引用的思想，再来实现一下拷贝。

这样，顺便把**临时对象的问题**也解决了。

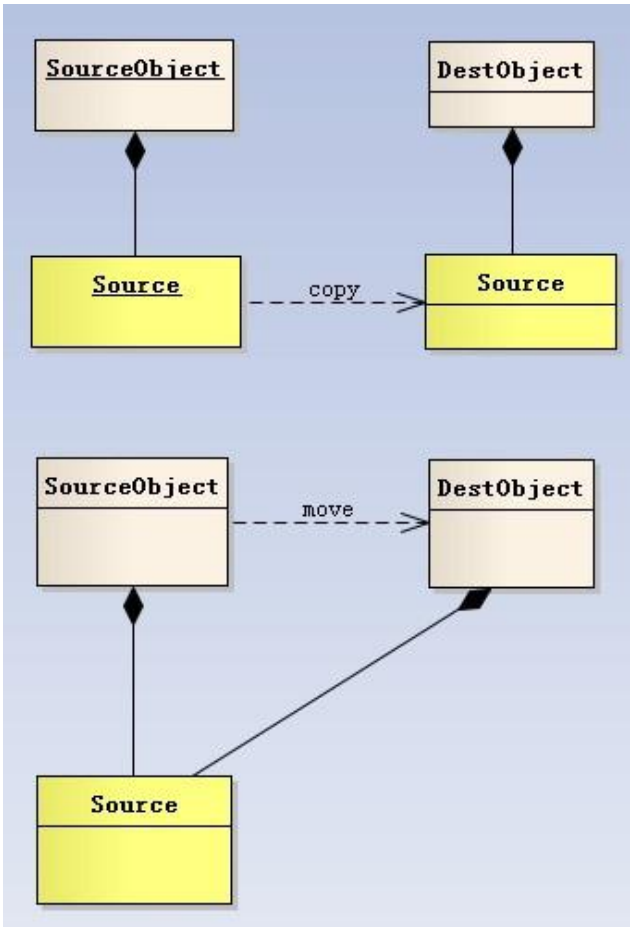
```
#include <iostream>
using namespace std;

class HasPtrMem
{
public:
    HasPtrMem():_d(new int(0))
    {
       cout<<"HasPtrMem()"<<this<<endl;
   }
   HasPtrMem(const HasPtrMem& another)
       :_d(new int(*another._d))
   {
       cout<<"HasPtrMem(const HasPtrMem& another)"<<
            this<<"->"<<&another<<endl;
   }
   HasPtrMem(HasPtrMem &&another)
   {
       cout<<this<<" Move resourse from "<<
            &another<<"->"<<another._d<<endl;

       _d = another._d;
       another._d = nullptr;
   }

    ~HasPtrMem()
    {
       if(_d != nullptr)
          delete _d;
       cout<<"~HasPtrMem()"<<this<<endl;
   }

   int * _d;
};

HasPtrMem getTemp()
{
   return HasPtrMem();
}
```

```
int main(int argc, char *argv[])
{
    HasPtrMem a  = getTemp();
    return 0;
}
```

3.3.3.移动构造（所有权）

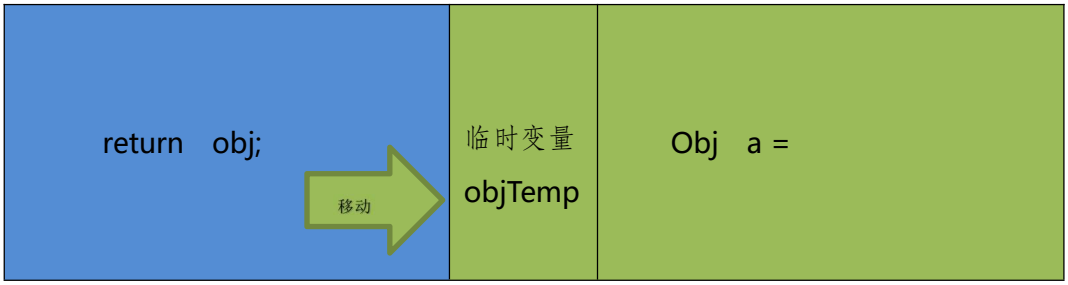
栈对象，返回，**拷贝不可避免**， 能不能减少拷贝的代价？？

如下是，移动构造函数。我们借用临时变量，将待返回对象的内容"偷"了过来。移动构造的本质，也是一种**浅拷贝**，但此时的浅拷贝已经**限定了其使用的场景**，故而是安全的。




```
HasPtrMem(HasPtrMem &&another)
{
    cout<<this<<" Move resourse from "<<
         &another<<"->"<<another._d<<endl;

    _d = another._d;
    another._d = nullptr;
```

```
}
```



移动构造函数，最大的用途避免同一份内存数据的不必要的变成两份甚至多份、过程中的变量传递导致的内存复制，另外解除了栈变量对内存的引用。

3.3.4.关于默认

对于不含有资源的对象来说，自实现拷贝与移动语义并没有意义，对于这样的类型而言移动就是拷贝，拷贝就是移动。

拷贝构造/赋值 和 移动构造/赋值，**必须同时提供或是同时不提供**。才能保证同时具有拷贝和移动语义。只声明一种的话，类只能实现一种语义。

只有拷贝语义的类，也就是 C++98 中的类。而只有移动语义的类，表明该类的变量所拥有的资源只能被移动，而不能被拷贝。那么这样的资源必须是唯一的。只有移动语义构造的类型往往时"资源型"的类型。

比如智能指针，文件流等。

class A		
类型	**默认**	**自实现**
构造 A()	空	一经实现不再提供
拷贝构造 A(const A&)	等位拷贝	若有深拷贝的需求，两者通常均要自实现。若拷贝自实现，**移动默认即无**，即本资源，可以拷贝不可以移动。
拷贝赋值 A&operator（const A &）	等位赋值	
移动构造 A(A&&)	等位拷贝	若有移动的需求，两者通常均要自实现。若移动自实现，**拷贝默认即无**，即本资源可以移动，不可以拷贝。
移动赋值 A&operator（A&& ）	等位赋值	
析构 ~A()	空	

```
#include <iostream>
#include <memory>
using namespace std;
```

```
class Copy
{
public:
    Copy(int i):_i(new int(i))
    {
        cout<<"Copy(int i)"<<endl;
    }
    Copy(const Copy & another)
        :_i(new int(*another._i))
    {
        cout<<"  Copy(const Copy & another)"<<endl;
    }
    Copy & operator=(const Copy & another)
    {
        cout<<"  Copy & operator=(Copy & another)"<<endl;
        if(this == &another) return *this;
        delete _i;
        _i = new int(*another._i);
        return *this;
    }

    ~Copy()
    {
        delete _i;
        cout<<"~Copy()"<<endl;
    }

    int *_i;
};

class Move
{
public:
    Move(int i):_i(new int(i))
    {
        cout<<"Move(int i)"<<endl;
    }
    Move( Move && another)
    {
        cout<<" Move( Move && another)"<<endl;
        _i = another._i;
        another._i = nullptr;
    }
    Move & operator=(Move && another)
    {
        cout<<" Move & operator=(Move && another)"<<endl;

        if(this != &another)
        {
            delete _i;
```

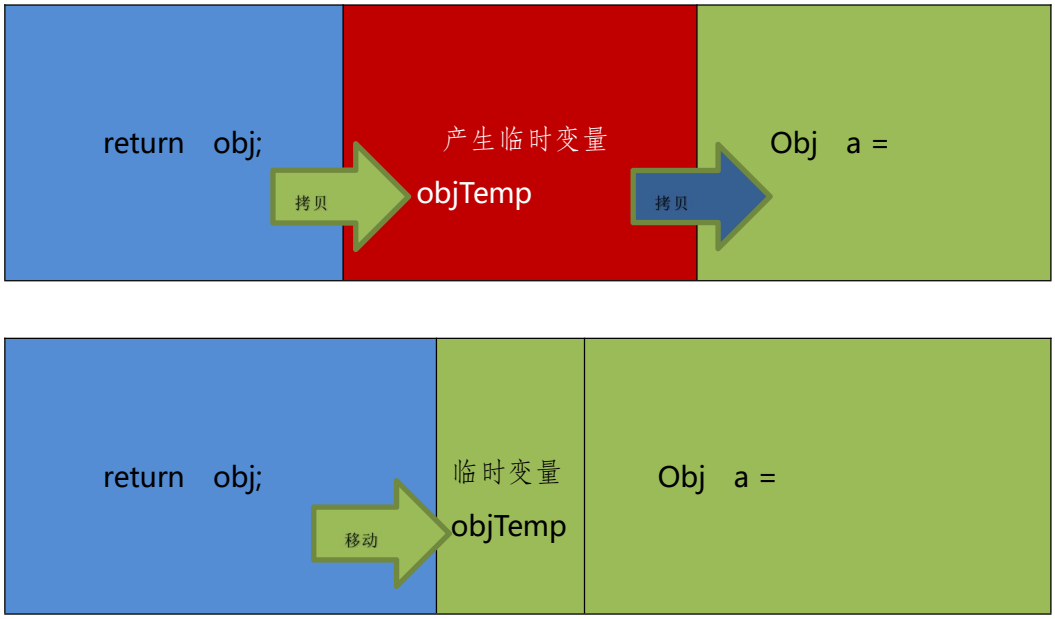
```
            _i = another._i;

            another._i = nullptr;
        }
        return *this;
    }

    ~Move()
    {
        if(_i != nullptr)
            delete _i;
        cout<<"~Move()"<<endl;
    }

    int *_i;
};

Copy getCopy()
{
    return Copy(1);
}

Move getMove()
{
    return Move(1);
}


int main(int argc, char *argv[])
{
//    Move m(100);
//    Move n(m);          //编译不过，没有拷贝构造器，两者的互斥性

//    Copy  rc = getCopy();
//    Move  rm = getMove();

    Copy  &&rrc = getCopy();
    Move  &&rrm = getMove();
    return 0;
}
```

3.3.5.慎用移动

拷贝，无疑是安全的，而移动，无疑是高效的。但是这种高效前提你对资源的深刻的理解，否则可能带来极大的安全隐患的，所以要慎用。

移动解决了，如下第一个拷贝的效率问题，引用，解决了，如下第二个拷贝的效率问题。

C++11 注定是高效的。



3.4.移动化 std::move

3.4.1.move 本质

3.4.1.1.语义

move 的名字本身比较迷惑，其本身并没有任何"移动"，它唯一的功能，就是将左值转化为右值，完成移动语义，继而我们可以使用右值。

如果说，产生临时对象，**隐式**的调用了移动构造，std::move 是一种显示调用移动构造的方法。即**刻意**营造一种可以使用移动的情景。

3.4.1.2.功用

1、减少内存复制成本

2、将不再需要的变量，取消它对原先持有变量(内存)的持有(修改)权限

3.4.1.3.测试

```
#include <iostream>
#include <memory>
using namespace std;

class Move
{
public:
    Move(int i):_i(new int(i))
    {
        cout<<"Move(int i)"<<endl;
    }
    Move( Move && another)
    {
```

```
        cout<<" Move( Move && another)"<<endl;
        _i = another._i;
        another._i = nullptr;
    }
    Move & operator=(Move && another)
    {
        cout<<" Move & operator=(Move && another)"<<endl;

        if(this != &another)
        {
            delete _i;
            _i = another._i;

            another._i = nullptr;
        }
        return *this;
    }

    ~Move()
    {
        if(_i != nullptr)
            delete _i;
        cout<<"~Move()"<<endl;
    }

    int *_i;
};


int main(int argc, char *argv[])
{
//    Move m(100);
//    Move n(m);          //编译不过，没有拷贝构造器，两者的互斥性

    Move  m(100);
    Move  n(std::move(m));
    return 0;
}
```

3.4.2.应用

可以 move 的类中，**其父类成员，和成员对象**的也要支持 move，就要借助 std::move，这样继承自父类的部分也可以 move。

```
#include <iostream>
using namespace std;
class Complex
{
public:
```

```
    Complex(int f = 0)
        :_f(new float(f))
    {
        cout<<"Complex()"<<endl;
    }
    Complex(const Complex & another)
        :_f(new float(*another._f))
    {
        cout<<"Complex(const Complex & another)"<<endl;
    }
    Complex(Complex && another)
        :_f(another._f)
    {
        another._f = nullptr;
        cout<<" Complex(Complex && another)"<<endl;
    }
    Complex& operator = (const Complex & another)
    {
        cout<<"Complex& operator = (const Complex & another)"<<endl;
        if(this != &another)
        {
            delete _f;
            _f = new float;
            *_f = *another._f;
        }
        return *this;
    }
    Complex& operator = ( Complex && another)
    {
        cout<<" Complex& operator = (const Complex && another)"<<endl;
        if(this != &another)
        {
            delete _f;
            _f = another._f;
        }
        return *this;
    }
    ~Complex()
    {
        if(_f != nullptr)
            delete _f;
    }
    float *_f;
};
class Moveable
{
public:
    Moveable(int i)
        :_i(new int(100)),_c(2.1)     //先父类,再类对象,再本类
    {
```

```
        cout<<" Moveable(int i)"<<endl;
    }
    Moveable(const Moveable &another )
        :_i(new int(*another._i))
        ,_c(another._c)
    {
        cout<<" Moveable(const Moveable &another )"<<endl;
    }
    Moveable(Moveable &&another)
    {
        cout<<"  Moveable(Moveable &&another)"<<endl;
        _i = another._i;
        another._i = nullptr;
        _c = std::move(another._c);     //走 Complex 移动赋值
    }
    ~Moveable()
    {
        if(_i != nullptr)
            delete _i;
    }
    int *_i;
    Complex _c;       //类对象成员
};
int main(int argc, char *argv[])
{
    Moveable  m(100);
    Moveable  n(std::move(m));  //走移动构造

    return 0;
}
```

3.5.移动赋值 move operator=

3.5.1.代码

此段代码，是微软 msdn 上给出的 MemoryBlock 的示例。非常标准的展示了移动构造，移动赋值，非常有借鉴意义。

```
https://msdn.microsoft.com/en-us/library/dd293665.aspx
```

```
// MemoryBlock.h
#include <iostream>
#include <algorithm>
class MemoryBlock
{
public:

   // Simple constructor that initializes the resource.
```

```
explicit MemoryBlock(size_t length)
   : _length(length)
   , _data(new int[length])
{
   std::cout << "In MemoryBlock(size_t). length = "<<
               _length << "." << std::endl;
}

// Destructor.
~MemoryBlock()
{
   std::cout << "In ~MemoryBlock(). length = "<<
               _length << ".";

   if (_data != nullptr)
   {
      std::cout << " Deleting resource.";
      // Delete the resource.
      delete[] _data;
   }

   std::cout << std::endl;
}

// Copy constructor.
MemoryBlock(const MemoryBlock& other)
   : _length(other._length)
   , _data(new int[other._length])
{
   std::cout << "In MemoryBlock(const MemoryBlock&). length = "<<
               other._length << ". Copying resource." << std::endl;

   std::copy(other._data, other._data + _length, _data);
}

// Copy assignment operator.
MemoryBlock& operator=(const MemoryBlock& other)
{
   std::cout << "In operator=(const MemoryBlock&). length = "<<
               other._length << ". Copying resource." << std::endl;

   if (this != &other)
   {
      // Free the existing resource.
      delete[] _data;

      _length = other._length;
      _data = new int[_length];
      std::copy(other._data, other._data + _length, _data);
   }
```

```
    return *this;
  }
// Move constructor.
MemoryBlock(MemoryBlock&& other)
  : _data(nullptr)
  , _length(0)
{
  std::cout << "In MemoryBlock(MemoryBlock&&). length = "<<
            other._length << ". Moving resource." << std::endl;

  // Copy the data pointer and its length from the
  // source object.
  _data = other._data;
  _length = other._length;

  // Release the data pointer from the source object so that
  // the destructor does not free the memory multiple times.
  other._data = nullptr;
  other._length = 0;
}
// Move assignment operator.
MemoryBlock& operator=(MemoryBlock&& other)
{
  std::cout << "In operator=(MemoryBlock&&). length = "<<
            other._length << "." << std::endl;

  if (this != &other)
  {
    // Free the existing resource.
    delete[] _data;

    // Copy the data pointer and its length from the
    // source object.
    _data = other._data;
    _length = other._length;

    // Release the data pointer from the source object so that
    // the destructor does not free the memory multiple times.
    other._data = nullptr;
    other._length = 0;
  }
  return *this;
}
// Retrieves the length of the data resource.
size_t Length() const
{
   return _length;
}
private:
  size_t _length; // The length of the resource.
```

```
   int* _data; // The resource.
};
```

3.5.2.更简洁的移动构造

```
// Move constructor.
MemoryBlock(MemoryBlock&& other)
   : _data(nullptr)
   , _length(0)
{
   *this = std::move(other);
}
```

3.5.3.测试

见下一章

3.6.SIX BIG DEFAULFT FUNCTION

```
#include <iostream>
using namespace std;
```

3.6.1.成员 class C

```
class C{
public:
    C(int c){}
    C(const C & another){
        cout<<"C&"<<endl;
    }
    C(C &&another){
        cout<<"C&&"<<endl;
    }
    C &operator=(const C & another){
        if(this != &another)
            cout<<"C="<<endl;
        return *this;
    }
    C &operator=(C && another){
        if(this != &another)
            cout<<"C&&="<<endl;
        return *this;
    }
    ~C(){}

};
```

3.6.2.父类 class A

```
class A{
public:
    A(int a){}
    A(const A & another){
        cout<<"A&"<<endl;
    }
    A(A &&another){
        cout<<"A&&"<<endl;
    }
    A &operator=(const A &another){
        if(this != &another)
            cout<<"A ="<<endl;
        return *this;
    }
    A &operator=(A &&another){
        if(this != &another)
            cout<<"A &&="<<endl;
        return *this;
    }
};
```

3.6.3.子类 class B

```
class B:public A{
public:
    B(int b, int a):A(a),c(1){
    }

    B(const B & another)
        :A(another),c(another.c){
        cout<<"B&"<<endl;
    }
    B(B &&another)
        :A(move(another)),c(move(another.c)){
        cout<<"B&&"<<endl;
    }
    B &operator=(const B & another){
        if(this != &another){
            A::operator=(another);
            cout<<"B ="<<endl;
            c = another.c;
        }
        return *this;
    }
    B &operator=(B && another){
        if(this != &another){
            c = std::move(another.c);
            A::operator=(std::move(another));
            cout<<"B &&="<<endl;
```

```
        }
        return *this;
    }
private:
    C c;
};

int main()
{
    //    B a(1,2);
    //    B b(a);
    //    B c(0,0);
    //    c = b;
    B  a(1,2);
    B b(0,0);
    b = move(a);

    return 0;
}
```

3.7.STL 中应用 move 语义

3.7.1.原理

STL 容器对象中的元素，均在堆上，而通常的作法是将栈对象压入，此时完了一次，栈对象的构造，然后再完成一次到堆对象的拷贝。此时效率极低。

第二个效率问题，就是像 vector 内存不足的时候，自动开辟新的空间，要对原有空间作一次深拷贝，此时效率也是极低的。

对象的创建是不可避免的，但是可以移动到容器中去。

3.7.2.应用 pushback

```
#include <iostream>
#include <vector>
#include <memory>
using namespace std;

class Move
{
public:
    Move(int i):_i(new int(i))
    {
        cout<<"Move(int i)"<<endl;
    }
    Move( Move && another)
    {
        cout<<"    Move( Move && another)"<<endl;
```

```
        _i = another._i;
        another._i = nullptr;
    }
    Move & operator=(Move && another)
    {
        cout<<"    Move & operator=(Move && another)"<<endl;

        if(this != &another)
        {
            delete _i;
            _i = another._i;

            another._i = nullptr;
        }
        return *this;
    }

    ~Move()
    {
        if(_i != nullptr)
            delete _i;
        cout<<"~Move()"<<endl;
    }

    int *_i;
};


int main(int argc, char *argv[])
{
    vector<Move>  vm;
//    vm.reserve(100);
    for(int i=0; i<100; i++)
        vm.push_back(Move(i));       //性能高的飞起
    return 0;
}
```

3.7.3.应用 insert

```
int main()
{
  // Create a vector object and add a few elements to it.
  vector<MemoryBlock> v;
  v.push_back(MemoryBlock(25));
  v.push_back(MemoryBlock(75));
   v.push_back(MemoryBlock(75));

  // Insert a new element into the second position of the vector.
  v.insert(v.begin() + 1, MemoryBlock(50));
   //插入高效 //性能高的飞起
```

```
}
```

3.8.完美转发 perfect forwarding

3.8.1.语义

右值引用类型是独立于值的，一个右值引用参数作为函数的形参，在函数内部再转发该参数的时候它已经变成一个左值了，并不是它原来的类型了。因此，我们需要一种方法能按照参数原来的类型转发到另一个函数，这种转发被称为完美转发。

所谓完美转发（perfect forwarding），是指在函数模板中，完全依照模板的参数的类型，将参数传递给函数模板中调用的另外一个函数。c++11 中提供了这样的一个函数 std::forward，它是为转发而生的，它会按照参数本来的类型来转发出去，不管参数类型是 T&&这种未定的引用类型还是明确的左值引用或者右值引用。

3.8.2.测试

```
#include <iostream>

using namespace std;

template<typename T>
void PrintT(T& t)
{
    cout << "lvaue" << endl;
}

template<typename T>
void PrintT(T && t)
{
    cout << "rvalue" << endl;
}

template<typename T>
void TestForward(T && v)
{
    PrintT(v);
    PrintT(std::forward<T>(v));
    PrintT(std::move(v));
}

int main()
{
    TestForward(1);
    int x = 1;
    TestForward(x);
    TestForward(std::forward<int>(x));
    return 0;
}
```

3.8.3.解析

TestForward(1);由于 1 是右值，所以未定的引用类型 T && v 被一个右值初始化后变成了一个右值引用，但是在 TestForward 函数体内部，调用 PrintT(v);时，v 又变成了一个左值，因为它这里已经变成了一个具名的变量，所以它是一个左值，因此第一个 PrintT 被调用，打印出"lvaue";

PrintT(std::forward<T>(v));由于 std::forward 会按参数原来的类型转发，因此，这时它还是一个右值（这里已经发生了类型推导，所以这里的 T&&不是一个未定的引用类型），所以会调用 void PrintT(T &&t)函数。

PrintT(std::move(v));是将 v 变成一个右值引用，虽然它本来也是右值引用，因此它和 PrintT(std::forward<T>(v));的输出结果是一样的。

3.8.4.引用折叠（Reference collapsing）规则：

1. 所有的右值引用叠加到右值引用上变成一个右值引用
2. 所有的其它引用类型叠加都变成一个左值引用
3. 左值或者右值是独立于它的类型的，也就是说一个右值引用类型的左值是合法的。

3.9.函数包装 Function wapper

3.9.1.引入

3.9.1.1.多态统一接口

```
#include <iostream>
#include <functional>
using namespace std;

class Oper
{
public:
    virtual int op(int,int) = 0;
};

class OperAdd:public Oper
{
public:
    int op(int i,int j)
    {
        return i+j;
    }
};
class OperMinus:public Oper
{
```

```
public:
    int op(int i,int j)
    {
        return i-j;
    }
};

int main()
{
    Oper *oper = new OperAdd;
    cout<<oper->op(4,5)<<endl;
    oper = new OperMinus;
    cout<<oper->op(10,4)<<endl;
    return 0;
}
```

3.9.2.function 语义

类模版 std::function 是**可调用对象**的包装器,可以包装**除了类成员函数之外**的所有可调用对象。包括，普通函数，函数指针， lambda，仿函数。

通过指定的模板参数，它可以用统一的方式保存，并延迟执行它们。所谓的延迟执行，就是回调了。

3.9.2.1.语法

```
template <class Ret, class... Args> class function<Ret(Args...)>;
```

3.9.2.2.统一接口

以下程序，分别对普通函数，函数指针，lambda 及仿函数作了接口统一处理。此举实现的高度的统一。

```
#include <iostream>
#include <functional>
#include <map>
using namespace std;

int add(int i,int j)  //普通函数
{
    return i+j;
}
int _minus(int i,int j)
{
    return i-j;
}

typedef int(*MINUS)(int,int);  //函数指针

auto multiply = [](int i,int j){return i*j;}; //lambda
```

```
class Divide  //仿函数
{
public:
    int operator()(int i, int j){
        return i/j;
    }
};

int main()
{
    std::function<int(int,int)> oper;
    oper = add;
    cout<<oper(1,2)<<endl;

    MINUS m = _minus;
    oper = m;
    cout<<oper(10,1)<<endl;

    oper = multiply;
    cout<<oper(1,2)<<endl;

    oper = Divide();
    cout<<oper(10,2)<<endl;

    map<string,std::function<int(int,int)>> math;

    math.insert({"+",add});
    math.insert({"-",_minus});
    math.insert({"*",multiply});
    math.insert({"/",Divide()});
    math.insert({"%",[](int i, int j){return i%j;}});

    cout<<math["+"](10,5)<<endl;
    cout<<math["-"](10,5)<<endl;
    cout<<math["*"](10,5)<<endl;
    cout<<math["/"](10,5)<<endl;
    cout<<math["%"](10,5)<<endl;


    return 0;
}
```

3.9.2.3. 多态

```
#include <iostream>
#include <list>
#include <functional>
using namespace std;
```

```
class FunctorA
{
public:
    void operator()(){
        cout<<"class Functor  A"<<endl;
    }
};
class FunctorB
{
public:
    void operator()(){
        cout<<"class Functor  B"<<endl;
    }
};
class FunctorC
{
public:
    void operator()(){
        cout<<"class Functor  C"<<endl;
    }
};


class Object
{
public:
    Object(FunctorA a, FunctorB b,FunctorC c)
    {
        _list.push_back(a);
        _list.push_back(b);
        _list.push_back(c);
    }
    void notify()
    {
        for(auto &item:_list)
        {
            item();
        }
    }

private:
    list<function<void(void)>> _list;
};

int main()
{
    FunctorA a; FunctorB b;  FunctorC c;

    Object  obj(a,b,c);
    obj.notify();
```

```
    return 0;
}
```

3.9.3.应用

3.9.3.1.fucntion 作参数类型实现回调

```
#include <iostream>
#include <functional>
using namespace std;

bool compare(int x, int y)
{
    return x>y;
}

void selectSort(int *p, int n, function<bool(int, int)> pCompare)
{
    for(int i=0; i<n-1; i++)
    {
        for(int j=i+1; j<n; j++)
        {
            if(pCompare(p[i] , p[j]))
            {
                p[i] = p[i] ^ p[j] ;
                p[j] = p[i] ^ p[j] ;
                p[i] = p[i] ^ p[j] ;
            }
        }
    }
}

int main()
{
    int arr[10] = {1,3,5,7,9};
    selectSort(arr,10,[](int x ,int y){return x>y;});

    for(auto i:arr)
        cout<<i<<endl;
    return 0;
}
```

3.9.3.2.fucntion 作类成员实现回调

```
#include <iostream>
#include <functional>

using namespace std;
```

```
class functor
{
public:
    void operator()()
    {
        cout<<__FUNCTION__<<endl;
    }
};

class A
{
public:
    A(const function<void()> & cb)
        :_callback(cb)
    {}
    void notify()
    {
        _callback();
    }

    function<void()> _callback;
};

int main(int argc, char *argv[])
{
    functor  fct; //事先定义好的函数

    A a(fct);
    a.notify();

    return 0;
}
```

观察者模式

```
#include <iostream>
#include <list>
#include <functional>
using namespace std;


class FunctorA
{
public:
    void operator()(){
        cout<<"class Functor  A"<<endl;
    }
};
class FunctorB
{
```

```
public:
    void operator()(){
        cout<<"class Functor  B"<<endl;
    }
};
class FunctorC
{
public:
    void operator()(){
        cout<<"class Functor  C"<<endl;
    }
};


class Object
{
public:
    Object(FunctorA a, FunctorB b,FunctorC c)
    {
        _list.push_back(a);
        _list.push_back(b);
        _list.push_back(c);
    }
    void notify()
    {
        for(auto &item:_list)
        {
            item();
        }
    }

private:
    list<function<void(void)>> _list;
};

int main()
{
    FunctorA a; FunctorB b;  FunctorC c;

    Object  obj(a,b,c);
    obj.notify();

    return 0;
}
```

3.9.3.3.cocos 中应用

cosos 中的屏幕触摸事件，就是采用了 std::function

```
typedef std::function<bool(Touch*, Event*)> ccTouchBeganCallback;
```

```
typedef std::function<void(Touch*, Event*)> ccTouchCallback;
ccTouchBeganCallback      onTouchBegan;
ccTouchCallback          onTouchMoved;
ccTouchCallback          onTouchEnded;
ccTouchCallback          onTouchCancelled;
```

```
auto listener = EventListenerTouchOneByOne::create();
listener->onTouchBegan = [](Touch *t, Event *e) {
   Point  p = t->getLocation();
   log("x = %f y = %f", p.x, p.y);
   return true;
};
listener->setSwallowTouches(true);
auto director = CCDirector::getInstance();
auto dispatcher = director->getEventDispatcher;
dispatcher->addEventListenerWithSceneGraphPriority(listener,this);
```

3.10.绑定函数参数 Bind function arguments

3.10.1.bind 语义

bind 用来将**可调用对象**和**参数**一起进行绑定。可调用对象包括普通函数、全局函数、静态函数、类静态函数甚至是类成员函数，参数包括普通参数和类成员。绑定后的结果，可以使用 std::function 进行保存，并**延迟调用**到我们需要的时候。

需要头文件:

```
#include <functional>
```

和命名空间:

```
using namespace std;
using namespace std::placeholders;
```

3.10.1.1.绑定普通函数与参数及占位

bind 绑定顺序，也就是函数中形参的声明顺序。placeholders::_x 中的序列是实参的顺序。

```
#include <iostream>
#include <functional>
using namespace std;


double myDivide (double x, double y)
{
   return x/y;
}

int main ()
```

```
{

    using namespace std::placeholders;

    // 零个参数 returns 10/2
    auto fn_five = std::bind (myDivide, 10, 2);
    cout << fn_five() << endl;


    // 一个参数 returns  x/2    auto 不可省
    auto fn_half = std::bind (myDivide,std::placeholders::_1,2);
    cout << fn_half(10) <<endl;
     // 一个参数 returns 2/x
    auto fn_half2 = std::bind (myDivide,2,std::placeholders::_1);
    cout << fn_half2(10) <<endl;


    // 两个参数 returns int  x/y
    auto fn_rounding = std::bind<int> (myDivide,_1,_2);
    cout << fn_rounding(10,3) << endl;


    // 反转参数 returns      y/x
    auto fn_invert = std::bind (myDivide, _2,  _1);
    cout << fn_invert(10,2) << endl;

    return 0;
}
```

3.10.1.2.绑定对象与成员及占位

绑定对象及成员函数时，顺序是调用成员，对象，[参数]。成员，对象及参数不**能有缺位，缺位可用 placeholders::_x 来占位**。否则编译不过。

```
#include <iostream>
#include <functional>
using namespace std;
using namespace std::placeholders;

struct MyPair
{
    double a;
    int b;
};
class MyPair2
{
public:
    int add(int x, int y)
```

```
    {
        return x + y;
    }
};

int main()
{
    struct MyPair mp = {1,2};

    auto bindObj = bind(&MyPair::a,mp);
    cout<<bindObj()<<endl;

    auto bindObj2 = bind(&MyPair::b,mp);
    cout<<bindObj2()<<endl;

    class MyPair2 mp2;
    auto bindObjfunc = bind(&MyPair2::add,mp2 , 2, 3);
    cout<<bindObjfunc()<<endl;

    auto bindObjfunc2 = bind(&MyPair2::add,_1, 2, 3);
    cout<<bindObjfunc2(mp2)<<endl;

    auto bindObjfunc3 = bind(&MyPair2::add,_1,_2, 3);
    cout<<bindObjfunc3(mp2,100)<<endl;

    auto bindObjfunc4 = bind(&MyPair2::add,_1,_2,_3);
    cout<<bindObjfunc4(mp2,100,200)<<endl;
    return 0;

}
```

3.10.1.3.函数重载情形

```
#include <iostream>
#include <functional>
using namespace std;
using namespace std::placeholders;


int add(int x, int y)
{
    cout<<"int add(int x, int y)"<<endl;
    return x + y;
}
double add(double x, double y)
{
    cout<<"double add(double x, double y)"<<endl;
    return x + y;
}
```

```
class MyPair
{
public:
    int add(int x, int y)
    {
        cout<<"MyPair:: int add(int x, int y)"<<endl;
        return x + y;
    }
    double add(double x, double y)
    {
        cout<<"double add(double x, double y)"<<endl;
        return x + y;
    }
};

int main()
{

    auto bindGlobalFunc =
          bind((int(*)(int,int))add,_1,_2);
    cout<<bindGlobalFunc(1,2)<<endl;

    auto bindGlobalFunc2 =
          bind(static_cast<double(*)(double,double)>(add),_1,_2);
    cout<<bindGlobalFunc2(1,2)<<endl;

    MyPair mp;

    auto bindMemberFunc =

    bind(static_cast<double(MyPair::*)(double,double)>(&MyPair::add),mp,
    1,2);
    cout<<bindMemberFunc()<<endl;

    return 0;
}
```

3.10.2.bind 语义小结

（1）bind 预先绑定的参数需要传具体的变量或值进去，对于预先绑定的参数，是 pass-by-value 的。

（2）对于不事先绑定的参数，需要传 std::placeholders 进去，从_1 开始，依次递增。placeholder 是 pass-by-reference 的。

（3）bind 的返回值是可调用实体，可以直接赋给 std::function 对象。

（4）对于绑定的指针、引用类型的参数，使用者需要保证在可调用实体调用之前，这些参数是可用的。

（5）**类的 this 可以通过对象或者指针来绑定。**

3.10.3.多态之 bind +fucntion

function 本是不可以包装类成员函数，但是 bind 的可以实现类成员函数的绑定，然后赋给 fucntion 对象，亦即实现了间接性的包装。

bind+function 可以实现接口统一的极大整合,这是多态吗?

```
#include <iostream>
#include <functional>
using namespace std;

void foo()
{
    cout<<"void foo()"<<endl;
}


void func(int a)
{
    cout<<"void func(int a)"<<endl;
}

class Foo
{
public:
    void method()
    {
        cout<<"Foo::void method()"<<endl;
    }
    void method2(string s)
    {
        cout<<"Foo:void method2()"<<endl;
    }
};

class Bar
{
public:
    void method3(int a ,string s)
    {
        cout<<"Bar:void method3()"<<endl;
    }
};
```

```
std::function<void(void)> f;


int main()
{
    f = std::bind(foo);
    f();

    f= std::bind(func,1);
    f();

    Foo foo;
    f = std::bind(&Foo::method,&foo);
    f();

    f = std::bind(&Foo::method2,&foo,"china");
    f();

    Bar bar;

    f = std::bind(&Bar::method3,&bar,10,"china");
    f();


    return 0;
}
```

3.10.4.cocos 中应用

3.10.4.1.bind 应用

```
auto closeItem =
MenuItemImage::create(
          "CloseNormal.png",
          "CloseSelected.png",
           CC_CALLBACK_1(HelloWorld::menuCloseCallback, this)
          );
```

```
MenuItemImage * MenuItemImage::create(
        const std::string& normalImage,
        const std::string& selectedImage,
        const ccMenuCallback& callback)
{
    return MenuItemImage::create(normalImage, selectedImage, "",
callback);
}
```

```
typedef std::function<void(Ref*)> ccMenuCallback;
```

3.10.4.2.bind 绑定

```
#define CC_CALLBACK_0(__selector__,__target__, ...)
          std::bind(&__selector__,__target__, ##__VA_ARGS__)

#define CC_CALLBACK_1(__selector__,__target__, ...)
           std::bind(&__selector__,__target__, std::placeholders::_1,
##__VA_ARGS__)

void HelloWorld::menuCloseCallback(Ref* sender)
{
   Director::getInstance()->end();

#if (CC_TARGET_PLATFORM == CC_PLATFORM_IOS)
   exit(0);
#endif
}
```

3.10.4.3.推导

```
ccMenuCallback
mc = CC_CALLBACK_1(HelloWorld::menuCloseCallback, this);

ccMenuCallback
mc std::bind(&HelloWorld::menuCloseCallback, this,
std::placeholders::_1);
```

3.10.4.4.调用

```
void MenuItem::activate()
{
   if (_enabled)
   {
      if( _callback )
      {
         _callback(this);
      }
#if CC_ENABLE_SCRIPT_BINDING
      if (kScriptTypeLua == _scriptType)
      {
         BasicScriptData data(this);
         ScriptEvent scriptEvent(kMenuClickedEvent, &data);

ScriptEngineManager::getInstance()->getScriptEngine()->sendEvent(&scriptEvent);
      }
#endif
   }
}
```

3.10.4.5.自实现

```
#include <iostream>
#include <functional>

using namespace std;


#define CC_CALLBACK_0(__selector__,__target__, ...)  \
    std::bind(&__selector__,__target__, ##__VA_ARGS__)

#define CC_CALLBACK_1(__selector__,__target__, ...)  \
    std::bind(&__selector__,__target__, std::placeholders::_1,
##__VA_ARGS__)

class A
{
public:
    void func();
    void foo(int a);
};

void A::func()
{
    cout<<"void A::func"<<endl;
}
void A::foo(int a)
{
    cout<<"void A::foo(int a)"<<a<<endl;
}

int main()
{

    A  a;
    function<void()> f = CC_CALLBACK_0(A::func,&a);
    f();

    function<void(int a)> f2 =  CC_CALLBACK_1(A::foo,&a);
    f2(100);


    return 0;
}
```

3.10.5.bind1st/bind2nd

略

4.STL in C++11

4.1.无序容器 Unordered Container

4.1.1.hash

4.1.1.1.原理

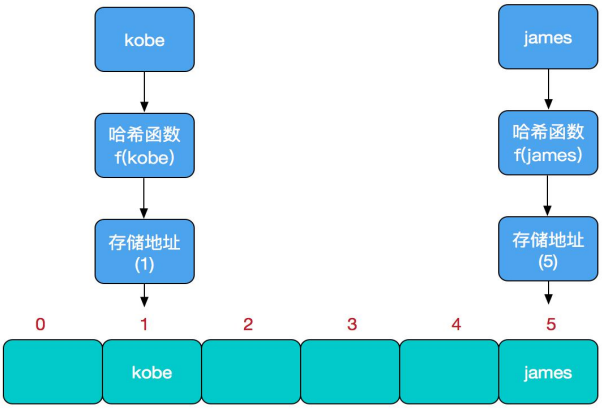
在哈希表中进行添加，删除，查找等操作，性能十分之高，不考虑哈希冲突的情况下，仅需一次定位即可完成，时间复杂度为 O(1)，接下来我们就来看看哈希表是如何实现达到惊艳的常数阶 O(1)的。

我们知道，数据结构的物理存储结构只有两种：顺序存储结构和链式存储结构（像栈，队列，树，图等是从逻辑结构去抽象的，映射到内存中，也这两种物理组织形式），而在上面我们提到过，在数组中根据下标查找某个元素，一次定位就可以达到，哈希表利用了这种特性，哈希表的主干就是数组。

比如我们要新增或查找某个元素，我们通过把当前元素的关键字 通过某个函数射到数组中的某个位置，通过数组下标一次定位就可完成操作。

存储位置 = f(关键字)

其中，这个函数 f 一般称为哈希函数，这个函数的设计好坏会直接影响到哈希表的优劣。

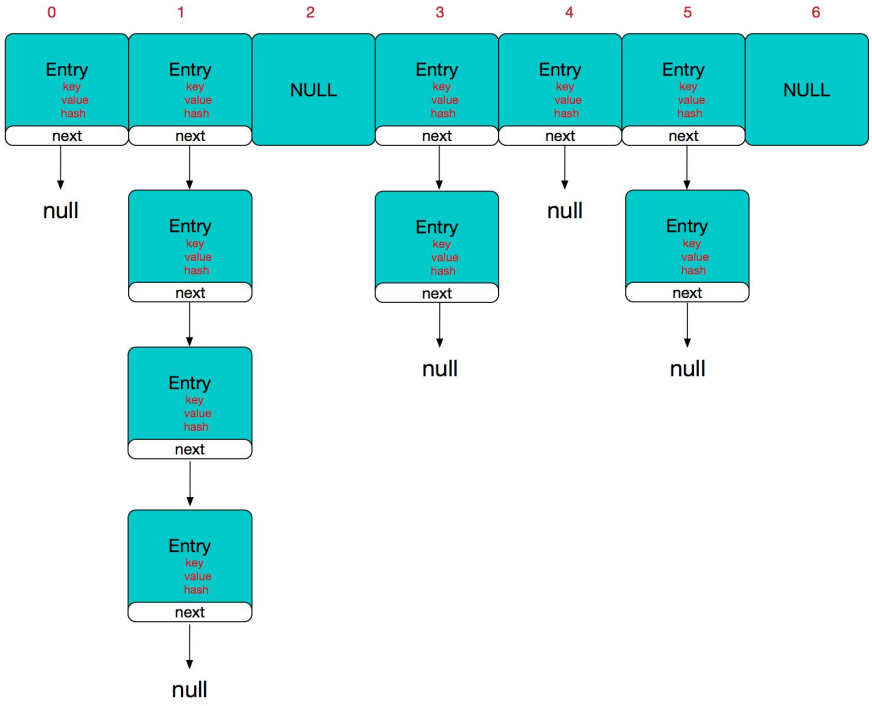


4.1.1.2.hash 冲突

如果两个不同的元素，通过哈希函数得出的实际存储地址相同怎么办？也就是说，当我们对某个元素进行哈希运算，得到一个存储地址，然后要进行插入的时候，发现已经被其他元素占用了，其实这就是所谓的哈希冲突，也叫哈希碰撞。

哈希函数的设计至关重要，好的哈希函数会尽可能地保证 计算简单和散列地址分布均匀,但是，我们需要清楚的是，数组是一块连续的固定长度的内存空间，再好的哈希函数也不能保证得到的存储地址绝对不发生冲突。

那么哈希冲突如何解决呢？哈希冲突的解决方案有多种:开放定址法（发生冲突，继续寻找下一块未被占用的存储地址），再散列函数法，链地址法，而 HashMap 即是采用了链地址法，也就是数组+链表的方式。



4.1.2.unordered_map

4.1.2.1.语义

unordered_map 提供了和 map 类似的接口，只是 map 是有序，而 unordered_map 因为采用 hash map 的数据结构，所以是无序的。

另外，因为 map 采用的是红黑树，所以查找性能是 O(log(n))。而 unordered_map 采用 hash map，所以查找性能是 O(1)。

所以一般来说小规模的数据适合采用 map(百 W 以下)，而大规模的数据适合 unordered_map(百 W 以上)。

4.1.2.2.声明

```
template < class Key,                    // unordered_map::key_type
         class T,                        // unordered_map::mapped_type
         class Hash = hash<Key>,         // unordered_map::hasher
         class Pred = equal_to<Key>, // unordered_map::key_equal
         class Alloc = allocator< pair<const Key,T> >
                                        // unordered_map::allocator_type
         > class unordered_map;
```

4.1.2.3.实战

```
#include <iostream>
#include <string>
#include <map>
#include <unordered_map>
```

```
#include <unordered_set>
using namespace std;


int main()
{

    unordered_map<string, string> u =
    {
        {"RED",  "#FF0000"},
        {"GREEN","#00FF00"},
        {"BLUE", "#0000FF"}
    };


    for( const auto& n : u ) {
        cout << "Key:[" << n.first << "] Value:[" << n.second << "]\n";
    }


    u["BLACK"] = "#000000";
    u["WHITE"] = "#FFFFFF";

    cout << "The HEX of color RED is:[" << u["RED"] << "]\n";
    cout << "The HEX of color BLACK is:[" << u["BLACK"] << "]\n";


    return 0;
}
```

4.1.3.unordered_set

4.1.3.1.声明

```
template < class Key,                    // unordered_set::key_type/value_type
         class Hash = hash<Key>,    // unordered_set::hasher
         class Pred = equal_to<Key>,     // unordered_set::key_equal
         class Alloc = allocator<Key>    // unordered_set::allocator_type
         > class unordered_set;
```

4.1.3.2.实战

```
#include <unordered_set>
#include <iostream>


void printUnorderSet(std::unordered_set<int>& m, char* pre) {
    std::unordered_set<int>::iterator it;
    std::cout << pre;
    for ( it = m.begin(); it != m.end(); it++ )
```

```
        std::cout << *it << " ";
    std::cout << std::endl;
}
void unoderSet() {
    std::unordered_set<int> foo1;


    // 普通插入
    foo1.insert(1);

    // 带暗示插入，std::pair<int,double>等价于上述的
    // std::unordered_set<int,double>::value_type
    foo1.insert(foo1.end(),2);

    // 插入一个范围
    std::unordered_set<int> foo2;
    foo2.insert(3);
    foo2.insert(4);
    foo2.insert(5);
    foo1.insert(foo2.begin(),foo2.end());

    printUnorderSet(foo1,"插入元素后的 foo1: ");

    // 查找主键 4
    std::unordered_set<int>::iterator it;
    it = foo1.find(4);
    if( it != foo1.end() )
    {
        std::cout << "foo1.find(4): ";
        std::cout << *it << std::endl;
    }

    // 删除上述找到的元素
    if( it != foo1.end() )
    {
        foo1.erase(it);
    }
    printUnorderSet(foo1,"删除主键为 4 的元素后的 foo1: ");

    // 遍历删除主键为 2 的元素
    for(it = foo1.begin();it != foo1.end();it++)
    {
        //遍历删除主键等于 2
        //注意，删除元素会使迭代范围发生变化
        if ( *it == 2 )
        {
            foo1.erase(it);
            break;
        }
    }
```

```
    printUnorderSet(foo1,"删除主键为 2 的元素后的 foo1: ");
    foo1.clear();
    printUnorderSet(foo1,"清空后的 foo1: ");
}
int main( )
{
    unoderSet();
    return 0;
}
```

4.2.Other

4.2.1.emplace

作用于容器，区别于 push、insert 等，如 push_back 是在容器尾部追加一个容器类型对象，emplace_back 是构造 1 个新对象并追加在容器尾部

对于标准类型没有变化，如 std:;vector<int>，push_back 和 emplace_back 效果一样

```
如自定义类型 class A，A 的构造函数接收一个 int 型参数，

那么对于 push_back 需要是：

std::vector<A> vec;

A a(10);

vec.push_back(a);

对于 emplace_back 则是：

std::vector<A> vec;

vec.emplace_back(10);
```

4.2.2.shrink_to_fit

5.自动内存管理 Auto Memory Manage

5.1.auto_ptr

5.1.1.RAII

RAII（Resource Acquisition Is Initialization）是一种利用对象生命周期来控制程序资源（如内存、文件句柄、网络连接、互斥量等等）的简单技术。

RAII 的一般做法是这样的：在对象构造时获取资源，接着控制对资源的访问使之在对象的生命周期内始终保持有效，最后在对象析构的时候释放资源。借此，我们实际上把管理一份资源的责任托管给了一个对象。这种做法有两大好处：

①不需要显式地释放资源。

②采用这种方式，对象所需的资源在其生命期内始终保持有效。

此时，所托管的资源，随对象的创建而获取，随对象的消失而消失，即所谓的 RAII 思想：资源获取即初始化。

```
#include <iostream>
#include <memory>

using namespace std;

class A
{
public:
    A()
    {
        cout<<"A()"<<endl;
    }
    ~A()
    {
        cout<<"~A()"<<endl;
    }
    void dis()
    {
        cout<<"A::void dis()"<<endl;
    }
};
void func()
{
    auto_ptr<A> a(new A);
    a->dis();
}

int main()
{
    func();
    return 0;
}
```

5.1.2.原理

原理上，是代理了被托管的对象指针，管理对象的生命周期，即实现自动释放。其行为类似于所托管的对象指针，原因是，重载了 operator->和 operator*。

如下是智能指针的模板(template)化实现。

```
template <class T>
class SmartPtr
{
public:
   explicit SmartPtr(T* pointee) : pointee_(pointee){}
    ~SmartPtr()
    {
       delete pointee_;
    }
   T& operator*() const
   {
      //...
      return *pointee_;
   }
   T* operator->() const
   {
      //...
      return pointee_;
   }
private:
   T* pointee_;
   //...
};
```

5.1.3.自实现

5.1.4.deprecated

5.1.4.1.引入:

防止两个 auto_ptr 对象拥有同一个对象（一块内存）。于 ap 与 ap2 都觉得指针 p 是归它管的。在析构时都试图删除 p, 两次删除同一个对象的行为在 C++标准中是没有定义的。

5.1.4.2.测试

```
#include <iostream>
#include <memory>
using namespace std;

int main()
```

```
{
    int *p = new int(10);

    {
        auto_ptr<int> ap(p);
        cout<<*ap<<endl;
    }

    auto_ptr<int> ap2(p);
    cout<<*ap2<<endl;
    return 0;
}
```

作参数传递的时，亦是会出现同样的情况，两个指针，对同一段资源产生了引用行为。

```
#include <iostream>
#include <memory>
using namespace std;

class Copy
{
public:
    Copy(int i):_i(new int(i))
    {
        cout<<"Copy(int i)"<<endl;
    }
    Copy(const Copy & another)
        :_i(new int(*another._i))
    {
        cout<<" Copy(const Copy & another)"<<endl;
    }
    ~Copy(){
        cout<<"~Copy()"<<endl;
    }

    int *_i;
};

void func(auto_ptr<Copy> apc)
{

}

int main(int argc, char *argv[])
{
    auto_ptr<Copy> apc(new Copy(10));
    cout<<*apc->_i<<endl;

    func(apc);
    cout<<*apc->_i<<endl;
```

```
    return 0;
}
```

5.1.4.3.专业注解

template <class X> class auto_ptr; Automatic Pointer [deprecated]
Note: This class template is deprecated as of C++11. unique_ptr is a new facility with a similar functionality, but with improved security (no fake copy assignments), added features (deleters) and support for arrays. See unique_ptr for additional information.

This class template provides a limited garbage collection facility for pointers, by allowing pointers to have the elements they point to automatically destroyed when the auto_ptr object is itself destroyed.

auto_ptr objects have the peculiarity of taking ownership of the pointers assigned to them: An auto_ptr object that has ownership over one element is in charge of destroying the element it points to and to deallocate the memory allocated to it when itself is destroyed. The destructor does this by calling operator delete automatically.

Therefore, no two auto_ptr objects should own the same element, since both would try to destruct them at some point. When an assignment operation takes place between two auto_ptr objects, ownership is transferred, which means that the object losing ownership is set to no longer point to the element (it is set to the null pointer).

5.2.unique_ptr

5.2.1.从 auto_ptr 到 unique_ptr

uniqu_ptr 的使用方法，基本上等同于 auto_ptr，不同之处，就在于实现了**资源的唯一**，既不可以拷贝也不可以赋值，正如其名字一样。

```
#include <iostream>
#include <memory>
using namespace std;

class Copy
{
public:
    Copy(int i):_i(new int(i))
    {
        cout<<"Copy(int i)"<<endl;
    }
```

```
    Copy(const Copy & another)
        :_i(new int(*another._i))
    {
        cout<<" Copy(const Copy & another)"<<endl;
    }
    ~Copy(){
        cout<<"~Copy()"<<endl;
    }

    int *_i;
};

void func(unique_ptr<Copy> upc)
{

}

int main(int argc, char *argv[])
{
    unique_ptr<Copy> upc(new Copy(10));
    cout<<*upc->_i<<endl;

    unique_ptr<Copy> upc2(upc);  //编译不过

    func(upc);                  //编译不过

    return 0;
}
```

unique_ptr 可以用于数组。

```
#include <iostream>
#include <memory>
using namespace std;

class A
{
public:
    A()
    {
        cout<<"A()"<<endl;
    }
    ~A()
    {
        cout<<"~A()"<<endl;
    }
    void dis()
    {
        cout<<"A::void dis()"<<endl;
    }
```

```
};

int main()
{
    unique_ptr<int[]> up(new int[10]{1,2,3,4});

    for(int i=0; i<10; i++)
        cout<<up[i]<<endl;

    unique_ptr<A[]>  up2(new A[20]);

    for(int i=0; i<20; i++)
        up2[i].dis();


    return 0;
}
```

5.2.2.常用接口

生命周期随构造者，reset 自动析构再重新构造，get 判断是否有效、支持放在容器内；真正意义智能指针。

不论是临时变量指针、类成员指针变量.....90%的指针都应该用这个。

public interface	explain
get	Get pointer (public member function) 获取所托管资源指针
operator bool	Check if not empty (public member function) 非空为真
release	Release pointer (public member function) 放弃托管，返回资源指针
reset	Reset pointer (public member function) 有参(管新资源，释放旧资源) 无参(释放旧资源)
operator*	Dereference object (public member function)
operator->	Dereference object member (public member function)

```
#include <iostream>
#include <memory>
using namespace std;

class Copy
{
public:
    Copy(int i):_i(new int(i))
    {
        cout<<"Copy(int i) "<<
             "this: "<<this<<" _i "<<_i<<endl;
    }
    Copy(const Copy & another)
```

```
        :_i(new int(*another._i))
    {
        cout<<" Copy(const Copy & another)"<<endl;
    }
    ~Copy(){
        cout<<" ~Copy() "<<this<<endl;
    }

    int *_i;
};

int main1()
{
    unique_ptr<Copy>  up;
    if(!up)
        cout<<"无资源托管"<<endl;
    unique_ptr<Copy> up2(new Copy(10));
    if(up2)
        cout<<"有资源托管"<<endl;
    {
        unique_ptr<Copy> up3(new Copy(99));
        cout<<"get="<<up3.get()<<endl;

//       up3.reset();
        Copy *p = up3.release();  //放弃托管，而非释放资源
        delete p;
        cout<<"+++++++++++++++"<<endl;
    }

}
int main2()
{
    {
        unique_ptr<Copy>  up(new Copy(10));
        up.reset(new Copy(100));
        cout<<"++++++++++++++++++"<<endl; //释放旧，获得新
    }
    cout<<"================="<<endl;
}
int main()
{
     unique_ptr<Copy>  up(new Copy(99));
     cout<<up.get()<<endl;
     unique_ptr<Copy> up2 =std::move(up);
     cout<<up2.get()<<endl;

}
```

注：release 是对所托管的对象，释放权限，并没有释放托管对象本身。

5.3.shared_ptr

5.3.1.原理

uinque_ptr 解决了 auto_ptr 中引用同一个对象的问题，方式就是**不可引用同一个对象。**

shared_ptr 解决了 auto_ptr 中引用同一个对象，共同拥有一个资源， 但不会重析构的问题，原理是，在内部保持一个引用计数，并且仅当引用计数为 0 才能被删除，不可以用数组。

5.3.2.常用接口

可以有多个持有者的共享指针，即所谓引用计数型指针，直到最后一个持有者 delete 释放时，其指向的资源才会真正被释放。

典型应用案例：如对同一个全局无锁队列对象由 shared_ptr 封装，多线程的多个持有者均持有对其的引用。直到全部线程都释放掉对其的引用时，该无锁队列对象才会被最终销毁。

也就是 shared_ptr 适合用于管理“全局动态资源”。

public interface	explain
operator=	shared_ptr assignment (public member function) 支持复制
reset get	Reset pointer (public member function) Get pointer (public member function)
use_count	Use count (public member function)
unique	Check if unique (public member function)
operator bool	Check if not null (public member function)
operator*	Dereference object (public member function)
operator->	Dereference object member (public member function)

5.3.2.1.基本类型计数测试

```
#include <iostream>
#include <memory>
using namespace std;

void func(shared_ptr<int> sp)
{
//    sp.reset(); //此处仅将自己所累积的计数减 1
    cout<<sp.use_count()<<endl;
    sp.reset(); //此时 reset 等价于 sp 对象消失，若己为零，则不再减 1.
}

int main()
{
    shared_ptr<int> sp(new int(10));
    cout<<sp.get()<<endl;
    if(sp)
        cout<<"有资源托管中"<<endl;
```

```
    cout<<sp.use_count()<<endl;
    shared_ptr<int> sp2 = sp;
    cout<<sp2.use_count()<<endl;
    cout<<sp.use_count()<<endl;

    func(sp);
    cout<<sp.use_count()<<endl;

    return 0;
}
```

5.3.2.2.对象计数测试

reset 跟参数，会托管新对象，释放旧对象，如若不跟参数话，会将当前对象的引用计数减 1.

```
#include <iostream>
#include <memory>
using namespace std;

class A
{
public:
    A()
    {
        cout<<"A()"<<this<<endl;
    }
    ~A()
    {
        cout<<"~A()"<<this<<endl;
    }
    void dis()
    {
        cout<<"A::void dis()"<<endl;
    }
};

int main1()
{
    {
        shared_ptr<A>  sp(new A);
        sp.reset(new A());
        cout<<"++++++++++++++++++++"<<endl;
    }
    cout<<"======================"<<endl;
}

int main()
{
    {
```

```
        shared_ptr<A>  sp(new A);
        sp.reset();sp.reset();sp.reset();
        cout<<"++++++++++++++++++++"<<endl;
    }
    cout<<"======================"<<endl;
}
```

5.3.2.3.对象传参测试

```
#include <iostream>
#include <memory>
using namespace std;

class A
{
public:
    A()
    {
        cout<<"A()"<<this<<endl;
    }
    ~A()
    {
        cout<<"~A()"<<this<<endl;
    }
    void dis()
    {
        cout<<"A::void dis()"<<endl;
    }
};
void func(shared_ptr<A> &sp)  //shared_ptr<A> &sp
{
    cout<<sp.use_count()<<endl;
    sp.reset(); sp.reset();
    cout<<"======================"<<endl;
}

int main()
{
    shared_ptr<A> sp(new A);
    cout<<sp.use_count()<<endl;
    func(sp);

    cout<<sp.use_count()<<endl;

    shared_ptr<A> sp2 = std::move(sp); //移动会将计数也一起移走
    cout<<sp2.use_count()<<endl;
    cout<<sp.use_count()<<endl; //此时资源为0

    return 0;
```

```
}
```

对同一个对象，多次 reset 的结果，是仅对自己增加的计数减 1。要保证，当前的对象的使用安全性。也不会对其它象的使用造成影响。

```
void func(shared_ptr<Copy> spc)
//&spc 若传递的是引用，则引用计数不会加 1。离开函数也不会减 1

shared_ptr<A> sp2 = std::move(sp); //移动会将计数也一起移走
```

5.4.weak_ptr

weak_ptr 基本上没有什么用处，此处略。

6.线程框架 Thread Frame

6.1.引入

6.1.1.串行 while

6.1.2.并行 while

6.2.Thread

6.2.1.线程定义

前面我们已经学习过系统编程和网络编程，线程原语层面的，我们不再赘述。线程可以理解为一个特立独行的函数。其存在的意义，就是并行，避免了主线程的阻塞。

6.2.2.线程创建

6.2.2.1.线程对象与线程

C++线程的启动，只需要#include <thread>即可。线程对象的创建，意味着线程的开始。

```
#include <iostream>
#include <thread>
#include <unistd.h>

using namespace std;

void func()
{
    cout<<"thread id:"<<this_thread::get_id()<<endl;
    cout<<"do some work"<<endl;
//    sleep(3);
    this_thread::sleep_for(chrono::seconds(10));
}

int main()
{
    cout<<"main thread id:"<<this_thread::get_id()<<endl;
    thread t(func);
    t.join();

    cout<<"main thread is waiting thread"<<endl;
    return 0;
}
```

6.2.2.2.join 与 detach

t.join 和 t.detach 标志着，线程对象和线程的关系。t.join 表识，线程与线程对象的同步关系。而 t.detach 表识，线程与线程对象的异步关系。

detach 后的线程，不能再 join，是否可以 join 可以通过 joinable 来判断。

```
#include <iostream>
#include <thread>
#include <unistd.h>

using namespace std;

void func()
{
    cout<<"thread id:"<<this_thread::get_id()<<endl;
    cout<<"do some work"<<endl;
//    sleep(3);
    this_thread::sleep_for(chrono::seconds(5));
}

int main()
{
    cout<<"main thread id:"<<this_thread::get_id()<<endl;
    thread t(func);
//    t.join();
    t.detach();

    if(!t.joinable())
    {
        this_thread::sleep_for(chrono::seconds(10));
        cout<<"main thread is waiting thread"<<endl;
    }
    else
    {
        t.join();
    }


    return 0;
}
```

join 是阻塞的。

```
#include <iostream>
#include <thread>
#include <chrono>

void foo()
{
    // 模拟昂贵操作
    std::this_thread::sleep_for(std::chrono::seconds(5));
}

void bar()
{
    // 模拟昂贵操作
```

```
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main()
{
    std::cout << "starting first helper...\n";
    std::thread helper1(foo);

    std::cout << "starting second helper...\n";
    std::thread helper2(bar);


    helper1.join();
    std::cout << "waiting for helpers1 to finish..." << std::endl;
    helper2.join();
    std::cout << "waiting for helpers2 to finish..." << std::endl;

    std::cout << "done!\n";
}
```

6.2.2.3.传参方式

线程，有自己独立的栈。可以共享全局的变量。在线程启动的时候可以传入启动参数。

```
#include <iostream>
#include <thread>
using namespace std;

void func(int n, string s)
{
    for(int i=0;i <n ;i++)
    {
        cout<<s<<endl;
    }
}

int main()
{
    thread t(func,5,"china");
    t.join();
    return 0;
}
```

除了传入参数，共享全局以外，还可以**传入本地变量**的引用。以下传递引用的方式，编译通不过。

```
#include <iostream>
#include <thread>
```

```
using namespace std;

void func(int &n, string &s)
{
    for(int i=0;i <n ;i++)
    {
        cout<<s<<endl;
    }
    n = 10;
    s = "america";
}

int main()
{
    int n = 5;
    string  s = "china";
    thread t(func,n, s);
    t.join();
    return 0;
}
```

采用 **std::ref** 的方式：

```
#include <iostream>
#include <thread>
using namespace std;


void func(int& n, string& s)
{
    for(int i=0;i <n ;i++)
    {
        cout<<s<<endl;
    }
    n = 10;
    s = "Japan";
}

int main()
{
    int n = 5;
    string str = "china";
    thread t(func,std::ref(n),std::ref(str));
    t.join();

    cout<<"n = "<<n<<" str = "<<str<<endl;


    return 0;
}
```

6.2.3.你的电脑能开多少线程?

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

#define N 1000

void foo()
{
    // 模拟昂贵操作
    this_thread::sleep_for(chrono::seconds(5));
    cout << "foo!\n";
}

int main()
{
    thread th[N];
    for(int i=0; i< N ; i++)
    {
        th[i] = thread(foo);
    }

    for(auto & t:th)
        t.join();
    cout << "done!\n";
}
```

6.3.同步之 mutex

std::mutex 是 C++中最基本的互斥量，std::mutex 对象提供了独占所有权的特性--即不支持递归地对 std::mutex 对象上锁，而 std::recursive_lock 则可以递归地对互斥量对象上锁。

A mutex is a *lockable object* that is designed to signal when critical sections of code need exclusive access, preventing other threads with the same protection from executing concurrently and access the same memory locations.

6.3.1.成员函数

1）构造函数：std::mutex 不允许拷贝构造，也不允许 move 拷贝，最初产生的 mutex 对象是处于 unlocked 状态的。

2）lock()：调用线程将锁住该互斥量，线程调用该函数会发生以下 3 种情况：

（a）如果该互斥量当前没有被锁住，则调用线程将该互斥量锁住，直到调用 unlock 之前，该线程一直拥有该锁。

（b）如果当前互斥量被其他线程锁住，则当前的调用线程被阻塞住。

（c）如果当前互斥量被当前调用线程锁住，则会产生死锁。

3）unlock()：解锁，释放对互斥量的所有权。

4）try_lock()：尝试锁住互斥量，如果互斥量被其他线程占有，则当前线程也不会被阻塞，线程调用该函数会出现下面 3 种情况：

（a）如果当前互斥量没有被其他线程占有，则该线程锁住互斥量，直到该线程调用 unlock 释放互斥量。

（b）如果当前互斥量被其他线程锁住，则当前调用线程返回 false，而并不会被阻塞掉。

（c）如果当前互斥量被当前调用线程锁住，则会产生死锁。

6.3.2.应用测试

6.3.2.1.lock/unlock

```
#include <iostream>
#include <mutex>
#include <thread>
using namespace std;

volatile int counter(0);   // non-atomic counter
mutex mtx;                // locks access to counter

void increase10Ktime()
{
    for(int i=0; i<10000; i++)
    {
        mtx.lock();
        counter++;
        mtx.unlock();
    }
}
int main()
{
    thread ths[10];
    for(int i=0; i<10; i++)
    {
        ths[i] = thread(increase10Ktime);
    }
    for(auto &th:ths)
        th.join();

    cout<<"after successful increase :"<<counter<<endl;

    return 0;
```

```
}
```

6.3.2.2.try_lock/unclok

```
#include <iostream>
#include <mutex>
#include <thread>
using namespace std;

volatile int counter(0);   // non-atomic counter
mutex mtx;              // locks access to counter

void increase10Ktime()
{
    for(int i=0; i<10000; i++)
    {
        mtx.try_lock();
        counter++;
        mtx.unlock();
    }
}
int main()
{
    thread ths[10];
    for(int i=0; i<10; i++)
    {
        ths[i] = thread(increase10Ktime);
    }
    for(auto &th:ths)
        th.join();

    cout<<"after successful increase :"<<counter<<endl;

    return 0;
}
```

6.3.2.3.lock_guard

在 lock_guard 对象构造时,传入的 Mutex 对象(即它所管理的 Mutex 对象)会被当前线程锁住。在 lock_guard 对象被析构时，它所管理的 Mutex 对象会自动解锁，由于不需要程序员手动调用 lock 和 unlock 对 Mutex 进行上锁和解锁操作，因此这也是**最简单安全的上锁和解锁方式**，尤其是在程序抛出异常后先前已被上锁的 Mutex 对象可以正确进行解锁操作，极大地简化了程序员编写与 Mutex 相关的异常处理代码。

与 Mutex **RAII** 相关，方便线程对互斥量上锁。using a local lock_guard to lock mtx guarantees unlocking on **destruction / exception:**

模拟 **exception**

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

mutex  mtx;

void printEven(int i)
{
    if( i%2 == 0)
        cout<< i <<" is even"<<endl;
    else
        throw logic_error("not even");
}

void printThreadId(int id)
{
    try{
        mtx.lock();
        printEven(id);
        mtx.unlock();

    }catch(logic_error & ){
        cout<<"exception caught"<<endl;
    }
}


int main()
{
    thread ths[10];   //spawn 10 threads

    for(int i=0; i<10; i++)
    {
        ths[i] = thread(printThreadId,i+1);
    }

    for(auto & th: ths)
        th.join();

    return 0;
}
```

改进版本

```
void printThreadId(int id)
{
    try{
        lock_guard<mutex> lck(mtx);    //栈自旋 抛出异常时栈对象自我析构。
```

```
        printEven(id);

    }catch(logic_error & ){
        cout<<"exception caught"<<endl;
    }
}
```

6.4.死锁

6.4.1.死锁成立

6.4.2.死锁测试

死锁的原因是，container 试图多次去获取锁已获得的锁。std::recursive_mutex 允许多次获取相同的 mutex。

C++中 STL 中的容器，是非线程安全的。

```
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
using namespace std;

template <typename T>
class container
{

public:
    void add(T element)
    {
        _mtx.lock();
        _elements.push_back(element);
        _mtx.unlock();
    }

    void addrange(int num, ...)
    {
        va_list arguments;

        va_start(arguments, num);

        for (int i = 0; i < num; i++)
        {
            _mtx.lock();
            add(va_arg(arguments, T));
            _mtx.unlock();
```

```
        }

        va_end(arguments);
    }

    void dump()
    {
        _mtx.lock();
        for(auto e : _elements)
            cout << e << endl;
        _mtx.unlock();
    }
private:
    mutex _mtx;
    vector<T> _elements;
};

void func(container<int>& cont)
{
    cont.addrange(3, rand(), rand(), rand());
}

int main()
{
    srand((unsigned int)time(0));

    container<int> cont;

    thread t1(func, ref(cont));
    thread t2(func, ref(cont));
    thread t3(func, ref(cont));

    t1.join();
    t2.join();
    t3.join();

    cont.dump();

    return 0;
}
```

改进之

```
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <stdarg.h>
using namespace std;

template <typename T>
class container
{

public:
    void add(T element)
    {
        lock_guard<recursive_mutex> lock(_mtx);
        _elements.push_back(element);
    }

    void addrange(int num, ...)
    {
        va_list arguments;

        va_start(arguments, num);

        for (int i = 0; i < num; i++)
        {
            lock_guard<recursive_mutex> lock(_mtx);
            add(va_arg(arguments, T));

        }

        va_end(arguments);
    }

    void dump()
    {
        lock_guard<recursive_mutex> lock(_mtx);
        for(auto e : _elements)
            cout << e << endl;
    }
private:
    recursive_mutex _mtx;
    vector<T> _elements;
};

void func(container<int>& cont)
{
    cont.addrange(3, rand(), rand(), rand());
}

int main()
{
    srand((unsigned int)time(0));
```

```
    container<int> cont;

    thread t1(func, ref(cont));
    thread t2(func, ref(cont));
    thread t3(func, ref(cont));

    t1.join();
    t2.join();
    t3.join();

    cont.dump();

    return 0;
}
```

6.5.同步之 condition val

条件变量(condition variable)是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待某个条件为真，而将自己挂起；另一个线程使的条件成立，并通知等待的线程继续。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。

C++11 中引入了条件变量，其相关内容均在<condition_variable>中。这里主要介绍 std::condition_variable 类。

条件变量 std::condition_variable 用于多线程之间的通信，它可以阻塞一个或同时阻塞多个线程。std::condition_variable 需要与 std::unique_lock 配合使用。std::condition_variable 效果上相当于包装了 pthread 库中的 pthread_cond_*()系列的函数。

当 std::condition_variable 对象的某个 wait 函数被调用的时候，它使用 std::unique_lock(通过 std::mutex)来锁住当前线程。当前线程会一直被阻塞，直到另外一个线程在相同的 std::condition_variable 对象上调用了 notification 函数来唤醒当前线程。

6.5.1.成员函数

(1)、构造函数：仅支持默认构造函数，拷贝、赋值和移动(move)均是被禁用的。

(2)、wait：当前线程调用 wait()后将被阻塞，直到另外某个线程调用 notify_*唤醒当前线程；当线程被阻塞时，该函数会自动调用 std::mutex 的 unlock()释放锁，使得其它被阻塞在锁竞争上的线程得以继续执行。一旦当前线程获得通知(notify，通常是另外某个线程调用 notify_*唤醒了当前线程)，wait()函数也是自动调用 std::mutex 的 lock()。wait 分为无条件被阻塞和带条件的被阻塞两种。

无条件被阻塞：调用该函数前，当前线程应该已经对 unique_lock<mutex> lck 完成了加锁。所有使用同一个条件变量的线程必须在 wait 函数中使用同一个 **unique_lock<mutex>。该 wait 函数内部会自动调用 lck.unlock()对互斥锁解锁**，使得其他被阻塞在互斥锁上的线程恢复执行。使用本函数被阻塞的当前线程在获得通知(notified，通过别的线程调用 notify_*系列的函数)而被唤醒后，wait()函数恢复执行并自

动调用 lck.lock()对互斥锁加锁。

带条件的被阻塞：wait 函数设置了谓词(Predicate)，只有当 pred 条件为 false 时调用该 wait 函数才会阻塞当前线程，并且在收到其它线程的通知后只有当 pred 为 true 时才会被解除阻塞。因此，等效于 while (!pred())　wait(lck).

(3)、wait_for：与 wait()类似，只是 wait_for 可以指定一个时间段，在当前线程收到通知或者指定的时间超时之前，该线程都会处于阻塞状态。而一旦超时或者收到了其它线程的通知，wait_for 返回，剩下的步骤和 wait 类似。

(4)、wait_until：与 wait_for 类似，只是 wait_until 可以指定一个时间点，在当前线程收到通知或者指定的时间点超时之前，该线程都会处于阻塞状态。而一旦超时或者收到了其它线程的通知，wait_until 返回，剩下的处理步骤和 wait 类似。

(5)、notify_all：唤醒所有的 wait 线程，如果当前没有等待线程，则该函数什么也不做。

(6)、notify_one：唤醒某个 wait 线程，如果当前没有等待线程，则该函数什么也不做；如果同时存在多个等待线程，则唤醒某个线程是不确定的(unspecified)。

6.5.2.应用测试

6.5.2.1.条件

10 个线程，争相去打印，传入的 id，第一个线程但是获得锁的，但为条件不满足，采用条件变量出让锁，等待条件，其它线程亦是如此。延时后的主要线程，获得锁，将条件置为真，并能过条件变量唤醒所有等待在条件变量上的 10 个线程。此时 10 个线程再次争相去获取锁，然后，判断条件为真，然后打印，释放锁给其它线程。

6.5.2.2.测试

```
#include <iostream>
#include <thread>
#include <chrono>
#include <mutex>
#include <condition_variable>

using namespace std;

condition_variable  cv;
mutex  mtx;
bool ready = false;

void printId(int id)
{
   unique_lock<mutex>  ql(mtx) ;
   while(!ready)
      cv.wait(ql);
   cout<<"thread id:"<<this_thread::get_id()
     <<" id = "<<id<<endl;
```

```
}

void go()
{
    unique_lock<mutex> ql(mtx); //主线程 中也要上锁的。
    ready = true;
    cv.notify_all();
}


int main()
{
    thread th[10];
    for(int i=0; i<10; i++)
        th[i] = thread(printId,i);

    this_thread::sleep_for(chrono::seconds(5));

    go();

    for(auto &t: th)
        t.join();
    return 0;
}
```

6.6.线程池自实现

7.时间 Chrono