1. 变量

*C/C++*程序内存分为堆区（自由存储区），栈区，全局区（静态区），常量区，代码区。

ELF文件分为代码段，*.data段*，*.bss段*，*.rodata*段以及自定义段。

已经初始化的全局变量以及静态局部变量存放在*.data*段，没有初始化的存放在*.bss*段。由于没有初始化数据，所以其实不占用空间，因此在*ELF*文件中，*.bss*只是一个占位符，只有当程序真正运行起来之后才会在内存上真正的开辟*.bss*空间，并且在*.bss*空间中开辟空间，并且自动初始化为0。程序运行之后存放在全局静态区。

常量（字符串常量，*const*变量）存放在*.rodata*段，程序运行后存放在常量区

代码存放在代码段，程序运行后存放在代码区。

当程序被加载到内存中后，操作系统负责加载上述各段，并为程序分配堆和栈。堆存放局部变量以及函数形参，有操作系统自行分配和释放；栈存放由*malloc*函数申请的空间，由程序员显示地分配和释放。

1.1 局部变量

定义在某个函数内部或者某个*statement*内部的变量，生命周期为函数或者*statement*开始到结束。

1.2 全局变量

定义在所有函数之外（包括*main*函数），在*main*函数执行之前进行初始化。

若想实现在*main*函数之前执行某个函数，可以定义一个全局变量，变量值为某个函数的返回值即可。

若想全局变量跨多个源文件，需要在一个文件里进行定义，其他文件进行*extern*声明。

1.3 静态变量

静态局部变量只定义一个，存放在全局静态区，生命周期同全局变量，但是仅局部可见；静态全局变量只在该文件可见，不能跨文件使用。

可执行二进制程序 = *text + bss(0) + data + rodata*

正在运行的程序 = *text + bss + data + rodata + stack + heap*

1.4 *heap*和*free store*释疑

网上有说法是*malloc*申请的内存在*heap*区而*new/new[]*申请的内存在*free store*区，但是根据我的查阅的资料发现二者是没有区别的。

首先是根据*Herb Sutter* (*served as secretary and convener of the ISO C++ standards committee for over 10 years*)对*heap*的说法:

> The heap is the other dynamic memory area,allocated/freed by malloc/free and their variants. Note that while the default global new and delete might be implemented in terms of malloc and free by a particular compiler, the heap is not the same as free store and memory allocated in one area cannot be safely deallocated in the other. Memory allocated from the heap can be used for objects of class type by placement-new construction and explicit destruction. If so used, the notes about free store object lifetime apply similarly here.

对于*free store*他觉得是：

> The free store is one of the two dynamic memory areas, allocated/freed by new/delete. Object lifetime can be less than the time the storage is allocated; that is, free store objects can have memory allocated without being immediately initialized, and can be destroyed without the memory being immediately deallocated. During the period when the storage is allocated but outside the object's lifetime, the storage may be accessed and manipulated through a void* but none of the proto-object's nonstatic members or member functions may be accessed, have their addresses taken, or be otherwise manipulated.

他觉得我们做这个区分是因为：

> We distinguish between "heap" and "free store" because the draft deliberately leaves unspecified the question of whether these two areas are related.

而*C++*之父对此的回应是：

> In other word, the "free store" vs "heap" distinction is Herb's attempt to distinguish malloc() allocation from new allocation.
>
> Because even though it is undefined from where new and malloc() get their memory, they typically get them from exactly the same place. It is common for new and malloc() to allocate and free storage from the same part of the computer's memory. In that case, "free store" and "heap" are synonyms. I consistently use "free store" and "heap" is not a defined term in the C++ standard (outside the heap standard library algorithms, which are unrelated to new and malloc()). In relation to new, "heap" is simply a word someone uses (typically as a synonym to "free store") - usually because they come from a different language background.

他认为标准没有定义*new*和*malloc*申请内存的位置，但是可以把*free store*和*heap*当成是一块区域。

另外经过我查看*GCC*或者*VSC++*源码发现*new operator*的实现都是调用的*malloc*：

```
//GCC gcc/libstdc++-v3/libsupc++/new_op.cc
_GLIBCXX_WEAK_DEFINITION void *
operator new (std::size_t sz) _GLIBCXX_THROW (std::bad_alloc)
{
  void *p;

  /* malloc (0) is unpredictable; avoid it.  */
  if (__builtin_expect (sz == 0, false))
    sz = 1;

  while ((p = malloc (sz)) == 0)
    {
      new_handler handler = std::get_new_handler ();
      if (! handler)
    _GLIBCXX_THROW_OR_ABORT(bad_alloc());
      handler ();
    }

  return p;
}
//VSC++ VC/Tools/MSVC/14.10.25017/crt/src/vcruntime/new_scalar.cpp
void* __CRTDECL operator new(size_t const size)
{
    for (;;)
    {
```

```
        if (void* const block = malloc(size))
        {
            return block;
        }

        if (_callnewh(size) == 0)
        {
            if (size == SIZE_MAX)
            {
                __scrt_throw_std_bad_array_new_length();
            }
            else
            {
                __scrt_throw_std_bad_alloc();
            }
        }
    }
}
```

所以我们不必纠结二者的差异，就当成一样的就好了。

reference:

> http://zamanbakshifirst.blogspot.com/2007/02/c-free-store-versus-heap.html
>
> http://www.gotw.ca/gotw/009.htm

2. 字符、字符串

`char a = ‘a’;`

`char *b = “abc”;`

上述两个类型为字符及字符串常量，存储在*ELF*文件的*.rodata*段，程序运行后加载到内存空间的常量区。

`char c[] = “abc”;` 此为*char*型数组，存放在堆区。

*string*类型是模板参数为*char*的*basic_string*模板类的类型定义：

`typedef basic_string<char, char_traits<char>, allocator<char>> string;`

3. 浮点类型

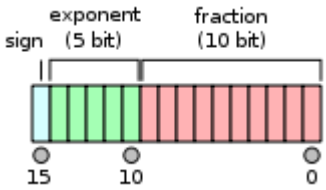
定义在*IEEE754*标准中，规约化计算方式为$(-1)*sign*(2^(exponent-1)-2^{e-1}+1)*1.(fraction)$,

非规约化计算方式为$(-1)*sign*(2^(-126))*0.(fraction)$;

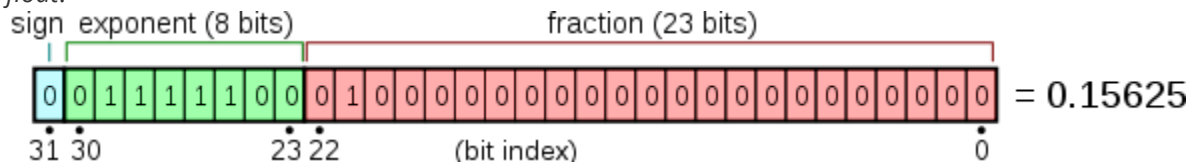
规约和非规约的区别为指数位是否大于0，若为0则为非规约数，指数位默认大小为$2^(-126)$；

若大于0则为规约数，指数为大小为$2^(exponent-1)-2^(e-1)+1$。

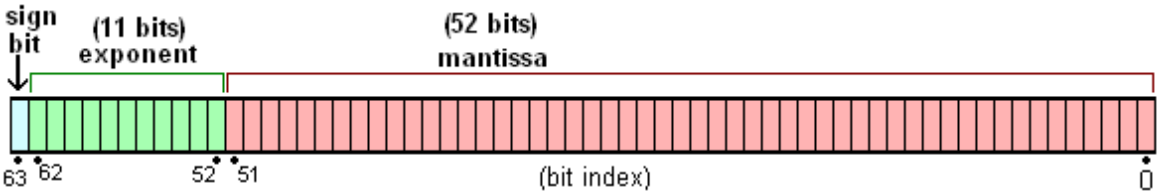
fp16:



float:



double:



4. 类型提升

表达式中可以使用整数的地方，就可以使用枚举类型，或有符号或无符号的字符、短整数、整数位域。如果一个*int*可以表示上述类型，则该值被转化为*int*类型的值；否则，该值被转化为*unsigned int*类型的值。这一过程被称作*integral promotion*。

整型提升的意义在于：表达式的整型运算要在CPU的相应运算器件内执行，CPU内整型运算器(ALU)的操作数的字节长度一般就是*int*的字节长度，同时也是CPU的通用寄存器的长度。因此，即使两个*char*类型的相加，在CPU执行时实际上也要先转换为CPU内整型操作数的标准长度。通用CPU（*general-purpose CPU*）是难以直接实现两个8比特字节直接相加运算（虽然机器指令中可能有这种字节相加指令）。所以，表达式中各种长度可能小于`int`长度的整型值，都必须先转换为*int*或*unsigned int*，然后才能送入CPU去执行运算。

各种整形的长度：

类型	16位系统(byte)	32位系统(byte)	64位系统(byte)
char	1	1	1
char*	2	4	8
short	2	2	2
int	2	4	4
long	4	4	8
long long	8	8	8

所以指针类型只能安全的转换成nullptr_t或者long long

5. 函数

函数可分为匿名函数和有名函数，匿名函数使用*lambda*表达式实现，匿名函数一般使用在该函数只使用一次的情况下，比如为*std::sort*定义一个比较函数：

```
using itype = vector<int,int>;
itype a;
itype b;
```

我们希望通过比较第二个*int*的值来排序，那么可以像这样调用

```
std::sort(a,b,[&](itype a, itype b ){
If(a[1]>b[1])
    return true;
return false;
});
```

function模板类可包装其他任何*callable*目标——函数、*lambda* 表达式、 *bind* 表达式或其他函数对象，还有指向成员函数指针和指向数据成员指针。我们可将匿名函数包装于*function*中使其成为有名函数，可使用*function*接受*bind*返回使成员函数作为线程入口函数：

```
using namespace std;
using namespace std::placeholder;
Class MyClass{
public:
void MyFunc(int a, int b)
{
    count = a + b;
}
private:
    int count;
};
MyClass mc;
auto func = bind(MyClass::MyFunc,mc,_1,_2);
int a = 10;
int b = 100;
Thread T(func, a, b);
T.join();
cout<<"mc.count="<<mc.count<<endl;
```

*std::bind*的返回值类型为*unspecified*，所以不能比较两个*bind*后的函数是否相等。

5.1 函数声明

函数声明不需要函数实现，不需要形参名。

5.2 函数定义

函数定义需要声明加实现

5.3 函数调用

第一个进栈的是主函数中函数调用后的下一条指令（函数调用语句结束的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的*C*编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

函数调用大致包括以下几个步骤：

参数入栈：将参数从右向左依次压入系统栈中
返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行
代码区跳转：处理器从当前代码区跳转到被调用函数的入口处
栈帧调整：具体包括
保存当前栈帧状态值，已备后面恢复本栈帧时使用（EBP入栈）
将当前栈帧切换到新栈帧。（将ESP值装入EBP，更新栈帧底部）
给新栈帧分配空间。（把ESP减去所需空间的大小，抬高栈顶）

ESP：栈指针寄存器(extended stack pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的栈顶
EBP：基址指针寄存器(extended base pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部
函数栈帧：ESP和EBP之间的内存空间为当前栈帧，EBP标识了当前栈帧的底部，ESP标识了当前栈帧的顶部。

EIP：指令寄存器(extended instruction pointer)， 其内存放着一个指针，该指针永远指向下一条待执行的指令地址。

5.4 函数副作用(*side effect*)

函数副作用指当调用函数时，除了返回函数值之外，还对主调用函数产生附加的影响。例如修改全局变量（函数外的变量），修改参数或改变外部存储。函数副作用会给程序设计带来不必要的麻烦，给程序带来十分难以查找的错误，并降低程序的可读性。严格的函数式语言要求函数必须无副作用。没有副作用的函数为纯函数。

5.5 函数参数传递

分为值传递和引用传递，引用传递形参就是实参，不会新构造一个副本。

5.6 缺省参数

函数参数设置默认值之后调用时可缺省参数，但是只可以从右往左设置默认值，不能够中间空没有设置默认值的参数。

5.7 *inline*函数

编译器会选择性在编译时将*inline*函数在调用的地方直接替换成函数体，省去了函数调用的开销。和宏定义的区别在于：

- 1.*inline*函数会进行参数检查
- 2.*inline*函数在编译时展开，宏定义在预编译时替换
- 3.*inline*函数可以进行*debug*

*inline*函数的函数体若小于调用函数产生的压栈出栈的代码大小则会使得程序占用内存变小，若函数体过大则会增大程序占用的内存大小。

由于函数体直接被替换到了调用处，编译器可根据上下文信息进行进一步的优化。

若没有使用*inline*函数，程序至函数调用处需要跳转去函数体所在位置的代码，一般函数调用位置和函数体位置并不相近，这样容易形成缺页中断。

5.8 函数重载

函数名称必须相同。

参数列表必须不同（个数不同、类型不同、参数排列顺序不同等）。

函数的返回类型可以相同也可以不相同。

仅仅返回类型不同不足以成为函数的重载。

*C++*通过*name mangling*来实现函数重载，以及域名空间等作用域的功能。

子类不能重载父类的函数，子类父类属于不同的域名空间，如果基类声明被重载了，则应该在派生类中重新定义所有的基类版本。**如果在派生类中只重新定义一个版本，其他父类版本将会被隐藏，派生类对象将无法使用它们**，但是可以转成父类指针再进行调用，因为同名成员函数其实就是函数的重载，不同类型对象指针会调用不同类型的重载函数。

```
class Base {
public:
    void func1() {
        cout << "Base func1" << endl;
    }
};

class Derive : public Base
{
public:
    void func1() {
        cout << "Derive func1" << endl;
    }
};

int main()
{
    Derive *d = new Derive();
    d->func1();//Derive func1
    static_cast<Base *>(d)->func1();//Base func1
    return 0;
}
```

简而言之，重新定义函数，并不是重载。在派生类中定义函数，将不是使用相同的函数特征标覆盖基类声明，而是隐藏同名的基类方法，不管参数的特征标如何。如果我们需要在子类中使用所有父类定义的某个函数但又不想重写，可以使用如下方法：

```
using A::print;
```

6. 类

在*C++*中，结构体是由关键词*struct*定义的一种数据类型。他的成员和基类默认为公有的（*public*）。由关键词*class*定义的成员和基类默认为私有的（*private*）。这是*C++*中结构体和类仅有的区别。

6.1 访问控制

*C++*类的重要属性就是封装和继承。因此，最关键的问题就是权限 的问题，*public*，*protected*，*private*控制的就是访问权限。

	public	protected	private
类成员是否可以访问	Y	Y	Y
友元函数是否可以访问	Y	Y	Y
子类是否可以访问	Y	Y	N
类的实例化对象是否可以访问	Y	N	N

三种继承方式导致的权限变化：

	public	protected	private
public继承	public	protected	private
protected继承	protected	protected	private
private继承	private	private	private

通过对象我们可以直接访问对象的成员函数以及成员变量，我们也可以通过*pointer to member*来对成员进行访问：

```
class MyClass{private: int value;};
int Myclass::*ptr = MyClass::value;
MyClass mc;
mc.value = 10;
cout<<mc.*ptr<<endl;
```

6.1.1 logical constness

有以下类 `BigArray`，其成员 `vector v;` 是一个数组数据结构，为了让外部可以访问该数组，此类提供了一个 `getItem` 接口，除此之外，为了计算外部访问数组的次数，该类还设置了一个计数器 `accessCounter` ，可以看到用户每次调用 `getItem` 接口，`accessCounter` 就会自增，很明显，这里的成员 `v` 是核心成员，而 `accessCounter` 是非核心成员，我们希望接口 `getItem` 不会修改核心成员，而不考虑非核心成员是否被修改，此时 `getItem` 所具备的 `const` 特性就被称为*logic constness*。

```
class BigArray {
    vector<int> v;
    int accessCounter;
public:
    int getItem(int index) const {
        accessCounter++;
        return v[index];
    }
};
```

但是，上面的代码不会通过编译，因为编译器不会考虑*logic constness*，于是就有了*bitwise constness*这个术语，可以理解为字面上的*constness*属性，编译器只认*bitwise constness*。为了解决这种矛盾，可以把 `accessCounter` 声明为 `mutable` 的成员，即：

```
class BigArray {
    mutable int accessCounter;
    // const_cast<BigArray*>(this)->accessCounter++; // 这样也行，但不建议这么做
    // ...
};
```

6.2 对象

- 1)　对象是类实例化（调用构造函数）之后的结果，仅对`public`成员有访问权限，释放时会自动调用析构函数。
- 2)　对象模型
  - a) *C++*中虚函数的作用主要是为了实现多态机制。多态，简单来说，是指在继承层次中，父类的指针可以具有多种形态——当它指向某个子类对象时，通过它能够调用到子类的函数，而非父类的函数。
  - b) 当一个类本身定义了虚函数，或其父类有虚函数时，为了支持多态机制，编译器将为该类添加一个虚函数指针（*vptr*）。虚函数指针一般都放在对象内存布局的第一个位置上，这是为了保证在多层继承或多重继承的情况下能以最高效率取到虚函数表。
  - c) 当*vprt*位于对象内存最前面时，对象的地址即为虚函数指针地址。我们可以取得虚函数表指针的地址：

```
Base b;
int * vptrAdree = (int *)(&b);
cout << "虚函数指针（vprt）的地址是：\t"<<vptrAdree << end;
```

-
  - 我们强行把类对象的地址转换为*int** 类型，取得了虚函数指针的地址。虚函数指针指向虚函数表,虚函数表中存储的是一系列虚函数的地址，虚函数地址出现的顺序与类中虚函数声明的顺序一致。对虚函数指针地址值，可以得到虚函数表的地址，也即是虚函数表第一个虚函数的地址:

```
typedef void(*Fun)(void);
Fun vfunc = (Fun)*( (int *)*(int*)(&b));
cout << "第一个虚函数的地址是：" << (int *)*(int*)(&b) << endl;
cout << "通过地址调用虚函数Base::print()：";
vfunc();
```

-
  - 我们把虚表指针的值取出来：**(int*)(&b)*，它是一个地址，第一个虚函数的地址

    把虚函数的地址强制转换成 *int** :*(int*) *(int*)(&b)*

    再把它转化成我们Fun指针类型：*(Fun)*(int *)*(int*)(&b)*

    这样，我们就取得了类中的第一个虚函数，我们可以通过函数指针访问它。

    同理,第二个虚函数的地址为：*(int*)(*(int*)(&b)+1)*

-
  - d) 子类若*override*了一个父类的虚函数，其虚函数表中对应被*override*的虚函数会替换成自己*override*的函数，若有新增虚函数则在虚函数表后面累加新的虚函数地址。
  - e) 所以继承类的对象内存分布为：

地址	内容
	第一个父类虚表指针
	第一个父类成员变量*n
	第二个父类虚表指针
	第二个父类成员变量*m
	自身的成员变量*k

- 3) 对象大小
  - a) 空类的大小为1；
  - b) 类的（静态）成员函数，静态成员变量不占用类的空间；
  - c) 若有虚函数增加一个虚函数表指针的大小；
  - d) 虚继承的子类也需要加上n个父类的虚函数表指针；

6.3 构造函数/析构函数

- 1) 构造函数在生成对象时调用，分为默认构造函数，拷贝构造函数，移动构造函数，赋值构造函数和移动赋值构造函数。
- 2) 在构造函数后面加*default*关键字可将某个构造函数设置成默认的构造函数，比如若果我们没有定义构造函数，编译器会帮助我们生成默认无参且函数体为空的默认构造函数；但如果我们定义了一个有参数的构造函数，编译器便不会帮助我们生成默认构造函数，此时我们不能够像这样定义对象*MyClass mc*，但是我们可以通过添加*default*关键字恢复：

  `MyClass() = default;`
- 3) 在构造函数后面加*delete*关键字可将某个构造函数设为禁用，比如我们不希望对象进行拷贝构造而希望其进行移动构造这样能避免不必要的内存分配和拷贝，这样我们可在拷贝构造函数和赋值拷贝构造函数后面添加*delete*关键字。
- 4）在一个类的成员变量只有*primitive type*时我们直接将*primitive type*赋值给对象会发生隐式转换，这种时候编译器不会报错并且能正常运行，但是可能并不是我们想要的结果。我们希望能早早地在编译阶段就发现这些问题，这时候我们可以将对应的构造函数声明为*explicit*，这样编译器在静态检查阶段便能发现问题。

```
class A
{
public:
    A(int x):x(x)
    {
        n = new int[x];
        cout << "A default construction" << endl;
    }
    A(A &a)
    {
        this->x = a.x;
        this->n = new int[x];
        memcpy(this->n, a.n, x * sizeof(typeid(n)));
        cout << "A copy construction" << endl;
    }
    A(A &&a)
    {
        this->x = a.x;
        this->n = a.n;
```

```
        a->n = nullptr;
        cout << "A move construction" << endl;
    }
    explicit A& operator=(A &a)
    {
        this->x = a.x;
        this->n = new int[x];
        memcpy(this->n, a.n, x * sizeof(typeid(n)));
        cout << "A copy assignment construction" << endl;
        return *this;
    }
    explicit A& operator=(A &&a)
    {
        this->x = a.x;
        this->n = a.n;
        a->n = nullptr;
        cout << "A move assignment construction" << endl;
        return *this;
    }
    virtual void foo() {}
    ~A()
    {
        if (this->n != nullptr) {
            delete this->n;
            this->n = nullptr;
        }
        cout << "A destruction" << endl;
    }
private:
    int x = 0;
    int *n;
};
A test()
{
        A a(1);
        return a;
}
A a = 10;//error，不能进行隐式转换
/*
A default construction
A move construction
A destruction
A move assignment construction
//若移动赋值构造函数返回值而非引用，这里会多一个拷贝构造函数和析构
A destruction
A destruction
*/
A a1;
a1 = test();
```

- 1) 默认构造函数为编译器为我们默认生成的构造函数，实际上没有任何操作。
- 2) 拷贝构造函数传入形参为对象的引用，声明对象时进行赋值或隐式赋值会调用拷贝构造函数
- 3) 移动构造函数传入形参为对象的右值引用，将赋值对象使用*move*函数强转成右值引用之后再进行如同拷贝构造函数的调用方式时会调用移动构造函数。
- 4) 函数返回对象会先将局部对象赋给一个临时变量，由于函数返回值为右值，所以此时调用的是移动构造函数，等局部变量析构后再将临时对象赋给函数外的对象，此时仍然是移动构造函数，最后再将临时对象析构。

- 5) 析构函数在对象到达生命周期时会自动调用，比如局部对象在函数结束时，对象指针被*delete*时。通过这种特性我们可以实现*RAII(resource acquisition is initialization)*，比如*c++11*中的*lock_guard*:

```
template <class Mutex> class lock_guard {
private:
    Mutex& mutex_;
public:
    lock_guard(Mutex& mutex) : mutex_(mutex) { mutex_.lock(); }
    ~lock_guard() { mutex_.unlock(); }
    lock_guard(lock_guard const&) = delete;
    lock_guard& operator=(lock_guard const&) = delete;
};
```

6.3.1 返回值优化(*RVO*)

https://www.ibm.com/developerworks/community/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/RVO_V_S_std_move?lang=en

6.4 类的静态成员函数和变量

- 1) 静态成员函数不能直接访问非静态成员变量，可以以传入对象的方式间接访问。
- 2) 非静态成员函数可以调用静态成员变量，因为静态成员变量属于整个类而非某个特定的对象，所有对象都共享该变量，在对象产生之前就有了，存储在全局静态存储区。
- 3) 使用静态成员变量实现多个对象之间的数据共享不会破坏隐藏的原则，保证了安全性还能节省内存。
- 4) 静态成员变量使用之前必须初始化（如`int MyClass::m_Number = 0;`），否则链接会出错。

6.5 友元

- 1) 友元函数是可以直接访问类的私有成员的非成员函数。它是定义在类外的普通函数，它不属于任何类，但需要在类的定义中加以声明，声明时只需在友元的名称前加上关键字*friend*。友元函数能访问对象的私有成员的意思是在友元函数内，对象可以直接访问私有成员变量而不需要通过成员函数，而不是友元函数可以直接访问成员变量：

```
class Derive
{
public:
    friend void FriendFunc(Derive d);
private:
    int a = 10;
};
friend void FriendFunc(Derive d)
{
    cout << "Derive's friend func" << d.a << endl;
}
int main()
{
    Derive *d = new Derive();
    FriendFunc(*d);
    delete d;
    return 0;
}
```

- 2) 友元函数的声明可以放在类的私有部分，也可以放在公有部分，它们是没有区别的，都说明是该类的一个友元函数。
- 3) 一个函数可以是多个类的友元函数，只需要在各个类中分别声明。
- 4) 友元函数的调用与一般函数的调用方式和原理一致。
- 5) 友元类的所有成员函数都是另一个类的友元函数，都可以访问另一个类中的隐藏信息（包括私有成员和保护成员）。
- 6) 当希望一个类可以存取另一个类的私有成员时，可以将该类声明为另一类的友元类。

```
class A
{
public:
friend class B;
};
```

6.6 操作符重载

- 1) 在类中声明的成员函数操作符重载只能被成员触发，所以对于双目运算符来说只能是运算符左边的对象来触发运算符重载。假如说我们希望实现 `primitive_type operator object;` 这种模式，我们只能用普通函数重载运算符而不是成员函数重载运算符：

```
class Register{
public:
    Register& operator+(Register &b) {
        this->regVal+= b.regVal;
        return *this;
    }
    friend  Register& operator+(int a, Register &b) {
        b.regId += a;
        return b;
    }
    int regVal;
};
```

- 2) 使用*ostream*对象进行<<操作只能定义为普通函数或者友元函数，因为无法修改*ostream*对象的<<代码实现，此时需要传入两个参数：

```
ostream& operator << (ostream& os, Test & test)
{
    os << test.a;
    return os;
}
```

6.7 基类

6.8 派生类

6.9 多态机制

- 1) 多态通过虚函数来实现，在运行时根据基类指针指向的具体子类调用特定被*override*的函数，具体虚函数表原理可查看*6.2*章节。

- 2) 若子类在需要重写的虚函数结尾加了*override*关键字则该函数一定是父类中存在可被重写的虚函数，否则在编译时会报错。

6.10 派生类的构造函数

- 1) 当创建一个派生类对象时，派生类的构造函数必须首先通过调用基类的构造函数来对基类的数据成员进行初始化，然后再执行派生类构造函数的函数体，对派生类新增的数据成员进行初始化。当派生类对象的生存期结束时，析构函数的调用顺序相反。
- 2) 派生类构造函数调用基类构造函数

隐式调用：不指定基类的构造函数，默认调用基类默认构造函数（不带参数或者带默认参数值的构造函数）

显式调用：指定调用基类的某个构造函数。除非基类有默认构造函数，否则都要用显示调用。

<派生类名>::<派生类名>(<形参声明>) : <基类名>(<参数表>)
{
<派生类构造函数的函数体>
}

6.11 抽象类

抽象类即为声明了纯虚函数的类，这种类不能实例化为对象，继承这种类的派生类需要实现对应的纯虚函数才能够进行实例化，否在在编译时会报错。通过这种方式可以提供一个单纯只提供接口的父类，而不需要对对应的接口进行某种特定的实现。

`virtual void MyFunc() = 0;`

6. 命名空间

`namespace`分为有名命名空间和无名命名空间，无名的由于没有名字所以其他文件无法引用，相当于该文件里面的`static`。`namespace`中的变量或者函数通过作用域符进行访问：

```
namespace MyNamespace{
int value = 10;
void MyFunc(int value){
    cout<<”value:”<<value<<endl;
}
}
MyNamespace::MyFunc(MyNamespace::value);
```

若作用域符前面没有任何`namespace`或者类名，则表示访问的是全局变量。但这个规则不适用于*C++98*风格的枚举型别中定义的枚举量。这些枚举量的名字属于包含着这个枚举型别的作用域，这就意味着在此作用域内不能有其他实体取相同的名字。

6.1 限定作用域枚举类型

先说一个通用规则，如果在一对大括号里声明一个名字，则该名字的可见性就被限定在括号括起来的作用于内。

```
enum Color { black, white, red};                    // black, white, red 和
auto white = false;                                 // 错误！因为 white
                                                        // 在这个定义域已经被声明过
```

事实就是枚举元素泄露到包含它的枚举类型所在的作用域中，对于这种类型的`enum`官方称作无作用域的（`unscoped`）。在`C++11`中对应的使用作用域的enums（`scoped enums`）不会造成这种泄露：

```
enum class Color { black, white, red};            // black, white, red
                                                      // 作用域为 Color
auto white = false;                               // fine, 在这个作用域内
                                                      // 没有其他的 "white"
Color c = white;                                  // 错误！在这个定义域中
                                                      // 没有叫"white"的枚举元素
Color c = Color::white;                           // fine
auto c = Color::white;                            // 同样没有问题（和条款5
                                                      // 的建议项吻合）
```

7. Name Mangling

*mangling*的目的就是为了给重载的函数不同的签名，以避免调用时的二义性调用。如果希望*C++*编译出来的代码不要被*mangling*，可以使用*extern "C" {}*来讲目标代码包含起来，这样能使得*C++*编译器编译出的二进制目标代码中的链接符号是未经过*C++*名字修饰过的，就像*C*编译器一样。

如果想将*mangling*的符号恢复成可读的，可以使用*linux*下的*c++filt*。

如果有一些编译器*mangling*的例子：

编译器	`void h(int)`	`void h(int, char)`	`void h(void)`
Intel C++ 8.0 for Linux	`_Z1hi`	`_Z1hic`	`_Z1hv`
HP aC++ A.05.55 IA-64	`_Z1hi`	`_Z1hic`	`_Z1hv`
IAR EWARM C++ 5.4 ARM	`_Z1hi`	`_Z1hic`	`_Z1hv`
GCC 3.*x* and 4.*x*	`_Z1hi`	`_Z1hic`	`_Z1hv`
GCC 2.9*x*	`h__Fi`	`h__Fic`	`h__Fv`
HP aC++ A.03.45 PA-RISC	`h__Fi`	`h__Fic`	`h__Fv`
Microsoft Visual C++ v6-v10	`?h@@YAXH@Z`	`?h@@YAXHD@Z`	`?h@@YAXXZ`
Digital Mars C++	`?h@@YAXH@Z`	`?h@@YAXHD@Z`	`?h@@YAXXZ`
Borland C++ v3.1	`@h$qi`	`@h$qizc`	`@h$qv`
OpenVMS C++ V6.5（ARM模式）	`H__XI`	`H__XIC`	`H__XV`
OpenVMS C++ V6.5（ANSI模式）	`CXX$__7H__FI0ARG51T`	`CXX$__7H__FIC26CDH77`	`CXX$__7H__FV2CB06E8`
OpenVMS C++ X7.1 IA-64	`CXX$_Z1HI2DSQ26A`	`CXX$_Z1HIC2NP3LI4`	`CXX$_Z1HV0BCA19V`
SunPro CC	`__1cBh6Fi_v_`	`__1cBh6Fic_v_`	`__1cBh6F_v_`
Tru64 C++ V6.5（ARM模式）	`h__Xi`	`h__Xic`	`h__Xv`
Tru64 C++ V6.5（ANSI模式）	`__7h__Fi`	`__7h__Fic`	`__7h__Fv`
Watcom C++ 10.6	`W?h$n(i)v`	`W?h$n(ia)v`	`W?h$n()v`

8. *new/delete*重载

8.1 *new*

new operator：指我们在C++里通常用到的关键字，比如`A* a = new A;`

operator new：它是一个操作符，并且可被重载(类似加减乘除的操作符重载)

当我们调用*new operator*时会调用*operator new*来分配内存，其中会调用*malloc*函数，接着会在分配内存上调用对象的构造函数，最后再返回这个指针。但是*operator new*却不一定在会调用*malloc*函数，我们可以通过*operator new*在指定内存上面调用构造函数，该地址可以是堆也可以是栈，比如我希望在其中一个构造函数内调用另外一个构造函数：

```
class A
{
public:
    A(int x):x(x){}
    A(A a)
    {
        //A(a.x);//这样调用返回的是一个临时A，我们需要在调用A(A a)这个构造函数作用域的地方还能维持该内存，所以我们需要像下面这样调用
        new (this) A(a.x);//this对于外面作用域是持续的，所以可以在this这个地址调用构造函数
    }
private:
    int x;
};
```

关于这两者的关系，我找到一段比较经典的描述（来自于www.cplusplus.com 见参考文献：

> operator new can be called explicitly as a regular function, but in C++, new is an operator with a very specific behavior: An expression with the new operator, first calls function operator new (i.e., this function) with the size of its type specifier as first argument, and if this is successful, it then automatically initializes or constructs the object (if needed). Finally, the expression evaluates as a pointer to the appropriate type.

通过重载这两个运算符我们可以记录内存分配删除信息来构建内存池，或者添加一些打印信息的功能。

```
#include <iostream>
using std::cout;
using std::endl;

void* operator new(size_t size, const char *file, unsigned int line) {
    cout << "file:" << file << " line:" << line << " size:" << size << " Call ::operator new" << endl;
    void* tmp = malloc(size);
    return tmp;
}
void operator delete(void *ptr, const char *file, unsigned int line) {
    cout << "Call ::operator delete" << endl;
    free(ptr);
}
class A {
public:
    A() {
        cout << "A constructor" << endl;
    }
    ~A()
    {
        cout << "A destructor" << endl;
    }
private:
    int a[20];
};

//#define new new(__FILE__,__LINE__)

int main()
{
#ifndef new
```

```
    A *a = reinterpret_cast<A*>(operator new(sizeof(A), __FILE__, __LINE__));
    new(a) A;
    delete a;

    A *b = new(__FILE__, __LINE__) A;//这么写会调用重载的operator new 再调用A的构造函
数
    delete b;
#else
    A *b = new A;
    delete b;
#endif
    system("pause");
    return 0;
}
```

new或者malloc出来的数组根据编译器的不同会把其大小存放在某个特定的内存，比如说VS2017中存放在了分配地址的前4个字节中，我们可以这么获得其大小：

```
int *a = new int[125];
cout << *((int*)a - 1) << endl;//打印125,只能是普通类型，如果是类结果还是125，这样就是错
的
```

linux下我们可以通过函数malloc_usable_size来获得普通类型数组的大小，如果是类则会得到0，可能需要其他方式得到，这些都是编译器的特性，千万不能再实际产品中使用这种危险的方式获得数组的大小。

8.2 *delete*

delete和new一样也分为delete operator和operator delete，operator delete 一样可以进行重载，在这里我们不多介绍。只说一个delete和delete[]的区别：

若指针为非基本类型，delete只会为指针的第一个元素调用析构函数，而delete[]会为每一个对象调用析构函数；若指针未基本类型则没有区别。

9. 类型转换

*C++*同样支持*C*风格的强制转换(*Type Cast*): `TypeName b = (TypeName)a;`

*C++*的四种类型转换：

- 1.*const_cast*
  - 常量指针被转化成非常量的指针，并且仍然指向原来的对象
  - 常量引用被转换成非常量的引用，并且仍然指向原来的对象
  - 常量指针是指向常量的指针，指指针指向的内容不能改变，声明为`const int *p;`或者`int const *p;`
  - 指针常量是指针的值为常量，指指针指向的地址不能改变，声明为`int * const p = &a;`
- 2.*static_cast*
  - *static_cast* 作用和C语言风格强制转换的效果基本一样，由于没有运行时类型检查来保证转换的安全性，所以这类型的强制转换和C语言风格的强制转换都有安全隐患。
  - 用于类层次结构中基类（父类）和派生类（子类）之间指针或引用的转换。注意：进行上行转换（把派生类的指针或引用转换成基类表示）是安全的；进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的。

    - 用于基本数据类型之间的转换，如把*int*转换成*char*，把*int*转换成*enum*。这种转换的安全性需要开发者来维护。
    - *static_cast*不能转换掉原有类型的*const*，*volatile*、或者 *__unaligned*属性。(前两种可以使用 *const_cast* 来去除)
    - 在*c++ primer*中说道：*C++* 的任何的隐式转换都是使用 *static_cast* 来实现。
- 3.*dynamic_cast*
    - 该转换涉及面向对象的多态性和程序运行时的状态,也与编译器的属性设置有关.所以不能完全使用C语言的强制转换替代
    - 从子类到基类不会有任何问题
    - 从基类到子类会对*RTTI*(*runtime type information*)进行检查，由于需要*RTTI*，所以需要有虚函数表，所以需要有虚函数，若没有虚函数则会出现编译错误；若子类拥有父类没有的成员，则函数会返回*nullptr*。

      *typeid*运算符也是读取*RTTI*的机制，同样需要有虚函数，在生成虚表之后上面会挂一个*type_info*结构体，通过这个结构体父类指针在运行时可以读取出相关的信息进行判断。而*sizeof*运算符则是在编译时求值，只关心静态声明的类型，不过也可以有运行时的语义，当*sizeof*的参数是Variable-Length Array时。

      > a) If expression is a glvalue expression that identifies an object of a polymorphic type (that is, a class that declares or inherits at least one virtual function), the typeid expression evaluates the expression and then refers to the std::type_info object that represents the dynamic type of the expression. If the glvalue expression is obtained by applying the unary * operator to a pointer and the pointer is a null pointer value, an exception of type std::bad_typeid or a type derived from std::bad_typeid is thrown.

- 4.*reinterpret_cast*
    - *reinterpret_cast*是强制类型转换符用来处理无关类型转换的，通常为操作数的位模式提供较低层次的重新解释。但是他仅仅是重新解释了给出的对象的比特模型，并没有进行二进制的转换。他是用在任意的指针之间的转换，引用之间的转换，指针和足够大的*int*型之间的转换，整数到指针的转换。

10. 异常机制

11. *auto*

- 对于*auto*而言，其意义在于*type deduce*，所以它不会允许没有初始化的声明，这样对于开发者来说是一个很好的习惯；
- 另外一个意义是简化代码，尤其是经常使用容器迭代器初始化；
- 如果想保存*lambda*表达式，不需要自己推导一个函数指针或者*function*类型变量，直接使用*auto*即可；

```
auto closure = [](const int&, const int&) {}
```

- 在*C++14*的泛型*lambda*表达式中，我们还可以将参数定义成*auto*类型，通过这个我们可以实现*ChurchNumber*，后续在13章作介绍；

```
auto closure = [](auto x, auto y) { return x * y;}
```

- 在*C++1X*中，配合*decltype*可以解决原来难以让编译器自动推导模板函数返回值的问题，原来我们需要先转换成万能的0，再转成指针，最后再取指针：

```
template<class T,class U>
decltype(*(T*)(0) * *(U*)(0)) mul(T x, U y)
{
    return x*y;
}
```

*C++11*可以直接这样：

```
template<class T,class U>
auto mul(T x, U y)->decltype(x * y)
{
    return x*y;
}
```

而在*C++14*中我们可以直接这样：

```
template<class T,class U>
decltype(auto) mul(T x, U y)
{
    return x*y;
}
```

其中的*auto*在编译时会替换成函数返回的表达式，再通过*decltype*来推导其类型。

- 当不声明为引用类型时，*auto*的初始化表达式即使是引用，编译器也并不会将该变量推导为对应类型的引用，*auto*的推导结果等同于初始化表达式去除引用和*const qualifier*，所以在实际编码中我们需要显式的声明引用。当声明为引用时*auto*的推导结果能推导出正确的类型，其结果能保留初始化表达式的*qualifier*：

```
#include <iostream>
using namespace std;
int main()
{
    int x = 0;
    auto *a = &x;//类型为int*，auto为int
    ++(*a);
    cout << "after ++(*a) *a:" << *a << " x:" << x << endl;
    auto b = &x;//类型为int*，auto为int*
    ++(*b);
    cout << "after ++(*b) *b:" << *b << " x:" << x << endl;
    auto &c = x;//类型为int&，auto为int
    ++c;
    cout << "after ++c c:" << c << " x:" << x << endl;
    auto d = c;//类型为int，auto为int
    ++d;
    cout << "after ++d d:" << d << " x:" << x << endl;

    const auto e = x;//类型为const int，auto为int
    //++e; error C3892: “e”: 不能给常量赋值
    auto f = e;//类型为int，auto为int
    ++f;
    cout << "after ++f f:" << f << " x:" << x << endl;
```

```
    const auto &g = x;//类型为const int&，auto为int
    //++g; error C3892: “g”: 不能给常量赋值
    auto& h = g;//推导为const int&,auto为const int
    //++h; error C3892: “g”: 不能给常量赋值
    auto *i = &e; //推导为 const int*，常量指针，不能改变指针指向的内容
    //++(*i);  error C3892: “i”: 不能给常量赋值
    auto j = i;
    //++(*j); error C3892: “j”: 不能给常量赋值
    return 0;
}
```

- *auto*有时候可能得不到我们预想的类型，其实主要是因为基础不够扎实，比如*vector<bool>*。*vector<bool>*是*vector<T>*的一个特例化(*template specialization*)，这个特例化要解决的问题是存储容量的问题。

  To optimize space allocation, a specialization of vector for bool elements is provided.

  所以，它一般来说是以位的方式来存储*bool*的值。从这里我们可以看出，如果使用位来提升空间效率可能引出的问题就是时间效率了，因为我们的计算机地址是以字节为单位的，根据网友的实验遍历访问*vector*要比其他*vector<int>*耗时40倍以上。

  对于*vector<bool>*来说，它的*operator[]*返回的不是对应元素的引用，而是一个*member class std::vector<bool>::reference* 。对于普通*vector<T>*来说，若是使用*auto*声明变量对保存*opertor[]*返回的值，会如同上一段所说的去除引用，而对于*vector<bool> operator*返回的*std::vector<bool>::reference*是一个*member class*：

  > *This embedded class is the type returned by members of non-const vector when directly accessing its elements. It accesses individual bits with an interface that emulates a reference to a `bool`.*

  根据定义这个类可以以*bit*为单位访问对应的*bool*，并且是以引用的方式，所以*auto*声明得到的*vector<bool>::operator[]*返回值是可以改变*vector<bool>*对应元素的内容的。如果我们试图用*auto&*来定义一个*vector<bool>::operator[]*返回的值，会出现如下编译错误：

  > error: invalid initialization of non-const reference of type 'std::*Bit_reference&' from an rvalue of type 'std::*Bit_iterator::reference {aka std::_Bit_reference}'

  这是因为对于*std::vector<bool>::reference*这种*proxy reference*来说，我们对其*dereference*并不会得到一个普通的*bool &*，取而代之的是一个临时对象，所以取引用会导致编译错误。这时候我们使用右值引用来定义这个变量可以解决*vector<bool>*想遍历修改的问题，右值引用对*vector<T>*的其他类型同样适用：

```
vector<bool> v = {true, false, false, true};
// Invert boolean status
for (auto&& x : v)  // <-- note use of "auto&&" for proxy iterators
    x = !x;
```

12. 移动语义和右值引用

首先我们来定义一下左值和右值，左值就是有名字的对象或者变量，可以被赋值或给别的对象或变量赋值，比如 `obj`，`*ptr`，`ptr[index]`，和 `++x` ；而右值就是临时变量（对象），不能被赋值，比如 `1729`，`x + y`，`std::string("meow")`，和 `x++`，另外还有函数的返回值。

12.1 右值引用

在*6.2*章节构造函数章节中我们介绍了移动构造函数，入参中的*A &&a*即为右值引用，当我们使用*move*函数将变量转换成右值引用之后再进行构造或者直接使用右值进行构造都会触发调用我们的移动构造函数。我们一般会在移动构造函数中实现移动语义的功能，就是不会新分配一块内存，而是将旧的内存直接赋给新的对象，并将旧的对象的指针赋成*nullptr*：

```
A(A &&a)
{
    this->n = a.n;
    a->n = nullptr;
    cout << "A move construction" << endl;
}
```

如果我们为对象实现了移动构造函数，在函数中返回局部对象时我们也会触发移动语义：

- 1) 局部变量赋给临时变量时触发移动构造函数（没有则触发拷贝构造函数）
- 2) 临时变量赋值给函数已经声明过的外部变量时会触发移动赋值构造函数（这种情况没有会编译出错）
- 3) 若是声明时赋值则触发移动构造函数（若没有可以触发拷贝构造函数）
- 4) 若移动赋值构造函数返回值会在临时对象赋值给外部对象时再调用一次复制构造函数，这是因为又实例化了一个新的对象
- 5) 这一段可结合*6.3*章节一起看

```
A test()
{
    A a;
    return a;
}
A a1 = test();//触发移动构造函数，若没有则触发拷贝构造函数
A a2;
a2 = test();//触发移动赋值构造函数，若没有编译出错： error: invalid initialization of
non-const reference of type ‘A&’ from an rvalue of type ‘A’
```

右值引用使*C++*标准库的实现在多种场景下消除了不必要的额外开销（如*std::vector*,*std::string*），也使得另外一些标准库（如*std::unique_ptr*,*std::function*）成为可能。右值引用的意义通常为两大作用：移动语义和完美转发。移动语义即为上述所示的移动构造函数，*std::vector*和*std::string*也可以通过移动构造函数来构造，这样就避免了重新给vetor或者string分配空间的开销：

```
template<typename T>
decltype(auto) move(T&& param)
{
    using ReturnType = remove_reference_t<T>&&;
    return static_cast<ReturnType>(param);

}
std::vector<int> vec(100,1);
std::vector<int> vec1 = move(vec);
```

而对于模板函数来说，`void func(T&& param)`其中的*T&&*并不一定表示的是右值，它绑定的类型是未知的，既可能是右值，也可能是左值，需要类型推导之后才会知道，比如：

```
template<typename T>
void func(T&& param){}
func(10); // 10是右值
int x = 10;
func(x);  // x是左值
```

12.1.1 右值引用接受参数

若我们的函数是右值引用，我们能接受哪些参数能，假设我们有一个函数：

```
class Data {};
void func(Data && data) {}
```

- 情形一：

  ```
  Data data;
  func(data); //[Error] cannot bind 'Data' lvalue to 'Data&&'
  ```

  data是个左值，不能绑定到右值上

- 情形二：

  ```
  Data data;
  Data & d = data;
  func(d); //[Error] cannot bind 'Data' lvalue to 'Data&&'
  ```

  d同样是一个左值

- 情形三：

  都说const 引用和 右值引用有相似之处，尝试传递const 引用

  ```
  Data data;
  const Data & d = data;
  func(d); // [Error] invalid initialization of reference of type, 'Data&&'
  from expression of type 'const Data'
  ```

  仍然不能传

- 情形四：

  ```
  func(Data());
  ```

  ok，匿名对象为右值

- 情形五：

  标准做法

  ```
  Data data;
  func(std::move(data));//OK
  ```

- 情形六：

  move一个做值引用：

```
Data data;
Data & p = data;
func(std::move(p)); //OK
```

- 情形七：

  把一个右值参数传递给const 引用类型

```
void func(const Data & data){}
void func_1(Data && data)
{
    func(data);//OK
}
Data data;
func_1(std::move(data));
```

- 情形八：

  直接声明一个右值引用，来做参数传递

```
Data p;
Data && p1 = std::move(p);
func(p1); // [Error] cannot bind 'Data' lvalue to 'Data&&'
```

  同样的错误，说明p1 还是左值 ， 我们可以通过这个方式验证一下

```
void func(Data && data){}
void func_1(Data && data)
{
    func(data);//[Error] cannot bind 'Data' lvalue to 'Data&&'
}
Data data;
func_1(std::move(data));
```

  这时候就需要我们下一节会介绍的万能引用出场了：

```
Data p;
Data && p1 = std::move(p);
func(std::forward<Data>(p1)); // OK
void func_1(Data && data)
{
   func(std::forward<Data>(data));
}
```

12.2 万能引用

这种未定的引用类型称为万能引用(*universal reference*)，这种类型必须被初始化，具体是什么类型取决于它的初始化。由于存在*T&&*这种未定的引用类型，当它作为参数时，有可能被一个左值引用或右值引用的参数初始化，这是经过类型推导的*T&&*类型，相比右值引用(*&&*)会发生类型的变化，这种变化就称为引用折叠，引用折叠规则如下：

- 1.所有右值引用折叠到右值引用上仍然是一个右值引用。（*T&& &&*变成 *T&&* ）
- 2.所有的其他引用类型之间的折叠都将变成左值引用。（*T& &* 变成 *T&*; *T& &&* 变成 *T&*; *T&& &* 变成 *T&* ）

对于万能引用，我们可能需要知道它什么时候是右值引用什么时候是左值引用，这时候我们就需要完美转发*std::forward<T>()*。如果传进来的参数是一个左值，*enter*函数会将T推导为*T&*，*forward*会实例化为*forward<T&>*，*T& &&*通过引用折叠会成为*T&*，所以传给*func*函数的还是左值；如果传进来的是一个右值，*enter*函数会将*T*推导为*T*，*forward*会实例化为*forward<T>*，*T&&*通过引用折叠还是*T&&*，所以传给*func*函数的还是右值：

```
template<typename T>
T&& forward(typename remove_reference<T>::type& param)
{
    return static_cast<T&&>(param); //会发生引用折叠
}

template <typename T>
void func(T t) {
    cout << "in func " << endl;
}

template <typename T>
void enter(T&& t) {
    cout << "in enter " << endl;
    func(std::forward<T>(t));
}
```

13. *lambda*表达式

*lambda*表达式对于*C++*的意义有两条：

- 第一条是可以在表达式中直接定义一个函数，而不需要将定义函数和表达式分开。
- 第二条是引入了闭包。闭包是指将当前作用域中的变量通过值或者引用的方式封装到*lambda*表达式当中，成为表达式的一部分，它使*lambda*表达式从一个普通的函数变成一个带隐藏参数的函数。

基本语法我们在函数章节有所介绍：

```
[capture list] (params list) -> return type {function body};
```

[]中我们可以定义好闭包变量的捕获方式：

```
[]        //未定义变量.试图在Lambda内使用任何外部变量都是错误的.
[x, &y]   //x 按值捕获, y 按引用捕获.
[&]       //用到的任何外部变量都隐式按引用捕获
[=]       //用到的任何外部变量都隐式按值捕获
[&, x]    //x显式地按值捕获. 其它变量按引用捕获
[=, &z]   //z按引用捕获. 其它变量按值捕获
```

对于成员函数中的*lambda*表达式如果我们期望成员变量按值捕获要注意不能直接用*[=]*让任何外部变量都隐式按值捕获，因为捕获了*this*之后其实是能够操作所有成员变量，这样所有成员变量实际是按引用捕获的，所以应该明确写出对*this*的捕获：

```
class MyClass {
public:
    void Foo()
    {
        int i = 0;
```

```
        auto Lambda = [=]() { Use(i, data_); };   // 不好: 看起来像是拷贝/按值捕获，成员变量实际上是按引用捕获
        data_ = 42;
        Lambda(); // 调用 use(42);
        data_ = 43;
        Lambda(); // 调用 use(43);
        auto Lambda2 = [i, this]() { Use(i, data_); }; // 好，显式指定按值捕获，最明确，最少的混淆
    }
private:
    int data_ = 0;
};
```

()中我们定义了需要直接直接传入*lambda*表达式的形参；

->之后我们定义的是返回值类型，当然我们可以将其隐藏，编译器会根据 *return*表达式进行类型推导。除了返回值可以类型推导，在*C++14*中我们使用generic lambda还能对形参进行类型推导；

通过闭包和返回值以及形参的类型推导我们可以进行函数式编程，比方说构造*Church Number*：

```
#include <iostream>
using namespace std;
//定义0函数
auto zero = [](auto f){
    return [=](auto x){
        return x;
    };
};
//定义后继函数
auto succ = [](auto num){
    return [=](auto f){
        return [=](auto x){
            return f(num(f)(x));
        };
    };
};
//定义加法函数
auto add = [](auto m, auto n){
    return [=](auto f){
        return [=](auto x){
            return m(f)(n(f)(x));
        };
    };
};
//定义乘法函数
auto mul = [](auto m, auto n){
    return [=](auto f){
        return [=](auto x){
            return m(n(f))(x);
        };
    };
};
int main()
{
    auto two = succ(one);//通过后继函数得到2
    auto three = add(one, two);//通过加法函数得到3
    auto six = mul(two, three);//通过乘法函数得到6
```

```
    auto f = [](auto x){cout<<"x:"<<x<<endl; return x+1;}//为了能具象数字，我们定义f
    six(f)(0);
    return 0;
}
```

14.智能指针

智能指针是在普通指针的基础上封装了一层*RAII*机制，这样一层封装机制的目的是为了使得指针可以方便的管理一个对象的生命周期。在程序员难以判断指针需要在什么时候释放，忘记释放，或者抛出异常时能安全的将内存释放。

智能指针分为四种：*auto_ptr*（摒弃），*unique_ptr*，*shared_ptr*和*weak_ptr*。旧的*auto_ptr*在对新的*auto_ptr*进行复制构造了之后旧的便会失效，而*unique_ptr*在*auto_ptr*的基础上禁止了复制构造，但是可以使用移动语义转移所有权，如果希望函数返回临时变量unique_ptr可将临时变量转成右值引用进行返回，这样就会触发unique_ptr的移动构造函数：

```
unique_ptr<int> up(new int);
unique_ptr<int> up1(ap); //error
unique_ptr<int> up2 = ap; //error
unique_ptr<int> GetVal( ){
    unique_ptr<int> up(new int);
    return up;
}
unique_ptr<int> uP3 = GetVal(); //ok
unique_ptr<int> uP4 = move(up); //ok
unique_ptr<int> test()//返回值不能为rvalue reference，否则会产生dangling reference，跟lvalue reference一样
{
    unique_ptr<int> a = unique_ptr<int>();
    return move(a);//vs2017可不用转，gcc5.4中需要否则编译错误
}
unique_ptr<int> a1 = move(test());//ok
```

*shared_ptr*则会更加灵活，在*unique_ptr*的基础上增加了引用计数，每一次显示或者是隐式构造都会增加引用计数（引用计数为原子操作，线程安全，但管理的内存需要自己来维护线程安全，除非使用*unique_ptr*），当引用计数归零之后会在其析构函数中调用*deleter*函数来释放其管理的内存。*shared_ptr*的默认*deleter*为`[](T *a){delete a;}`，所以如果让*shared_ptr*管理对象数组时需要指定*deleter*为`[](T *a){delete[] a;}`，否则*shared_ptr*不能正确调用所有对象的析构函数。而由于*unique_ptr*不能在定义时管理数组，在*reset*时又不能*reset deleter*，所以*unique_ptr*在*C++17*前都不能管理数组。

在实际使用*shared_ptr*的过程中我们不可避免的会出现循环引用的情况，比如下面这种情况：

```
class ClassB;
class ClassA
{
public:
    ClassA() { cout << "ClassA Constructor..." << endl; }
    ~ClassA() { cout << "ClassA Destructor..." << endl; }
    shared_ptr<ClassB> pb;  // 在A中引用B
};

class ClassB
{
```

```
public:
    ClassB() { cout << "ClassB Constructor..." << endl; }
    ~ClassB() { cout << "ClassB Destructor..." << endl; }
    shared_ptr<ClassA> pa;  // 在B中引用A
};

int main() {
    shared_ptr<ClassA> spa = make_shared<ClassA>();
    shared_ptr<ClassB> spb = make_shared<ClassB>();
    spa->pb = spb;
    spb->pa = spa;
    return 0;
}
```

这时候我们就需要用到*weak_ptr*。*weak_ptr*是为了配合*shared_ptr*引入的一种智能指针，它指向一个由*shared_ptr*管理的对象而不影响所指对象的生命周期，也就是将一个*weak_ptr*绑定到一个*shared_ptr*不会改变*shared_ptr*的引用计数。不论是否有*weak_ptr*指向，一旦最后一个指向对象的*shared_ptr*被销毁，对象就会被释放。从这个角度看，*weak_ptr*更像是*shared_ptr*的一个助手而不是智能指针。

由于*weak_ptr*访问的指针可能被释放，所以我们不能直接访问指向的内存，我们可以用成员函数*lock*来判断，如果内存未释放，则返回一个指向内存的*shared_ptr*，若释放了则返回一个值为*nullptr*的*shared_ptr*：

```
class A
{
public:
    A() { cout << "A Constructor..." << endl; }
    ~A() { cout << "A Destructor..." << endl; }
};

int main() {
    shared_ptr<A> sp(new A());
    weak_ptr<A> wp(sp);
    //sp.reset();
    if (shared_ptr<A> pa = wp.lock()) {
        cout << pa->a << endl;
    } else {
        cout << "wp指向对象为空" << endl;
    }
}
```

*shared_ptr*在*C++17*之前都不支持动态数组，所以在这之前如果用*shared_ptr.reset()*数组之后需要自定义*deleter*，使用*delete[]*来进行释放：

```
std::shared_ptr<int[]> sp1(new int[10]()); // 错误，c++17前不能传递数组类型作为shared_ptr的模板参数
std::unique_ptr<int[]> up1(new int[10]()); // ok, unique_ptr对此做了特化
std::shared_ptr<int> sp2(new int[10]()); // 错误，可以编译，但会产生未定义行为
```

14.1 常用函数

```
get(); //返回管理的裸指针
shared_ptr<ClassName> sp(new ClassName,[](ClassName* p){delete p;});//构造函数，自定义deleter
reset(p, Del);//重新设置维护的指针及其对应的deleter,只有shared_ptr可以reset deleter，unique_ptr不行
get_deleter();//获得智能指针的deleter
template <class T, class... Args>
shared_ptr<T> make_shared (Args&&... args);//相当于调用T类的构造函数，
```

14.2 *enable_shared_from_this*

使用智能指针难以避免的场景之一就是需要把在类的成员函数里把当前类的对象作为参数传给其他异步函数，这时候需要在成员函数里获得一个管理*this*指针的shared_ptr，我们可能想要这么做：

```
class Foo
{
public:
    void Bar(std::function<void(Foo*)> fnCallback)
    {
        std::thread t(fnCallback,this);
        t.detach();
    }
};
```

但是我们不能保证在fnCallback异步调用的时候Foo对象没有被析构，所以我们可能想要给fnCallback回调传一个shared_ptr管理的Foo对象指针，像这样：

```
class Foo
{
public:
    void Bar(std::function<void(std::shared_ptr<Foo>)> fnCallback)
    {
        std::shared_ptr<Foo> pFoo(this);
        std::thread t(fnCallback,pFoo);
        t.detach();
    }
};
```

然而这样就让两个*shared_ptr*来管理一个对象，两个*shared_ptr*不共享引用计数，各自都是1，所以仍然会让成员函数外的*shared_ptr*析构之后释放*Foo*。这时候我们就需要j继承*enable_shared_from_this*类来帮我们获得一个和外面*shared_ptr*共享引用计数的新的*shared_ptr*：

```
class CallbackClass : public enable_shared_from_this<CallbackClass> {
public:
    void CallCallbackFunc(std::function<void(shared_ptr<CallbackClass>)>
CallbackFunc) {
        auto sp = shared_from_this();
        thread t(CallbackFunc,sp);
        t.detach();
        //t.join();
        return;
    }
    CallbackClass():a(0) {
        cout << "CallbackClass constructor" << endl;
```

```
    }
    ~CallbackClass() {
        cout << "CallbackClass destructor" << endl;
    }
    int Adda() {
        a++;
        return a;
    }
private:
    int a;
};

void Foo()
{
    shared_ptr<CallbackClass> sp = make_shared<CallbackClass>();
    sp->CallCallbackFunc([](shared_ptr<CallbackClass> sp) {
        Sleep(100);
        cout << "sleep 100 ms done" << endl;
        for (int i = 0; i < 1000; ++i) {
            cout << "a:" << sp->Adda() << endl;
        }
    });
    return;
}
int main(int argc, char* argv[])
{
    Foo();
    cout << "Foo func done" << endl;
    system("pause");
    return 0;
}
```

*enable_shared_from_this*中包含一个*weak_ptr*，在初始化*shared_ptr*时，构造函数会检测到这个该类派生于*enable_shared_from_this*，于是将这个*weak_ptr*指向初始化的*shared_ptr*。调用*shared_from_this*，本质上就是*weak_ptr*的一个*lock*操作。

15. *STL*容器

15.1 顺序容器(*sequence container*)

15.1.1 *vector*

*vector<bool>*为该模板类的特例化，我们在前文中已经介绍过了，不再赘述。

15.1.1.1 数据结构

*vector*的底层数据结构是动态数组，因此，*vector*的数据安排以及操作方式与*std::array*十很相似，它们间的唯一差别在于对空间的运用灵活性上。*array*为静态数组，有着静态数组最大的缺点：每次只能分配一定大小的存储空间，当有新元素插入时，要经历 “找到更大的内存空间”->“把数据复制到新空间” ->“销毁旧空间” 三部曲， 对于*std::array*而言，这种空间管理的任务压在使用它的用户身上，用户必须把握好数据的数量，尽量在第一次分配时就给数据分配合理的空间（这有时很难做到），以防止“三部曲”带来的代价，而数据溢出也是静态数组使用者需要注意的问题。而*vector*用户不需要亲自处理空间运用问题。*vector*是动态空间，随着新元素的插入，旧存储空间不够用时，*vector*内部机制会自行扩充空间以容纳新元素，当然，这种空间扩充大部分情况下（几乎是）也逃脱不了“三部曲”，只是不需要用户自

己处理，而且*vector*处理得更加安全高效。*vector*的实现技术关键就在于对其大小的控制以及重新配置时数据移动效率。

15.1.1.2 迭代器类型

对于*Cstyle*数组，我们使用普通指针就可以对数组进行各种操作。*vector*维护的是一个连续线性空间，与数组一样，所以无论其元素型别为何，普通指针都可以作为*vector*的迭代器而满足所有必要的条件。*vector*所需要的迭代器操作，包括*operator*,*operator->*,*operator++*,*operator--*,*operator+=*,*operator-=*等，普通指针都具有。因此，普通指针即可满足*vector*对迭代器的需求。所以，*vector*提供了*Random Access Iterators*。

15.1.1.3 内存分配

标准库的实现者使用了这样的内存分配策略：以最小的代价连续存储元素。为了使*vector*容器实现快速的内存分配，其实际分配的容量要比当前所需的空间多一些(预留空间)，*vector*容器预留了这些额外的存储区用于存放添加的新元素，于是不必为每个新元素进行一次内存分配。当继续向容器中加入元素导致备用空间被用光（超过了容量*capacity*），此时再加入元素时*vector*的内存管理机制便会扩充容量至两倍，如果两倍容量仍不足，就扩张至足够大的容量。容量扩张必须经历"重新配置、元素移动、释放原空间"这个浩大的工程。按照《*STL*源码剖析》中提供的*vector*源码，*vector*的内存配置原则为：

- 如果*vector*原大小为0，则配置1，也即一个元素的大小。
- 如果原大小不为0，则配置原大小的两倍。

当然，*vector*的每种实现都可以自由地选择自己的内存分配策略，分配多少内存取决于其实现方式，不同的库采用不同的分配策略。

15.1.1.4 迭代器失效

对于迭代器失效的问题，*vector*有三种情况会导致迭代器失效：

- *vector*管理的是连续的内存空间，在容器中插入（或删除）元素时，插入（或删除）点后面的所有元素都需要向后（或向前）移动一个位置，指向发生移动的元素的迭代器都失效。
- 随着元素的插入，原来分配的连续内存空间已经不够且无法在原地拓展新的内存空间，整个容器会被*copy*到另外一块内存上，此时指向原来容器元素的所有迭代器通通失效。
- 删除元素后，指向被删除元素的迭代器失效。

15.1.1.5 常用成员函数

```
explicit vector (size_type n, const value_type& val = value_type(), const
allocator_type& alloc = allocator_type());
template <class InputIterator> vector (InputIterator first, InputIterator last,
const allocator_type& alloc = allocator_type());
vector (const vector& x);
vector& operator= (const vector& x);//包含赋值和移动版本
reference at (size_type n);//以此替换operator[]，at函数会做边界检测
iterator insert (iterator position, const value_type& val);//Inserting new
elements before the element at the specified position. Return an iterator that
points to the first of the newly inserted elements.
void insert (iterator position, size_type n, const value_type& val);
template <class InputIterator>
void insert (iterator position, InputIterator first, InputIterator last);
template <class... Args>
iterator emplace (const_iterator position, Args&&... args);//和insert功能类型，返回
值也一样，只不过只能插入一个值的右值引用
iterator erase (iterator position);//Return an iterator pointing to the new
location of the element that followed the last element erased by the function
call.
```

```
iterator erase (iterator first, iterator last);
value_type* data() noexcept;//返回vertor管理的内存首地址
```

15.1.2 *list*

15.1.2.1 数据结构

list同样是一个模板类，它底层数据结构为双向循环链表。因此，它支持任意位置*O(1)*的插入/删除操作，不支持快速随机访问。*list*的迭代器具备前移、后移的能力，所以*list*提供的是*Bidirectional iterator*(双向迭代器)。由于采用的是双向迭代器，自然也很方便在指定元素之前插入新节点，所以*list*很正常地提供了*insert()*/*emplace()*操作与*push_back()*/*pop_back()*/*emplace_front()*/*emplace_back()*操作。

15.1.2.2 内存分配

*list*的空间配置策略，自然是像我们普通双向链表那样，有多少元素申请多少内存。它不像*vector*那样需要预留空间供新元素的分配，也不会因找不到连续的空间而引起整个容器的内存迁移。

15.1.2.3 迭代器失效

插入操作（*insert*）与接合操作（*splice*）都不会造成原有的list迭代器失效。这在*vector*是不成立的，因为*vector*的插入可能引起空间的重新配置，导致原来的迭代器全部失效。*list*的迭代器失效，只会出现在删除的时候，指向删除元素的那个迭代器在删除后失效。

15.1.2.4 常用成员函数

```
//构造函数同vector类似
void remove (const value_type& val);//Remove elements with specific value
iterator insert (iterator position, const value_type& val);//inserting new
elements before the element at the specified position. Return An iterator that
points to the first of the newly inserted elements.
void insert (iterator position, size_type n, const value_type& val);
template <class InputIterator>
void insert (iterator position, InputIterator first, InputIterator last);
iterator erase (iterator position);//Return An iterator pointing to the element
that followed the last element erased by the function call.
iterator erase (iterator first, iterator last);
```

15.1.3 *deque*

15.1.3.1 数据结构

*vector*是单向开口的线性连续空间，*deque*则是一种双向开口的连续数据空间。所谓的双向开口，意思是可以在头尾两端分别做元素的插入和删除操作。当然*vector*也可以在头尾两端进行操作，但是其头部操作效果奇差，所以标准库没有为*vector*提供*push_front*或*pop_front*操作。与*vector*类似，*deque*支持元素的快速随机访问。

*deque*由一段一段的定量连续空间构成。一旦有必要在*deque*的前端或尾端增加新空间，便配置一段定量连续空间，串接在整个*deque*的头端或尾端。*deque*的最大任务，便是在这些分段的定量连续空间上，维护其整体连续的假象，并提供随机存取的接口。避开了“重新配置、复制、释放”的轮回，代价则是复杂的迭代器架构。

受到分段连续线性空间的字面影响，我们可能以为*deque*的实现复杂度和*vector*相比差不太多，其实不然。主要因为，既是分段连续线性空间，就必须有中央控制，而为了维持整体连续的假象，数据结构的设计及迭代器前进后退等操作都颇为繁琐。*deque*的实现代码分量远比*vector*或*list*都多得多。

*deque*采用一块所谓的*map*（注意，不是*STL*的*map*容器）作为主控。这里所谓*map*是一小块连续空间，其中每个元素（此处称为一个节点，*node*）都是指针，指向另一段（较大的）连续线性空间，称为缓冲区。缓冲区才是*deque*的储存空间主体。*SGI STL*允许我们指定缓冲区大小，默认值0表示将使用512*bytes*缓冲区。

15.1.3.2 迭代器类型

*deque*的迭代器必须能够指出分段连续空间（亦即缓冲区）在哪里，其次它必须能够判断自己是否已经处于其所在缓冲区的边缘，如果是，一旦前进或后退就必须跳跃至下一个或上一个缓冲区。为了能够正确跳跃，*deque*必须随时掌握管控中心（*map*）。所以在迭代器中需要定义：当前元素的指针，当前元素所在缓冲区的起始指针，当前元素所在缓冲区的尾指针，指向*map*中指向所在缓区地址的指针，分别为*cur*，*first*，*last*，*node*。

15.1.3.3 迭代器失效

- 在deque容器首部或者尾部插入元素不会使得任何迭代器失效。
- 在其首部或尾部删除元素则只会使指向被删除元素的迭代器失效。
- 在deque容器的任何其他位置的插入和删除操作将使指向该容器元素的所有迭代器失效。

15.2 容器适配器(*container adaptor*)

如果说容器是*STL*中能保存数据的数据类型，那么容器适配器就是*STL*中为了适配容器提供特定接口的数据类型，所以底层是以关联容器为基础实现的。*C++*提供了三种容器适配器：*stack*，*queue*和*priority_queue*。*stack*和*queue*基于*deque*实现，*priority_queue*基于*vector*实现。容器适配器不支持任何类型的迭代器，即迭代器不能用于这些类型的容器。

15.2.1 *stack*

*stack*为了提供LILO的数据结构。常用成员函数为：

```
template <class... Args> void emplace (Args&&... args);
bool empty() const;
void pop();
void push (const value_type& val);
size_type size() const;
void swap (stack& x) noexcept;
value_type& top();
```

我自己测试发现底层容器改成vector会比用default的deque要快。

15.2.2 *queue*

*queue*为了提供FIFO的数据结构。常用成员函数为：

```
value_type& back();
template <class... Args> void emplace (Args&&... args);
bool empty() const;
value_type& front();
void pop();
void push (const value_type& val);
size_type size() const;
void swap (queue& x) noexcept;
```

15.2.3 *priority_queue*

*priority_queue*为了提供优先队列，数据结构为大根堆，能够常数时间获得最大的元素，插入删除时间复杂度为O(lgn)。常用成员函数为：

```
template <class... Args> void emplace (Args&&... args);
bool empty() const;
void pop();
void push (const value_type& val);
size_type size() const;
void swap (priority_queue& x) noexcept;
const value_type& top() const;
```

15.3 关联容器(*associative container*)

关联容器的关联指的是存储的元素的位置是和其值是相关联的，而不是像顺序容器一样是绝对的位置。其存储顺序可分为有序和无序两种，有序的是*map/set*，其内部数据结构为*RBT*；无序的就是*unordered_map/set*，其内部数据结构为*hashmap*。

15.3.1 *map/multimap*

常用成员函数：

```
mapped_type& at ( const key_type& k );//If k does not match the key of any element in the container, the function throws an out_of_range exception.
template <class P> pair<iterator,bool> insert (P&& val);//插入成功返回指向新元素的迭代器和true的pair，如有相同元素则返回原有元素迭代器和false的pair
template <class P> iterator insert (const_iterator position, P&& val);//和顺序容器的区别在于其不能随意存储，在这个重载中的第一个迭代器的含义不是像顺序存储一样让你把新的值插入这个迭代器后面，它仍然会进行排序插入正确的位置。只不过若插入恰巧发生在hint前的位置（用upper_bound()获得），则时间复杂度为常数
template <class InputIterator>
void insert (InputIterator first, InputIterator last);
mapped_type& operator[] (key_type&& k);//等于(*((this->insert(make_pair(k,mapped_type()))).first)).second，所以无论其实是否包含该元素都会做一次insert操作
iterator  erase (const_iterator position);//返回删除元素的下一个元素的迭代器
iterator  erase (const_iterator first, const_iterator last);
size_type erase (const key_type& k);//返回删除元素个数
iterator find (const key_type& k);
size_type count (const key_type& k) const;//map只返回0或1
pair<iterator,iterator> equal_range (const key_type& k);
key_compare key_comp() const;
template <class... Args>
pair<iterator,bool> emplace (Args&&... args);
map& operator= (const map& x);
map& operator= (map&& x);
map& operator= (initializer_list<value_type> il);
```

15.3.2 *set/multiset*

set/multiset跟map不同，它通过值来进行排序，所以不能对存储的数据进行随意修改。思考一下，set和map都只能通过迭代器来进行访问，若set能随意修改迭代器对应的值那这个迭代器就失效了，因为值改变了需要重新排序。正因如此，set的迭代器使用operator*返回的类型是const的，就是防止你对其进行修改。

常用成员函数除*operator[]*外和*map*系列一致。

15.3.3 *unordered_map/unordered_multimap*

*unordered_map*内部使用*bucket hash*实现的*hashmap*，属于开放地址法的一种，默认构造函数后桶个数初始化为*11*。开放地址法是所有的元素都存放在散列表里，发生地址冲突时，按照某种方法继续探测*Hash*表中其它存储单元，直到找到空位置为止 。除了这种开放地址法我们还有封闭地址的方法，也叫拉链法，即冲突后再节点后面用链表延伸冲突的*key*，也可以改成用二叉树。

其中的哈希函数又采用的是*Fowler–Noll–Vo*算法，属于非密码学哈希函数，目前有三种，分别是*FNV-1*，*FNV-1a*和*FNV-0*，但是*FNV-0*算法已经被丢弃了。*FNV*算法的哈希结果有*32*、*64*、*128*、*256*、*512*和*1024*位等长度。如果需要哈希结果长度不属于以上任意一种，也可以根据*Changing the FNV hash size - xor-folding*上面的指导进行变换得到。4字节的特例化代码如下：

```
///usr/include/c++/5.4.0/tr1/functional_hash.h
template<>
struct _Fnv_hash_base<4>
{
    template<typename _Tp>
    static size_t
    hash(const _Tp* __ptr, size_t __clength)
    {
        size_t __result = static_cast<size_t>(2166136261UL);
        const char* __cptr = reinterpret_cast<const char*>(__ptr);
        for (; __clength; --__clength){
            __result ^= static_cast<size_t>(*__cptr++);
            __result *= static_cast<size_t>(16777619UL);
        }
        return __result;
    }
};
```

另外密码学中常用的哈希算法还有*MD5*、*SHA1*、*SHA2*、*SHA256*、*SHA512*、*SHA3*、*RIPEMD160*。

常用成员函数除和map一样的之外还有：

```
void rehash( size_type n );//Sets the number of buckets in the container to n or more.
float load_factor() const noexcept;//load_factor = size / bucket_count
hasher hash_function() const;
size_type bucket_count() const noexcept;
```

15.3.4 *unordered_set/unordered_multiset*

常用成员函数除*operator[]*外和*unordered_map*系列一致。

15.4 迭代器

迭代器分为5种，由一般到特殊(“高级”)可以把它分成五类，如下表所示：

iterator分类	能力	由谁提供
Output iterator 输出迭代器	向前写	ostream, inserter
Input Input 输入迭代器	向前读, 每个元素只能读一次	istream
		`forward list`, unordered

iterator分类	能力	由谁提供
Forward iterator 前向迭代器	向前读	rs(无序容器)
Bidirectional iterator双向迭代器	可向前向后两个方向读取	list, set(multiset), map(multimap)
Random-access iterator 随机访问迭代器	随机读取	array, vector, deque, string, C风格数组

STL中的各种算法，包括`<algorithm>`中的以及各种容器的成员函数，还有各种功能函数，比如迭代器辅助函数(advance, next, prev, distance, iter_swap)等，有很多都是以迭代器作为输入参数的，这些函数中，形参类型越是“一般”，说明其使用范围越大。

比如std::sort快排只能接受RandomAccessIterator，那么std::sort就只能对array, vector, deque, string这几类容器进行排序：

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

这里所说的“一般”指的就是上面5类迭代器中“低级”的迭代器，比如Input iterator和Output iterator就比Forward iterator一般，Forward iterator比Bidirectional iterator一般，Bidirectional iterator又比Random-access iterator一般。

假设有两个算法，f和g，f接受Input iterator类型的参数，而g接受Random-access类型的参数，那么f的作用范围就比g大，因为所有的Forward, Bidirectional和Random-access迭代器都可以作为f的参数，而g只能使用Random-access参数。

15.4.1 Output iterator 输出迭代器

支持的操作	功能描述
*iter = value	把value写入迭代器iter所指向位置的元素
++iter	向前移动一个位置，返回新的位置
iter++	向前移动一个位置，返回旧的位置
TYPE(iter)	拷贝构造函数

15.4.2 Input iterator 输入迭代器

一个纯粹的Input iterator 类型的迭代器，只能挨个元素向前只读地访问元素. 典型示例是读取标准键盘输入的迭代器，每个元素只能读取一次，且只能向前, 只能读取，不能修改。

支持的操作	功能描述
*iter	读取元素
iter->member	访问iter出元素的成员
++iter	向前移动一个位置，返回新元素位置
iter++	向前移动一个位置, 不要求返回值。通常是返回旧元素位置
iter1 == iter2	判等
iter1 != iter2	判不等
TYPE(iter)	拷贝构造函数

15.4.3 Forward iterator 前向迭代器

Forward iterator 是一种特殊的Input iterator, 它在Input iterator的基础上提供了额外的保证：

它保证两个指向同一个元素的迭代器pos1, pos2， pos1 == pos2 返回true，并且对pos1, pos2调用自增操作符之后，二者仍然指向相同元素。

支持的操作	功能描述
*iter	读取元素
iter->member	访问iter出元素的成员
++iter	向前移动一个位置，返回新元素位置
iter++	向前移动一个位置, 不要求返回值。通常是返回旧元素位置
iter1 == iter2	判等
iter1 != iter2	判不等
TYPE()	使用默认构造函数创建一个iterator
TYPE(iter)	拷贝构造函数
iter1 = iter2	赋值

15.3.4 Bidirectional iterator 双向迭代器

Bidirectional iterator是提供了回头访问能力的Forward iterator, 在Forward iterator支持的操作基础上，它提供了以下两个“回头”操作：

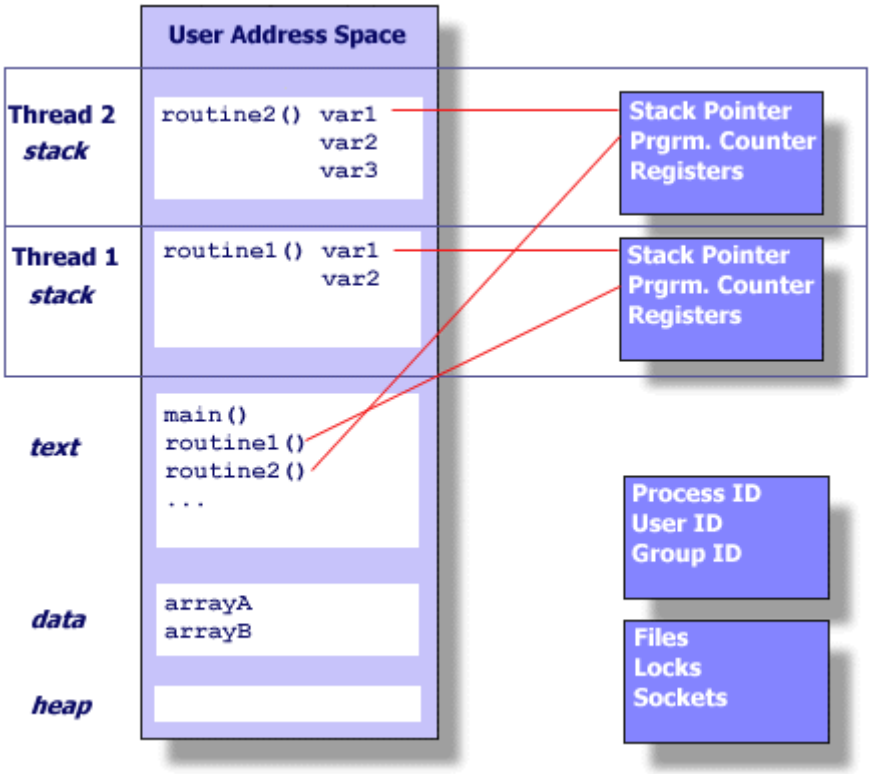
支持的操作	功能描述
–iter	回头走一步，返回新位置
iter–	回头走一步，返回旧位置

15.3.5 Random-access iterator 随机访问迭代器

Random-access iterator是功能最强大的迭代器类型，在Bidirectional iterator基础上提供了随机访问的功能，因此支持迭代器运算，类比指针运算。

支持的操作	功能描述
iter[n]	取第n个位置的元素
iter += n	移动n个位置, 向前后取决于n的符号
iter -= n	移动n个位置, 向前后取决于n的符号
iter + n	返回移动n个位置后的迭代器
iter - n	返回移动n个位置后的迭代器
iter1 - iter2	返回iter1和iter2间的距离
iter1 < iter2	iter1比iter2考前？
iter1 <= iter2	iter1不比iter2靠后？
iter1 > iter2	iter1比iter2靠后？
iter1 >= iter2	iter1不比iter2靠前？

16. C++11并发编程



*C++11*的*std::thread*是经过良好设计并且跨平台的线程表示方式，在类*Unix*平台上是对*pthread*进行的面向对象封装（增加了易用性，也损失了一些功能，所以*pthread*是*C++11*并发编程库的超集），比如*std::thread*的构造函数中调用的就是*pthread_create*来创建线程。如果在代码中使用了*std::thread*，还需要另外链接*libpthread.so*；而在*Windows*上有自己另外的封装。

我们首先来介绍一下*pthread*。

16.1 *pthread*

pthread是POSIX的线程标准，定义了创建和操纵线程的一套API。实现POSIX 线程标准的库常被称作 **Pthreads**，一般用于Unix-like POSIX 系统，如Linux、Solaris。Linux的pthread实现是*Native POSIX Thread Library* (*NPTL*)。在Linux2.6之前进程是内核调度的实体，在内核中并不能真正支持线程。但是它的确可以通过 `clone()` 系统调用将进程作为可调度的实体。这个调用创建了调用进程（calling process）的一个拷贝，这个拷贝与调用进程共享相同的地址空间。LinuxThreads 项目使用这个调用来完全在用户空间模拟对线程的支持。不幸的是，这种方法有一些缺点，尤其是在信号处理、调度和进程间同步原语方面都存在问题。另外，这个线程模型也不符合 POSIX 的要求。要改进 LinuxThreads，非常明显我们需要内核的支持，并且需要重写线程库。有两个相互竞争的项目开始来满足这些要求。一个包括 IBM 的开发人员的团队开展了 NGPT（Next-Generation POSIX Threads）项目。同时，Red Hat 的一些开发人员开展了 NPTL 项目。NGPT 在 2003 年中期被放弃了，把这个领域完全留给了 NPTL。

> reference:
>
> https://www.ibm.com/developerworks/cn/linux/l-threading.html
>
> https://computing.llnl.gov/tutorials/pthreads/

pthread定义了一套C语言的类型、函数与常量，它以`pthread.h`头文件和一个线程库实现。

pthread API中大致共有100个函数调用，全都以"pthread_"开头，并可以分为四类：

- 线程管理，例如创建线程，等待(join)线程，查询线程状态等。
- 互斥锁（Mutex）：创建、摧毁、锁定、解锁、设置属性等操作
- 条件变量（Condition Variable）：创建、摧毁、等待、通知、设置与查询属性等操作
- 使用了互斥锁的线程间的同步管理

POSIX的Semaphore API可以和pthread协同工作，但这并不是pthread的标准。因而这部分API是以"sem_"打头，而非"pthread_"。下面是一个简单用例：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>

static void wait(void)
{
    time_t start_time = time(NULL);
    while (time(NULL) == start_time){
        /* do nothing except chew CPU slices for up to one second */
    }
}

static void *thread_func(void *vptr_args)
{
    int i;
    for (i = 0; i < 20; i++){
        fputs("  b\n", stderr);
        wait();
    }
    return NULL;
}

int main(void)
{
    int i;
    pthread_t thread;

    if (pthread_create(&thread, NULL, thread_func, NULL) != 0){
```

```
        return EXIT_FAILURE;
    }
    for (i = 0; i < 20; i++){
        puts("a");
        wait();
    }
    if (pthread_join(thread, NULL) != 0){
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

16.2 *std::thread*

*std::thread*封装了*pthread*的线程管理接口，相对于*pthread*最方便的地方在于不需要将参数打包在一个*void**中进行参数传入，*std::thread*使用模板函数对多参数进行了打包从而让我们能将参数一个一个的传入。

> reference:https://www.zhihu.com/question/30553807

常用的成员函数有：

```
thread() noexcept;//默认构造函数并不代表有线程开始执行，non-joinable
template <class Fn, class... Args>
explicit thread (Fn&& fn, Args&&... args);//若参数为引用类型我们可以用std::ref进行转换
thread (const thread&) = delete;//禁止拷贝构造函数但是允许移动语义
thread (thread&& x) noexcept;
void detach();//不能被其他线程回收或杀死的，资源在终止时由系统自动释放
void join();//和detach一样执行之后都变成non-joinable，但是join为阻塞的
native_handle_type native_handle();//This member function is only present in
class thread if the library implementation supports it. 例如在linux就获得pthread_t
id get_id() const noexcept;
~thread();//如果析构时thread仍然是joinable的，则会调用std::terminate()抛出异常
```

前面说*std::thread*损失了一些功能，比如说设置线程的*cpu affinity*，这时候我们可以通过`native_handle()`获得*pthread*的句柄，然后通过*pthread*的接口来设置*cpu affinity*：

```
#include <thread>
#include <chrono>
#include <unistd.h>
#include <pthread.h>
using namespace std;

void Foo()
{
    auto start = std::chrono::high_resolution_clock::now();
    auto end = std::chrono::high_resolution_clock::now();
    float ms = 0.0;
    while(1){
        int sum = 0;
        start = std::chrono::high_resolution_clock::now();
        for(int i=0;i<100000;++i){
            sum += 1;
        }
        end = std::chrono::high_resolution_clock::now();
        ms = std::chrono::duration<float, std::milli>(end - start).count();
```

```
        usleep(ms * 1000);//让cpu占用率为50%
    }
}
int main()
{
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(2, &cpuset);//设置cpu2的亲和性
    thread t1(Foo);
    pthread_t nativeThread = t1.native_handle();
    pthread_setaffinity_np(nativeThread, sizeof(cpu_set_t), &cpuset);
    t1.join();
    return 0;
}
```

16.3 *std::mutex*

*std::mutex*互斥锁用来对临界区域加锁（可以理解为值为0或1的semaphore），和自旋锁(*spin lock*)的区别在于mutex是sleep-waiting。就是说当没有获得mutex时，会有上下文切换，将当前线程阻塞加到等待队列中，直到持有mutex的线程释放mutex并唤醒当前线程，这时CPU是空闲的，可以调度别的任务处理；而自旋锁是busy-waiting的，就是说当没有可用的锁时，就一直忙等待并不停的进行锁请求，直到得到这个锁为止，这个过程中CPU始终处于繁忙状态不能处理别的任务。

16.3.1 pthread_mutex_t加锁原理

linux平台的std::mutex是pthread_mutex_t封装，我们可以查看pthread_mutex_t源码来看std::mutex的实现，其结构体内容如下：

```
typedef union
{
    struct __pthread_mutex_s
    {
        int __lock;//mutex状态，0表示未占用，1表示占用
        unsigned int __count;//用于可重入锁，记录owner线程持有锁的次数
        int __owner;//owner线程ID
        unsigned int __nusers;
        /* KIND must stay at this position in the structure to maintain
       binary compatibility.  */
        int __kind;//记录mutex的类型
        int __spins;
        __pthread_list_t __list;
  } __data;
  ......
} pthread_mutex_t;
```

其中`__kind`有四种模式，分别为：

```
PTHREAD_MUTEX_TIMED_NP//这是缺省值，也就是普通锁。
PTHREAD_MUTEX_RECURSIVE_NP//可重入锁，允许同一个线程对同一个锁成功获得多次，并通过多次unlock解锁。
PTHREAD_MUTEX_ERRORCHECK_NP//检错锁，如果同一个线程重复请求同一个锁，则返回EDEADLK，否则与PTHREAD_MUTEX_TIMED_NP类型相同。
PTHREAD_MUTEX_ADAPTIVE_NP//自适应锁，自旋锁与普通锁的混合。
```

而C++11其实只实现了普通锁std::mutex和重入锁std::recursive_mutex，自旋锁需要我们自己实现，我们会在16.5章节中使用std::atomic来实现。

pthread中使用 pthread_mutex_lock接口来对4种锁进行加锁，我们可以看一下其中的操作：

```
if (__builtin_expect (type, PTHREAD_MUTEX_TIMED_NP) == PTHREAD_MUTEX_TIMED_NP)
{
    simple:
    /* Normal mutex.普通锁 */
    LLL_MUTEX_LOCK (mutex);//调用LLL_MUTEX_LOCK宏获得锁
    assert (mutex->__data.__owner == 0);
} else if (__builtin_expect (type == PTHREAD_MUTEX_RECURSIVE_NP, 1)) {
      /* Recursive mutex.当发现owner就是自身，只是简单的自增__count成员即返回。否则，调用LLL_MUTEX_LOCK宏获得锁，若能成功获得，设置__count = 1，否则挂起。*/

    /* Check whether we already hold the mutex.  */
    if (mutex->__data.__owner == id)
    {
        /* Just bump the counter.  */
        if (__builtin_expect (mutex->__data.__count + 1 == 0, 0))
        /* Overflow of the counter.  */
            return EAGAIN;
        ++mutex->__data.__count;
        return 0;
    }

    /* We have to get the mutex.  */
    LLL_MUTEX_LOCK (mutex);

    assert (mutex->__data.__owner == 0);
    mutex->__data.__count = 1;
} else if (__builtin_expect (type == PTHREAD_MUTEX_ADAPTIVE_NP, 1)) {
    /*spin lock，这种锁分两个阶段。第一阶段是自旋锁（spin lock），忙等待一段时间后，若还不能获得锁，则转变成普通锁。所谓“忙等待”，在x86处理器下是重复执行nop指令，nop是x86的小延迟函数：*/
    if (! __is_smp)
        goto simple;

    if (LLL_MUTEX_TRYLOCK (mutex) != 0)
    {
        int cnt = 0;
        int max_cnt = MIN (MAX_ADAPTIVE_COUNT, mutex->__data.__spins * 2 + 10);
        do
        {
            if (cnt++ >= max_cnt)
            {
                LLL_MUTEX_LOCK (mutex);
                break;
            }

#ifdef BUSY_WAIT_NOP
            BUSY_WAIT_NOP;//#define BUSY_WAIT_NOP   asm ("rep; nop")
#endif
        }
        while (LLL_MUTEX_TRYLOCK (mutex) != 0);

        mutex->__data.__spins += (cnt - mutex->__data.__spins) / 8;
```

```
    }
    assert (mutex->__data.__owner == 0);
} else {
    assert (type == PTHREAD_MUTEX_ERRORCHECK_NP);
      /* Check whether we already hold the mutex.它会侦测一个线程重复申请锁的情况，如遇到，报EDEADLK，从而避免这种最简单的死锁情形。若无死锁情形，goto simple语句会跳到普通锁的处理流程。*/
    if (__builtin_expect (mutex->__data.__owner == id, 0))
        return EDEADLK;
    goto simple;
}
```

通过上面的代码我们可以看到获取锁的核心代码是*LLL_MUTEX_LOCK*宏，该宏的实现为：

```
#define LLL_MUTEX_LOCK(mutex) \
    lll_lock ((mutex)->__data.__lock, PTHREAD_MUTEX_PSHARED
(mutex))//PTHREAD_MUTEX_PSHARED宏表示该锁是进程锁还是线程锁，0表示线程锁，128表示进程锁
```

通过该宏我们可以看到将mutex的__data.__lock字段传入了lll_lock函数中进行lock状态的修改，lll_lock的实现代码如下：

```
__lll_lock (int *futex, int private)
{
    int val = atomic_compare_and_exchange_val_24_acq (futex, 1, 0);
    if (__glibc_unlikely (val != 0))
    {
        if (__builtin_constant_p (private) && private == LLL_PRIVATE)
            __lll_lock_wait_private (futex);
        else
            __lll_lock_wait (futex, private);
    }
}
```

*atomic_compare_and_exchange_val_24_acq*和我们std::atomic的成员函数compare_exchange_strong的功能一样，若futex的值等于0，表示锁可以被当前线程占用，则将其置为1，val返回0；若futex值不等0，表示锁被其他线程占用，则futex不变，val返回1。后面判断若`val != 0`则调用`__lll_lock_wait`进行等待：

```
/*
futex有三种状态
0 锁空闲
1 没有waiter，解锁之后无需调用futex_wake
2 有waiter，那么解锁之后需要调用futex_wake
*/
void __lll_lock_wait (int *futex, int private)
{
    /* 非第一个线程会阻塞在这里 */
    if (*futex == 2)
        lll_futex_wait (futex, 2, private); /* Wait if *futex == 2.  */

    /* 第一个线程会阻塞在这里，atomic_exchange_acq返回当前futex值并将其赋为2*/
    while (atomic_exchange_acq (futex, 2) != 0)
        lll_futex_wait (futex, 2, private); /* Wait if *futex == 2.  */
}
```

最终在`__lll_lock_wait`中调用`lll_futex_wait`，`lll_futex_wait`是个宏，展开后为：

```
#define lll_futex_wait(futex, val) \
({ \
...
__asm __volatile (LLL_EBX_LOAD \
    LLL_ENTER_KERNEL \
    LLL_EBX_LOAD \
    : "=a" (__status) \
    : "0" (SYS_futex), LLL_EBX_REG (futex), "S" (0), \
    "c" (FUTEX_WAIT), "d" (_val), \
    "i" (offsetof (tcbhead_t, sysinfo)) \
    : "memory"); \
... \
})
```

可以看到当发生竞争的时候，会调用SYS_futex系统调用， 调用futex系统调用的futex_wait操作进行排队。因为用户空间并不知道内核的futex队列中是否还有其它锁竞争的任务在等待，所以系统调用阻塞唤醒回到用户空间，对futex尝试上锁，必须以锁竞争状态来上锁，以使自己解锁时，会调用futex_wake。 futex的优点在于只有当处于竞争状态的时候才会调用系统调用陷入内核。

16.3.2 常用函数

```
void lock();//如果当前mutx被其他线程锁定，则该接口会阻塞当前线程直至解锁；如果呗同一个线程锁定，则会造成死锁
native_handle_type native_handle();//和native_handle类似，在linux下会获得pthread_mutex_t
bool try_lock();//若锁被其他线程占用会返回false，若被自己占用会造成死锁
void unlock();//If the mutex is not currently locked by the calling thread, it causes undefined behavior.
```

16.3.3 相关类

16.3.3.1 lock_guard

这个接口我们在前文介绍过，通过RAII实现，生成对象时就加锁，在析构时进行解锁，所以锁的生命周期和对象的生命周期一样，使用方式像下面这样：

```
std::mutex mtx;
{
    std::lock_guard<std::mutex> lck (mtx);
}//此时生命周期为大括号
```

对象不能复制只能移动。

16.3.3.2 unique_lock

unique_lock和lock_guard类似，默认情况下锁的生命周期也是和对象一样，但是我们可以通过传入不同的参数进行灵活修改：

value	description
(no tag)	Lock on construction by calling member lock.
try_to_lock	Attempt to lock on construction by calling member try_lock.
defer_lock	Do not lock on construction (and assume it is not already locked by thread).
adopt_lock	Adopt current lock (assume it is already locked by thread).

所以unique_lock有如下构造函数：

```
unique_lock() noexcept;
explicit unique_lock (mutex_type& m);
unique_lock (mutex_type& m, try_to_lock_t tag);
unique_lock (mutex_type& m, defer_lock_t tag) noexcept;
unique_lock (mutex_type& m, adopt_lock_t tag);
template <class Rep, class Period>
unique_lock (mutex_type& m, const chrono::duration<Rep,Period>& rel_time);
template <class Clock, class Duration>
unique_lock (mutex_type& m, const chrono::time_point<Clock,Duration>& abs_time);
unique_lock (const unique_lock&) = delete;
unique_lock (unique_lock&& x);
```

可以看到我们还能设定加锁时间，另外我们也可以不在构造函数时设定这些属性，可以通过成员函数重新设定：

```
explicit operator bool() const noexcept;//true is the object owns a lock on the managed mutex object.
bool owns_lock() const noexcept;//true is the object owns a lock on the managed mutex object.
mutex_type* release() noexcept;//Returns a pointer to the managed mutex object, releasing ownership over it.
void lock();//Calling lock on a mutex object that has already been locked by other threads causes the current thread to block (wait) until it can own a lock to it.
bool try_lock();
template <class Rep, class Period>
bool try_lock_for (const chrono::duration<Rep,Period>& rel_time);//加锁一段时间，加锁成功返回true
template <class Clock, class Duration>
bool try_lock_until (const chrono::time_point<Clock,Duration>& abs_time);//一段时间后加锁，加锁成功返回true
```

16.3.4 recursive_mutex

16.3.5 shared_mutex

mutex可以分为递归锁(recursive mutex)和非递归锁(non-recursive mutex)。可递归锁也可称为可重入锁(reentrant mutex)，非递归锁又叫不可重入锁(non-reentrant mutex)。

二者唯一的区别是，同一个线程可以多次获取同一个递归锁，不会产生死锁。而如果一个线程多次获取同一个非递归锁，则会产生死锁。

16.4 *std::condition_variable*

条件变量用于阻塞当前线程直至有信号量通知，举个例子来说：

```
std::mutex mutex;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;
void Worker() {
    std::unique_lock<std::mutex> lock(mutex);
    cv.wait(lock, [] { return ready; });
    std::cout << "worker is processing data..." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(1));
    data += " done";
    processed = true;
    std::cout << "worker notify main thread" << std::endl;
    lock.unlock();
    cv.notify_one();
}
int main() {
    std::thread worker(Worker);
    {
        std::lock_guard<std::mutex> lock(mutex);
        std::cout << "main thread is preparing for data..." << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        data = "sample data";
        ready = true;
        std::cout << "main thread get ready for data" << std::endl;
    }
    cv.notify_one();
    {
        std::unique_lock<std::mutex> lock(mutex);
        cv.wait(lock, [] { return processed; });
    }
    std::cout << "back to main thread, data:" << data << std::endl;
    worker.join();
    return 0;
}
```

程序一开始先启动worker线程并获得mutex锁，之后调用std::condition_variable::wait，由于当前ready为false则释放mutex锁并调用wait阻塞当前线程等待被唤醒。与此同时主线程继续执行，使用lock_guard获得mutex锁，生成数据，将ready置成true，离开大括号作用域后lock_guard析构释放mutex锁并调用std::condition_variable::notify_one()通知wait的线程。此时主线程重新阻塞去竞争获得mutex锁。worker线程被唤醒后wait函数判断此时ready返回true不再调用wait阻塞当前线程，并获得mutex进行加锁，处理完数据之后将processd置true并对mutex进行解锁，最后通知主线程。此时主线程有可能被阻塞在竞争mutex的地方被唤醒，判断processed为true直接继续执行程序；有可能在wait被唤醒，重新判断processed返回true后继续运行，最终结束程序。

16.4.1 *Lost Wakeup and Spurious Wakeup*

- *Lost Wakeup*是指一个线程的*std::condition_variable::notify_one()*发生在了另外一个线程std::condition_variable::wait()之前，这样调用std::condition_variable::wait()的线程就不会被唤醒
- *Spurious Wakeup*是指一个调用std::condition_variable::wait()的线程被系统中断(EINTR)唤醒而不是被真正等待的线程唤醒，这是一种虚假的唤醒现象

为了防止这两种情况发生，根据CppCoreGuidelines CP.42: Don't `wait` without a condition，即调用wait函数时一定要加上一个Predicate函数，wait函数当Predicate返回false时才会真正调用wait函数，返回true时并不会去调用wait阻塞等待锁并加锁。

16.4.2 常用函数

```
void notify_all() noexcept;//还有非成员函数void notify_all_at_thread_exit
(condition_variable& cond, unique_lock<mutex> lck);
void notify_one() noexcept;//如果有多个线程都在等待，会随机唤醒一个线程
void wait (unique_lock<mutex>& lck);//该函数会阻塞当前线程直到被唤醒，被阻塞的同时当前线
程会主动调用lck.unlock()释放其管理的锁
template <class Predicate>
void wait (unique_lock<mutex>& lck, Predicate pred);//该函数和上一个重载的区别在于当
callable的pred返回false才会执行wait，返回true的时候不会加锁，该操作相当于while (!pred())
wait(lck);(which is specially useful to check against spurious wake-up calls)
template <class Clock, class Duration>
    cv_status wait_until (unique_lock<mutex>& lck,
                          const chrono::time_point<Clock,Duration>& abs_time);//当
等待时间到了会返回cv_status::timeout，否则为cv_status::no_timeout
template <class Clock, class Duration, class Predicate>
       bool wait_until (unique_lock<mutex>& lck,
                          const chrono::time_point<Clock,Duration>& abs_time,
                          Predicate pred);
```

16.5 *std::atomic*

std::atomic模板类，生成一个T类型的原子对象，并提供了系列原子操作函数。其中T是trivially copyable type满足：要么全部定义了拷贝/移动/赋值函数，要么全部没定义；没有虚成员；基类或其它任何非static成员都是trivally copyable。典型的内置类型bool、int等属于trivally copyable。再如class triviall{public: int x};也是。T能够被memcpy、memcmp函数使用，从而支持compare/exchange系列函数。有一条规则：不要在保护数据中通过用户自定义类型T通过参数指针或引用使得共享数据超出保护的作用域。atomic编译器通常会使用一个内部锁保护，而如果用户自定义类型T通过参数指针或引用可能产生死锁。总之限制T可以更利于原子指令。注意某些原子操作可能会失败，比如atomic、atomic在compare_exchange_strong()时和expected相等但是内置的值表示形式不同于expected，还是返回false，没有原子算术操作针对浮点数;同理一些用户自定义的类型T由于内存的不同表示形式导致memcmp失败，从而使得一些相等的值仍返回false。

16.5.1 原子操作原理

C++11新引入的std::atomic主要是通过硬件的cmpxchgl(CAS,compare and swap)指令实现，linux内核将cmpxchgl封装成函数cmpxchg，实现如下：

```
#define cmpxchg( ptr, _old, _new ) { \
    volatile uint32_t *__ptr = (volatile uint32_t *)(ptr);   \
    uint32_t __ret;                                          \
    asm volatile( "lock; cmpxchgl %2,%1"           \
    : "=a" (__ret), "+m" (*__ptr)                \
    : "r" (_new), "0" (_old)                     \
    : "memory");                  \
    );                                             \
    __ret;                                         \
}
```

作用为将ptr的保存值和_old进行比较，若相等则将_new存入ptr，否则返回ptr保存的值。可以看到cmpxchgl指令前加了lock前缀，lock保证了指令不会受其他处理器或cpu核的影响。在PentiumPro之前，lock的实现，是通过锁住bus（总线），从而阻止其他cpu核的内存访问。可想而知，这种实现是非常低效的。从PentiumPro开始，lock只会阻塞其他cpu核对相关内存的缓存块的访问。

std::mutex的加锁过程其实也是有cmpxchg参与，只不过当发生竞争的时候会陷入内核进行等待，这时候性能会比较低，所以能用原子操作的尽量使用原子操作。

16.5.2 ABA问题

CAS在执行过程中有可能会因为ABA问题导致结果错误，我们通过atomic实现一个stack来介绍什么是ABA问题：

```
template<typename _Ty>
struct LockFreeStackT
{
    struct Node
    {
        _Ty val;
        Node* next;
    };
    LockFreeStackT() : head_(nullptr) {}
    void push(const _Ty& val)
    {
        Node* node = new Node{ val, head_.load() };
        while (!head_.compare_exchange_strong(node->next, node));
    }
    void pop()
    {
        Node* node = head_.load();
        while (node && !head_.compare_exchange_strong(node, node->next);
        if (node) delete node;
    }
    std::atomic<Node*> head_;
};
```

整个逻辑很简单，如果新元素的next和栈顶一样，证明在你之前没人操作它，使用新元素替换栈顶退出即可；如果不一样，证明在你之前已经有人操作它，head_在新建node之后被其他线程改动，而node->next仍然指向之前的head_，此时栈顶已发生改变，该函数会自动更新新元素的next值为改变后的栈顶；然后继续循环检测直到状态1成立退出。

假设现有两条线程，栈为A->B，此时线程1对栈进行pop操作，在CAS之前CPU切换去处理线程2。线程2此时连pop两次，将A和B都pop出来，又进行push操作，由于操作系统很可能会分配刚刚释放的内存，所以重新new的数据可能就是刚刚释放地址。此时CPU切到线程1，线程1进行CAS判断此时的head仍然是A，所以将A pop出来将B这个已经释放的内存设为栈顶。解决ABA问题的办法无非就是通过打标签的方式给每个节点进行打标签，而不是通过地址进行判断。

16.5.3 常用函数

```
bool is_lock_free() const volatile;//判断atomic<T>中的T对象是否为lock free的，若是返回true。lock free(锁无关)指多个线程并发访问T不会出现data race，任何线程在任何时刻都可以不受限制的访问T
bool is_lock_free() const;
atomic() = default;//默认构造函数，T未初始化，可能后面被atomic_init(atomic<T>* obj,T val )函数初始化
constexpr atomic(T val);//T由val初始化
```

```
atomic(const atomic &) = delete;//禁止拷贝
atomic & operator=(const atomic &) = delete;//atomic对象间的相互赋值被禁止，但是可以显示转换再赋值，如atomic<int> a=static_cast<int>(b)这里假设atomic<int> b
atomic & operator=(const atomic &) volatile = delete;//atomic间不能赋值
T operator=(T val) volatile;//可以通过T类型对atomic赋值，如：atomic<int> a;a=10;
T operator=(T val);
operator T() const volatile;//读取被封装的T类型值，是个类型转换操作，默认内存序是memory_order_seq需要其它内存序则调用load
operator T() const;//如：atomic<int> a,a==0或者cout<<a<<endl都使用了类型转换函数
//以下函数可以指定内存序memory_order
T exchange(T val, memory_order = memory_order_seq_cst) volatile;//将T的值置为val，并返回原来T的值
T exchange(T val, memory_order = memory_order_seq_cst);
void store(T val, memory_order = memory_order_seq_cst) volatile;//将T值设为val
void store(T val, memory_order = memory_order_seq_cst);
T load(memory_order = memory_order_seq_cst) const volatile;//访问T值
T load(memory_order = memory_order_seq_cst) const;
bool compare_exchange_weak(T& expected, T val, memory_order = memory_order_seq_cst) volatile;//该函数直接比较原子对象所封装的值与参数expected的物理内容，所以某些情况下，对象的比较操作在使用 operator==()判断时相等，但compare_exchange_weak判断时却可能失败，因为对象底层的物理内容中可能存在位对齐或其他逻辑表示相同但是物理表示不同的值(比如true和2或3，它们在逻辑上都表示"真"，但在物理上两者的表示并不相同)。可以虚假的返回false(和expected相同)。若本atomic的T值和expected相同则用val值替换本atomic的T值，返回true;若不同则用本atomic的T值替换expected，返回false。
bool compare_exchange_weak(T &, T, memory_order = memory_order_seq_cst);
bool compare_exchange_strong(T &, T, memory_order = memory_order_seq_cst) volatile;//与compare_exchange_weak不同,strong版本的compare-and-exchange操作不允许返回 false，即原子对象所封装的值与参数expected的物理内容相同，比较操作一定会为true。不过在某些平台下，如果算法本身需要循环操作来做检查，compare_exchange_weak的性能会更好。因此对于某些不需要采用循环操作的算法而言,通常采用compare_exchange_strong更好
bool compare_exchange_strong(T &, T, memory_order = memory_order_seq_cst);
```

16.5.4 自旋锁实现

通过原子变量，我们可以自行实现标准库中没有的自旋锁：

```
class spin_mutex {
    std::atomic<bool> flag = ATOMIC_VAR_INIT(false);
public:
    spin_mutex() = default;
    spin_mutex(const spin_mutex&) = delete;
    spin_mutex& operator= (const spin_mutex&) = delete;
    void lock() {
        bool expected = false;
        while(!flag.compare_exchange_strong(expected, true))
        expected = false;
    }
    void unlock() {
        flag.store(false);
    }
};
```

从网上的性能测试来看所有平台的自旋锁性能都无限接近无锁实现，并且使用方式和互斥锁几乎没有差别，但是仍然看场景，场景我们在16.3讨论过。

16.6 *std::async*

*std::async()*是一个接受回调(函数或函数对象)作为参数的函数模板，并有可能异步执行它们。*std::async*返回一个*std::future*，它存储由*std::async()*执行的函数对象返回的值。函数的参数可以作为函数指针参数后面的参数传递给*std::async()*。

std::async中的第一个参数是启动策略，它控制std::async的异步行为，我们可以用三种不同的启动策略来创建std::async：
std::launch::async：保证异步行为，即传递函数将在单独的线程中执行
std::launch::deferred：当其他线程的future调用get()或者wait()才执行该函数
std::launch::async | std::launch::deferred：默认行为。有了这个启动策略，它可以异步运行或不运行，这取决于系统的负载，但我们无法控制它。

16.7 *std::promise*

promise对象可以通过set_value保存某一类型 T 的值，该值可被 future 对象通过get阻塞读取（可能在另外一个线程中），因此 promise 也提供了一种线程同步的手段。在 promise 对象构造时可以和一个共享状态（通常是std::future，通过get_future）相关联，并可以在相关联的共享状态(std::future)上保存一个类型为 T 的值。

可以通过 get_future 来获取与该 promise 对象相关联的 future 对象，调用该函数之后，两个对象共享相同的共享状态(shared state)

- promise 对象是异步 Provider，它可以在某一时刻设置共享状态的值。
- future 对象可以异步返回共享状态的值，或者在必要的情况下阻塞调用者并等待共享状态标志变为 ready，然后才能获取共享状态的值。

举个例子来说明promise是如何使用的：

```
#include <iostream>       // std::cout
#include <functional>     // std::ref
#include <thread>         // std::thread
#include <future>         // std::promise, std::future

void print_int(std::future<int>& fut) {
    int x = fut.get(); // 获取共享状态的值.
    std::cout << "value: " << x << '\n'; // 打印 value: 10.
}

int main ()
{
    std::promise<int> prom; // 生成一个 std::promise<int> 对象.
    std::future<int> fut = prom.get_future(); // 和 future 关联.
    std::thread t(print_int, std::ref(fut)); // 将 future 交给另外一个线程t.
    prom.set_value(10); // 设置共享状态的值, 此处和线程t保持同步.
    t.join();
    return 0;
}
```

future在主线程中和promise绑定，然后将future传给副线程，future在副线程中调用get阻塞等待共享变量ready。当主线程中的promise调用set_value后共享变量ready，副线程中的future唤醒线程并获得主线程set_value的值。

常用函数有：

```
future<T> get_future();//返回一个和promise绑定的future对象
promise& operator= (promise&& rhs) noexcept;
promise& operator= (const promise&) = delete;//禁用复制，允许移动构造
void set_exception (exception_ptr p);//设置异常，当future调用get的时候抛出异常
void set_exception_at_thread_exit (exception_ptr p);//线程结束时设置future同步的值
void set_value (const T& val);
void set_value (T&& val);
void promise<R&>::set_value (R& val);   // when T is a reference type (R&)
void promise<void>::set_value (void);   // when T is void
void set_value_at_thread_exit (const T& val);
void set_value_at_thread_exit (T&& val);
void promise<R&>::set_value_at_thread_exit (R& val);// when T is a reference
type (R&)
void promise<void>::set_value_at_thread_exit (void);// when T is void
```

16.8 *std::future*

std::future 可以用来获取异步任务的结果，因此可以把它当成一种简单的线程间同步的手段。std::future 通常由某个 Provider 创建，你可以把 Provider 想象成一个异步任务的提供者，Provider 在某个线程中设置共享状态的值，与该shared state相关联的 std::future 对象调用 get（通常在另外一个线程中）获取该值，如果共享状态的标志不为 ready，则调用 std::future::get 会阻塞当前的调用者，直到 Provider 设置了共享状态的值（此时共享状态的标志变为 ready），std::future::get 返回异步任务的值或异常（如果发生了异常）。

一个有效(valid)的 std::future 对象通常由以下三种 Provider 创建，并和某个共享状态相关联。Provider 可以是函数或者类，其实我们前面都已经提到了，他们分别是：

- std::async构造函数返回一个future。
- std::promise::get_future，get_future 为 promise 类的成员函数。
- std::packaged_task::get_future，此时 get_future为 packaged_task 的成员函数。

std::shared_future与std::future 类似，但是std::shared_future 可以拷贝、多个std::shared_future可以共享某个共享状态的最终结果(即共享状态的某个值或者异常)。shared_future可以通过某个std::future对象隐式转换（参见std::shared_future的构造函数），或者通过std::future::share()显示转换，无论哪种转换，被转换的那个 std::future对象都会变为not-valid。一个有效的std::future对象只能通过std::async()，std::future::get_future或者std::packaged_task::get_future来初始化，可通过valid()函数来判断一个future对象是否valid。具体例子可以看16.7。

常用函数有：

```
T get();//Returns the value stored in the shared state (or throws its exception)
when the shared state is ready.
future& operator= (future&& rhs) noexcept;
future& operator= (const future&) = delete;
shared_future<T> share();//Returns a shared_future object that acquires the
shared state of the future object.执行后原来的future变成valid
bool valid() const noexcept;//默认构造函数的future和调用过get的future都返回false，除非
被其他valid的future进行移动赋值
void wait() const;//等待直到shared state变成ready
template <class Rep, class Period>
future_status wait_for (const chrono::duration<Rep,Period>& rel_time) const;//等
待shared state变成ready或者时间到。若时间到了shared state还未ready则返回
future_status::timeout；若ready返回future_status::ready；若future对象由async构造函数
返回，并且async包含的是一个std::launch::deferred同步执行函数，则返回
future_status::deferred
template <class Clock, class Duration>
future_status wait_until (const chrono::time_point<Clock,Duration>& abs_time)
const;//等待直到一个特定时间点，若等待时间点在当前之前则返回future_status::timeout
```

17. 并行编程

17.1 指令集并行

CPU流水线相关

17.2 数据级并行

17.2.1 *SIMD(Single instruction multidata)*

SIMD即单指令流多数据流，是一种采用一个控制器来控制多个处理器，同时对一组数据（又称“数据向量”）中的每一个分别执行相同的操作从而实现空间上的并行性的技术。简单来说就是一个指令能够同时处理多个数据。

1996年Intel推出了X86的MMX(MultiMedia eXtension)指令集扩展，MMX定义了8个寄存器，称为MM0到MM7，以及对这些寄存器进行操作的指令。每个寄存器为64位宽，可用于以“压缩”格式保存64位整数或多个较小整数，然后可以将单个指令一次应用于两个32位整数，四个16位整数或8个8位整数。

intel在1999年又推出了全面覆盖MMX的SSE(*Streaming SIMD Extensions*， 流式SIMD扩展)指令集，并将其应用到Pentium III系列处理器上，SSE添加了八个新的128位寄存器(XMM0至XMM7)，而后来的X86-64扩展又在原来的基础上添加了8个寄存器(XMM8至XMM15)。SSE支持单个寄存器存储4个32位单精度浮点数，之后的SSE2则支持单个寄存器存储2个64位双精度浮点数，2个64位整数或4个32位整数或8个16位短整形。SSE2之后还有SSE3，SSE4以及AVX，AVX2等扩展指令集。

AVX引入了16个256位寄存器(YMM0至YMM15)，AVX的256位寄存器和SSE的128位寄存器存在着相互重叠的关系(XMM寄存器为YMM寄存器的低位)，所以最好不要混用AVX与SSE指令集，否在会导致transition penalty（过渡处罚）。

17.2.2 *SIMT(Single instruction multithread)*

首先厘清概念：

- SIMD：单指令多数据，首先获取多个数据，同时使用一条指令处理
- SMT：同时多线程，不同线程之间的指令可以并行执行

- SIMT：二者折中方案，单指令多线程，线程内部执行相同指令，但比SIMD更灵活，比SMT效率更高

其次，对比SIMT与SIMD，上文说到，SIMT比SIMD更灵活，其主要体现在以下三点

1. 单指令，可以访问多个寄存器组。
2. 单指令，多种寻址方式。
3. 单指令，多种执行路径

每组线程中，如果出现分支指令，则不同线程之间串行执行，直到分支指令执行完毕，每组线程继续并行执行相同指令，下文会提供一种分支指令预测机制。

最后，对比SIMT与SMT，上文说到，SIMT比SMT效率更高，主要体现在SIMT可同时运行的线程更多、寄存器更多这两点：

1. 足够多的线程，可以获得足够高的吞吐率
2. 一方面延迟是竭力避免的，另一方面寄存器的价格是可以接受的。

17.3 线程级并行

17.3.1 *SMT(Simultaneous multithreading)*

我们前面介绍过pthread，一个多线程库，这是软件层面的概念。如果多个线程想运行在同一个core上我们只能让任务分时服用，而硬件上如果一个CPU支持SMT，就是在能让多个线程共用一个CPU，但是分别用CPU上的不同的资源。CPU在执行一条机器指令时，并不会完全地利用所有的CPU资源，而且实际上，是有大量资源被闲置着的。超线程技术允许两个线程同时不冲突地使用CPU中的资源。比如一条整数运算指令只会用到整数运算单元，此时浮点运算单元就空闲了，若使用了超线程技术，且另一个线程刚好此时要执行一个浮点运算指令，CPU就允许属于两个不同线程的整数运算指令和浮点运算指令同时执行，这是真的并行。超线程的原理主要是两个逻辑核心各自有一套自己的线程状态存储设施：控制寄存器，通用寄存器。从而调度器可以同时调度两个线程，这个是关键。最终这么做的目的是充分利用执行引擎。

17.3.2 OpenMP

17.4 进程级并行

17.4.1 MPI

MPI是一个跨语言的通讯协议，用于编写进程级并行程序，包括协议和和语义说明，他们指明其如何在各种实现中发挥其特性。MPI的目标是高性能，大规模性，和可移植性。一般用于HPC等大型计算集群场景。

MPI有很多的实现，包括OpenMPI/IntelMPI/MPICH2/MVAPICH等。Nvidia的NCCL其实也算是MPI接口的一种实现，当前NCCL支持多GPU和多节点的通信，兼容MPI接口。

以OpenMPI为例，节点之间的通信除了基本的TCP协议之外还支持RoCE和Infiniband等协议，能尽量减少节点之间通信的时延。

除了用于HPC之外在当前深度学习训练场景下也常常用到MPI。Inspur公司就基于caffe实现了MPI版本来提升集群训练的性能。当然在当前NCCL已经兼容了MPI接口并且还支持GPUDirect的情况下也就不需要再使用其他版本的MPI了。除了caffe之外在tensorflow中做集群通信的主要是使用的自家的gRPC（当然GPU之间通信肯定是使用的NCCL），gRPC当然也能基于RDMA，但是由于gRPC本身层级过高，基于RDMA的性能并不如MPI，所以科大目前有团队把tensorflow改成了MPI版，但并不是主流。

18. *CPU*

CPU执行计算任务时都需要遵从一定的规范，程序在被执行前都需要先翻译为CPU可以理解的语言。这种规范或语言就是指令集（ISA，Instruction Set Architecture）。程序被按照某种指令集的规范翻译为CPU可识别的底层代码的过程叫做编译（compile）。x86、ARM v8、MIPS都是指令集的代号。指令集可以被扩展，如x86增加64位支持就有了x86-64。厂商开发兼容某种指令集的CPU需要指令集专利持有者授权，典型例子如Intel授权AMD，使后者可以开发兼容x86指令集的CPU。

核心的实现方式被称为微架构（microarchitecture）。微架构的设计影响核心可以达到的最高频率、核心在一定频率下能执行的运算量、一定工艺水平下核心的能耗水平等等。此外，不同微架构执行各类程序的偏向也不同，例如90年代末期Intel的P6微架构就在浮点类程序上表现优异，但在整数类应用中不如同频下的对手。

常见的代号如Haswell、Cortex-A15等都是微架构的称号。注意微架构与指令集是两个概念：指令集是CPU选择的语言，而微架构是具体的实现。i7-4770的核心是Haswell微架构，这种微架构兼容x86指令集。对于兼容ARM指令集的芯片来说这两个概念尤其容易混淆：ARM公司将自己研发的指令集叫做ARM指令集，同时它还研发具体的微架构如Cortex系列并对外授权。但是，一款CPU使用了ARM指令集不等于它就使用了ARM研发的微架构。Intel、高通、苹果、Nvidia等厂商都自行开发了兼容ARM指令集的微架构，同时还有许多厂商使用ARM开发的微架构来制造CPU。通常，业界认为只有具备独立的微架构研发能力的企业才算具备了CPU研发能力，而是否使用自行研发的指令集无关紧要。