

VECTOR

- `vector` : variable array 长度可变的数组
- `deque` : dual-end queue 双端队列
- `list` : double-linked-list 双向链表
- `forward_list` : as it
- `array` : as "array" 在C++中是一个关键字，是stl中的一个类
- `string` : char. array
- Constructors

`vector<Elem> c;`

`vector<Elem> c1(c2);` 把容器c2给到c1建立新容器

- Simple Methods

`V.size() // num items`

`V.empty() // empty?`

`==, !=, <, >, <=, >=`

`V.swap(v2) // swap`

- Iterators

`I.begin() // first position`

`I.end() // last position`

- Element access

```
V.at(index)
```

两者都可取指定index的值

```
V[index]
```

```
V.front() // first item
```

给出头尾的元素，和begin end不一样

```
V.back() // last item
```

- Add/Remove/Find

```
V.push_back(e)
```

```
V.pop_back()
```

```
v.insert(pos, e)
```

pos需要是itrator

```
V.erase(pos) → 空着
```

```
V.clear()
```

```
V.find(first, last, item)
```

- V.at(index)

- 该方法会进行边界检查，如果越界，编译器会抛出异常，更加安全

- V[index]

- 注意：不能用这种方法修改元素！

- 该方法不会做边界检查，如果越界的话，则行为不可预测，是未定义的行为（Undefined Behaviour），因此速度快，但不安全

`insert(index, count, value)` 表示在 index 位置插入 count 个 value

List

- Ability to assign items to a list, remove items

```
x.push_back(item)
```

```
x.push_front(item)
```

```
x.pop_back()
```

```
x.pop_front()
```

```
x.erase(pos1, pos2)
```

但 `std::list` 的迭代器是 双向迭代器 (BidirectionalIterator) , 它只支持:

- `++it` 和 `--it`
- `==` 和 `!=`

它 不支持 :

- `it1 < it2`
- `it1 > it2`
- `it + n` , `it - n`
- `it[n]`



`s.insert(p, t);` List的insert是插入在传入的位置前一个

- Use `vector` unless you have other reasons
- Don't use `list` or `forward_list` if your program has lots of small elements and space overhead matters
- Use `vector` or `deque` if the program requires random access to elements
- Use `list` or `forward_list` if the program needs to insert elements in the middle of the container
- Use `deque` if the program needs to insert elements at the front and the back, but not in the middle

3. `std::deque` --- 双端队列 (`<deque>`)

✓ 特点：

- 支持高效的头部和尾部插入/删除 (均为 $O(1)$)。
- 支持随机访问 (可以通过下标访问元素)。
- 内部采用分段连续存储，避免了单块连续内存的限制。
- 比 `vector` 更适合频繁在首部插入的情况。

⚠ 缺点：

- 中间插入/删除效率不如链表 (需要移动大量元素)。
- 容量增长策略复杂，某些情况下不如 `vector` 简洁高效。

✓ 使用场景：

- 需要频繁在头部或尾部插入/删除元素。
- 同时又需要快速访问任意位置。
- 如：滑动窗口、缓存、双端队列等。

MAP

```

map<long,int> root;
root[4] = 2;
root[1000000] = 1000;
long l;
cin >> l;      判断key是否存在
if (root.count(l))
    cout<<root[l]
else cout<<"Not perfect square";

```

Typdef

- Annoying to type long names
 - map<string, list<string>> phonebook;
 - map<string, list<string>>::iterator finger;
- Simplify with typedef
 - typedef map<string, list<string>> PB;
 - PB phonebook;
 - PB::iterator finger;
- Or use the using statement:
 - using PB = map<string, list<string>>;

statement is. 双引号搜索范围更大

- #include "xx.h" : search in the current directory firstly, then the directories declared somewhere
- #include <xx.h> : search in the specified directories
- #include <xx> : same as #include <xx.h>

```

#ifndef HEADER_FLAG
#define HEADER_FLAG
    // Type declaration here...
#endif // HEADER_FLAG

```

Storage Allocation vs Initialization

存储分配vs构造函数调用

- The compiler allocates all the storage for a scope at the opening brace of that scope.
- The constructor call doesn't happen until the sequence point where the object is defined.

存储分配	编译器在作用域开头为所有变量分配存储空间，这是静态的。
构造函数调用	构造函数调用发生在对象定义的序列点，这是动态的。

Friend 授权访问私有变量

- is a way to explicitly grant access to a function that isn't a member of the structure
- The class itself controls which code has access to its members.
- Can declare a global function as a friend, as well as a member function of another class, or even an entire class, as a friend.

默认
class defaults to private

struct defaults to public

5. Static Member Functions (静态成员函数)

- 定义：在类中使用 `static` 修饰的成员函数。
- 含义：共享于所有实例，并且只能访问静态成员（静态成员变量或静态成员函数）。
- 特点：
 - 静态成员函数没有隐式的 `this` 指针，因此不能直接访问非静态成员。
 - 示例：

```
cpp
1 v class MyClass {
2   public:
3     static int getCount() {
4       return count; // 只能访问静态成员变量
5     }
6
7   MyClass() {
8     count++;
9   }
10
11 private:
12   static int count; // 静态成员变量
13 };
```

- And a function with a static X object

```
void f() {           静态对象只有当第一次被调用的时候被构造 且其生命周期也持续到整个程序结束
    static X my_X(10, 20);
    ...
}
```

```
int MyClass::count = 0; // 不需要再次使用 `static`
```

- References

- can't be null
- are dependent on an existing variable, they are an alias for an variable
- can't change to a new "address" location

- Pointers

- can be set to null
- pointer is independent of existing objects
- can change to point to a different address

- No pointers to references

```
int &*p;           // illegal
```

- Reference to pointer is ok

```
void f(int *&p);
```

```
int x=20; // left-value
int&& rx = x * 2;
// the result of x*2 is a right-value, rx extends its lifttime
int y = rx + 2; // In this way it can be reused:42
rx = 100;
// Once a right-value reference is initialized,
// this variable becomes a left-value that can be assigned
int&& rrx1 = x;
// Illegal: right-value reference can not be initialed by a left-value
const int&& rrx2 = x;
// Illegal: right-value reference can not be initialed by a left-value
```

```
cpp
```

```
1 int x;
2 cin >> x;
3 const int size = x;
4 double classAverage[size]; // error!
```

- `x` 是一个普通的整型变量，从标准输入读取值。
- `size` 被声明为 `const int`，但它的值是从运行时输入的 `x` 得到的。
- 在 C++ 中，数组的大小必须是一个编译时常量（即在编译时已知的值）。
- `size` 是一个运行时计算的值（依赖于用户输入），因此它不是一个编译时常量。
- 因此，使用 `size` 来定义数组 `classAverage` 是非法的，会导致编译错误。

```
char * const q = "abc"; // q is const
*q = 'c'; // OK
q++; // ERROR
const char *p = "ABCD"; // (*p) is a const char
*p = 'b'; // ERROR! (*p) is the const
```

	<code>int i;</code>	<code>const int ci = 3;</code>
<code>int *ip;</code>	<code>ip = &i;</code>	<code>ip = &ci; // ERROR</code>
<code>const int *cip</code>	<code>cip = &i;</code>	<code>cip = &ci;</code>

```
1 v class HasArray {
2     const int size;
3     int array[size]; // ERROR!
4 };
```

习题分析：

- `size` 是一个成员变量，即使它是 `const` 类型，它仍然是一个普通成员变量。
- 在类定义时，`size` 的值是未知的，因为它需要在对象实例化时通过构造函数初始化。
- 数组大小必须是一个编译时常量（即在编译时已知的值），而 `size` 是一个运行时确定的值（依赖于对象的构造过程）。
- 因此，`int array[size]` 导致编译错误，因为数组大小不是编译时常量。

```
T && a = ReturnRvalue();
```

- `ReturnRvalue` 函数返回的右值在表达式语句结束后，其生命也就终结了（通常我们也称其具有表达式生命期），而通过右值引用的声明，该右值又“重获新生”，其生命期将与右值引用类型变量 `a` 的生命期一样。只要 `a` 还“活着”，该右值临时量将会一直“存活”下去
- 所以相比于以下语句的声明方式：

```
T b = ReturnRvalue();
```

- 右值引用变量声明就会少一次对象的析构及一次对象的构造。因为 `a` 是右值引用，直接绑定了 `ReturnRvalue()` 返回的临时量，而 `b` 只是由临时值构造而成的，而临时量在表达式结束后会析构因应就会多一次析构和构造的开销

- the compiler tries to avoid creating storage for a const -- holds the value in its symbol table.

赋初值了，
直接嵌入

- `extern forces storage to be allocated.` 分配空间，表示它仍然是个变量，因此全局常量仍然是个变量
(可能会被外部引用)

类似的，如果 `const` 成员变量定义时没有赋初值，要靠后面传参数赋值，则也要分配空间，也是个变量，而不是在编译时直接作为一个常数/字符串来处理

<code>void f() const;</code>	如果对象是 <code>const</code> 的，会调用 <code>const</code> 的 <code>f()</code> ；
<code>void f();</code>	如果对象不是 <code>const</code> ，则默认调用非 <code>const</code> 的 <code>f()</code>

```
int harpo(int n, int m = 4, int j = 5);
int chico(int n, int m = 6, int j); // illegal
int groucho(int k = 1, int m = 2, int n = 3);
```

默认值必须从右往左写，即出现一个默认值后后面必须都是默认值

内联函数，编译器会直接把函数当成代码放到调用的地方，没有进出栈的操作 但占用内存会增加

Inline function

- An inline function is to be expanded in place, like a preprocessor macro, so the overhead of the function call is eliminated.

```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
inline int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

- 为什么需要放在头文件中？

- `inline` 函数的函数体必须在所有调用它的源文件中可见，因为编译器需要在编译时将函数体直接嵌入到调用点。
- 如果将 `inline` 函数的定义放在 `.cpp` 文件中，其他源文件在编译时无法看到函数体，会导致链接错误（例如未定义符号）。
- 因此，通常将 `inline` 函数的定义放在头文件中，通过 `#include` 让所有需要调用该函数的源文件都能看到其函数体。

题目列出了 `inline` 变量的三个特性：

1. Single Definition Across Translation Units :

- 编译器确保即使变量在多个文件中包含，也只有一个实例存在。

2. Initialization in Header Files :

- 你可以在头文件中声明和初始化变量，使代码更加简洁。

3. Global or Static Variables :

- 适用于全局常量、类的静态成员或单例模式。

```
1 // header.h
2 inline int count = 0; // inline 变量
3
4 // source1.cpp
5 #include "header.h"
6 void increment() {
7     count++;
8 }
9
10 // source2.cpp
11 #include "header.h"
12 void printCount() {
13     std::cout << "Count: " << count << std::endl;
14 }
```

- `count` 是一个 `inline` 变量，即使在多个源文件中包含 `header.h`，编译器也会确保每个翻译单元中的 `count` 是独立的。
- 这种设计避免了链接冲突，同时保持了代码的简洁性。

1. 全局常量：

```
cpp
1 // header.h
2 inline const int MAX_SIZE = 100; // 全局常量
```

- `MAX_SIZE` 是一个全局常量，可以在多个源文件中使用，且只有一个实例。

2. 类的静态成员：

```
cpp
1 // header.h
2 class MyClass {
3 public:
4     static inline int instance_count = 0; // 静态成员
5 };
```

- `instance_count` 是一个静态成员，通过 `inline` 确保在多个翻译单元中只有一个实例。

大对象或消耗大量内存的变量	避免使用 <code>inline</code> ，因为会导致每个翻译单元都有一个副本，造成不必要的内存浪费。
需要唯一实例的变量	不适合使用 <code>inline</code> ，因为 <code>inline</code> 变量在多个翻译单元中共享同一个实例。
可修改的变量	对于运行时可修改的变量，使用 <code>inline</code> 要谨慎，因为所有翻译单元共享同一个实例，可能导致意外的副作用。

Static (静态) 一词关键体现在两个方面：永久的存储和受限的访问范围

- 静态全局变量只能在定义它的编译单元（即一个 .cpp 文件）中使用，不能在其他编译单元中使用

Namespaces are open

- Multiple namespace declarations add to the same namespace.
 - Namespace can be distributed across multiple files.

```
//header1.h
namespace X {
    void f();
}
```

```
// header2.h
namespace X {
    void g(); // X now has f() and g();
}
```

- (private) member variables 子类可以得到父类的私有变量，但无法直接访问
父类中成员变量的初始化也在初始化列表中完成
- public member functions
- private member functions
- protected members 父类中 `protected` 的成员变量子类可以直接访问
，如父类A中有一个 `protected` 的成员变量i，则在子类B中可以直接用变量i
- static members

`protected` 的成员变量只能传递给下一个子类，如果是 `public` 继承，则在子类中也是 `protected`

Protected Members

- They are fully accessible in the derived class

More on constructors

- Base class is always constructed first
- If no explicit arguments are passed to base class
 - Default constructor will be called
- Destructors are called in exactly the reverse order of the constructors.

What is not inherited?

- Constructors
 - synthesized constructors use memberwise initialization
 - In explicit copy ctor, explicitly call base-class copy ctor or the default ctor will be called instead.
- Destructors
- Assignment operation 赋值操作
 - synthesized operator= uses memberwise assignment
 - explicit operator= be sure to explicitly call the base class version of operator=
- Private data is hidden, but still present

1. Constructors (构造函数)

- 关键点：构造函数不会被继承。
- 原因：
 - 构造函数的作用是初始化对象，而每个类的构造过程是独立的。
 - 派生类需要显式调用基类的构造函数，而不是直接继承基类的构造函数。
 - 如果派生类没有显式调用基类的构造函数，编译器会自动调用基类的默认构造函数（如果存在）。

赋值操作符的作用是实现对象的复制行为，而每个类的赋值逻辑是独立的。

派生类需要显式调用基类的赋值操作符，而不是直接继承基类的赋值操作符。

如果派生类没有显式定义赋值操作符，编译器会生成一个默认的赋值操作符，但这个操作符只会对成员变量逐个赋值，不会调用基类的赋值操作符。

```
cpp
1 v class Base {
2 public:
3     Base& operator=(const Base& other) { /* ... */ }
4 };
5
6 v class Derived : public Base {
7 public:
8     Derived& operator=(const Derived& other) {
9         Base::operator=(other); // 显式调用基类的赋值操作符
10        return *this;
11    }
12};
```

Base class member access specifier

表头一行指A中原来是public/protected/private的member

Inheritance Type (B is)	public	protected	private
public A 真正oop意义上的继承	public in B	protected in B	hidden
private A	private in B	private in B	hidden
protected A	protected in B	protected in B	hidden

public继承

private继承--父类public的东西
(变量、函数) 变为private,
最后的结果类似composition

基类中原来private的成员在派生类中会不可见，无法直接访问

cpp

```
1 v class Derived : protected Base {  
2     public:  
3         void test() {  
4             publicFunc(); // OK: public -> protected  
5             protectedFunc(); // OK: protected -> protected  
6             // privateFunc(); // 错误: private 成员仍然不可见  
7         }  
8     };  
9  
10    int main() {  
11        Derived d;  
12        d.publicFunc(); // ✗ 错误! publicFunc() 在 Derived 中变成了 protected, 外部无法访问  
13    }
```

C++ 在访问保护当中还有另一种关键词为 `friend`，可以让一个函数或者类访问另一个类的私有成员

- 由类本身来控制访问权限
- `friend` 可以是：
 - 一个函数
 - 一个成员函数或者其他类
 - 甚至是整个类

指向子类的指针总是可以被赋给指向父类的指针

Upcasting examples

```
Manager pete( "Pete", "444-55-6666", "Bakery");  
Employee* ep = &pete; // Upcast  
Employee& er = pete; // Upcast
```

- Lose type information about the object:

```
ep->print( cout ); // prints base class version
```

如果基类有prt，派生类自己也定义了prt，那么直接这样upcasting得到的基类指针/引用调用prt函数时会用基类的版本

- 如果基类 `Employee` 定义了一个成员函数 `prt`，并且派生类 `Manager` 也定义了自己的 `prt` 函数（可能是重写基类的 `prt`）。
- 当通过向上转型后的指针或引用（如 `ep` 或 `er`）调用 `prt` 函数时，如果没有使用虚函数机制，编译器会根据指针或引用的静态类型（即 `Employee`）来决定调用哪个版本的 `prt`。
- 结果是调用了基类 `Employee` 的 `prt` 函数，而不是派生类 `Manager` 的 `prt` 函数。

• 解决方法：

- 如果希望调用派生类的 `prt` 函数，需要确保基类中的 `prt` 函数被声明为 虚函数。这样，在运行时会根据实际对象的类型动态绑定到正确的派生类实现。

```
void render(Shape* p) {
    p->render(); // calls correct render function
} // for given Shape! void func() {
    Ellipse ell(10, 20);
ell.render(); // static -- Ellipse::render();
Circle circ(40);
circ.render(); // static -- Circle::render();
render(&ell); // dynamic -- Ellipse::render();
render(&circ); // dynamic -- Circle::render()
```

如果是upcasting:
1. 检查Shape有无render，无render报错
2. 检查Shape的render是不是virtual
3. 是，动态绑定；反之静态绑定

这两个直接对象. 是静态绑定

这两个发生了upcasting，就要按上面的步骤进行

• 静态绑定 (Static Binding) :

- 发生在编译时，根据调用者的类型决定调用哪个函数。
- 示例：`ell.render()` 和 `circ.render()`。
- 编译器知道 `ell` 是 `Ellipse` 类型，`circ` 是 `Circle` 类型，因此直接调用各自的 `render` 方法。

• 动态绑定 (Dynamic Binding) :

- 发生在运行时，根据实际对象的类型决定调用哪个函数。
- 示例：`render(&ell)` 和 `render(&circ)`。
- 由于 `render` 函数接受的是基类指针 `Shape*`，编译器无法在编译时确定具体调用哪个版本的 `render`，因此需要在运行时根据实际对象的类型动态绑定到正确的实现。

Static and dynamic type

- The declared type of a variable is its static type.
- The type of the object a variable refers to is its dynamic type.
- The compiler's job is to check for static-type violations.

```
for(Item item : items) {
    item.print(); // Compile-time error, given no print() defined in Item
}
```

对一个对象调用函数时，其static type里一定要有这个函数（不管是虚的还是什么），不然会报错

by the way 虚函数应该是可以继承的，比如说如果Shape, Ellipse和Circle中都有render函数，并为依次派生的关系；若Shape中的render是virtual，给Shape的指针赋Circle (upcasting)，此时调用->render仍然会动态绑定到Circle上而不用管Ellipse，即动态绑定直接关注的是actual type

Virtual functions

从另一种继承的角度讲，如果子类中没有某个函数prt，父类中有virtual的prt，那么是可以对子类调用prt的

- Non-virtual functions
 - Compiler generates static, or direct call to stated type – Faster to execute
- Virtual functions
 - 覆盖 (重写) —— 条件：1.父类子类关系 2.同名同参数表 3.父类是virtual
 - Can be transparently overridden in a derived class
 - Objects carry a pack of their virtual functions
 - Compiler checks pack and dynamically calls the right function
 - If compiler knows the function at compile-time, it can generate a static call

overload则是对于一个类内部而言的，同名不同参数表的函数

What happens if

```
Ellipse elly(20F, 40F);    直接做类的对象的赋值:  
Circle circ(60F);  
elly = circ; // 10 in 5? 这样赋值以后circ的VPRT并不会赋值给elly, 调用elly的函数
```

- Area of `circ` is sliced off
 - (Only the part of `circ` that fits in `elly` gets copied)
- Vtable from `circ` is ignored
- the vtable in `elly` is the Ellipse vtable
 - `elly.render(); // Ellipse::render()`

What happens with pointers?

```
Ellipse* elly = new Ellipse(20F, 40F);  
Circle* circ = new Circle(60F);  
elly = circ;
```

- Well, the original Ellipse for `elly` is lost....
- `elly` and `circ` point to the same Circle object!

```
elly->render(); // Circle::render()
```

Virtuals and reference arguments

```
void func(Ellipse& elly)  
{  
    elly.render();  
}  
  
Circle circ(60F);  
func(circ);
```

- References act like pointers
- `Circle::render()` is called

- Subclass method is called at runtime – it overrides the superclass version.

Relaxation example

子类重写函数的返回：可以与父类一样，但当父类返回类的指针和引用时子类可以返回父类返回类型的子类的指针和引用

```
class Expr {
public:
    virtual Expr* newExpr();
    virtual Expr& clone();
    virtual Expr self();
};

class BinaryExpr : public Expr {
public:
    virtual BinaryExpr* newExpr(); // Ok
    virtual BinaryExpr& clone(); // Ok
    virtual BinaryExpr self(); // Error!
};
```

3. 覆盖重载函数的规则：

- 如果在派生类中覆盖了基类中的某个重载函数，必须覆盖所有相关的重载函数变体。
- 不能只覆盖其中一个，否则会导致某些函数被隐藏（Name Hiding）。

4. 名字隐藏（Name Hiding）：

- 如果派生类中定义了一个与基类同名的函数，但没有使用 `override`，则基类中的同名函数会被隐藏。
- 这可能导致某些函数无法通过派生类对象调用。

Virtual in Ctor?

```
class A {
public:
    A() { f(); }
    virtual void f() { cout << "A::f()"; }
};

class B : public A {
public:
    B() { f(); }
    void f() { cout << "B::f()"; }
};
```

如果实例化一个B的对象b
B b
则会调用A的构造，B的构造，A的构造中的f虽然是虚函数，但会表现为静态的，即调用A的f；B的f另外会调用一次

步骤 2：在 `A()` 中调用 `f()`

- 此时，`f()` 是一个虚函数，但由于是在构造函数中调用的，不会发生动态绑定。
- 编译器会根据当前对象的类型（即正在构造的对象的类型）来决定调用哪个版本的 `f()`。
- 此时，`B` 对象尚未完全构造完成，因此它的派生部分（包括覆盖的 `f()` 实现）还未生效。
- 因此，调用的是基类 `A` 的 `f()` 实现，输出 "`A::f()`"。

在 C++ 中，抽象类（Abstract Class）是指至少包含一个纯虚函数（pure virtual function）的类。

```

class B1 { int m_i; };
class D1 : public B1 {};
class D2 : public B1 {};
class M : public D1, public D2 {};

void main() {
    M m; //OK
    B1* p = new M; // ERROR: which B1
    B1* p2 = dynamic_cast<D1*>(new M); // OK
}

```

“析构函数可以而且应该被声明为虚函数（特别是在基类中）；

构造函数不能是虚函数。”

Virtual bases

```

class B1 { int m_i; };
class D1 : virtual public B1 {};
class D2 : virtual public B1 {};
class M : public D1, public D2 {};
void main() {
    M m; //OK
    m.m_i++; // OK, there is only one B1 in m.
    B1* p = new M; // OK
}

```

拷贝构造什么时候发生：

1. 函数的参数是类的对象，则传入对象时会发生拷贝构造
2. 用一个对象去初始化另一个对象
3. 函数返回类型是一个对象
(注意函数参数如果是类的指针或类的引用时，是不会拷贝构造的)

浅拷贝vs深拷贝

有指针时就会直接令两个指针相同，则地址共享，一个对象析构的时候就直接把整块地址删了

- 浅拷贝：编译器自动产生的拷贝构造函数，会执行member-wise copy
- 当有成员变量是指针时，这种拷贝是有害的

Operators you can't overload

.()

.*

::

?:

通过对对象本身调用元素/函数

- For binary operators (+, -, *, etc) member functions require one argument.

单目运算符

```
const Integer operator-() const {
    return Integer(-i);
}

...
z = -x; // z.operator=(x.operator-());
```

Tips: Members vs. Free Functions

- Unary operators should be members
这四个必须是成员的
- = [] ->() ->* must be members
- All other binary operators as non-members

单目的做成成员的

双目的做成全局的 (两个都可以type conversion) (要访问private的话可以用友元赋权)

```
class Integer {
public:
    ...
    const Integer& operator++();      //prefix++ ++x
    const Integer operator++(int);    //postfix++ x++
    const Integer& operator--();      //prefix-- --
    const Integer operator--(int);    //postfix-- --
    ...
};
```

- New keyword: explicit 指不能用来做类型转换，只能做构造

```

class Rational {
public:
    没有返回类型 operator double() const; // Rational to double
}
Rational::operator double() const {
    return numerator_/(double)denominator_;
}
Rational r(1,3); double d = 1.3 * r; // r->double

```

编译器会去寻找是否有转成double的函数

[]

- 必须是一个成员函数
- 单参数
- 它表明对象的行为类似一个数组，所以它需要返回一个引用

如果要把这些函数在类外定义，则在类外定义也需要是一个函数模板

如

```

template<typename T>
Vector& Vector<T>::operator=(const Vector&)
{
    ...
}

```

- Template arguments can be constant expressions • Non-Type parameters
 - can have a default argument

```

template <class T, int bounds = 100>
class FixedVector {           默认
public:
    FixedVector();
    // ...
    T& operator[](int);
private:
    T elements[bounds]; // fixed size array!
};                           就算没有上面的=100, bounds也是模板的参数而不是代码的参数，编译时会把bounds对应的变量转成字面量

```

- 模板可以继承自普通的类
- Templates can inherit from non-template classes

```

template <class A>     Derive是Base类子类的模板
class Derived : public Base { ...

```

- 类模板也可以继承自类模板
- Templates can inherit from template classes

```

template <class A>     这里List传进去的还是未知类型的A，因此还是个模板
class Derived : public List<A> { ...

```

- 普通类的父类不能是模板，但可以继承自类模板实例化后的类
- Non-template classes can inherit from templates

```

class SupervisorGroup : public
List<Employee*> { ...

```

```

template <class T>
T& Vector<T>::operator[](int indx) {
    if (indx < 0 || indx >= m_size) {
        // throw is a keyword
        // exception is raised at this point
        throw <<something>>;
    }
    return m_elements[indx];
}

void outer() {
    try {
        func();
        func2();           //与throw出的类型是匹配的
    } catch (VectorIndexError& e) {
        e.diagnostic();
        // This exception does not propagate
    }
    cout << "Control is here after exception";
}

```

扔出异常后后面的代码不会再执行，会一层层向上寻找相应的catch

异常处理后不会再回去，如func抛了异常，func后的语句都不会再执行

如果是正在new的对象在构造函数中抛异常，对象在堆中无法自动回收，未正常构造出异常后也不再有其指针用于delete，因此不能让构造函数本身抛出异常

Failure in constructors...

- If you constructor can't complete, throw an exception.
- Dtors for objects whose ctor didn't complete won't be called.
- Clean up allocated resources before throwing.

Uncaught exceptions

- If an exception is thrown by not caught `std::terminate()` will be called.
- `terminate()` can also be intercepted.

```

void my_terminate() { /* ... */ }           可以自己写
...
set_terminate(my_terminate);

```

- `std::unique_ptr`: 独占所有权，不能被复制
 - 支持移动操作 (`std::move`)，允许将所有权从一个 `std::unique_ptr` 转移到另一个。当所有权转移后，原 `std::unique_ptr` 不再指向该对象
- `std::shared_ptr`: 共享所有权，可以被复制
 - 它内部通过引用计数(reference count) 机制来跟踪有多少个 `std::shared_ptr` 实例正共享该对象
 - 引用计数：
 - 当一个新的 `std::shared_ptr` 指向该对象（例如通过拷贝构造或赋值），引用计数会增加
 - 当一个 `std::shared_ptr` 被销毁或不再指向该对象时，引用计数会减少
 - 自动销毁：只有当最后一个指向对象的 `std::shared_ptr` 被销毁，使得引用计数降为零时，该对象才会被自动删除

<code>static_cast</code>	相关类型 (upcast, 基本类型等)	✗ 编译时	✓ 一般	类型转换、upcast
<code>dynamic_cast</code>	多态类间的 downcast/upcast	✓ 运行时	✓ 安全	安全向下转型
<code>const_cast</code>	添加/移除 const/volatile	✗ 编译时	✗ 危险	修改 const
<code>reinterpret_cast</code>	任意类型 (尤其是底层类型转换)	✗ 编译时	✗ 非常危险	底层操作、跨类型访问

cpp

```

1 Base* b = new Derived();
2 Derived* d = dynamic_cast<Derived*>(b); // 安全的 downcast
3
4 if (d) {
5     // 成功转换
6 } else {
7     // 转换失败, b 实际不是 Derived 类型
8 }
```

! 注意:

- 必须用于含有虚函数的类 (即启用 RTTI)
- 指针转换失败返回 `nullptr`
- 引用转换失败抛出 `std::bad_cast` 异常
- 性能略低于 `static_cast`

用于添加或移除 `const/volatile` 属性

▣ 支持的转换包括:

- `const T*` → `T*`
- `volatile T*` → `T*`

✓ 示例:

cpp

```

1 const int a = 10;
2 int* p = const_cast<int*>(&a);
3 *p = 20; // 未定义行为! 因为 a 是 const
```

! 注意:

- 只能用于指针或引用
- 移除 `const` 后修改原对象是 **未定义行为 (Undefined Behavior)**