Contents

[1. Introduction 6](#_Toc85896828)

[1.1. First program 6](#_Toc85896829)

[1.2. cmath.h 6](#_Toc85896830)

[1.3. bool type 6](#_Toc85896831)

[1.4. namespace 7](#_Toc85896832)

[1.5. NULL pointer 8](#_Toc85896833)

[1.6. sizeof() 9](#_Toc85896834)

[1.7. overloading operators 11](#_Toc85896835)

[1.8. sequence points 11](#_Toc85896836)

[2. OOP 13](#_Toc85896837)

[2.1. class and object 13](#_Toc85896838)

[2.1.1. constructor and instantiation 13](#_Toc85896839)

[2.1.2. attributes 17](#_Toc85896840)

[2.1.1. change constructor 19](#_Toc85896841)

[2.1.2. destructor 25](#_Toc85896842)

[2.1.3. static 28](#_Toc85896843)

[2.1.4. const class variable 30](#_Toc85896844)

[2.1.5. object pointer 30](#_Toc85896845)

[2.2. inheritance 31](#_Toc85896846)

[2.2.1. inheriting constructors 33](#_Toc85896847)

[2.2.2. upcast and downcast 38](#_Toc85896848)

[2.2.3. errors of inheritance 40](#_Toc85896849)

[2.2.1. access specifier 43](#_Toc85896850)

[2.2.2. virtual methods 45](#_Toc85896851)

[2.3. overloading 50](#_Toc85896852)

[2.3.1. overload attributes 50](#_Toc85896853)

[2.3.2. overloading of constructor 51](#_Toc85896854)

[2.3.3. overloading methods 52](#_Toc85896855)

[2.3.4. overload operator 53](#_Toc85896856)

[2.3.5. syntax errors 58](#_Toc85896857)

[2.4. composition 61](#_Toc85896858)

[2.5. special class 61](#_Toc85896859)

[2.5.1. friend class 61](#_Toc85896860)

[2.5.2. const class 63](#_Toc85896861)

[2.5.3. abstract classes 63](#_Toc85896862)

[2.6. function objects 64](#_Toc85896863)

[2.6.1. struct 64](#_Toc85896864)

[2.7. objects as elements in container 68](#_Toc85896865)

[2.1. coding errors 70](#_Toc85896866)

[3. STL: container 72](#_Toc85896867)

[3.1. methods 74](#_Toc85896868)

[3.2. Create 77](#_Toc85896869)

[3.3. Get 80](#_Toc85896870)

[3.4. find 83](#_Toc85896871)

[3.5. Modify 86](#_Toc85896872)

[3.5.1. string 89](#_Toc85896873)

[3.5.2. vector 90](#_Toc85896874)

[3.5.3. map/multimap 92](#_Toc85896875)

[3.5.4. set/multiset 93](#_Toc85896876)

[3.6. other 96](#_Toc85896877)

[3.6.1. merge/split 96](#_Toc85896878)

[3.6.2. Compare string 96](#_Toc85896879)

[3.6.3. Copy 97](#_Toc85896880)

[3.6.4. Size or length 98](#_Toc85896881)

[4. Iterator 103](#_Toc85896882)

[4.1. basic 103](#_Toc85896883)

[4.2. pointer/reference/iterator 103](#_Toc85896884)

[4.3. container iteration 106](#_Toc85896885)

[4.3.1. string 106](#_Toc85896886)

[4.3.2. vector 107](#_Toc85896887)

[4.3.3. map 109](#_Toc85896888)

[4.3.4. transform 110](#_Toc85896889)

[4.4. errors 111](#_Toc85896890)

[4.4.1. syntax errors 111](#_Toc85896891)

[4.4.2. Invalidation 111](#_Toc85896892)

[5. STL: algorithm 114](#_Toc85896893)

[5.1. math 115](#_Toc85896894)

[5.1.1. max/min 115](#_Toc85896895)

[5.1.2. counting 116](#_Toc85896896)

[5.2. merge 117](#_Toc85896897)

[5.2.1. concatenate 117](#_Toc85896898)

[5.2.2. combine/sum/product 118](#_Toc85896899)

[5.2.3. union 119](#_Toc85896900)

[5.3. partition 121](#_Toc85896901)

[5.3.1. partition 121](#_Toc85896902)

[5.3.1. split 122](#_Toc85896903)

[5.4. Search 123](#_Toc85896904)

[5.4.1. find/search 123](#_Toc85896905)

[5.4.2. Compare 127](#_Toc85896906)

[5.4.3. binary search 129](#_Toc85896907)

[5.4.4. check consistency 130](#_Toc85896908)

[5.5. random 130](#_Toc85896909)

[5.6. Sort 131](#_Toc85896910)

[5.7. update 133](#_Toc85896911)

[5.7.1. replace/fill 133](#_Toc85896912)

[5.7.1. remove duplicates 135](#_Toc85896913)

[5.8. Modify 136](#_Toc85896914)

[5.8.1. Copy 136](#_Toc85896915)

[5.8.2. Reverse 139](#_Toc85896916)

[5.8.3. swap/rotate 140](#_Toc85896917)

[5.8.4. move 143](#_Toc85896918)

[5.8.5. Convert 144](#_Toc85896919)

[6. Template 147](#_Toc85896920)

[6.1. define templates 147](#_Toc85896921)

[6.2. STL sequential container 148](#_Toc85896922)

[6.3. STL associative containers 148](#_Toc85896923)

[6.4. non-modifying STL algorithms 148](#_Toc85896924)

[6.5. modifying STL algorithms 148](#_Toc85896925)

[6.6. sorting STL operations 148](#_Toc85896926)

[6.7. STL merge operations 148](#_Toc85896927)

[6.8. STL utilities an functional libary 148](#_Toc85896928)

[7. Input/Output 149](#_Toc85896929)

[7.1. <iostream> 149](#_Toc85896930)

[7.2. text file: <fstream> 151](#_Toc85896931)

[7.3. <sstream> 153](#_Toc85896932)

[8. Basic data type 156](#_Toc85896933)

[8.1. digits and character 156](#_Toc85896934)

[8.1.1. digits 156](#_Toc85896935)

[8.1.2. type conversion 157](#_Toc85896936)

[8.2. enum 159](#_Toc85896937)

[8.2.1. syntax errors 160](#_Toc85896938)

[9. function 161](#_Toc85896939)

[9.1. basic 161](#_Toc85896940)

[9.1.1. declare, define, and call 161](#_Toc85896941)

[9.1.2. arguments and return 163](#_Toc85896942)

[9.1.3. pass arguments to function 164](#_Toc85896943)

[9.1.4. pass pointer or reference 168](#_Toc85896944)

[9.2. <functional> 172](#_Toc85896945)

[9.2.1. operators 172](#_Toc85896946)

[9.2.2. other function 172](#_Toc85896947)

[9.3. other 173](#_Toc85896948)

[9.3.1. inline function 173](#_Toc85896949)

[9.3.2. overloading 174](#_Toc85896950)

[9.3.3. syntax errors 175](#_Toc85896951)

[10. exceptions 176](#_Toc85896952)

[10.1. throw exceptions 176](#_Toc85896953)

[10.1.1. throw various data type: 176](#_Toc85896954)

[10.1.2. throw in function: 180](#_Toc85896955)

[10.2. hierarchy of exceptions 180](#_Toc85896956)

[10.3. customary exceptions 182](#_Toc85896957)

[10.4. syntax errors 182](#_Toc85896958)

[11. memory 185](#_Toc85896959)

[11.1. dynamic memory: <new> 185](#_Toc85896960)

[11.2. memory leak 186](#_Toc85896961)

[11.3. The new and delete keyword 186](#_Toc85896962)

# Introduction

## First program

|  |
| --- |
| //preprocessor directive  #include <iostream>  //namespace should be declared outside of functions  using namespace std;  int main(void){  cout << "hello\n" << endl;  } |

The header file "iostream" is file name

"cout" is one instance of the class iostream

A namespace is an abstract container or environment created to hold a logical grouping of unique entitles(blocks). "std" will make the standard facilities available throughout the program.

<< is defined in std::ostream::operator known as insertion operator. That overloaded as a member function.

std::cout<< "hi"

std::cin>>

## cmath.h

|  |  |
| --- | --- |
|  | sqrtf() square root |
|  |  |
|  |  |
|  |  |
|  |  |

## bool type

|  |  |
| --- | --- |
| bool yes = !false;  bool no = !yes;  if(!no) cout << "true";  else cout << "false"; | output: true |
| bool yes = !false;  bool no = !yes;  if(!no) cout << true;  else cout << false; | output: 1 |
|  |  |
|  |  |

## namespace

define namespace:

namespace <space name>{ …}

use namespace:

using namespace <space name>;

<space name>::<name>

use one object in namespace:

using <space name>::<name>

|  |  |
| --- | --- |
| #include <iostream>  use namespace std; | cout, cin, and edl belong to std |
| namespace Universe{int Galaxy=1;}  namespace Universe{int Planet = Galaxy+2; }  int main(void){  Universe::Galaxy \*=2;  //scope of namespace  **{**  **using namespace Universe;**  **Planet++;**  **}**  cout << Universe::Galaxy << Universe::Planet;  return 0;  } | output: 24  it is ok to define multiple block for one namespace.  There is scope limits for namespace {} |
| namespace space { char a= 'a', b= 'b';}  int a=1, b=2;  int main(void) {  **using space::a;**  **cout << a << " " << b;**  return 0;  } | output: a b  specify namespace  **Note: using namespace space::a; is invalid** |
| namespace space { char a= 'a', b= 'b';}  int a=1, b=2;  int main(void) {  using space;  cout << space::a << " " << space::b;  return 0;  } | compiling error: expected nested-name-specifier before 'space'  correct: using namespace space; |
| namespace outspace { int x=1; int y=2;}  namespace inspace { float x=3.0; float y=4.0;}  int main(void) {  { using namespace inspace;  cout << x; }  { using namespace outspace;    using inspace::y; cout << y; }  return 0;  } | output: 34  Note: "using namespace outspace;" is useless,"using inspace::y" override outspace::y. |

syntax error

|  |  |
| --- | --- |
| namespace A{ int A;}  namespace B{int A;}  int main(void){  A::A=B::A=1;  cout << A+1;  return 0;  } | compiling error.  correct: cout << B::A+1; |
| namespace space { char a= 'a', b= 'b';}  int a=1, b=2;  int main(void) {  using namespace space;  **cout << a << b;**  return 0;  } | compiling error:  ambiguous a and b  correct: cout << space::a << space::b; |

## NULL pointer

In C, NULL pointer is defined by ==NULL. But in default, "#define NULL 0;" which may cause confusion. In C++11, the pointer literal nullptr is introduced.

In C++, for null pointer ptr, ptr==NULL, ==0, !ptr, or ptr=nullptr will return 1

|  |  |
| --- | --- |
| void \*ptr = nullptr;  if(!ptr){  cout << NULL;  } | void \*ptr = 0;  void \*ptr=NULL;  are equal |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## sizeof()

The sizeof keyword gives the amount of storage, in bytes, associated with a variable or a type (including aggregate types). This keyword returns a value of type size\_t.

sizeof(string)=24

sizeof(char)=1

sizeof(short)=2, sizeof(int)=4, sizeof(long)=4, sizeof(long long)=8

sizeof(float)=4, sizeof(double)=8, sizeof(long double)=12,

string type

In sizeof(string), string is std::string. sizeof(std::string) returns the size of the class instance and its data members =24 bytes depending on platform, not the length of the string.

|  |  |  |
| --- | --- | --- |
|  | output |  |
| string a="abc";  cout<< sizeof(string); | 24 | 4\*3+4=  string class instance contains a pointer, and a struct |
| cout << (sizeof string); | 24 | the same as the above |
| string a="abc";  cout<< sizeof(a); | 24 | sizeof(a) is equal to sizeof(string) or sizeof(std::string)  string class instance contains a pointer, and a struct |
| string a="a\0bc\0";  cout<< sizeof(a); | 24 |  |
| cout<< sizeof("a\0bc"); | 5 | 4+1 characters |
| cout<< sizeof("a\0bc\0"); | 6 | 5+1 characters |
| string a="abc", \*p;  cout<< sizeof(p); | 4 | pointer size = 4 |
| string a="abc", \*p;  cout<< sizeof(\*p); | 24 | ==sizeof(a) |
|  |  |  |

size of int, float, double

|  |  |  |
| --- | --- | --- |
|  | output |  |
| int a;  cout<< sizeof(a=10); | 4 | expression a=10 succeeds return 1, which memory storage is 4 bytes |
| int a=100;  cout<< sizeof(a); | 4 | In C++, a is instance of int class. the pointer size is 4 bytes |
| double a=100.23;  cout<< sizeof(a); | 8 |  |
| int a=1;  cout<< sizeof(a=3) << a; | 14 | sizeof(a=3)=1, a=4 |
|  |  |  |
|  |  |  |
|  |  |  |

size of function

|  |  |  |
| --- | --- | --- |
|  | output |  |
| int f(){ return 100; }  int main(void){  cout<< sizeof(f()) << endl;  return 0;  } | 4 | sizeof(f()) == sizeof (int) |
| double f(){ return 10.23; }  int main(void){  cout<< sizeof(f());  return 0;  } | 8 |  |
|  |  |  |
|  |  |  |
|  |  |  |
| void f(){ int i=100; }  int main(void){  cout<< sizeof(f()) << endl;  return 0;  } | 1 | sizeof(void)=1 |
| void f(int &p){ p+=100; }  int main(void){  int i=100;  **cout<< sizeof(f(i)) << i**;  return 0;  } | 1100 | in sizeof(), f() can't be called because f(i) will be replaced with int type when compiling. In this snippet, the function is not being called. |

pointer

|  |  |  |
| --- | --- | --- |
|  | output |  |
| int i=4, \*p; p=&i;  cout<< sizeof(p)<<sizeof(\*p); | 44 |  |
| double i=4, \*p; p=&i;  cout<< sizeof(p)<<sizeof(\*p); | 48 |  |
| float i=4, \*p; p=&i;  cout<< sizeof(p)<<sizeof(\*p); | 44 |  |
| char i='a', \*p; p=&i;  cout<< sizeof(p)<<sizeof(\*p); | 41 |  |
| string i="abc", \*p; p=&i;  cout<< sizeof(p)<<sizeof(\*p); | 424 |  |
| float i[]={1,2,3}, \*p; p=i;  cout<< sizeof(p)<<sizeof(\*p); | 44 | \*p is equal to value of i[0], then equal to sizeof(float) |
| char i[]="ABC", \*p; p=i;  cout<< sizeof(p)<<sizeof(\*p); | 41 | sizeof(\*p) -> sizeof(i[0]) -> sizeof(char) |

## overloading operators

<string> + is used for concatenate multiple strings.

<money>

## sequence points

The order of execution is unspecified.

|  |  |
| --- | --- |
| class A{  public: A(){ a[0]=1;a[1]=0; }  int a[2];  int b(void){  int x=a[0]; a[0]=a[1]; a[1]=x;  return x;  }  };  int main(){  A a; a.b();  **cout << a.b() << a.a[1] << endl;**  return 0;  } | output: 01  Note: for cout statement, a.b() is not executed before a.a[1] |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# OOP

## class and object

object is instances of class

### constructor and instantiation

constructor a special non-static member function of a class. constructor is used for initializing objects of a class.

1.constructors must have the same name with class name and cann't be called directly like other class methods.

2. constructors without arguments are called default constructors. No constructor is ok for a class.

3. A constructor will be automatic run once when an object is created. If no constructor is define, a default constructor will be created for the object.

4. constructors that take another object of the same type as the arguments are copy constructor. In that case, constructors don't need run again.

1. class should be instantiated to objects before use except static methods. objects could be updated by obj=<value> (only one attr) or obj.<attr>=<value> (multiple attr).

class A{

int a;

public:

A(){ a=0; }

A(int i){ a=i; }

void get(){ cout << this->a;}

};

Here are all possible instantiations

|  |  |
| --- | --- |
| **A a;**  **a.get();** | output: 0  no arguments |
| A a(4);  a.get(); | output: 4  arguments constructor overload |
| A a=3;  a.get(); | output: 3;  equal to a(3) |
| A a;  a=6;  a.get(); | output: 6  update constructor |
| A a=4, b;  b=a; b.get(); | output:4  assign constructor |
| A \*p = new A();  \*p=6;  p->get(); | output: 6  pointer pattern |
| A a();  a.get(); | wrong |
|  |  |

2. public and private: attributes usually are private, which can't be accessed by objects. methods are public. If not explicit, all are private. **keep eyes on declaration of public and private.**

3. attributes are usually defined in private, and initialized in constructor. If no initialization, unknown value are returned but no compiling error. private attributes or methods can' be access by objects. **Keep eyes on the use like "obj.var=0;" in main(). can't use "cls::var" outside of class. Inside class, it is ok to use "var=0;", "this->var=0;", "cls::var=0;"**

**use class or objects as attribute**

|  |  |
| --- | --- |
| class A{  public:  string s;  A(string s){ this->s = s;}  };  class B{  public:  string s;  **B(A a){ this->s = a.s;}**  void print(){cout<<s;}  };  int main(void) {  A a("hello");  **B b=a; //equal to "B b(a);"**  b.print();  return 0;  } | **output: hello** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

4. constructor has not return, and can't be as other methods in class. overloading constructor: void, values, references of objects

5. destructor would better be defined after constructor. All destructors matching all objects would be executed when returning in main().

|  |  |
| --- | --- |
| class A{  public:  A(){ a[0]=1; a[1]=0; }  int a[2];  int b(void){  int x=a[0];a[0]=a[1];a[1]=x;  return x;  }  };  int main(){  A a; a.b();  **cout << a.b() << a.a[1] << endl;**  return 0;  } | output:01  **Note: execution order doesn't follow the order in cout statements** |
| class A{  public:  A(){ a.a=a.b=1;}  struct {int a,b;} a;  int b(void);  };  int A::b(void){  int x=a.a; a.a=a.b; a.b=x;  return x;  };  int main(){  A a; a.a.a = 0; a.b();  cout << a.b() << a.a.b << endl;  return 0;  } | output: 10  Note: a.a.b is executed before a.b() in cout statement. |
| class A{  public:  int a;  A() { a=0; cout << "A" << a; }  A(int b) {cout << "B" << b;}  };  int main(){  **A a(0);**  **a=4;**  return 0;  } | output: B0B4  **Update instance: instantiate class A with a(0), and then update instantiation.** |
| #include <iostream>  using namespace std;  class Class {  public:  Class(void) { cout << "constructed-1" << endl; }  Class(int v) { value = v; cout << "constructed-2" << endl; }  ~Class(void) { cout << "destructed val = " << value << endl; }  void IncAndPrint(void) {  cout << "value = " << ++value << endl;  }  int value;  };  int main(void) {  Class \*ptr1, \*ptr2;    ptr1 = new Class;  ptr2 = new Class(2);  ptr1 -> value = 1;  ptr1 -> IncAndPrint();  ptr2 -> IncAndPrint();  delete ptr2;  delete ptr1;  return 0;  } |  |
| class A{  public:  int data[3];  private:  int cnt;  public:  void put(int v){data[cnt++]=v;}  int take() {**int c=cnt;cnt=0;**return c;}  };  int main(){  A a;  a.take();  a.put(a.take());  a.put(1);  cout << a.data[0] << endl;  return 0;  } | output:0  attribute initialization: Run a.take() first time: "int c=cnt;" c is not initialized because cnt is not initialized. |
| class A{  float v;  public:  A(){v=1.0;}  **A(A &a) {A::v=a.v; cout<<"1";}**  ~A(){cout<<"0";}  float set(float v){  A::v=v;return v;}  float get(float v){return A::v;}  };  int main(){  A a, \*b=new A(a),  \*c=new A(\*b);  c->get(b->get(a.set(1.0)));  delete b;  delete c;  return 0;  } | output: 11000  note: 1. overloading: pass reference of object  2. three instances: run constructor three times, and run destructor three times when return in main(). |
| class A{  public:  A(float v){A::v=v;}  float v, b;  float set(float v){A::v=v; return v;}  float get(float v){return A::v;}  };  int main(){  **A \*a=new A(1.0), \*b=new A(\*a);**  cout<< a->get(b->set(a->v));  return 0;  } | **output:1**  **Note: A \*b=new A(\*b); but constructor doesn't define pointer argument** |

### attributes

1. set default values of attributes in constructor

2. declare attributes is default private. It is ok to public or protected.

3.for private attributes, build methods to access or update private methods. and configure those methods as public, protected.

|  |  |
| --- | --- |
| class A{  int x,y,z;  public:  A(){x=y=z=0;}  **A(int a, int b): x(a), y(b){ z=x+y; }**  void print(){ cout << x << y << z; }  };  int main(void){  A a(1,2); a.print();  A b; b.print();  return 0;  } | output:123000  1. two constructors: one default gives default values. another pass arguments  2.**"A(int a, int b): x(a), y(b){ z=x+y; }" is equal to "**A(int a, int b) { x=a, y=b, z=x+y; }**".**  **"x(a),y(b)" is initializer list.** |
| class A{  **public: int x,y,z;**  void print(){ cout << x << y << z; }  };  int main(void){  A b; b.print();  return 0;  } | output: unpredictable values  though no compiling errors, attributes x,y,y are not initialized but allocated memory address. |
| class A{  **public: int x,y,z;**  void print(){ cout << x << y << z; }  };  int main(void){  **A b; b.x=b.y=b.z=0; b.print();**  return 0;  } | output:000  initialize/update attributes of objects. In real world, it not suggested. And attributes should be public type. |
| class A{  **private: int x,y,z;**  public:  A(){x=y=z=0;}  A(int a, int b) { x=a, y=b, z=x+y; }  **void setX(int a){ x=a; }**  **void getX(){ return x;}**  void print(){ cout << x << y << z; }  };  int main(void){  A b; b.setX(1); b.print();  return 0;  } | output:100  1.For security, attributes are usually configured as private, which can't be accessed by other classes. and build methods for updating or getting attributes. |
| class A{  private: int x,y,z;  public:  A(){x=y=z=0;}  A(int a, int b) { x=a, y=b, z=x+y; }  void print(){ cout << x << y << z; }  protected:  void setX(int a){x=a;}  void getX(){return x;}  };  int main(void){  A b; b.setX(1); b.print();  return 0;  } | compiling error.  main() can't access attributes because those are protected methods, which is only open to derived class. |
|  |  |
|  |  |
|  |  |

### change constructor

1. assign constructor differing from initialization

For class A,

copy constructor: "A a1; A a2=a1;" object a1 is initialized, and a2 is initialized by the value of as is passed by value of a1. The two share the same memory

assign constructor. "A a1, a2; a2=a2" objects a1 and a2 are initialized. both of them have values. Then a2's values are updated by a1.

|  |  |
| --- | --- |
| class Uno{  public: Uno() { cout << "x";}  };  Uno foo(Uno d){  **Uno e=d; return e;**  }  int main(void) {  Uno u; foo(u);  return 0;  } | output: x  Uno e=d; copy instructor.  the two objects are different |
| class Uno{  public: Uno() { cout << "x";}  };  Uno foo(Uno d){  **Uno e; e=d; return e;**  }  int main(void) {  Uno u; foo(u);  return 0;  } | output: xx  Uno e; e=d; e is different object |
|  |  |
|  |  |
|  |  |

2. copy constructor only if class type is same.

copy newly objects: "A a; A b=a;"

|  |  |
| --- | --- |
| class A{  int val;  public:  A() {val=1;}  int get(){return ++val;}  };  int main(){  // copy object  **A a, b=a;**  b.get();  cout << a.get() << b.get();  return 0;  } | output:23  "A a, b=a;" and "A a,b; b=a;" result in same result |
| class A{  int \*val;  public:  A() {val = new int; \*val=0;}  int get(){return ++(\*val);}  };  int main(){  A a, b=a;  cout << a.get() << b.get();  return 0;  } | output:21 |
| class A{  int \*val;  public:  **A() {val=new int; \*val=0;}**  **A(A &a){val=new int; \*val = a.get();}**  int get(){return ++(\*val);}  };  int main(){  A a, b=a;  cout << a.get() << b.get();  return 0;  } | output:22  Note: "b=a" equal "b(a)"  b, and a are two different objects. initialized using different constructor. |
| class N{  public: float x;  N(){ x=0.0; }  N(float a){ x=a; }  N(N &n){ x=n.x; }  string operator==(N &n){  if(this!= &n) return "true"; else return "false";  }  };  int main(void) {  N a(1.1), \*b=&a, c=a, d; d=a;  cout << (a==\*b) << (a==c) << (a==d);  return 0;  } | output: falsetruetrue  except object pointer \*b, the object c and d are copy of a, differing from a. |
|  |  |
|  |  |
|  |  |

pass objects as arguments or return statement into function: "class A{}; A a; void f(A a);"

2. convert constructor: child-> parent, parent

static\_cast<A \*>(b); child->parent. done when compiling

dynamic\_cast<B \*>(a); child->parent, or parent->child in some cases

|  |  |
| --- | --- |
| class X{};  class Y: public X{};  class Z: public X{};  int main(){  Z \*z=new Z(); X \*x=new X(); Y \*y=new Y();  **x=z;  cout<< (x==z) <<endl;**  // wrong: z=y; not inheritance  //wrong: y=z; no inheritance  //wrong: z=x; can't convert parent to child  return 0;  } | output: 1  correct: convert child object to parent.  can't convert parent class to child  can't convert if no inheritance |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

3. pass constructor by value or by reference

|  |  |
| --- | --- |
| class A{  int \*val;  public:  A() {val=new int; \*val=0;}  A(A &a){val=new int; \*val = a.get();}  int get(){return ++(\*val);}  };  int main(){  **A a, b=a;**  cout << a.get() << b.get();  return 0;  } | output: 22 |
|  |  |
| class A{  public: void out(){cout << "A" << endl;}  };  class B: public A{  public: void out(){cout << "B" << endl;}  };  int main(){  **A \*a = new B();**  a->out();  **static\_cast<B\*>(a)->out();**  return 0;  } | output: AB  object of child class could be converted to that of parent class or reverse back to child class.  Note: but child class can't be converted to parent class. |
| class A{  public:  A():val(0) {}  int val;  virtual void run(){cout<<val;}  };  class B: public A {};  class C: public B {  public:  void run(){cout<<val+2;}  };  void Do(A \*a){  B \*b;  C \*c;  if(b=dynamic\_cast<B\*>(a))  b->run();  if(c=dynamic\_cast<C\*>(a))  c->run();  a->run();  }  int main(){  A \*a=new C();  Do(a);  return 0;  } |  |
| #include <iostream>  using namespace std;  class Class {  int data;  public:  Class(int value) : data(value) {}  void increment(void) { data++; }  int value(void) { return data; }  };  int main(void) {  Class o1(123);  **Class o2 = o1;**  **Class o3(o2);**    o1.increment();  cout << o1.value() << endl;  cout << o2.value() << endl;  cout << o3.value() << endl;  return 0;  } | the object o2 and o3 are copy of object o1. but they are different object and each has its own data field.  A(A &) |
| class A{  public:  A(): val(0){}  int val;  void inc(){++val;}  };  **void Do(A a){a.inc();}**  int main(){  A a;  Do(a);  a.inc();  cout<< a.val;  return 0;  } | output:1  pass object to function. The object is local. inner changes would not affect on the object. |
| class A{  public:  A(): val(0){}  int val;  **int inc(){ ++val; return val--;}**  };  void Do(A \*a){   a->val = a->inc();}  int main(){  A a;  Do(&a);  cout<< a.val << a.inc();  return 0;  } | output:12 |
|  |  |
|  |  |
|  |  |

5. update attributes by constructor

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

### destructor

For class X:

if "X x;", destructor will be automatically run within the scope.

if "X \*x = new X();", destructor will not run unless "delete x;".

|  |  |
| --- | --- |
| class X{  public:  X() { cout << 1; };  ~X() { cout << 2; };  };  int main(void) {  X x;  cout << 0;  return 0;  } | output:102  **"X x;" so destructor will be run once main() is done.** |
| class X{  public:  X() { cout << 1; };  ~X() { cout << 2; };  };  void exec(){ X x; }  int main(void) {  X \*x = new X();cout << 0;  return 0;  } | **output: 10**  **create class pointer. destructor will not run if no delete statement** |
| class X{  public:  X() { cout << 1; };  ~X() { cout << 2; };  };  int main(void) {  **X \*x = new X();delete x;**cout << 0;  return 0;  } | output: 120  **compared with the above**  **for class pointer, "delete x;" will trigger destructor.**  **If remove "delete x;" behind of "cout<<0;", the output will be 102.** |
| class A{  public:  string s;  A(string s){ this->s = s; cout << "Con";}  **~A(){cout << "De";}**  };  int main(void) {  A a("hello");  **a.~A();**  return 0;  } | output: ConDeDe  destructor could be launched separately. but constructor can't run again. for example "a.A("good");" is wrong. |
| class X{  public:  X() { cout << 1; };  ~X() { cout << 2; };  };  **void exec(){ { X x;} }**  int main(void) {  exec();cout << 0;return 0;  } | output:120  Note: the scope of "X x;" is within exec() rather main(). so destructor is run before "cout<<0;" |
| class E { public: E(int i) { cout << i; } };  class X{  static int c;  public:  X() { if(c++>2) throw E(c);};  ~X() { if(c++>2) throw E(c);};  };  int X::c=0;  void f(int i) { X a, b;  cout << i; }  int main(void) {  try { f(0); f(1); }  catch(...){ cout << 1; }  return 0;  } | output: 041  0 is determined by f(0)  4 is determined by constructor of f(0)  1 is determined by cout<<1 |
| class X{  public:  X() { cout << 1; };  ~X() { cout << 2; };  };  X \*exec(){  X \*x = new X();  throw string("0");  return x;  }  int main(void) {  X \*x;  try { delete exec(); }  catch (string &s){ cout << s;}  return 0;  } | output: 10  Note: 1. "X\* x;" has not impact on outcome.  2. delete have not impact on outcome because exception is thrown before return statement. |
| class A{  string a;  public:  A(){a="a"; cout << "A";}  A(string a){cout << a;}  ~A(){cout << "a";}  };  void f(A\* d){  **A e("3"); \*d = e;**  }  int main(void) {  **A \*a = new A(); f(a); delete a;**  return 0;  } | ouput:A3aa  1. two patterns to instantiate class. for pointer pattern, delete should be run or no destructor.  2. \*d=e is equal to d=&e |
| class A{  public:  string s;  A(string s){ this->s = s; }  **~A(){ delete s;}**  };  int main(void) {  A a("hello");  return 0;  } | compiling error:  don't use "delete s;" in destructor because "~A(){}" will delete all attributes without deleting them one by one. |
| class X{  public:  X() { cout << 1; };  ~X() { cout << 2; };  };  void exec(){  **{ X x;}**  **throw string("0");**  }  int main(void) {  **try { exec();}**  **catch(string &s){ cout << s; }**  return 0;  } | output:120  {X x;} constructor and destructor will run in the scope once throw exception |

### static

A static variable is never allocated on a stack. They are allocated space on different static storage. when we declare a static variable in a class, this variable is shared by all the objects of that class.

As static variables are initialized only once and are shared by all objects of a class, **the static variables are never initialized by a constructor.** Instead, the static variable should be explicitly initialized outside the class only once using the scope resolution operator (::).

When the first object is created, all static data of primitive type is initialized to zero when no other initialization is present.

|  |  |
| --- | --- |
| class Test {  public:  static void funS1(void) { cout << "static" << endl; }  static void funS2(void) { funS1(); }  };  int main(void) {  Test object; Test::funS2();object.funS2();  return 0;  } | output:  static  static  class->static methods or  instance->static methods |
| class Test {  public:  void funN1(void) { cout << "non-static" << endl; }  static void funS1(void) { funN1(); }  };  int main(void) {  Test object;Test::funS1();object.funS1();  return 0;  } | compiling error  static method can't access not-static methods |
| class Test {  public:  static void funS1(void) { cout << "static" << endl; }  void funN1(void) { funS1(); }  };  int main(void) {  Test object;object.funN1();  } | non-static can access static method |
|  |  |
|  |  |

static attributes

1. static attribute should be declared in class. That can't be initialized in constructor, but out of the class.

2. can't be call static attribute by class cls::<var> or by object obj.<var>

|  |  |
| --- | --- |
| class A{  **public:static int a;**  A() {a++;}  A(A &aa) {a++;}  };  **int A::a=1; // initialization out of class**  int main(){  A a, b(a), c(b);  **cout << A::a << endl;**  **cout << a.a << endl;**  return 0;  } | output: 4 4  define static variable |
| class A{  public: static int age;  A(){}  ~A(){}  void setAge(int a=10){age=a;}  void print(){ cout<<age;}  };  int A::age = 0;  int main(void) {  A a, c, \*b;   b = new A();  **a.setAge(30);b->setAge(20);  c.setAge();**  a.print();b->print();c.print();  return 0;  } | output:101010  though define multiple objects, A::age is always determined by last setting. |
| class E {};  class X{  **public: static int c;**  X(int a) { c=a; };  ~X() { if(c++>2) throw new E; }  };  **int X::c=0;**  void f(int i){  X\*t[2];  for(int j=0;j<i;j++)  t[j]=new X(i+1);  for(int j=0;j<i;j++)  delete t[j];  }  int main(void) {  try { f(2); }  catch(...) { cout << X::c;}  return 0;  } | output:4 |

### const class variable

**A**const**class field must be initialized inside an initialization list within any of the class constructors.** Any other assignment will be rejected.

a const **object mustn’t be modified during its life**.

|  |  |
| --- | --- |
|  |  |
| const a =3.14;  const b= a\*a;  cout << b; | compiling error  correct: const float a=3.14; |
|  |  |
|  |  |
|  |  |

### object pointer

object has class type. object type casting is not allowed except inheritance.

type casting of object should be explicitly done by object pointer.

|  |  |
| --- | --- |
| class A{public: void print(){cout<<"A";}};  class B{public: void print(){cout<<"B";}};  int main(void) {  B sc[2]; sc[0].print();  **A \*bc = (A\*) sc;**  for(int i=0; i<2;i++)  (bc++)->print();  return 0;  } | output:BAA  Note: class A and B have not relation.  **wrong: A\*bc = sc;** |
| class A{  public: void print(){cout<<"A";}  };  class B: public A{  public: void print(){cout<<"B";}  };  **void fun(A \*obj){ obj->print(); }**  int main(void) {  A a; fun(&a);  B b; fun(&b);  return 0;  } | output:AA  The argument of fun() only accept object of base class A. convert B to A  wrong: **void fun(B \*obj){ obj->print(); } can't convert from based class 'A\*' to derived class 'B\*'** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## inheritance

Pattern:

class X{…};

class Y {…};

class Z {…};

class A : public X, Y, Z { … };

Note: except attributes and method, constructor, destructor and operator= cannot inherit from base class.

class inheritance: derived class inherit attributes and methods from base class.

|  |  |
| --- | --- |
| class X{  public:int v;  void put(int x){ v=x; }  int get(void){ return v; }  };  **class Y: public X{**  public:  Y(){ put(0); }  void write(int x){ put(x+1); }  int read(void){ return get()-1; }  };  int main(){  **Y \*y=new Y(); //derived class**  y->write(1);cout<<y->read()<<endl;  return 0;  } | output: 1  class Y inherit attribute v, methods put() and get() from class X  Note: this is simplest case. 1. There is no overloading on attributes/constructor/methods. 2. No upcast or downcast of object. 3. access is public. 4. one inheritance. no inheritance chain or other relationship. |

pass object to class is equal to inheritance

|  |  |
| --- | --- |
| class A{  public:int a;  A() {a =0;}  A(int b) {a=b+1;}  };  class B{  public:**A a;**  **B(): a(0){}**  };  int main(){  B \*b= new B();   cout << b->a.a << endl;  return 0;  } | output: 1  "B(): a(0){}" explicit define constructor in class B, which inherit from constructor of class A. |
| class A{  public:  int a;  A() {a =0; cout << "AA";}  A(int b) {a=b+1;cout << "A";}  };  class B{  public: A a;  B(): a(0){a=4; cout << "B";}  };  int main(){  B \*b= new B(); cout << b->a.a << endl;  return 0;  } | output: AAB5  constructor of class B inherit that from class A. a(0) is object of class A in constructor. Then the object a is updated with 4. |
| class A{  public:int a;  A() {a=1;}  A(int aa) {a=2;}  **A(A &aa) {a=3;}**  };  int main(){  A a(1), b(a);  cout << a.a + b.a << endl;  return 0;  } | output:5  Note: one overloading constructor is pass instance of itself. |

### inheriting constructors

Order of Constructor Call with Inheritance in C++. The order of the destructor is exactly the reverse

1. Construction always starts with the base class. If there are multiple base classes then, construction starts with the leftmost base. If there is a virtual inheritance then it's given higher preference).

2. Then the member fields are constructed. They are initialized in the order they are declared

3. Finally, the class itself is constructed

|  |  |
| --- | --- |
| class A{  **public:A(){cout<<"Ano";}**  };  class B: public A{  **public:B(int s){cout<<"Bint";}**  };  int main(void) {  B b1(10);  return 0;  } | output:AnoBint  construction order: start from base class: A() -> B(int s) |
| class A{  public:  A(){cout<<"Ano";}  **A(int s){cout<<"Aint";}**  };  class B: public A{  **public:B(int s){cout<<"Bint";}**  };  int main(void) {  B b1(10);  return 0;  } | output:AnoBint  construction order: A() -> B(int s)  **Note: constructor A(int s) is overridden by constructor B(int s)** |
| class A{  public:  **A(int s){cout<<"Aint";}**  };  class B: public A{  **public:B(int s){cout<<"Bint";}**  };  int main(void) {  B b1(10);  return 0;  } | compiling error  "A(){cout<<"Ano";}" is missing. calling constructor always start from base class. Or remove all constructor of class A. If constructors are required in class A, at least A() should be defined. |
| class A{  public:  A(){cout<<"Ano";}  A(string s){cout<<"Astr";}  A(A &a){cout<<"Aobj";}  };  class B: public A{  public:  B(){cout<<"Bno";}  B(string s){cout<<"Bstr";}  B(int s){cout<<"Bint";}  };  int main(void) {  B b1(10);B b2(b1);  return 0;  } | output: AnoBintAobj  order: A::A() -> B::(int s), A::(A &a) |
| class A{  public:  A(){cout<<"Ano";}  A(string s){cout<<"Astr";}  A(A &a){cout<<"Aobj";}  };  class B: public A{  public:  B(){cout<<"Bno";}  B(string s){cout<<"Bstr";}  B(int s){cout<<"Bint";}  };  int main(void) {  B b1(10);  B b2("good");  return 0;  } | output: AnoBintAnoBstr  order : A::A() -> B::(int s), A::A() ->B::(string s) |
| class A{  public:  A() {cout<<"A";}  ~A() {cout<<"a";}  };  class B: public A{  public:  B() {cout<<"B";}  ~B() {cout<<"b";}  };  class C: private B{  public:  C() {cout<<"C";}  ~C() {cout<<"c";}  };  int main(void) {  B b; C c;  return 0;  } | output: ABABCcbaba  constructor storage is stack type: "last in, and first out.":  for constructor: order starts from based to derived.  "B b;" constructor: AB  "C c" constructor: ABC  for destructor, order is reversed. first c and then b  "C c" destructor: cba  "B b;" destructor: ba |
| class A{  public:  A() {cout<<"A";}  ~A() {cout<<"a";}  };  class B{  public:  B() {cout<<"B";}  ~B() {cout<<"b";}  };  class C: public A, public B{  public:  C() {cout<<"C";}  ~C() {cout<<"c";}  };  int main(void) {  C c;  return 0;  } | output: ABCcba  inheritance: C->A, C->B |
| class A{  public: int x;  A(){x=0;}  };  class B{  public: int x;  B(){x=1;}  };  class C: public A, public B{  public: int x;  C(int x){ this->x=x; A::x=x+1; }  void print(){ cout<<x<<A::x<<B::x; }  };  int main(void) {  C c(1); c.print();  return 0;  } | output:121 |

Here are inheritance of three classes

Constructors start from base class in order and finally is derived class.

Destructors are in reversed order.

|  |  |
| --- | --- |
| class A{  public:A() {cout<<"A";}~A() {cout<<"a";}  };  **class B: public A{**  public: B() {cout<<"B";}~B() {cout<<"b";}  };  **class C: private B{**  public: C() {cout<<"C";}~C() {cout<<"c";}  };  int main(void) {  B b; C c;  return 0;  } | output: ABABCcbaba  **inheritance: C->B->A**  constructor storage is stack type: "last in, and first out.":  for constructor: order starts from based to derived.  "B b;" constructor: AB  "C c" constructor: ABC  for destructor, order is reversed. first c and then b  "C c" destructor: cba  "B b;" destructor: ba  Note: private A will not affect on running of constructors |
| class A{  public:A() {cout<<"A";}~A() {cout<<"a";}  };  class B{  public: B() {cout<<"B";}~B() {cout<<"b";}  };  **class C: public A, public B{**  public: C() {cout<<"C";}~C() {cout<<"c";}  };  int main(void) {  C c;  return 0;  } | output: ABCcba  **inheritance: C->A, C->B** |
| class A{  public:A() {cout<<"A";}~A() {cout<<"a";}  };  **class B: public A{**  public: B() {cout<<"B";}~B() {cout<<"b";}  };  **class C: protected A, protected B{**  public: C() {cout<<"C";}~C() {cout<<"c";}  };  int main(void) {  C c;  return 0;  } | output: AABCcbaa  inheritance: B->A, C->A,B  Note: "**class C: protected A, protected B " or** "**class C: protected B, protected A " will not raise compiling error, but should be " class C:  protected B "** |
| class A{  public: int x;  A(){x=0;}  };  class B{  public: int x;  B(){x=1;}  };  class C: public A, public B{  public: int x;  C(int x){ this->x=x; A::x=x+1; }  void print(){ cout<<x<<A::x<<B::x; }  };  int main(void) {  C c(1); c.print();  return 0;  } | output:121 |
| class B {  private:  B() { cout<<"B"; }~B() { cout<<"b"; }  };  **class C: public B{**  public: C() {cout<<"C";}~C() {cout<<"c";}  };  int main(void) {  C c;  return 0;  } | compiling error  can't set constructor as private in class B. |

### upcast and downcast

upcasting is to convert object or pointer of derived-class to base class. treat derived-class as base class.

parent\_obj = child\_obj.

downcast is to convert object or pointer of based class to derived class. downcasting is not allowed without an explicit type cast because "is-a" relationship.

child\_obj = parent\_obj is wrong.

class A{

public: void out(){cout<<"A";}

};

class B: public A{

public: void out(){cout<<"B";}

};

**class B inherit from class A. Access type is public.**

|  |  |  |
| --- | --- | --- |
| A a; a.out();  B b; b.out(); | AB | common approach.  for b, B::out() override A::out() |
| A \*pa = new A(); pa->out();  B \*pb = new B(); pb->out(); | AB | pointer approach |
| **A a = B(); a.out();** | **A** | upcast: it is ok to convert child to parent |
| **pa = pb; pa->out();**  **pa = new B(); pa->out();** | **AA** | upcast |
| **B b = A();** |  | wrong: can't convert parent to child |
| pb=pa; |  | wrong: can't convert parent to child |
| pb = (B\*) &a; pb->out();  pb = (B\*) pa; pb->out(); | BB | downcast explicitly |
|  |  |  |
|  |  |  |

|  |  |
| --- | --- |
| class A{  public: void out(void){ cout << "A"; }  };  class B: public A{  public: void out(){ cout << "B"; }  };  int main(){  A \*a = new B();  **a->out(); static\_cast<B \*>(a)->out();**  return 0;  } | output: AB  upcast  class B inherits from class A |
| class zero {public: void out(){cout<<0;}};  class one: public zero {public: void out(){cout<<1;} };  class two: public zero {public: void out(){cout<<2;} };  int main(void) {  one obj1; two obj2;  **zero \*ptr;**  ptr=&obj1; ptr->out();  ptr=&obj2; ptr->out();  **zero obj;**  obj=obj1; obj.out();  obj=obj2; obj.out();  return 0;  } | output: 0000  inheritance: zero<-one, zero<-two  pointer \*ptr or object obj is class zero. |
| class zero {public: void out(){cout<<0;}};  class one: public zero {public: void out(){cout<<1;} };  class two: public one {public: void out(){cout<<2;} };  int main(void) {  zero obj0, \*ptr0;  one obj1, \*ptr1;  two obj2, \*ptr2;  //wrong: ptr1=&obj0; ptr1->out();  ptr1=&obj2; ptr1->out();  //wrong: obj1=obj0; obj1.out();  obj1=obj2; obj1.out();  return 0;  } | output:11  inheritance: zero<-one<-two  wrong: obj1=obj0; obj1.out();  no known conversion for argument   1 from 'zero' to 'one&&' |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### errors of inheritance

1. can't convert class in initialization if the two classes have not relation

2. child class access private members of parent class

3. duplicate loading: loaded parent classes have inheritance.

4. duplicate definition of class

|  |  |
| --- | --- |
| class A{  public: void out(void){ cout << "A"; }  };  class B: public A{  public: void out(){ cout << "B"; }  };  class C: public B,A{  }; | compiling error  super class B and A can't co exists because B inherit from A |
| class A{  public: void out(void){ cout << "A"; }  };  class B{  public: void out(){ cout << "B"; }  };  class C: public A,B{  }; | compiling error  super class A and B have the same name members which are ambiguous. |
| class B{  public:  void out(){ cout << "B"; }  void toB(){ cout << "b"; }  };  class C{  public:  void toC(){ cout << "c"; }  };  int main(){  B \*b = new C();  return 0;  } | compiling error  class B and C haven't not relationship. can't convert \*C to \*B in initialization. |
| class B{  public: void out(){ cout << "B"; }  };  class C: B{};  int main(){  C \*c = new C(); c->out();  return 0;  } | compiling error: B is not accessible based of C  correct: "class C: public B{};" |
| class B{  void out(){ cout << "B"; }  };  class C: public B{};  int main(){  C \*c = new C();  c->out();  return 0;  } | compiling error: inaccessible  B::out() is private. |
| class A{  int a=3;  public:  void out(void){ cout << "out"; }  };  class C: public A{};  int main(){  C \*c = new C();  cout << c->a;  return 0;  } | compiling error: inaccessible  A::a is private. |
| class A{  public: void out(void){ cout << "out"; }  };  class A{  public: void in(){ cout << "in"; }  };  class C: public A{};  int main(){  C \*c = new C();  c->out();  return 0;  } | compiling error  previous definition of class A |
| class X{};  class Y: public X{};  class Z: public X{};  int main(){  Z \*z=new Z();  Y \*y=new Y();  z=y;  cout<< (z==y) <<endl;  return 0;  } | can't convert Y\* to Z\* in assignment  Note: objects of various class (different class and no inheritance)can't be converted |
| class A{  friend void f();  private: int field;  public:  int set(int x){return field = ++x;}  int get(){return ++field;}  };  void f(A &a){a.field /= 2;}  int main(){  A a;  a.set(2);  f(a);  cout << a.get() << endl;  return 0;  } | errro: inaccessible to private attribute field  Note: though f() is friend of class A , attribute field is private |

### access specifier

public, private, or protect

A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

public inheritance: public in base class keep public in derived class. protected remain protected. can't access private

protected inheritance: public and protected in base class are protected in derived class

private inheritance: public and protected in base class are private in derived class

*if a component is declared as public and its class is inherited as public, the resulting access is public*.

|  |  |
| --- | --- |
| class Uno{  protected: char y;  public: char z;  };  **class Due: protected Uno{**  public:  void set(){ y='a'; z='z'; }  void out(){cout << ++y << --z;}  };  int main(void) {  Due b;  b.set(); b.out();  return 0;  } | output: by  Either "class Due: protected Uno" or "class Due: public Uno" is ok in this case. |
| class A{  char a;  protected: char b;  public: char c;  A(){a='a';b='b';c='c';}  };  class B: public A{  char d;  public:  void set(){ c='e'; d='d';}  void get(){ cout << c << d;}  };  int main(void) {  B b;  b.set();   b.get();  return 0;  } | output: ed |
|  |  |
|  |  |
| class A{  public: int p;  A(){r=2;}  private: int q;  **protected: int r;**  };  **class B: private A{};**  **class C: public B{**  public: void out(void){ cout<< r;}  };  int main(void) {  C c;c.out();  return 0;  } | error: 'int A::r' is protected within this context  in Class C, can access protected r defined in grandparent class A usually. but parent of B is private A. so A.r is private for child class C. |
| class A{  char a;  protected: char b;  public: char c;  A(){a='a';b='b';c='c';}  };  class B: public A{  char d;  public: void set(){ c='e'; d='d';}  };  int main(void) {  B b;b.set();   cout << b.c << b.d;  return 0;  } | 'char B::d' is private within this context  char d in class B is private. can't be used in main() |
|  |  |
|  |  |

### virtual methods

A virtual function is a member function which is declared within a base class and is overriden by a derived class.

virtual methods are bound at execution time. non-virtual methods are bound at compiling.

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
4. The prototype of virtual functions should be the same in the base as well as derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have [virtual destructor](https://www.geeksforgeeks.org/virtual-destructor/) but it cannot have a virtual constructor.

|  |  |
| --- | --- |
| class A{  public: virtual void out(){cout << "A";}  };  class B: public A{  **public: virtual void out(){cout << "B";}**  };  class C: public A{  **public: virtual void out(){cout << "C";}**  };  int main(void) {  **A \*a = new B(),  \*b = new C();**  b->out();   a->out();  return 0;  } | output: CB  Note: inheritance: A🡨B, A🡨C  implementation inheritance: virtual methods defined  class A,B, C could act as base class  A::\*a-> B::out(), A::\*b -> C::out()  **Note: In this case, removing virtual from class B and C would not impact on outcome** |
| class A{  **public: void out(){cout << "A";}**  };  class B: public A{  **public: void out(){cout << "B";}**  };  class C: public A{  **public: void out(){cout << "C";}**  };  int main(void) {  A \*a = new B(),  \*b = new C();  b->out();   a->out();  return 0;  } | output: AA  Not virtual keyword included  A::\*a, A::\*b is upcasting |
| class A{  **public: void out(){cout << "A";}**  };  class B: public A{  **public: virtual void out(){cout << "B";}**  };  class C: public A{  **public: virtual void out(){cout << "C";}**  };  int main(void) {  A \*a = new B(),  \*b = new C();  b->out();   a->out();  return 0;  } | output: AA  virtual exist in derived class will not impact on outcomes |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| class base {  public:  virtual void print()  {  cout << "print base class" << endl;  }     void show() {  cout << "show base class" << endl;  }  };    class derived : public base {  public:  void print(){  cout << "print derived class" << endl;  }    void show(){  cout << "show derived class" << endl;  }  };    int main(){  base\* bptr;  derived d;  bptr = &d;    // virtual function, binded at runtime  bptr->print();    // Non-virtual function, binded at compile time  bptr->show();  } | output:  print derived class  show base class |
| class A{  public:  A():val(0) {}  int val;  **virtual void run(){cout<<val;}**  };  class B: public A {};  class C: public B {public:  void run(){cout<<val+2;}};  void Do(A \*a){  B \*b; C \*c;  **if(b=dynamic\_cast<B\*>(a))** b->run();  **if(c=dynamic\_cast<C\*>(a))**c->run();  a->run();  }  int main(){  **A \*a=new C();**  Do(a);  return 0;  } | output: 222  **"A \*a=new C();" is equal to**  **A\* a; C c; a=&c;**  if use dynamic\_cast, the base class A must define virtual method. because dynamic\_cast do conversion during execution time. |
| class A{  public:  A():val(0) {}  int val;  void run(){cout<<val;}  };  class B: public A {public: virtual void run(){cout<<val+2;}};  class C: public B { };  void Do(A \*a){  B \*b; C \*c;  if(b=static\_cast<B\*>(a)) b->run();  if(c=dynamic\_cast<C\*>(b))c->run();  a->run();  }  int main(){  A \*a=new C();  Do(a);  return 0;  } | output:220 |
|  |  |
| class X{public: void shout(){cout<<"X";}};  class Y: public X {public: void shout(){cout<<"Y";}};  class Z: public Y {public: void shout(){cout<<"Z";}};  int main(){  Z \*z = new Z();  z->shout();  static\_cast<Y \*>(z)->shout();  static\_cast<X \*>(z)->shout();  Y \*y = new Z();  y->shout();  static\_cast<Z \*>(y)->shout();  static\_cast<X \*>(y)->shout();  return 0;  } | output: ZYXYZX  1. z.shout() override parent. convert to Y \*, then y.shout() override z.shout() and x.shout()  2. static\_cast and dynamic\_cast are equal  3**. it is ok to convert child to parent. illegal to convert parent to child** |
| class X{public: virtual void shout(){cout<<"X";}};  class Y: public X {public: void shout(){cout<<"Y";}};  class Z: public Y {public: void shout(){cout<<"Z";}};  int main(){  X \*x = new Z();  x->shout();  dynamic\_cast<Y \*>(x)->shout();  dynamic\_cast<Z \*>(x)->shout();  cout<<endl;  Y \*y = new Z();  y->shout();  dynamic\_cast<X \*>(y)->shout();  dynamic\_cast<Z \*>(y)->shout();  cout<<endl;  X \*xx = new Y();  xx->shout();  dynamic\_cast<Y \*>(xx)->shout();  dynamic\_cast<Z \*>(xx)->shout();  cout<<endl;  return 0;  } | output:ZZZ ZZZ YY  x.shout() is virtual method. \*x or \*t is overridden by z.out(). \*xx is overridden by y.shou(). |
| **class X{public: virtual void shout(){cout<<"X";}};**  class Y: public X {public: void shout(){cout<<"Y";}};  class Z: public Y {public: void shout(){cout<<"Z";}};  int main(){  Z \*z = new Z();  z->shout();  static\_cast<X \*>(z)->shout();  static\_cast<Y \*>(z)->shout();  return 0;  } | output:ZZZ  x.shout() is virtual method. x.shout() and y.shout() is overridden by z.shout() |
| int main(){  Y \*y = new Z();  y->shout();  dynamic\_cast<X \*>(y)->shout();  dynamic\_cast<Z \*>(y)->shout();  cout << endl;  X \*x = new Z();  x->shout();  // wrong: dynamic\_cast<Y \*>(x)->shout();  // wrong: dynamic\_cast<Z \*>(x)->shout();  cout << endl;  Y \*yy = new Y();  dynamic\_cast<X \*>(yy)->shout();  // wrong: dynamic\_cast<Z \*>(yy)->shout();  cout << endl;  return 0;  } | output:ZXZ X X  y.shout() is overridden by z.shout(). convert class Y\* (children) to class X\*(parent). y.shout() is virtual method, overridden by x.shout() |
|  |  |

## overloading

**13 Overloading**

1 When two or more different declarations are specified for a single name in the same scope, that name is said to be overloaded. By extension, two declarations in the same scope that declare the same name but with different types are called overloaded declarations. Only function and function template declarations can be overloaded; variable and type declarations cannot be overloaded.

### overload attributes

|  |  |
| --- | --- |
| class A{  public:string s;  A(){s="A"; cout<<s;}  A(string s){cout<<"Astr";}  };  class B: public A{  public:string s;  B(){cout<<"Bno";}  B(string s){s=s; cout<<s;}  };  int main(void) {  B b2("good");  return 0;  } | output:Agood  when b2 is initialized, B::s overrides A::s |
| class A{  public:int x;  A(){x=0;}  A(int x){this->x=x;}  };  class B: public A{  public:using A::x;  B(){}  B(int x){this->x = x;}  };  int main(void) {  B c1, c2(5);  cout << c1.x << c2.x << endl;  return 0;  } | output:05  Note Adding "using A::x" will not impact outcome. this statement just explicitly declare inheritance |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### overloading of constructor

|  |  |
| --- | --- |
| class A{  public:  int s;  A(){cout<<"Ano";}  A(string s){cout<<"Astr";}  };  class B: public A{  public:  B(){cout<<"Bno";}  B(string s){cout<<"Bstr";}  };  int main(void) {  B b2("good");  return 0;  } | output: AnoBstr  B::B(string s) override A::A(string s) |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### overloading methods

|  |  |
| --- | --- |
| class A{  public: void out(void){ cout << "A"; }  };  class B: public A{  public: void out(){ cout << "B"; }  };  class C: public B{  public: void out(){ cout << "C"; }  };  int main(){  A a;B b;C c;  a.out();  b.out();  c.out();  return 0;  } | output:ABC  B.out() override A.out() in B  C.out() override B.out() in C |
| class A{  public: void out(void){ cout << "A"; }  };  class B: public A{  public: void out(){ cout << "B"; }  };  class C: public A{  };  int main(){  A a;B b;C c;  a.out();  b.out();  c.out();  return 0;  } | output: ABA |
| class A{  public:  void out(void){ cout << "A"; }  void toA(){ cout << "a"; }  };  class B: public A{  public:  void out(){ cout << "B"; }  void toB(){ cout << "b"; }  };  class C: public B{  public:  void toC(){ cout << "c"; }  };  int main(){  **A \*a = new B();  B \*b = new C();**  a->out(); b->out();  return 0;  } | output: AB  though new B() but the \*a is instance of class A  convert \*B to \*A, \*C to \*B in initialization. |
|  |  |
|  |  |

### overload operator

Three approach:

1. defined as class methods or templates

2. defined as function, which is not member of class

3. defined as struct methods or templates

overload bool operator==(): compare age. The approach below are equal

|  |  |
| --- | --- |
| class person{  private: int age;  public: person(int a){  this->age=a; }  // member function  **inline bool operator==(const person &ps) const{**  if (this->age==ps.age) return true;  return false;  }  };  int main(){    person p1(10), p2(20);    if(p1==p2) cout<< "Age is equal!";    else cout<< "Age is different";    return 1;  } | Age is different  operator==() is member method.  Note: "inline" could be removed. |
| class person{  public: int age;  person(int a){ this->age=a; }  bool operator==(person const &ps2);  };  // outside method  bool person::operator==(person const &ps2){  if (this->age==ps2.age) return true;  return false;  }  int main(){    person p1(10), p2(20);    if(p1==p2) cout<< "Age is equal!";    else cout<< "Age is different";    return 1;  } | operator==() is outside class methods |
| template <typename T> class person{  private: T age;  public:  person(const T &a){ this->age=a; }  bool operator==(const person &ps){  if (this->age==ps.age) return true;  return false;  }  };  int main(){  person<int> p1(10), p2(20);  if(p1==p2) cout<< "Age is equal!";  else cout<< "Age is different";  return 1;  } | operator==() is defined in template class |
| class person{  public: int age;  person(int a){  this->age=a; }  };  //operator==() function  bool operator==(person const &ps1, person const &ps2){  if (ps1.age==ps2.age) return true;  return false;  }  int main(){    person p1(10), p2(20);    if(p1==p2) cout<< "Age is equal!";    else cout<< "Age is different";    return 1;  } | operator==() is function out of class. |
| struct person{  int age;  person(int a){ this->age=a; }  bool operator==(const person &ps){  if (this->age==ps.age) return true;  return false;  }  };  int main(){  **struct person p1(10), p2(20);**  if(p1==p2) cout<< "Age is equal!";  else cout<< "Age is different";  return 1;  } | use struct. Compared with class approach, all members of struct is public. |
| template<class T> struct person{  T age;  person(const T &a){ this->age=a; }  bool operator==(const person &ps){  if (this->age==ps.age) return true;  return false;  }  };  int main(){  **person<int> p1(10), p2(20);**  if(p1==p2) cout<< "Age is equal!";  else cout<< "Age is different";  return 1;  } | build struct template |

operator+

|  |  |
| --- | --- |
| enum T { A=2, B=-1, C };  **T operator+(T t, int i){**  T C = T(2);  switch(t){  case A: return T(A);  case B: return static\_cast<T>(i);  default: return (T)1;  }  }  int main(void) {  **T i = A+2;**  cout << i << C;  return 0;  } | output:20  1. pass enum value to operator+. so T C =T(2) will not affect outer enum T. So C=0.  2. overload enum: "A+2" is illegal for enum type, but operator+ is overloaded. |
| class A{  double re, im;  public:  A(): re(1), im(1) {}  A(double r, double i): re(r), im(i) {}  **A operator+(A &); //declaration**  void out(){cout << re << im;}  };  //define operator+  **A A::operator+ (A &a){**  **A c(this->re + a.re, this->im + a.im);**  **return c;**  **}**  int main(void) {  A x(1,2), y, z;  z = x+ y; z.out();  return 0;  } | output: 23 |
| class complex{  double re, im;  public:  complex(): re(1), im(0.4) {}  complex operator> (complex &t);  void Print(){ cout << re <<" "<< im; }  };  complex complex::operator> (complex &t){  complex temp;  temp.re = this->re ? t.re : 2;  temp.im = this->im ? t.im : 0.6;  return temp;  }  int main(void) {  complex c1, c2, c3;  c3= c1 > c2;  c3.Print();  return 0;  } | output: 1 0.4 |
| class Box{  private: int length, width, height;  public:  Box(int length, int width, int height){  this->length = length;  this->width = width;  this->height = height;  }  Box(int length){  this->length = this->width = this->height = length;  }  int getVolume(void){  int v= this->length \* this->height \* this->width;  return v;  }  **int operator+(const Box &b2){**  **this->length += b2.length;**  **this->width += b2.width;**  **this->height += b2.height;**  **}**  };  int main(){  Box b1(1,2,4), b2(5);  cout << b1.getVolume() << endl;  cout << b2.getVolume() << endl;  **b1 + b2;**  cout << b1.getVolume() << endl;  return 1;  } |  |

operator ++ and operator --

// Declare prefix and postfix increment operators.

Point& operator++(); // Prefix increment operator.

// Declare prefix and postfix decrement operators.

Point& operator--(); // Prefix decrement operator.

post operator

Point operator++(int);

Point operator--(int);

|  |  |
| --- | --- |
| enum e {a=1,b,c,d};  //prefix operator  **e& operator--(e &x){ x=b; return x; }**  int main(void) {  e f=c;  **cout << int(--f) <<endl;**  return 0;  } | output:2 |
| enum e {a=1,b,c,d};  **e& operator--(e &x, int){ x=b; return x; }**  int main(void) {  e f=c;  **cout << int(f--) <<endl;**  return 0;  } |  |
| class Int{  public:  int v;  Int(int a){v=a;}  Int& operator--(){ ++v; return \*this; }  Int& operator--(int v){ v+=2; return \*this; }  };  ostream& operator<< (ostream &o, Int &a){  return o << a.v++;  }  int main(void) {  Int i=0; cout << --i << i--;  return 0;  } | output:12 |
| enum T {A=2, B=-1,C};  class Int{  public: T v;  Int(T a){ v = a; }  **Int& operator++(){ v = T(v+2); return \*this;}**  };  **ostream& operator<<(ostream &o, Int &a){**  **++a;   return o << a.v;**  **}**  int main(void) {  Int i = B;cout << i;  return 0;  } | output: 1 |

**operator<<()**

**binary operator,** should pass two reference arguments

The approaches below are equal:

|  |  |
| --- | --- |
| class Int{  public: int v;  Int(int a){v=a;}  };  ostream& operator<<(ostream &o, Int &a){  return o << a.v;  }  int main(void) {  **Int i = 1; cout << i;**  return 1;  } | output:1  2. The below is ok.  void operator<<(ostream &o, Int &a){  o << a.v++;  } |
| class Int{  private: int v;  public:  Int(int a){v=a;}  ostream& operator<<(ostream &o){  return o << this->v;  }  };  int main(void) {  **Int i = 1; i << cout;**  return 1;  } | defined as class method  Note: "i<<cout;" |
| class Int{  private: int v;  public:  Int(int a){v=a;}  **ostream& operator<<(ostream &o);**  };  **ostream& Int::operator<<(ostream &o){**  return o << this->v;  }  int main(void) {  Int i = 1; i << cout;  return 1;  } | same as the above. but move method out of class. |
| **template <typename T> class Int{**  private: T v;  public:  Int(T a){ v=a; }  ostream& operator<<(ostream &o){  return o << this->v;  }  };  int main(void) {  **Int<int> i(1); i << cout;**  return 1;  } | operator<<() defined in template class. |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
| class Int{  public: int v;  Int(int a){v=a;}  Int& operator[](int x){ v += x; return \*this;}  };  ostream& operator<<(ostream &o, Int &a){  return o << a.v;  }  int main(void) {  Int i=2;  **cout << i.v << i[2];**  return 0;  } | output: 44  **Both  "cout << i[2];" and  "cout << i.v << i[2];" are ok.** |
| class Int{  public: int v;  Int(int a){v=a;}  Int& operator++(int x){ v += 2; return \*this;}  };  ostream& operator<<(ostream &o, Int &a){  return o << a.v;  }  int main(void) {  int i=0;  **cout << i++ << i.v;**  return 0;  } | output:20  The first << is overloaded by operator<<, but the 2nd << is not. so first run i.v=0, then trigger i++->i.v=2.  if "cout << i.v << i++;", the ouput is 22. |
|  |  |
|  |  |
|  |  |
|  |  |

### syntax errors

1. only overload class or enum type operator. Built-in data type can't be overloaded.

|  |  |
| --- | --- |
| class A{  public: int v;  A():v(1) {}  A(int i): v(i) {}  **void operator\*\*(int a) { v\*=a; }**  };  int main(void) {  A i(2);  **i\*\*2;**  cout << i.v <<endl;  return 0;  } | \*\* can't be overloaded. \* is unary operator |
| enum e {a=1,b,c,d};  e& operator--(e &x){ x=b; return x; }  int main(void) {  e f=c;  cout << int(f--) <<endl;  return 0;  } | overloading should be prefix operator. |
| string operator> (float l, float r){  if( int(l) > int(r) ) return "true";  return "false";  }  int main(void) {  float l=2.0, r=2.9999;  cout << (l > r);  return 0;  } | must have an argument of class or enumerated type  It is not allowed to overload an operator for a built-in type including int, float, double, char, etc.  correct: create class ,remove global operator to class operator |
| class Int{  public: int v;  Int(int a){v=a;}  };  ostream& operator<<(Int &a){ return cout << a.v; }  int main(void) {  int i=1; cout << i;  return 0;  } | compiling error.  operator<< is binary operator. should pass two arguments  should be  ostream& operator<<(ostream &o, Int &a){  return o << a.v;  } |
| class N{  public: float x;  N(){x=0.0;}  N(float a){x=a;}  N(N& n){x=n.x;}  **N& operator<<(N &y){**  **return \*new N(x\*10);**  **}**  };  int main(void) {  N a(2.0), b(4.0);  N c = a << 1; cout << c.x;  return 0;  } | compiling error  mismatch data type: a<<1;  correct a<<b; |
| class Int{  public: int v;  Int(int a){v=a;}  **Int& operator--(){ ++v; return \*this;}**  };  ostream& operator<<(ostream &o, Int &a){  return o << --a.v;  }  int main(void) {  int i=2; cout << i;  return 0;  } | output:1  "o << --a.v" doesn't apply operator-- because a.v is int type rather than object. Soo a.v=2, --a.v, a.v=1  **The below will load operator-- : o << (--a).v;** |
| class N{  public: float x;  N(){ x = 0.0; }  N(float a){ x=a; }  N(N& n){ x=n.x; }  };  N& operator=(N &y, float f){ return \*new N(f); }  int main(void) {  N a; a = 2.0;  cout << a.x;  return 0;  } | compiling error   'N& operator=(N&, float)' must be a nonstatic member function  operator= can not be defined as global operator |
|  |  |

## composition

|  |  |
| --- | --- |
| using namespace std;  class A {  public:  void Do(void) { cout << "A is doing something" << endl; }  };  class B {  public:  void Do(void) { cout << "B is doing something" << endl; }  };  class Compo {  public:  A f1;  B f2;  };  int main(void) {  Compo co;  co.f1.Do();  co.f2.Do();  return 0;  } |  |
|  |  |
|  |  |
|  |  |

## special class

### friend class

**Friend Class** A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example, a LinkedList class may be allowed to access private members of Node.

Following are some important points about friend functions and classes:   
**1)** Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.  
**2)** Friendship is not mutual. If class A is a friend of B, then B doesn’t become a friend of A automatically.  
**3)** Friendship is not inherited (See [this](https://www.geeksforgeeks.org/g-fact-34/)for more details)  
**4)** The concept of friends is not there in Java.

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;  class A {  friend class B;  friend class C;  private: int field;  protected: void print(void) { cout << "It's a secret, that field = " << field << endl; }  };  class C {  public: void DoIt(A &a) { a.print(); }  };  class B {  public: void DoIt(A &a, C &c) {  a.field = 111; c.DoIt(a); }  };  int main(void) {  A a; B b; C c;  b.DoIt(a,c);  return 0;  } |  |
| class A{  friend class B;  int a;  public:  A(): a(1) {}  int f() { return a; }  };  class B{ public:static void f(A &a){a.a++;} };  int main(){  A a; B::f(a);  cout<< a.f()<<endl;  return 0;  } | output: 2 |
| **class B;**  class A{  **friend class B;**  int a;  public:  A(): a(4) {}  **void f(B &b, A &a);**  int out(void) { return a; }  };  class B{  **friend class A;**  int b;  public:  B(): b(2){}  **void f(A &a){a.a /=b;}**  };  **void A::f(B &b, A&a){ b.f(\*this); }**  int main(){  A a; B b;a.f(b, a);  cout<< a.out()<<endl;  return 0;  } | output: 2  Note: pay attention to order of class A and B. "Class B;" should be declared before definition.  " void A::f(B &b, A&a){b.f(\*this);}" should declared after class B. |
|  |  |
|  |  |
|  |  |

### const class

### abstract classes

Abstract methods defined in a based class provide an interface for derived classes. Abstract can not be implemented at base class. Declaring abstract is specified by placing "=0" in it declaration as follows

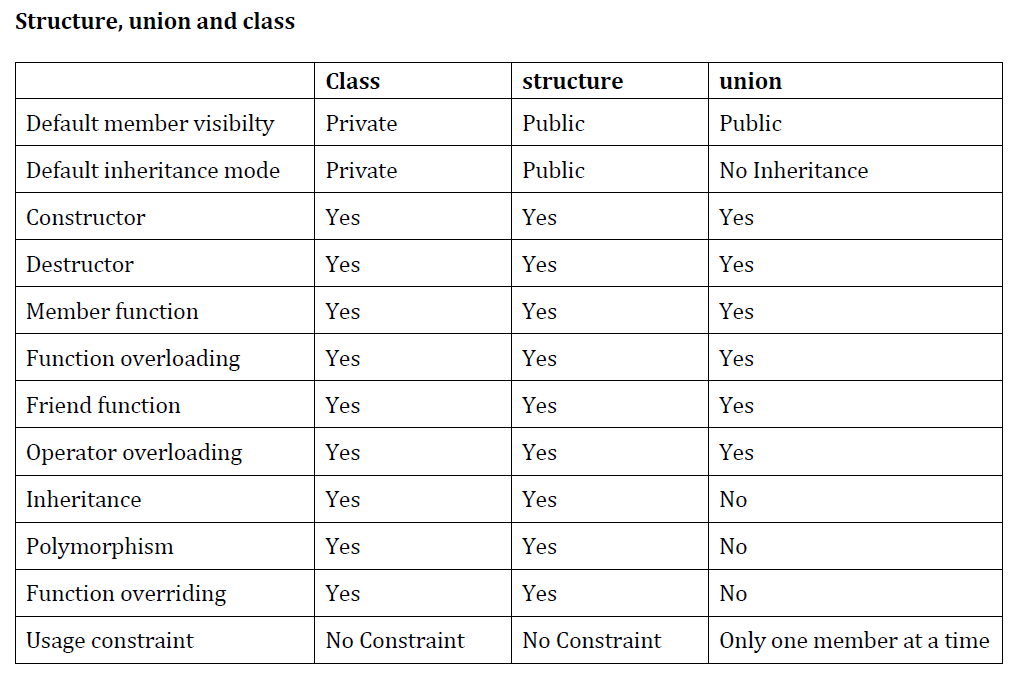
|  |  |
| --- | --- |
| class A {  public: int length;  **virtual double getVolume() = 0;**  };  class Box: public A {  public:  Box(int x){length=x;}  **double getVolume(){**  **return length\*length\*length;**  **}**  };  int main(void) {  Box b(10); cout<< b.getVolume();  return 0;  } | output: 1000 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| class A {  public: int length;  virtual double getVolume() = 0;  void print(){cout<<"A";}  };  int main(void) {  **A a;**  return 0;  } | error: cannot declare variable 'a'  to be of abstract type 'A'  class A is abstract class, which can't be implementated. |
| class A {  public: int length;  **virtual double getVolume() = 0;**  };  class Box: public A {  public:  Box(int x){length=x;}  int getVolume(){  return length\*length\*length;  }  }; | conflicting return type specified for  virtual method  abstract is double, implementation should also be double. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## function objects

### struct

In C, struct can be used. In C++, either class or struct is available. Difference of class and struct for encapsulation is their default access level. All members of struct should be public, so struct should be defined out of main(). Members (constructor and methods) of class could be defined as public, and attributes are defined as private in default.



define a struct

|  |  |
| --- | --- |
| struct str{  int t[3];  char s[3];  };  str a = {1,2,3, 'a','b','c'};  str b = {5,6,7,'x','y','z'};  cout << char(b.s[0]+a.t[0]) << int(a.s[2] - a.s[0]) << int(b.s[2]-b.s[1]); | output: y21  char(b.s[0]+a.t[0])=char('x'+1)='y'  int(a.s[2] - a.s[0])=int(3-1)=2  int(b.s[2]-b.s[1])=int('z'-'y')=1 |
| struct sct{ int t[2]; };  struct str { sct t[2]; };  str t[2] = {{0,2,4,6},{1,3,5,7}};  cout << t[1].t[0].t[1] << t[0].t[1].t[0]; | output: 34  nested struct array |
| struct T{  int v;  struct T \*next;  };  struct T t1, t2;  t2.v=2, t2.next = NULL;  t1.v=1, t1.next = &t2;  cout << t1.v << t1.next->v; | output:12 |
|  |  |
|  |  |

use struct as function object known as functor.

1. Struct should define a function-call operator(), which receive arguments and may have return. That is function-call operator.

2. Struct could be instantiated like class as object by overloading operator(). The object is known as function objects.

3. Struct could define constructor inside. additional arguments could be passed into struct objects.

|  |  |
| --- | --- |
| **struct Out{**  **//function-call operator**  **void operator() (int x){ cout << x; }**  **} obj;//obj is function objects**  int main(){  int t[]={10,5,9,6,2,4,7,8,3,1};  vector<int> v(t,t+10);  **for\_each(v.begin(), v.end(), obj);**  return 1;  } | output: 10596247831  3rd argument of for\_each() should be function. obj is object type function. |
| struct Out{  void operator() (int x){ cout << x; }  };  int main(){  int t[]={10,5,9,6,2,4,7,8,3,1};  vector<int> v(t,t+10);  //initiate function objects at calling  **for\_each(v.begin(), v.end(), Out());**  return 1;  } | output: 10596247831  same as the above |
| **typedef struct{**  **void operator() (int x){ cout << x; }**  **} Out;**  int main(){  int t[]={10,5,9,6,2,4,7,8,3,1};  vector<int> v(t,t+10);  for\_each(v.begin(), v.end(), Out());  return 1;  } | output: 10596247831  same as the above  use typedef |
| struct Out{  int val;  **Out(int val): val(val){cout << val;}**  void operator() (int x){ cout << x; }  };  int main(){  int t[]={10,5,9,6,2,4,7,8,3,1};  vector<int> v(t,t+10);  **for\_each(v.begin(), v.end(), Out(9));**  return 1;  } | 910596247831  define constructor inside struct Out. could pass additional arguments into struct objects. |
| struct Out{  ostream &out;  Out(ostream &o): out(o){}  void operator() (int x){ out << x; }  };  int main(){  int t[]={10,5,9,6,2,4,7,8,3,1};  vector<int> v(t,t+10);  for\_each(v.begin(), v.end(), Out(cout));  return 1;  } | 10596247831  pass std::cout into object |
| **template<class T>struct Out{**  **ostream &out;**  **Out(ostream &o): out(o){}**  **void operator() (const T &x){ out << x; }**  **};**  int main(){  int t[]={10,5,9,6,2,4,7,8,3,1};  vector<int> v(t,t+10);  **for\_each(v.begin(), v.end(), Out<int>(cout));**  return 1;  } | 10596247831  define struct template which accept various type. Here, struct is called with int.  **Note: must be "const T" in operator().** |
|  |  |

syntax error

|  |  |
| --- | --- |
| struct T{  int v;  struct T next;  }; | error: field 'next' has incomplete type 'main()::T'  correct: struct T{ int v; struct T \*next;}; |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## objects as elements

|  |
| --- |
| class B{  int val;  public:  B(){val=0;}  B(int v):val(v){}  int getVal() const { return val; }  int setVal(int a) { return this->val=a; }  **// find() use operator==**  bool operator==(const B &b) const {  return (this->val == b.getVal()) ? true : false;  };  **// upper\_bound() use operator<**  bool operator<(const B &b) const {  return (this->val < b.getVal()) ? true : false;  };  // sort() use operator()  bool operator()(const B &a, const B &b) const {  return (a.getVal() < b.getVal()) ? true : false;  };  //std::cout, << is unary operator here  ostream& operator<<(ostream &out){  out << this->val;  return out;  };  }; |

use vector, of which elements of vector is class B.

|  |  |
| --- | --- |
| **vector<B> v(10);**  cout << v[0].getVal(); | 0  The constructor B() is called. |
| vector<B> v;  v.push\_back(4);  cout << v[0].getVal(); | 4  The constructor B(int v) is called. |
| int a[]={1,2,3};  vector<B> v(a, a+3);  cout << v[1].getVal(); | 2  The constructor B(int v) is called. |
| int a[]={1,2,3};  vector<B> v(a, a+3);  vector<B>::iterator iter;  for(iter=v.begin(); iter!=v.end(); ++iter)  cout << iter->getVal(); | 123 |
| int a[]={1,2,3};  vector<B> v(a, a+3);  vector<B>::iterator iter;  **iter = find(v.begin(), v.end(), B(2));**  cout << iter->getVal(); | 2  for algorithm::find(), operator== should be overloaded or compile fail. The element of vector is objects of class B. |
| int t[]={8,10,5,1,4,6,2,7,9,3};  deque<B> v(t, t+10);  sort(v.begin(), v.end(), B());  deque<B>::iterator iter=v.begin();  **iter = upper\_bound(v.begin(), v.end(), B(4));**  for(iter; iter!=v.end();++iter)  cout << iter->getVal(); | 5678910  overloaded operator() is used by sort()  overloaded operator< is used by upper\_bound() |
| int t[]={8,10,5,1,4,6,2,7,9,3};  deque<B> v(t, t+10);  deque<B>::iterator iter=v.begin();  for(iter; iter!=v.end();++iter)  **\*iter << cout;** | 81051462793  overload operator<<, which is unary operator. |
| class B{  int val;  public:  B(){val=0;}  B(int v):val(v){}  int getVal() const { return val; }  int setVal(int a) { return this->val=a; }  **friend ostream& operator<<(ostream &out, const B &b);**  };  **//std::cout, << is binary operator here**  **ostream& operator<<(ostream &out, const B &b){**  **out << b.getVal();**  **return out;**  **};**  int main(){  int t[]={8,10,5,1,4,6,2,7,9,3};  deque<B> v(t, t+10);  deque<B>::iterator iter=v.begin();  for(iter; iter!=v.end();++iter)  **cout << \*iter;**  return 1;  } | compared with the above  Define operator<< as binary operator. but that should be defined as none-member function |

Here is more general example of class object, which could be applied for multiple containers.

1. constructor A(int x): initialization

2.overload operator<(): map/set insertion; sort(), lower\_bound(), upper\_bound().

3.overload operator==(): find().

4.overload operator<<(): print element

|  |
| --- |
| class A{  private: int val;  public:  A():val(0){}  A(int x):val(x){}  int getVal(){ return val; }  void setVal(int x){ val=x; }  // int operator[](const A \*b){return b.val;}  bool operator< (const A &b) const{  return (this->val < b.val);  }  };  template<class T> void operator<<(ostream &out, T &a){  out << a.getVal();  }  template<class T> void print(T start, T end){  while(start != end){  cout << \*start;  start++;  }  } |

|  |  |
| --- | --- |
|  |  |
| int main(void) {  A a[]={1,2,3}; //constructor A(int x)  cout << a[0]; // overload operator<<()  return 1;  } | for array. |
| int main(void) {  int a[]={3,1,2};  //create: constructor A(int x)  vector<A> v(a, a+3);  //access: overload operator<<()  cout << v[0]; //=3  cout << v.at(1); //=1  vector<A>::iterator iter=v.begin();  cout << \*next(iter,2); //=2  //overload operator<()  sort(v.begin(),v.end());// v=123  //for loop  for(A &b: v) cout << b; //=123  for\_each(v.begin(), v.end(), [](A &b)  { cout << b; }); //=123  // modify vector  v.emplace(v.begin(), A(0));//v=0123  v.emplace\_back(A(4)); //v=01234  v.insert(v.begin()+2, A(2));//v=012234  //overload operator==()  iter = find(v.begin(), v.end(), A(2));  cout << distance(iter, v.end()); //=4  iter = lower\_bound(v.begin(), v.end(), A(2));  cout << distance(v.begin(), iter); //=2  iter = upper\_bound(v.begin(), v.end(), A(2));  cout << distance(iter, v.end()); //=2  return 1;  } | for vector |
| int main(void) {  int t[]={3,4,2,1,6,5,7,9,8,10};  string s[] = {"three", "four", "two", "one",\  "six", "five", "seven", "nine", "eight", "ten"};  map<A, string> m;  for(int i=0; i<10; i++)  // overload operator<()  m.insert(pair<A, string>(A(t[i]), s[i]));  // overload operator<()  m.erase(m.lower\_bound(2), m.upper\_bound(7));  map<A, string>::iterator iter=m.begin();  for(;iter!=m.end();iter++)  cout<< iter->second << ',';  return 1;  } | output: one,eight,nine,ten,  Apply objects elements in map  by fault, map and set use operator< to sort key. So operator<() should be overloaded. |

## coding errors

1. object access private attribute

2. wrong data type of arguments when instantiate class.

3. initialize static variable in constructor.

4. miss semicolon end of class

5. access attribute which is not initialized.

|  |  |
| --- | --- |
| class A{ int a;};  int main(){  A a; a.a=1/2;  cout << a.a << endl;  return 0;  } | attribute of class A.a is private by default. can't access a.a directly.  correct: public: int a; |
| class A{  public: static int a;  A() {a=1;a++;}  A(A &aa) {a++;}  };  int main(){  A a, b(a), c(b);  cout << A.a << endl;  return 0;  } | Two errors:  1.static variable a can't be initialized within constructor. that should be initialized outside of class: "int A::a=1;"  2. can't directly apply "A.a". should be "A::a" or "a.a". A is class. a is instance of class A. |
| class Test {  bool isVal() const { return isVal; }  private: bool isVal;  }; | overloading is ok for functions. variable isVal can't overload the function isVal() |
| class A{  float v;  A(){v=1.0;}  A(A &a) {A::v=a.v; cout<<"1";}  ~A(){cout<<"0";}  float set(float v){ A::v=v; return v; }  float get(float v){ return A::v; }  };  int main(){  A a, \*b=new A(a),  \*c=new A(\*b);  c->get(b->get(a.set(1.0)));  delete b; delete c;  return 0;  } | All attributes and methods are private if no explicit declaration.  correct: insert "public: " before constructor. |
| class A{  public: A():val(0) {}  int val;  void run(){cout<<val;}  };  class B: public A {  public: virtual void run(){cout<<val+2;}  };  class C: public B { };  void Do(A \*a){  B \*b; C \*c;  if(b=dynamic\_cast<B\*>(a)) b->run();  if(c=dynamic\_cast<C\*>(b)) c->run();  a->run();  }  int main(){  A \*a=new C();  Do(a);  return 0;  } | cannot dynamic\_cast 'a' (of type 'class A\*') to type 'class B\*' (source type is not polymorphic)  correct: if(b=static\_cast<B\*>(a)) |
| class X{ protected: int v; };  class Y: protected X{ Y(): v(0){} };  int main(){  Y \*y = new Y();  cout << y->v;  delete y;  return 0;  } | 'int X::v' is protected within this context |
|  |  |
|  |  |

# STL: container

sequence containers: sequential means data are ordered by the order of inputs, and stored in contiguous memory. STL defines **string, vector, linked list, deque**.

association containers ( element is key-value pattern): map, multi-map, set, multi-set

container adapters: stack, queue, priority-queue

|  |  |  |
| --- | --- | --- |
|  | **action** | **Note** |
| **sequential container**:  1. data is ordered by the input order. | string  <string> | In C, string is not a built-in data type, and denoted as a char array. In C++, string is defined by class string like char vector. |
| array  <array> | the same as C. size-fixed array. can't be changed after it is initialized |
| vector  <vector> | dynamic array. allow random access. |
| deque  <deque> | double-ended queue (de-que). More like vector. |
|  |  |
| list  <list> | double\_linked list (struct) |
| forward\_list  <list> | single\_linked list (struct) |
|  |  |  |
| **associative container**:  1. store data in key-value pair.  2. values are sorted automatically. | map  <map> | 1. map only stores unique keys.  2. There is only one value for every key  3. Keys are in sorted order.Searching for an element is very fast. |
| unordered-map  <map> | keys are not ordered. |
| multimap  <map> | enable multiple identical values of a key |
| unordered-multimap  <map> | keys are not ordered. |
| set  <set> | distinct and sorted values.  All values must be in ascending or descending.  All values must be unique. |
| multiset  <set> | sorted values. but allow duplicate values |
|  |  |  |
| **underlying**  **container** | stack  <stack> | a sequential container with one end. Stack only works on the top element namely access/add/delete the top one. "Last In - First Out"  methods: push(), pop(), peek(), isFull(), isEmpty()  don't support random access |
| queue  <queue> | a sequential container with front and rear side. dequeue is to remove element from the front side. Enqueue is to add element from the rear side. "First In - First Out"  methods: push(), pop(), peek(), isFull(), isEmpty()  don't support random access |
| priority-queue  <queue> |  |
|  |  |  |
| tree | heap | a complete binary tree where all the levels of a tress are fully filled-- any parent must have exact two children. Among all nodes, values of parents are either >= or <= values of children.  min-heap: an ordered balanced binary tree. value of parent(root) is <= that of children.  max-heap: an ordered balanced binary tree. value of parent(root) is >= that of children. |
|  |  |

**Difference of array, vector, deque, and list:**

They stores sequential elements.

**Array** is lower-level data structure in C/C++. An array stores a fixed-size sequential collection of elements with same data type. The memory size is fixed, equal to data type times by size of elements. being smaller compared with vector, and more access efficient.

**Vector** is defined in the class <vector> of STL of C++. Vector is class-object based. Vector is dynamics: could insert update, delete or replace or insert elements. So vector required more memory.

Elements of array and vector are stored in contiguous memory. Array contains elements with their memory location, and access element by memory address. Vector enable direct access to certain element using subscript operator, which takes more times but are safer.

**Deque** is double-linked queue. Elements are stored consecutively. Deque supports random access, add/pop from front or back, and insert/delete from the middle.

**List** is defined in the class <list> of STL of C++. Elements are not contiguous stored, which internally works as **double-liked list** (struct type) in C. List has not index for elements. It can't allow random access, for example to access a certain element by index.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | array | vector | deque | list |  |  |
|  | static | dynamic | dynamic | dynamic |  |  |
| store elements | consecutive | consecutive | consecutive | random |  |  |
| memory usage | smallest | +- | + | ++ |  |  |
| resize | No | resize with capacity | resize with capacity | resize |  |  |
| random access | Y, fast | Y | Y | No |  |  |
| insertion | No | Y | Y | Y |  |  |
| deletion | No | Y | Y | Y |  |  |
| iterator | Y | Y | Y | Y |  |  |

## methods

The summary as showed the below will assist you keep methods in minds:

"size" is size of containers in int type.

"first" and "last" could be iterators of c.begin() and c.end().

"val"/"el" could be element of containers.

"iter" is current iterator.

"new\_iter" is new iterator returned by methods.

"p\_var" is pointer variable

**vector**

vector<int> c;

vector<int>::iterator iter;

|  |  |
| --- | --- |
| iterator | iter=c.begin(),c.end(),c.rbegin(),c.rend()  c.cbegin(),c.cend(),c.crbegin(),c.crend() |
| capacity | bool=c.empty()  size=c.size(), size=c.max\_size(), c.re\_size(size)  int=c.capacity(), c.reserve(int), c.shrink\_to\_fit() |
| element access | ref=operator[pos]  ref=c.at(pos), ref=c.front(), ref=c.back()  p\_var = c.data() |
| modifier | c.assgin(size, val)  c.push\_back(val), c.pop\_back()  **new\_iter=c.erase(iter)/c.erase(first,last)**  **new\_iter=c.insert(iter, val)/c.insert(iter, first,last)/c.insert(iter,size,val)**  new\_iter=c.emplace(iter, val), c.emplace\_back(val)  c.swap(c2)  c.clear() |
|  |  |

**list**

example:

list<int> c;

list<int>::iterator iter;

|  |  |
| --- | --- |
| iterator | iter=c.begin(),c.end(),c.rbegin(),c.rend()  c.cbegin(),c.cend(),c.crbegin(),c.crend() |
| capacity | bool=c.empty()  size=c.size(), int=c.max\_size(), c.resize(size) |
| element access | ref=c.front()  ref=c.back() |
| modifier | c.assgin(size, val)  c.push\_front(val), c.pop\_front()  c.push\_back(val), c.pop\_back()  **new\_iter=c.erase(iter)/c.erase(first,last)**  **new\_iter=c.insert(iter, val)/ c.insert(iter, first,last)/c.insert(iter,size,val)**  new\_iter=c.emplace(iter, val), c.emplace\_front(val), c.emplace\_back(val)  c.swap(c2)  c.clear() |
| operations | c.splice(iter, c2)/ c.splice(iter, first, last)  c.remove(val), c.remov\_if(val, func)  c.unique()/c.unique(func)  c.merge(c2)  c.sort()/c.sort(func)  c.reverse() |

**map**

example:

pair<map<char, int>::iterator, bool> ret;

map<char, int>::iterator iter;

pair< map<char, int>::iterator, map<char, int>::iterator> range;

|  |  |
| --- | --- |
| iterator | c.begin(),c.end(),c.rbegin(),c.rend()  c.cbegin(),c.cend(),c.crbegin(),c.crend() |
| capacity | c.empty()  c.size(), c.max\_size() |
| element access | val=operator[key]  val=c.at(key) |
| modifier | ret=c.insert(<pair>)  size=c.erase(key), c.clear()  ret=c.emplace(key,val), ret=c.emplace\_hint(iter, key, val)  c.swap(c2) |
| operations | **iter=c.find(<key>)**  c.count(key) return 1 or 0  iter=c.lower\_bound(key), iter=c.upper\_bound(key)  range=c.equal\_range(key) |
|  |  |

stack

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Major learning points, of which the most are described in chapter of containers and algorithm. Here, I just list general points for better memory.

|  |  |
| --- | --- |
| string | 1. encompassed by "" rather than ''. 2. string ends with '\0'. |
| constructor |  |
| iterator | str.begin(), str.end(), str.rbegin(), str.rend()  str.cbegin(), str.cend(),str.crbegin(), str.crend(), |
| capacity | return int size: str.size()/str.length(), str.max\_size(), str.resize()  return storage space: str.capacity(), str.reserve(), str.shrink\_to\_fit()  clear and test: str.clear(), str.empty() |
| element access | return reference  overload operator[]: str[pos]  str.front(), str.back(), str.at(pos) (or throw exception of out\_of\_range) |
| modify | return reference  str.assign(<int>, <char>), str.swap(str2)  str.append(<str>), str.push\_back(<char>)  str.insert(pos1, str2, <str2-pos1>,<str2-pos2>)  str.replace(pos1, pos2, str2, <str2-pos1>,<str2-pos2>)  str.erase(pos1, <pos2>), str.pop\_back() |
| string operations | string \*p = str.data()  str.copy(str2, <pos1>,<pos2>)  bool f = str.compare(pos1, pos2, str2, <str2-pos1>,<str2-pos2>)  string sub = str.substr(pos, <length>)  auto pos = str.find(<string/char>), str.rfind()  str.find\_first\_of(str2, <str2-pos1>,<str2-pos2>), str.find\_last\_of()  str.find\_first\_not\_of(),str.find\_last\_not\_of() |
| overload operators | concatenate: str1 += st2  compare: str1 == str2, str1 >=str2, str1 !=str2  stream: istringstream >> str, ostringstream<< str, getline(cin,str) |

## Create

For creating a new container, usual for patterns are followed no matter what the container is. For example, String is defined by class string. declare a string is equal to instantiate object of class string. So Value assignment to string object, or the use of constructor are acceptable.

copy constructor: **string s="ABC";**

create empty object and then assign it. : **string s; s="ABC";**

initializer list: **string s "ABC";**

create object: **string s("ABC");**

initial identical elements: **c.assign(<number>, <value> )**.

|  |  |  |
| --- | --- | --- |
| **action** | **code** | **Note** |
| array | int arr[3]={1,2,3}; //int  float arr[]={.4,.7,10.3}; //float  bool arr[]={1,1,true,false}; // bool  char arr[]={'A', 'B', '\0'};  char arr[]="AB";// same as the above  char arr[]={"ABC"};//same as the above | integer, float, char, bool, string, array |
|  | int row=2, col=3;  **int \*\*p = new int \*[row];**  for(int i = 0; i < row; i++){  **p[i] = new int[col];**  for(int j = 0; j < col; j++){  **p[i][j] = i+j;**  cout << p[i][j];  }  cout << endl;  } | 2D array: 012, 123 |
|  | int \*b=NULL;  b = new int [1];  b[0]=3;  cout << \*b << endl; |  |
|  |  |  |
| string | #include <string>  string s1 = "ABC"; //declare and initialize  string s2; s2="ABC"; //declare then initialize  string s3("ABC"); // constructor pattern  string \*p; p = new string("pizza"); //pointer pattern |  |
| string sky;  **sky.assign(4, '\*');** | sky=\*\*\*\* |
| vector | #include <vector>  vector <int> v1 = {1, 2, 3};  vector <int> v2 {1, 2, 3};  vector <int> v3(3); v3={1, 2, 3};  vector <int> v4; v4={1, 2, 3}; | vector is dynamic array  2. "vector <int> v2();" is wrong |
|  | vector<int> v;  v.assign(3, 3); | v=333 |
| list | #include <list>  list <int> mylist = {1,2,3};  list <int> mylist {1,2,3};  list <int> mylist; mylist.push\_back(1);  list <int> mylist(10, 0);//size=10. all is 0.  list <int> mylist(); | "list <int> mylist[4] = {1,2,3};  " is wrong. |
| deque | #include <deque>  deque <int> d = {1,2,3};  deque <int> d {1,2,3};  deque <int> d; d.push\_back(1);  deque <int> d(); //empty deque  deque <int> d(10);//size=10  deque <int> d(10, 0);//size=10,values=0 |  |
| vector<int> v={1,2,3};  deque<int> d(v.begin(),v.end());//d=123  deque<int> d(v.rbegin(),v.rend());//d=321  deque<int> d; d.insert(d.begin(), v.begin(), v.end()); | convert vector to deque |
| int a[]={1,2,3};  deque<int> d(a, a+3); | convert array to deque |
| pair | #include <utility>  pair<char, int> p = {'a',1};  pair<char, int> p {'a',1};  pair<char, int> p, p= {'a',1};  pair<char, int> p, p.first = 'a', p.second=1; |  |
| map | #include <map>  map <char, int> m {{'a',3}};  map <char, int> m = {{'a',3}};  map <char, int> m{};//empty  map <char, int> m, m['a']=4;  map <char, int> m, m.insert(pair<char, int> p {'d',4}); |  |
| multi-map | #include <map>  typedef multimap<const char, int> myType;  myType m;  m.insert(myType::value\_type('a',2));  m.insert(myType::value\_type('a',3)); |  |
| multimap<char, int> m {{'b',2}, {'a',1},{'a',2}, {'d',4}};  multimap<char, int> m2(m);  multimap<char, int> m2=m; |  |
| set | #include <set>  set<int> s = {1,2,3}; | s=123  wrong: set<int> s {1,2,3}; |
| set<int> s; s.insert(1); | s=1 |
| int a[]={1,2,3};  set<int> s; s.insert(a[0]); | s=1 |
| int a[]={1,2,3};  set<int> s; **s.insert(a, a+3);** | s=123  array to set |
| vector<int> v={1,2,3};  set<int> s(v.begin(),v.end()); | s=123  vector to set |
| set<int> s;  s.insert(1); s.insert(1);  cout << s.size(); | 1  all elements are unique. |
| set<int> s;  s.insert(2); s.insert(1);  cout << \*s.begin(); | 1  All elements are sorted automatically |
| multi-set | #include <set>  multiset<int> s = {1,1,3};  multiset<int> s; s.insert(1);  int a[]={1,1,3}; multiset<int> s; s.insert(a[0]);  vector<int> v={1,1,3};  multiset<int> s(v.begin(),v.end()); |  |
|  |  |  |
| stack | #include <stack>  //brace-enclosed initializer is not working.  **stack<int> st;**  st.push(3); cout << st.top();  //copy constructor of another stack  **stack<int>st2(st);** | stack is "last-in, first-out" (LIFO). The two basic operations is PUSH (put an element on the top), and POP (remove the top element). |
| vector<int> v={1,2,3};  stack<int, vector<int>> st(v);  //wrong: stack<int> st(v); | st=321  convert vector to stack |
| deque<int> v = {1,2,3};  stack<int, deque<int>> st(v); | st=321  convert deque to stack |
| list<int> v = {1,2,3};  stack<int, list<int>> st(v); | st=321  convert list to stack |
|  |  |
| queue | #include <queue>  queue<int> q;  q.push(3); q.push(4);  cout << q.front(); << q.back(); | 3  create empty queue and push one by one. |
| deque<int> d (3,100);  queue<int> q(d);  cout << q.front(); | 3  convert deque to queue |
| priority-queue |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**create vector**

vector is dynamic array, so don't need to specify size when declaring it.

initialize a int vector

|  |  |
| --- | --- |
| vector <int> v;  v.push\_back(20); | int vector |
| vector <int> v(3); | int vector with size=3, value=0 |
| vector <int> v(3, 10); | int vector with size=3, value=10 |
| vector <int> v{1,2,3}; | int vector with various values |
| vector <int> v={1,2,3}; | the same as the above |
| vector<int> v(3);  fill(v.begin()+1,v.end(), 10); | int vector with size=3, and update the values of 2nd-end with 10 |
|  |  |

|  |  |
| --- | --- |
| vector <int> a={1,2,3};  cout << a[0]; | integer type vector |
| vector <char> a={'A','B'};  cout << a[1]; | char type vector |
| vector <string> a={"ABC", "DEF"};  cout << a[1]; | string type vector |
| vector <float> t;  t = {3.,2.,1.}; |  |
| vector <float \*> t = {new float[1], new float[1], new float[1] };  for(int i=0; i<t.size(); i++){  float \*p = t[i];  \*p=i;  }  cout << \*t[0] << \*t[1] << \*t[2] << endl;  for(int i=0; i<t.size(); i++){  delete [] t[i];  } | float pointer array type |
| vector <char> t(5) | it is ok to specify size |

## Get

1. overload operator[]:

Random access by index in array, string, vector.

Access value by key in map/multimap.

2. Iterator: array, string, vector, map, set

\*c.begin() return the first element

\*c.rbegin()

\*c.crbegin()

\*(c.begin()+n) return the (n+1)th element

\*(c.end()-1) returns the last element

\*c.rend()

\*c.crend()

\*next(c, n) return the (n+1)th element

<container>::iterator iter=c.begin(); advance(iter,n); \*iter; get the (n+1)th element

3. reference:

For array, string, vector, deque:

c.front(): return reference to the first element

c.back(): return reference to the last element

c.at(<n>): return reference to the element at position n.

For list:

c.front(): return reference to the first element

c.back(): return reference to the last element

For forward\_list:

c.front(): return reference to the first element

For map, multimap, unordered\_map

c.at(<key>): return reference to the element with the key.

For stack

stack.top() return reference to the top element

For queue

queue.front(): return reference to the first element

queue.end(): return reference to the last element

|  |  |  |
| --- | --- | --- |
| **action** | **code** | **Note** |
| string  #include <string>  string s = "abc"; | cout << s[0]; | a |
| cout << s[3] <<s[-1]; | no output if index is out of bounds or is negative. |
| cout << \*s.begin()<< \*(s.begin+1); | ab. |
| cout << \*(s.end()-1); | c. get the last element.  **Note: \*s.end() return '\0'** |
| cout << \*next(s.begin(),2); | c. |
| cout << \*next(s.end(),-2); | b |
| string::iterator iter=s.begin();  advance(iter, 2); cout << \*iter; | c.  Note: advance() has no returns |
| cout << s.front() << s.back(); | ab |
| cout << s.at(2); | c |
|  |  |
| vector  #include <vector>  vector<int> v = {1,2,3}; | cout << v.front() << v.back(); | 13, return reference |
| cout << v.at(2) | 3 |
| cout << \*v.data(); | 1, return pointer |
| cout << \*v.begin(); | 1, return iterator |
| cout << \*(v.begin()+1); | 2 |
| cout << \*next(v.begin(),1); | 2 |
| cout << \*next(v.end(),-2); |  |
| cout << \*(v.end()-2); | 2 |
| vector<int>::iterator iter = v.begin();  advance(iter, 1); cout << \*iter; | 2 |
| cout << v.at(3);  cout << v[6]; | runtime error: out\_of\_range  or unpredictable value |
|  |  |
|  |  |
| deque |  |  |
| list |  |  |
| forward\_list |  |  |
|  |  |  |
|  |  |  |
| map  #include <map>  map<char, int> m = {{'c',3}, {'a',1}, {'b',2}}; | cout << m['b']; | output: 2 |
| cout << m.at('b'); | 2 |
| map<char, int>::iterator iter = m.begin();  cout<< iter->first << iter->second; | a3 |
| map<char, int>::iterator iter = m.begin();  advance(iter, 1);  cout<< iter->first << iter->second; | b2  **Note: map is sorted by key even if the input is in disorder.** |
| cout << m['d']; | output:0. the key 'C' does not exist |
| multi-map  #include <map> | #include <map>  multimap<char, int> m = {{'v',3}, {'a',1}, {'b',2}};  cout << m.at('b'); | runtime error  multimap can’t use operator[] and c.at() |
| stack  #include <stack> | stack<int> st;  st.push(3);  st.push(2);  cout << st.top(); | 2 |
| queue  #include <queue> | queue<int> q;  q.push(3);  q.push(5);  q.push(7);  q.push(2);  cout << q.front() << q.back(); | 32 |
| deque  #include <deque>  deque<int> d={1,4,2,3,6}; | cout << d[3]; | 3 |
| cout << d.front() << d.back();  cout << d.at(2); | 162 |
| cout << \*d.begin() << \*(d.end()-1); | 16 |
| cout << \*next(d.begin(),2); | 2 |
| cout << \*next(d.end(),-2); | 3 |
| deque<int>::iterator iter=d.begin();  advance(iter,2);  cout << \*iter; | 2  Note:advance() has not return |
| deque<int>::iterator iter=d.begin()+2;  advance(iter,-2);  cout << \*iter; | 1 |

## find

**String and associative containers:**

For string:

**pos=str1.find(str2/char, <beginning index>)**. return index if succeeds or string::npos if finding fails.

**pos=str1.find\_first\_of(str2)**. return index if the first occurrence of any one of characters in str2 is detected in str1. If nothing is matched, return -1.

For map, multimap:

iter=map.find(<key>). Return iterator of pair. Return map.end() if nothing is detected.

**pair=c.equal\_range(<key>)**. returns pair of iterators with lower-bound and upper-bound iterators. Get the certain pair and the next pair in the container. sorting is not required when creating associative containers.

pair=c.lower\_bound(<key>) return iterator

pair=c.upper\_bound(<key>) return iterator

For set, multiset

iter=set.find(<value>). Return iterator of the last occurrence. Return set.end() if nothing is detected.

**pair=c.equal\_range(<key>)**. returns pair of iterators with lower-bound and upper-bound iterators.

pair=c.lower\_bound(<value>) return iterator

pair=c.upper\_bound(<value>) return iterator

|  |  |  |
| --- | --- | --- |
| **action** | **code** | **Note** |
| string | string a = "abcde", b = "de";  **cout << a.find(b);** | 3 |
| string a = "GGGAAAAGTC", b = "GAA";  cout << a.find(b + string(b.rbegin(), b.rend())); | 2 |
| string a = "abcde";  cout << a.find('c'); | 2 |
| string a = "abcdebc", b = "bc";  **cout << a.find(b, 4);** | 5 |
| string a = "abcdebc", b = "dd";  auto pos = a.find(b);  cout << (pos==string::npos); | 1 |
| string s = "ATXGCNGTI", sub = "INX";  **int p = s.find\_first\_of(sub);**  cout << p; | p=2 |
| string s = "AT", sub = "INX";  int p = s.find\_first\_of(sub);  cout << p; | p=-1  nothing detected. |
|  |  |
| map | map<char, int> m {{'b',2}, {'c',-2}, {'e',4}};  map<char, int>::iterator iter;  iter = m.find('e');  cout << iter->first << iter->second; | e4 |
| map<char, int> m {{'b',2}, {'c',-2}, {'e',4}};  map<char, int>::iterator iter;  **iter = m.find('d');**  cout << (iter==m.end()); | 1  return iterator to c.end() if no found. |
| map<char, int> m {{'b',2}, {'c',-2}, {'e',4}};  **map<char, int>::iterator iter;**  **iter = m.lower\_bound('d');**  cout << iter->first << iter->second; | e4  The key 'd' doesn't exit. return the neighboring larger one. |
| map<char, int> m {{'b',2}, {'c',-2}, {'e',4}};  **map<char, int>::iterator iter;**  **iter = m.upper\_bound('c');**  cout << iter->first << iter->second; | d4 |
| map<char, int> m {{'b',2}, {'a',1},{'c',-2}, {'d',4}};  **pair<map<char, int>::iterator, map<char, int>::iterator> range;**  **range = m.equal\_range('b');**  cout << range.first->first << range.first->second;  cout << range.second->first << range.second->second; | b2c-2 |
| multi-map | multimap<char, int> m {{'a',2}, {'a',1}, {'a',4}};  multimap<char, int>::iterator iter = m.find('a');  cout << iter->first << iter->second; | a2 |
| multimap<char, int> m {{'a',2}, {'a',1}, {'a',4}};  multimap<char, int>::iterator iter = m.find('d');  cout << (iter== m.end()); | 1 |
| multimap<char, int> m {{'b',2}, {'a',1},{'a',2}, {'a',4}};  **pair<multimap<char, int>::iterator, multimap<char, int>::iterator> mypair;**  **mypair = m.equal\_range('a');**  multimap<char, int>::iterator iter = mypair.first;  for(iter; iter!=mypair.second; ++iter)  cout << iter->first << iter->second; | a1a2a4 |
| set | set<int> st {1,2,3,5,6,7};  set<int>::iterator lb = st.lower\_bound(2);  set<int>::iterator ub = st.upper\_bound(6);  set<int>::iterator iter = lb;  for(iter; iter != ub; ++iter)  cout << \*iter; | 2356 |
| set<int> st {4,1,2,3};  pair<set<int>::iterator, set<int>::iterator> mypair;  mypair = st.equal\_range(3);  cout << \*mypair.first << \*mypair.second; | 34 |
| multi-set | multiset<int> m {1,4,2,3,2};  for(int i: m) cout<< i;  multiset<int>::iterator iter = m.find(2);  cout << ',' << \*iter; | 12234,2 |
| multiset<int> m {1,4,2,3,2};  multiset<int>::iterator iter = m.find(6);  cout << (iter==m.end()); | 1 |
| multiset<int> st {4,2,1,2,3,2,};  multiset<int>::iterator lb = st.lower\_bound(2);  multiset<int>::iterator ub = st.upper\_bound(2);  set<int>::iterator iter = lb;  for(iter; iter != ub; ++iter)  cout << \*iter; |  |
| multiset<int> st {4,2,1,2,3,2,};  pair<set<int>::iterator, set<int>::iterator> mypair;  **mypair = st.equal\_range(2);**  set<int>::iterator iter = mypair.first;  for(iter; iter != mypair.second; ++iter)  cout << \*iter; | 222  mypair.second=3 |
| multiset<int> st {3,4,1,1,6,5,7,9,8,6};  pair<set<int>::iterator, set<int>::iterator> range;  range = st.equal\_range(6);  while(range.first != range.second){  cout << \*range.first;  range.first++;  } | 66  Note: range.first=6, range.second=7 |

## Modify

insert, add/append, delete, replace/update

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |
|  | array | **string** | vector | list | **forward**  **\_list** | stack | queue | deque | map | set |
| c.swap(c2) | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| return iterator/pair |  |  |  |  |  |  |  |  |  |  |
| iter.erase() | - | **Y** | Y | Y | **erase\_after** | - | - | Y | Y | Y |
| insert() | - | **Y** | Y | Y | **insert\_after** | - | - | Y | Y | Y |
| emplace() | - | **replace** | Y | Y | **-** | **Y** | **Y** | Y | Y | Y |
| emplace\_front() | - | **-** | - | Y | **Y** | - | - | Y | - | - |
| emplace\_back() | - | **-** | Y | Y | **Y** | - | - | Y | - | - |
| emplace\_hint() | - | **-** | - | - | **-** | - | - | - | Y | Y |
|  |  |  |  |  |  |  |  |  |  |  |
| no return: |  |  |  |  |  |  |  |  |  |  |
| c.push\_front(val) | -- | -- | -- | Y | Y | c.push | c.push | Y | -- | -- |
| c.push\_back(val | -- | Y  append(val) | Y | Y | -- |  |  | Y | -- | -- |
| c.pop\_front(val) | -- | -- | -- | Y | Y | c.pop | c.pop | Y | -- | -- |
| c.pop\_back(val) | -- | Y | Y | Y | -- |  |  | Y | -- | -- |

**replace:**

str.replace(str.begin(), str.end(), str2, str2.begin(), str2.end())

str.replace(str-pos1, str-pos-2, str2, <str2-pos1>, <str2-pos2>)

|  |  |  |  |
| --- | --- | --- | --- |
|  | c.insert() | c.emplace()  c.emplace\_front()  c.emplace\_back()  c.emplace\_hint() | c.erase() |
|  | new\_iter=c.insert(iter, <el>)/c.insert(iter, first, last)/c.insert(iter, size, <el>).  Make a copy of container instance.  Return the iter to the new element | construct a new container instance.  Insert or append a new element into the place.  c.emplace(iter, val).  c.emplace\_front(val). Add to the front.  c.emplace\_back(val). Append to the end. | new\_iter=c.erase(iter)/ c.erase(first, last) |
| string | ref=c.insert(pos, char)  ref=c.insert(pos, char, size)  ref=c.insert(pos, char [])  ref=c.insert(pos, str)  ref=c.insert(pos, str, str-pos1, str-pos3)  iter=c.insert(iter, char)  void c.insert(iter, size, char)  void c.insert(iter, first, last) | ref=c.replace(pos, len, char[])  ref=c.replace(pos, len, char[], char[]-index)  ref=c.replace(pos, len, str)  ref=c.replace(pos, len, str, str-pos1, str-pos2)  ref=c.replace(pos, len, size, char)  ref=c.replace(first, last, str)  ref=c.replace(first, last, char[])  ref=c.replace(first, last, char[], char[]-index)  ref=c.replace(iter1,iter2, size, char) | ref=c.erase(pos)  ref=c.erase(pos, len)  iter=c.erase(iter)  iter=c.erase(first, last) |
| vector | iter=c.insert(iter, val)  void c.insert(iter, size, val)  void c.insert(iter, first, last) | c.emplace(iter, val)  c.emplace\_back(val) | iter=c.erase(iter)  iter=c.erase(first, last) |
| deque | iter=c.insert(iter, val)  void c.insert(iter, size, val)  void c.insert(iter, first, last) | c.emplace(iter, val)  c.emplace\_front(val)  c.emplace\_back(val) | iter=c.erase(iter)  iter=c.erase(first, last) |
| queue | - | c.emplace(val) | - |
| stack | - | c.emplace(val) | - |
| list | iter=c.insert(iter, val)  void c.insert(iter, size, val)  void c.insert(iter, first, last) | c.emplace(iter, val)  c.emplace\_front(val)  c.emplace\_back(val) | iter=c.erase(iter)  iter=c.erase(first, last) |
| forward\_list | iter=c.insert\_after(iter, val)  iter=c.insert\_after(iter, size, val)  iter=c.insert\_after(iter, first, last)  iter=c.insert\_after(iter, list2) | c.emplace\_front(iter, val)  c.emplace\_back(iter, val) | iter=c.erase\_after(iter)  iter=c.erase\_after(first, last) |
| map | pair<iter, bool>=c.insert(pair)  iter=c.insert(iter, pair)  void c.insert(first, last) | pair<iter, bool>=c.emplace(key, val)  iter=c.emplace\_hint(iter, key, val) | int=c.erase(key)  void c.erase(iter)  void c.erase(first, last) |
| multimap | iter=c.insert(pair)  iter=c.insert(iter, pair)  void c.insert(iter, first, last) | iter=c.emplace(key, val)  iter=c.emplace\_hint(iter, key,val) | int=c.erase(key)  void c.erase(iter)  void c.erase(first, last) |
| set | pair<iter, bool>=c.insert(val)  iter=c.insert(iter, val)  void c.insert(first, last) | pair<iter, bool>=c.emplace(val)  iter=c.emplace\_hint(iter, val) | int=c.erase(val)  void c.erase(iter)  void c.erase(first, last) |
| multiset | iter=c.insert(val)  iter=c.insert(iter, val)  void c.insert(first, last) | iter=c.emplace(val)  iter=c.emplace\_hint(iter, val) | int=c.erase(val)  void c.erase(iter)  void c.erase(first, last) |

Note: compare c.insert() with c.emplace()

c.insert() copy of a containers before insertion. c.emplace() construct instance before insertion.

c.insert() could insert container or one element. c.emplace() insert one new element only

c.insert() return iterator to the next element. c.emplace() has no return.

### string

string s= "ABC";

|  |  |  |
| --- | --- | --- |
|  | code | Note |
| push to the first | s='a'+s;  s="abc"+s;  s.insert(0, "abc"); | overload operator+ |
| append the last | s.push\_back('a');  s.push\_back('abc'); | s="ABCa" |
| s +='a';  s+= "abc"; | overload operator+ |
| s.append("abc"); | output:ABCabc |
| s.append(2, 'a'); | output: ABCaa |
| insert string | s.insert(2, "abc"); | output:ABabcC |
| string a = "a";  string b = a+"b";  b=a+"b"+b;cout<< b; | output: abab  use + to concatenate strings. |
| string s = "A" + "B"; | wrong: concatenate operator + can't work for two literals. At least one if variable |
| s.insert(2, 'a'); | compiling error  should be string type |
| string a = "a";  string b=a+'2'; | b is a2  concatenate character |
| replace | string s("GAATTC"); string ss;  **s.replace(0, 3, "TCC");** | s=TCCTTC |
| string s("usr/bin/test.text");  **s.replace(s.find('.'), 5, "");** | s=usr/bin/test  remove path extension |
| string s("a,b,c,");  **s.replace(s.begin(), s.begin()+3, "#");** | s=#,c, |
| string line = "this@ is@ a test string!";  char\* str = "12345";  //char array  **line = line.replace(0, 9, str, 4);** | line= 1234 a test string!  Note: str should be char array. string type will cause compiling error. |
|  |  |
| string s="abc"; string s2="123";  **s.swap(s2);** cout << s << ","<< s2; | 123,abc |
| string a ="ABCDEF", b;  b = a.substr(1,1)+a.substr(4)+a.substr();  a = b.substr(1,1)+b.substr(4)+b.substr(); | b=B+EF+ABCDEF  a=E+BCDEF+BEFABCDEF |
| c.erase();  string s = "ABC"; | s.erase(0,2); | delete the first two |
| **s.erase(s.begin(), s.end());** | remove all characters |
| s.erase(s.begin(), s.begin()+2); | remove first 2 characters |
| s.erase(s.end()-2, s.end()); | remove last 2 characters |
| s.erase(s.begin()+2); | remove the 3rd character |
| is empty | string s = "";  cout << s.empty(); | output:1  check if empty |
| delete all | string s="ABC";  **s.clear();**  cout << s.empty(); | 1  remove all characters |
| delete the last | string s="ABC";  **s.pop\_back();** | s=AB |
|  |  |  |
|  |  |  |

codes

|  |  |
| --- | --- |
| string quote = "Whyserious?", anyword = "monsoon";    quote.insert(3, 2, ' ').insert(4, anyword, 3, 2);    cout << quote; | output: Why so serious? |
| string s="AB";  s.append(s).push\_back(s[s.length()-1]);  cout<<s; | ABABB |
| string s1[]={"A", "Z"}, s="";  for(int i=0; i<2;i++)  cout << s.append(s1[i]).insert(1,"\_"); | output: A\_A\_\_Z |
| string to\_do = "I'll think about that in one hour";  string schedule = "today yesterday tomorrow";  **to\_do.replace(22, 12, schedule, 16, 8);** | to\_do=I'll think about that tomorrow  compared with the above |

### vector

Example: vector <int> v={1,2,3};

|  |  |  |
| --- | --- | --- |
|  | code | output |
| push to the first | vector<int>::iterator iter;  iter=v.insert(v.begin(),0);  cout << \*iter; | 0  v=0123 |
| append the last | v.push\_back(4); | v=1234 |
| vector<int>::iterator iter;  iter=v.insert(v.end(),4);  cout << \*iter; | 4 |
| insert one element | v.insert(v.begin()+1, 4); | output: 1,4,2,3 |
| insert some elements | v.insert(v.begin()+1, {4,5}); | output: 1,4,5,2,3 |
| vector<int> v2 {4,5};  vector<int>::iterator iter=v.begin();  iter=v.insert(iter,v2.begin(),v2.end());  for(int& i: v) cout << i; | v=45123 |
| vector<int>::iterator iter=v.begin();  iter=v.insert(iter,3, 8);  for(int& i: v) cout << i; | 888123 |
| concatenate two vectors | v.insert(v.end(), {4,5}); | 12345 |
| vector <int> v2={4,5};  v.insert(v.end(), v2.begin(), v2.end()); | 12345 |
|  |  |
| insert single element in place | vector<int> v {1,4,6,8};  v.emplace(v.begin()+2, 5);  for(int& i: v) cout << i; | v=14568  insert in the middle |
| vector<int> v {1,4,6,8};  v.emplace(v.end()-1, 5) | v=14658 |
| vector<int> v {1,4,6,8};  v.emplace(v.begin(), 1); | v=11468  insert on the first |
| vector<int> v {1,4,6,8};  v.emplace(v.end(), 1); | v=14681  push back |
| vector<int> v {1,4,6,1};  v.emplace\_back(1); | v=14611  push back |
| vector<int> v {1,4,6};  v.emplace(v.begin()+5, 1); | runtime error  out of bounds |
| swap two vectors | vector<int> v {1,2};  vector<int> v2 {3};  **v.swap(v2);** cout<< v[0]; | 3 |
| delete | v.erase(v.begin()+1); | v=13 |
| vector<int> vec{0,1,2,3,4,5,6};  vec.erase(vec.begin()+1,vec.end()-2); | delete 2nd-5nd elements. The vector is 056. |
| vector<int>::iterator iter;  iter = v.begin();  iter=v.erase(iter);  cout << \*iter; | 2 |
| vector <string> s = {};  cout << s.empty(); |  |
| delete the last | vector<int> v {1,2,3};  **v.pop\_back();**  for(int& i: v) cout<< i; | 12 |
| delete all | vector<int> v {1,2,3};  v.clear(); cout << v.empty(); | 1 |
| vector<int> vec{0,1,0,0,5,0};  vector<int>::iterator iter;  for(iter=vec.begin(); iter!=vec.end();){  if(\*iter==0){  // iterator point to the next if succeed  iter=vec.erase(iter);  cout << \*iter<<endl;  }else{ iter++; }  } | delete all elements with 0. the vector is 1256.  Note: after erase() used, the iterator should be redirect to the beginning or returns to iter. Otherwise iter would be "wild pointer". |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### deque

#include <deque>

deque<int> d = {1,2,3};

|  |  |  |
| --- | --- | --- |
| insert  one or more elements | deque<int>::iterator iter;  iter=d.insert(d.end(),4);  d.push\_back(4); // the same | d=1234  iter->4; |
| d.insert(d.begin(), 0);  d.push\_front(0); // the same | d=0123 |
| d.insert(d.rbegin().base(),4);  d.insert(d.rbegin(),4);//wrong | d=1234  **d.rbegin().base()=d.end()** |
| d.insert(d.end(), 2, 4); | d=12344 |
| d.insert(d.begin()+2, d.begin(), d.end()); | d=121233 |
| d.insert(d.end(), d.rbegin()+1, d.rend()); | d=12321 |
| d.insert(d.end(), d.rbegin()-1, d.rend()); | d=1230321 or runtime error due to out of ranges |
| emplace one element | d.emplace(d.end(), 4); | d=1234 |
| d.emplace(d.rbegin().base(), 4); | d=1234 |
| d.emplace\_front(0); | d=0123 |
| d.emplace\_back(4); | d=1234 |
| erase | d.erase(d.begin());  d.pop\_front(); //the same | d=23, delete the first |
| d.erase(d.begin(), d.end());  cout << d.empty(); | 1, delete all |
| d.erase(d.end());  d.pop\_back(); // the same | d=12, delete the last |
| d.erase(d.end()-2, d.end()); | d=1, delete the last two |
|  |  |  |
|  |  |  |

### map/multimap

c.insert() and c.emplace() will not work if the key exists already.

map

|  |  |  |
| --- | --- | --- |
| insert new pair | map <char, int> m = {{'a',1}};  **m['b']=2;** | {'a':1, 'b':2} |
| map<char, int> m {{'b',2}, {'a',1}};  m['c']=0; cout << m['c']; | 0  inert one |
| map <char, int> m = {{'a',1}};  **m.insert(pair<char, int> ('b',2));** | {'a':1, 'b':2} |
| map <char, int> m = {{'c',2}, {'a',2}};  map <char, int>::iterator iter;  m.insert(pair<char, int>('a',10));  for(pair<char, int> el: m)  cout << el.first << el.second; | a2c2  **Note: insert() is not working because the key exist** |
| map<char, int> m = {{'c',2}, {'a',2}};  **pair<map<char, int>::iterator, bool> ret;**  **ret = m.insert(pair<char, int>('a',10));**  **cout << ret.first->first << ret.second;** | a0 |
| update value by key | map<char, int> m {{'b',2}, {'a',1}};  m['c']=0; m.at('c') = 3;  cout << m['c']; | 3  update value |
| insert one pair in place | map<char, int> m {{'b',2}, {'a',1}};  m.emplace('c',3); cout << m['c']; | 3  insert a new element |
| map<char, int> m {{'b',2}, {'a',1}};  m.emplace('a',3);  cout << m['a']; | 1  The key 'a' exists. emplace() will not update value. |
| map <char, int> m = {{'c',2}, {'a',2}};  m.emplace\_hint(m.end(), 'd', 3);  for(pair<char, int> el: m)  cout << el.first << el.second; | m=a2c2d3 |
| delete one pair | map <char, int> m = {{'a',1}, {'b',2}};  m.erase('a');  cout << m['a']; | output:0  delete key='a' |
| map <char, int> m = {{'a',1}, {'b',2}};  m.erase(m.find('b')); | delete by key |
| delete all | m.erase(m.begin(), m.end()); |  |
| m.clear();  cout << m.empty(); | 1 |
|  |  |
|  |  |  |
|  |  |
|  |  |
| delete element in multimap | multimap <char, int> m = {{'a',1}, {'a',2}};  **auto pos = m.find('a');**  **if (pos!=m.end()) m.erase(pos);**  for(pair<char, int> el: m)  cout << el.first << el.second; | a2 |
| multimap <char, int> m = {{'a',1}, {'a',2}, {'b',3}};  pair<multimap<char, int>::iterator,  multimap<char, int>::iterator> range;  range = m.equal\_range('a');  **if(range.first != m.end())**  **m.erase(range.first, range.second);**  for(pair<char, int> el: m)  cout << el.first << el.second; | b3  delete multiple elements by key |
|  |  |

### set/multiset

|  |  |  |
| --- | --- | --- |
| insert elements in set | set<int> s={1,7,4,1};  s.insert(3);  for(const int& i: s) cout<< i; | 1347  pair<iter,bool>=insert(val) |
| set<int> s={1,7,4,1};  **pair<set<int>::iterator, bool> ret;**  **ret=s.insert(3);**  cout << \*ret.first << ret.second; | 31 |
| set<int> s={1,7,4,1};  s.insert(s.begin(), 3);  for(const int& i: s) cout<< i; | 1347  iter=insert(iter,val) |
| set<int> s={1,7,4,1};  **set<int>::iterator iter=s.begin();**  **iter = s.insert(iter,3);**  cout << \*iter; | 3 |
| set<int> s={1,7,4,1}, s2={3,6};  s.insert(s2.begin(), s2.end()); | 13467  void insert(firt,last) |
| set<int> s={1,7,4,1};  s.insert(next(s.begin(),2), 8);  for(const int& i: s) cout<< i; | 1478  use next() rather begin()+n |
| set<int> s={1,7,4,1};  s.insert(s.begin()+2, 8); | compiling error |
| insert single element in place | set<int> s={1,7,4,1};  pair<set<int>::iterator, bool> ret;  ret=s.emplace(3);  cout << \*ret.first << ret.second; | 31  pair<iter,bool>=emplace(val) |
| set<int> s={1,7,4,1};  s.emplace\_hint(s.begin(),4);  for(const int& i: s) cout<< i; | 147 |
| set<int> s={1,7,4,8};  set<int>::iterator iter=s.begin();  iter=s.emplace\_hint(s.begin(), 4);  cout << \*iter;  cout << distance(iter, s.end()) <<endl; | 43  iter=emplace\_hint(iter,val) |
|  |  |  |
|  |  |
|  |  |
|  |  |  |
|  |  |
|  |  |

multiset

|  |  |  |
| --- | --- | --- |
| insert elements | multiset<int> s={3,2,1};  multiset<int>::iterator iter=s.begin();  iter=s.insert(2);  cout << \*iter << distance(iter,s.end()); | 22  s=123->1223  iter=c.insert(val), iterator to the 2nd 2 |
| multiset<int> s={3,2,1};  multiset<int>::iterator iter=s.begin();  **iter=s.insert(s.begin(), 2);**  cout << \*iter << distance(iter,s.end()); | 23  s=123->1223  iterator to the 1st 2. |
| multiset<int> s={3,2,1};  multiset<int>::iterator iter=s.begin();  **iter=s.insert(s.end(), 2);**  cout << \*iter << distance(iter,s.end()); | 22  s.end() is meaningless |
| multiset<int> s={3,2,1}, s2={2,6};  s.insert(s2.begin(), s2.end());  for(const int& i: s) cout<< i; | 12236  void c.insert(first,last) |
| insert one element in place | multiset<int> s={3,2,1};  multiset<int>::iterator iter=s.begin();  iter=s.emplace\_hint(s.begin(), 8);  cout << \*iter << endl; | 8  iter=emplace(val) |
| multiset<int> s={3,2,1};  multiset<int>::iterator iter=s.begin();  iter=s.emplace\_hint(s.begin(), 8);  cout << \*iter << endl; | 8  s=1238  iter=emplace\_hint(iter,val) |
| delete elements  iter=erase(val)  iter=erase(iter)  iter=erase(first,last) | multiset<int> st {1,1,2,3};  **st.erase(1);**  multiset<int>::iterator iter = st.begin();  for(iter; iter != st.end(); ++iter)  cout << \*iter; | 23  remove all elements value=1 |
| multiset<int> st {1,1,2,3};  **st.erase(st.begin());**  multiset<int>::iterator iter = st.begin();  for(iter; iter != st.end(); ++iter)  cout << \*iter; | 123  remove the first element |
|  |  |
| multiset<int> st {1,2,3,5,2,4,5,6,7};  **st.erase(st.begin(), next(st.begin(),2));**  multiset<int>::iterator iter = st.begin();  for(iter; iter != st.end(); ++iter)  cout << \*iter; | 2345567  remove the first two elements  Note: "st.erase(st.begin(), st.begin()+2);" is wrong |
| multiset<int> st {1,2,3,5,2,4,5,6,7};  **st.erase(st.find(5), st.end());**  multiset<int>::iterator iter = st.begin();  for(iter; iter != st.end(); ++iter)  cout << \*iter; | 12234  remove elements starting from 5 to the end. |
| multiset<int> st {1,2,3,5,2,4,5,6,7};  **multiset<int>::iterator lb = st.lower\_bound(2);**  **multiset<int>::iterator ub = st.upper\_bound(5);**  **st.erase(lb,ub);**  multiset<int>::iterator iter = st.begin();  for(iter; iter != st.end(); ++iter)  cout << \*iter; | 167  remove all elements within a range. |
| #include <set>  set<int> st {1,2,3,5,6,7};  set<int>::iterator lb = st.lower\_bound(2);  set<int>::iterator ub = st.upper\_bound(6);  st.erase(lb,ub);  set<int>::iterator iter = st.begin();  for(iter; iter != st.end(); ++iter)  cout << \*iter; | 17  delete elements give a range of 2-6 |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## operation

Operations on containers could be copy, compare, merge/concatenate, split, swap, etc.

* c.merge(c2, <binary func>) is only for list or forward\_list
* only string support overload operator+=, ==, <, >
* stack and queue can't support overload operator=()

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |
|  | array | string | vector | list | forward  \_list | stack | queue | deque | map | set |
| **operator=** | **Y** | **Y** | **Y** | **Y** | **Y** | **-** | **-** | **Y** | **Y** | **Y** |
| operator+= | - | Y | - | - | - | - | - | - | - | - |
| operator==  operator< | - | Y | - | - | - | - | - | - | - | - |
| **operator[pos]** | **Y** | **Y** | **Y** | **-** | **-** | **-** | **-** | **Y** | **Y** | **-** |
| c.merge(c2, func) | - | - | - | Y | Y | - |  | - | - | - |
| c.swap(c2) | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |

**void c.merge(c2, <binary func>)**.

### string

Note: 1. string must end with '\0'. 2. bounds for string.

substring:

* str.substr(<start position>, <length>). length could be missing. Go through '\0' or meeting the length.

compare string:

* overload operator==(). str1==str2, if equal, str1<str2 if less than by ASCII code. str1>str2 if greater than by ASCII code.
* str1.compare(str2): equal return 0, less than return -1, greater than return 1

copy string

* overload operator=().
* iterator: c(c.begin(), c.end())
* string::str.copy(<char array>, <length>, <position index>). str.copy() works for char array rather than string type. the last element of char array should be '\0'.

|  |  |  |
| --- | --- | --- |
| substring | string a = "abcd";  cout << a.substr(3); | d |
| string a = "abcdef";  cout << a.substr(0,3); | abc |
| string a = "a\0bcdef";  cout << a.substr(0,3); | a  Note: contains '\0' |
| compare string | string a = "ad"; string b= "bc";  if(a<=b) cout<< a; | output: ad  compare ASCII code one by one. |
| string a = "ad"; string b= "bc";  if(a.compare(b)==-1) cout<< a; | str1.compare(str2)>0 str1>str2  str1.compare(str2)==0 str1==str2  str1.compare(str2)<0 str1<str2 |
| string a = "abcde"; string b = "de";  cout << a.compare(3, a.size()-1, b) << endl; | output: 0 |
| string a = "abcde"; string b = "de";  cout << a.compare(3, 10, b) << endl; | output: 0  the 2n argument the end could be longer than the length of the string |
| copy string | string s1="A", s2=s1; cout << s1 << s2;  s2="B"; cout << s1 << s2; | output: AAAB |
| string a = "abc";  string b(a.begin(), a.end()); |  |
| string str = "abc";  char ss[4]; s[3]='\0';  str.copy(ss, str.size()); | ss=abc |
| string str = "abcdef";  char ss[5]; ss[4]='\0';  str.copy(ss, 4, 2); | ss=cdef |
| string str = "abcdefgh";  char ss[3]; ss[2]='\0';  str.copy(ss, str.size()); | unpredicted value  ss is shorter than str. |
| string str = "ab\0c";  const int i = str.size();  cout << "size=" << i;  char ss[i+1]; ss[i]='\0';  str.copy(ss, i,0); | ss=ab.  '\0' ends a string. |
| string str = "ab\0cde";  char ss[7]; ss[6]='\0';  str.copy(ss, 6,0); | ss=ab |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### vector

|  |  |  |
| --- | --- | --- |
| copy | int arr[] = {1, 2, 3, 4, 5, 6};  int n = sizeof(arr) / sizeof(arr[0]);  vector<int> v(arr+1, arr + n); | copy array of 2nd-end into a vector |
| vector <int> v={1,2,3};  vector <int> v2=v;  v[0]=4; cout << v2[0]; | v2:123, v:423 |
| vector <int> v1={1,2,3};  vector <int> v2(v1.begin(), v1.end()); |  |
| vector<int> v {1,2,3,4}; vector<int> v2;  **for\_each(v.begin(), v.end(),\**  **[&](const int& i){ return v2.push\_back(i);});**  for(int& i: v2) cout << i << ','; |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### list

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### Size or length

Except Array, all STL containers is dynamic data type.

1. They automatically "resize". when you insert elements. For example, when you use c.insert() to insert elements into a container, you don't have to resize containers unless you reach some physical limit c.max\_size().

2. c.size() return the number of elements in a container. Those elements should be stored consecutively such string, vector, deque, list, stack, and queue. Array is static type, of which the size is declared when instantiation, and can't be changed in the future. So calculation size of array makes no sense. Iterators of forward\_list is not stored adjacently. It takes time to deallocate memory for calculating the size.

Both string::size and string::length are synonyms and return the same value. return number of characters before the character '\0'.

3. c.resize() allow assign overhead memory to a container such as string, vector, deque, list, forward\_list. Allocation enough memory will prevent from exception of bad\_alloc when do insertion and concatenation because lements of those containers should be stored continuous memory block. It is better to use c.resize() if c.insert() or c.emplace() would in used later.

But for associative containers such map and set. Map will allocate storage for each map element individually. "resize" is useless for those containers.

4. c.capacity() return the memory the string allocated to hold its contents.

c.shrink\_to\_fit() shrink capacity to the size.

c.reserve(): change capacity.

c.capacity() and c.reserve() are only working for string and vector. Because elements are stored consecutive in one memory block. They are relying on hardware and OS.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | array | string | vector | deque | list | forward  \_list | map | set | stack | queue |
| bool=c.empty() | - | Y | Y | Y | Y | Y | Y | Y | Y | Y |
|  |  |  |  |  |  |  |  |  |  |  |
| size=c.size() | - | Y | Y | Y | Y | - | Y | Y | Y | Y |
| size=c.max\_size() | - | Y | Y | Y | Y | Y | Y | Y | - | - |
| c.resize(size) | - | Y | Y | Y | Y | Y | - | - | - | - |
|  |  |  |  |  |  |  |  |  |  |  |
| size=c.capacity() | - | Y | Y | - | - | - | - | - | - | - |
| c.reserve(size) | - | Y | Y | - | - | - | - | - | - | - |
| c.shrink\_to\_fit() | - | Y | Y | Y | - | - | - | - | - | - |

sizeof(<container>) is size of container class:. sizeof(string)=24, sizeof(vector)=12

c.max\_size(): maximum length of the string can reach

|  |  |  |
| --- | --- | --- |
| **action** | **code** | **Note** |
| string | string s="ABC";  cout << s.size() << s.length(); | 33 |
| string s="ABC";  cout << sizeof(s) << sizeof(string); | 2424 |
| string s="";  cout << s.max\_size(); | one character = 2^8-1=0-255  2,147,483,647= 111,1111,1111,1111,1111,1111,1111,1111 = 2^31-1 |
| string s; cout << s.size();  s.resize(10); cout << s.size(); | 010 |
| string s="";  cout << s.capacity(); | 15  depending on compiler, min size memory of a string allocated is 15 characters.  "string s="";" differs from "string s;". The later is not allocated with memory |
| array | int a[10] = {1,2};  cout << "size of array" << end(a)-begin(a) <<endl; | end(arr), begin(arr) return address of start and end pointer. |
| int a[10] = {1,2};  cout << sizeof(a) << endl; | output:40 |
| vector | vector<int> v {1,2,3,4};  cout << v.size(); | 4 |
| vector<int> v {1,2,3,4};  cout << sizeof(v) << sizeof(vector<int>); | 1212 |
| vector<int> v {};  cout << v.capacity(); | 0 |
| vector<int> v {};  cout << v.max\_size(); | 1,073,741,823=2^30-1 |
| vector<float> v {}; cout << v.size();  v.resize(10); cout << v.size(); | 010 |
| linked list |  |  |
| deque |  |  |
|  |  |  |
| map | map<char, int> m {{'a',1}};  cout << m.size(); | 1 |
| map<char, int> m {{'a',1}};  cout << m.capacity(); | no map.capacity() |
| map<char, int> m {{'a',1}};  cout << m.max\_size(); | 178,956,970= |
| map<char, int> m {{'a',1}};  m.resize(10);  cout << m.size(); | no map.resize() |
|  |  |
|  |  |
| multi-map |  |  |
| set | set<int> s={1,2};    cout << s.size(); | 2 |
| set<int> s={1,1};    cout << s.size(); | 1. unique values |
| multi-set |  |  |
|  |  |  |
| stack |  |  |
| queue |  |  |
| priority-queue |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

The sizeof keyword gives the amount of storage, in bytes, associated with a variable or a type (including aggregate types). This keyword returns a value of type size\_t.

In C++, string a is the class instance. sizeof(a) calculate memory size of this class instance, equal to sizeof(string). The output is not affected by contents of the strong. The output is determined by the compiler. could be 4 (size of pointer), or 12,or 24 or 28 or 32

In C, string a is char array. the output below is 5.

char a[]="a\0bc";

printf("%d", sizeof(a));

|  |  |
| --- | --- |
| string fun(string t, string s="x", int r=1){  while(--r)  s +=s;  t +=s;  return s;  }  int main(void){  string name ="a";  cout << fun(name) << name;  return 0;  } | output:xa |
|  |  |
| string a="A";  a.append(a).append(a).append(a);  cout << a.length() << a.size() << endl; | output: 88  AA->AAAA->AAAAAAAA  length() is equal to size() in C++ |
| string a="a\0bc";  cout<< a.length() << endl;  cout<< a.size() << endl;  cout<< a.max\_size() << endl; | output:  1  1  2,147,483,634 |
| string a="a\0bc";  cout<< sizeof(a) << endl;  cout<< sizeof("a\0bc") << endl;  cout<< a.capacity() << endl; | **output:**  **24:== sizeof(string) give the size of class instance std::string no matter what the string is.**  **5: 4 characters + '\0'**  **15** |
| #include <iostream>  #include <cmath>  #include <iomanip>  using namespace std;  string fun(string &t, string s="", int r=2){  while(--r)   s +=s;  t +=s;  return s;  }  int main(void){  string name ="x";  cout << fun(name, name) << name;  return 0;  } | output: xxxxx  pass string value, string reference to function |

# Iterator

**declare iterator:**

**#include <iterator>**

**<container>::iterator iter;**

## iterator functions

**1.Get iterator**

iter=begin(c) / iter=c.begin()

iter=end(c) / iter=c.end()

new\_iter = next(iter, n): return the new iterator after advancing the iterator n positions

new\_iter = prev(iter, n): return the new iterator after decrementing the iterator n positions

**2. move iterator**

advance(iter, n): incrementally move the iterator with n position. No returns

**3. double iterators**

distance(iter\_1, iter\_2)

**4. insert iterator**

front\_inserter(c)

back\_inserter(c)

inserter(c, iter)

lambda functions:

[capture list] (parameter list) -> return type { function body }

## pointer/reference/iterator

pointers is c-style. iterators will be more widely used in C++.

**sequential container**

c.data(): return pointer to the first element

**iterator**

c.begin(): return iterator to the beginning

c.end(): return iterator to the end

c.rbegin(): return reverse iterator to the end

c.rend(): return reverse iterator to the beginning

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |
|  | array | string | vector | deque | list | forward  \_list | map | set | stack | queue |
| iter=c.begin() | Y | Y | Y | Y | Y | Y | Y | Y | - | - |
| iter=c.end() | Y | Y | Y | Y | Y | Y | Y | Y | - | - |
| iter=c.rbegin() | Y | Y | Y | Y | Y | - | Y | Y | - | - |
| iter=c.rend() | Y | Y | Y | Y | Y | - | Y | Y | - | - |
| iter=c.cbegin() | Y | Y | Y | Y | Y | Y | Y | Y | - | - |
| iter=c.cend() | Y | Y | Y | Y | Y | Y | Y | Y | - | - |
| iter=c.crbegin() | Y | Y | Y | Y | Y | - | Y | Y | - | - |
| iter=c.crend() | Y | Y | Y | Y | Y | - | Y | Y | - | - |

**reference**

c.front(): return reference to the first element

c.back(): return reference to the last element

c.at(n): return reference to the element at position n. Containers must support random access.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |
|  | array | string | vector | deque | list | forward  \_list | stack | queue | map | set |
| return reference |  |  |  |  |  |  |  |  |  |  |
| **ref=c.front()** | **Y** | **Y** | **Y** | **Y** | Y | Y | c.top | Y | -- | -- |
| **ref=c.back()** | **Y** | **Y** | **Y** | **Y** | Y | -- | -- | Y | -- | -- |
| **ref=c.at(<pos>)** | **Y** | **Y** | **Y** | **Y** | -- | -- | -- | -- | c.at(key) | -- |
| no return: |  |  |  |  |  |  |  |  |  |  |
| c.push\_front(<value>) | -- | -- | -- | Y | Y | Y | c.push | c.push | -- | -- |
| c.push\_back(<value>) | -- | Y | Y | Y | Y | -- | -- | -- |
| c.pop\_front(<value>) | -- | -- | -- | Y | Y | Y | c.pop | c.pop | -- | -- |
| c.pop\_back(<value>) | -- | Y | Y | Y | Y | -- | -- | -- |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |
|  | array | string | vector | deque | list | forward  \_list | map | set | stack | queue |
| c1.swap(c2) | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| bool=c.empty() | - | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| c.clear() | - | Y | Y | Y | Y | Y | Y | Y | - | - |
|  |  |  |  |  |  |  |  |  |  |  |
| iter.erase(iter) | - | Y | Y | Y | Y | erase\_after | Y | Y | - | - |
| insert() | - | Y | Y | Y | Y | insert\_after | Y | Y | - | - |
| emplace() | - | replace | Y | Y | Y | - | Y | Y | Y | Y |
|  |  |  |  |  |  |  |  |  |  |  |
| p\_var=c.data() | Y | Y | Y | - | - | - | - | - | - | - |
| c.find(<val>) | - | Y | - | - | - | - | Y | Y | - | - |
| c.assign(first, last) | fill | Y | Y | Y | Y | Y | - | - | - | - |

stack/queue: emplace()

map/set: emplace(), emplace\_hint()

vector: emplace(), emplace\_back()

deque/list: emplace(), emplace\_front(), emplace\_back()

forward\_list: emplace\_front(), emplace\_back()

|  |  |  |
| --- | --- | --- |
| **action** | **code** | **Note** |
| string | string s="ABC";  cout << \*s.data(); | A |
| string s="ABC";  cout << \*(s.begin()+1); | B |
| string s="ABC";  cout << s.front()+1; | 66  s.front()+1='A'+1=65+1=66 |
| string s="ABC";  cout << s.at(3); | runtime error: out of bounds |
| string s="ABC";  cout << \*s.rbegin(); | C |
| vector | vector<float> v {1,2,3};  cout << \*v.begin() << \*v.end()-1; | 1-1 |
| vector<float> v {1,2,3};  cout << int(\*v.end()); | 0  \*v.end()='\0' |
| vector<float> v {1,2,3};  cout << v.back(); | 3  return |
| vector<int> v {1,2,3};  cout << v.at(1); | 2 |
| vector <char> t(5);  char \*p1 = t.data()+2;  char \*p2 = p1+2;  cout<< p2 - t.data(); | output:4  vector.data() |
| map | map<char, int> m {{'b',2}, {'a',1}};  map<char, int>::iterator iter=m.begin();  cout << iter->first << iter->second; | a1 |
| map<char, int> m {{'b',2}, {'a',1}};  **m['c']=0; m.at('c') = 3;**  map<char, int>::iterator iter = m.begin();  advance(iter,2);  cout << iter->first << iter->second; | c3  Note: the key should exist using at(), or return error. |
|  |  |
|  |  |
| stack |  |  |
| queue |  |  |
| priority-queue |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## container iteration

1. for loop using int size as step

2. for loop using iterator as step

3. algorithm::for\_each()

4. while loop using iterator

5.algorithm::transform()

### string

Define a string:

#include <string>

string str = "hello";

The below approaches are equal. The output is "h,e,l,l,o,"

|  |  |
| --- | --- |
| for(int i=0; i<str.length(); i++) {  cout << str[i] << ',';  } | for loop |
| char \*p = &str[0];  while(\*p!='\0') {  cout << \*p++ << ',';  } | pointer |
| **for(char& c: str) {**  cout << c << ',';  } | for loop |
| string::iterator iter;  for(iter=str.begin(); iter != str.end(); ++iter) {  cout << \*iter << ',';  } | iterator+for loop |
| string::iterator iter=str.begin();      for(; iter != str.end(); ++iter) {          cout << \*iter << ',';      } | compared with the above |
| string::iterator iter**=str.begin()**;  **for(iter; iter != str.end(); advance(iter, 1)) {**          cout << \*iter << ',';      } |  |
| string::iterator iter**=str.begin()**;  **for(iter; iter != str.end(); iter=next(iter)) {**          cout << \*iter << ',';      } |  |
| string::iterator iter = str.begin();  while(iter != str.end()) {  cout << \*iter << ',';  ++iter;  } | iterator+while loop |
|  |  |

The below approaches are equal. The output is " o,l,l,e,h,"

|  |  |
| --- | --- |
|  |  |
|  |  |
| string::iterator iter=str.end()-1;  **for(iter; iter>=str.begin(); --iter){**          cout << \*iter << ',';      } |  |
| string::iterator iter=str.end()-1;  **for(iter; iter>=str.begin(); advance(iter,-1)){**          cout << \*iter << ',';      } |  |
| string::iterator iter=str.end()-1;  **for(iter; iter>=str.begin(); iter=prev(iter,1)){**          cout << \*iter << ',';      } |  |
| string::iterator iter=str.end()-1;      do{          cout << \*iter << ',';          iter--;      }while(iter != (str.begin()-1) ); |  |
|  |  |
|  |  |

### vector

For example: vector <int> v={1,2,3};

forward loop, output: 123

|  |  |
| --- | --- |
| for(int i=0; i<v.size(); i++) {      cout<< v[i];  } |  |
| vector<int>::iterator iter;  for(iter=v.begin(); iter!=v.end(); iter++) {      cout<< \*iter;  } |  |
| vector<int>::iterator iter;  for(iter=v.begin(); iter!=v.end(); advance(iter,1)) {      cout<< \*iter;  } |  |
| vector<int>::iterator iter;  for(iter=v.begin(); iter!=v.end(); iter=next(iter)) {      cout<< \*iter;  } |  |
| for(int& i:v){      cout << i;  } |  |
| for(int i:v){      cout << i;  } |  |
|  |  |
|  |  |

backward loop, output:321

|  |  |
| --- | --- |
| for(int i=v.size()-1; i>=0; i--) {      cout<< v[i];  } |  |
| vector<int>::iterator iter=v.end()-1;  for(iter; iter>=v.begin(); iter--) {      cout<< \*iter;  } |  |
| vector<int>::iterator iter=v.end()-1;  **for(iter; iter>=v.begin(); advance(iter, -1)) {**      cout<< \*iter;  } |  |
| vector<int>::iterator iter=v.end()-1;  **for(iter; iter>=v.begin(); iter=prev(iter)) {**      cout<< \*iter;  } |  |
| **for(pair<char, int> el: m){**      cout << el.first << el.second;  } |  |
| #include <algorithm>  **for\_each(m.begin(), m.end(), [](pair<char, int> el){**      cout << el.first << el.second;  }); | 1. for\_each( iter\_start, iter\_end, func() ) defined in algorithm.h  2. lambda function |
|  |  |
|  |  |

### map

For example: map<char, int> m = {{'a',1}, {'b',2}, {'c',3}};

|  |  |
| --- | --- |
|  |  |
| **map<char, int>::iterator iter=m.begin();**  **for(iter;iter!=m.end();iter++){**      cout << iter->first << iter->second;  } | iterator |
| **for(pair<char, int> el: m) {**      cout << el.first << el.second;  } | pair defined in utility.h |
| **for\_each(m.begin(), m.end(), \**  **[](pair<char, int> el) {**       cout << el.first << el.second;  }); | for\_each defined in algrothm.h |
| map<char, int> m = {{'a',1}, {'b',2}, {'c',3}};  map<char, int> m2;  transform(m.begin(), m.end(), inserter(m2, m2.begin()), \      [](pair<char, int>& el){          cout << el.first << el.second;          return 4;  }); | ???  can't pass reference in lambda functions for pair  correct :     [](const pair<char, int>& el) |
|  |  |
|  |  |
|  |  |

set

|  |  |
| --- | --- |
| set<int> s={1,7,4,1};  for(const int& i: s) cout<< i; | 147 |
| set<int> s={1,7,4,1};  for(int& i: s) cout<< i; | compiling error: const int& i |

### transform()

**algorithm::transform() is like apply() in R. function always have returns.**

for the objects of string or vector

last\_iter2 = transform(first, last, c2.begin()/back\_inserter(c2), <unary operator>). Pass one element at a time and push the return to v2.

last\_iter2= transform(first, last, first1, c2.begin()/back\_inserter(c2), <binary operator>). Pass two elements for one or two containers at a time.

Consider the details below when applying transforms()

1. How to use functions: c2.begin(), back\_inserter(c2), inserter(c2, iter)

2. How to use forward-/backward-iterator: c2.begin(), c2.rebegin()

3. How to use unary or binary functions

4. Use function objector: define lambada functions, use functions from <functional>, use objects from class or struct, use objects from template class or struct.

5. 1-3 containers are allowed.

**string**

|  |  |
| --- | --- |
| string s = "abc";  transform(s.begin(), s.end(), s.begin(), \  ::toupper); | s=ABC  cctype::toupper |
| string s = "ATGCT";  transform(s.begin(), s.end(), **s.rbegin()**, \  ::toupper); | s=ATGTA  use backward\_iterator |
| string s = "ABC"; string ss;  transform(s.begin(), s.end(), **back\_inserter(ss)**, \  ::tolower); | ss=abc |
| string s = "ATGC"; string ss;  transform(**s.rbegin(), s.rend()**, back\_inserter(ss), \  ::toupper); | ss=CGTA  backward iterator at [first, last] |
|  |  |
| map<char, char> nt {{'A','T'}, {'T','A'}, {'G','C'}, {'C','G'}};  string s = "ATGC"; string ss;  transform(s.rbegin(), s.rend(), back\_inserter(ss),\  **[&nt](const char& c){return nt[c];}**); | ssGCAT  reverse\_complementary |
| string s = "ATATGC";  string s2 = "ATATCG"; string ss;  transform(s.begin(), s.end(),s2.begin(), \  back\_inserter(ss), [](char& a, char& b){  return (a==b) ? a : 'N';  }); | ss=ATATNN  lambda function |
| char Complement(char &c){  switch(c){  case 'A': return 'T'; break;  case 'T': return 'A'; break;  case 'G': return 'C'; break;  case 'C': return 'G'; break;  default: return 'N';  };  };  int main(void) {  string s = "ATGC"; string ss;  transform(s.rbegin(), s.rend(), \  back\_inserter(ss), Complement);  cout << ss;  return 1;  } | ss=GCAT  use function in transform |
| struct Complement{  char operator()(char &c){  switch(c){  case 'A': return 'T'; break;  case 'T': return 'A'; break;  case 'G': return 'C'; break;  case 'C': return 'G'; break;  default: return 'N';  };  }  };  int main(void) {  string s = "ATGC"; string ss;  transform(s.rbegin(), s.rend(), \  back\_inserter(ss), Complement());  cout << ss;  return 1;  } | ss=GCAT  use struct object in transform() |
|  |  |

vector

|  |  |
| --- | --- |
| vector<int> v {1,2,3,4};  transform(v.begin(), v.end(), v.begin(),  [](const int& i){ return i\*i; });  for(int& i : v) cout << i; | output:14916 |
| vector<int> v {1,2,3,4};  vector<int> v2;  transform(v.begin(), v.end(), back\_inserter(v2),  [](const int& i){ return i\*i; });  for(int& i : v2) cout << i; | output:14916 |
| vector<int> v {1,4,6,10};  vector<int> v2;  **transform(v.begin(),v.end()-1, v.begin()+1, back\_inserter(v2),\**  **[](int a, int b){return b-a;});**  for(int& i: v2) cout << i; | v2=324  loop with double containers |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Here is one example: vector<int> v={1,2,3,4,5}; vector<int> v2;

Minus one from each element of vector v, then output should be v2=01234.

Compare multiple approaches when applying function object:

1. functions in functional. 2. lambda functions. 3. function.

|  |  |
| --- | --- |
| transform(v.begin(),v.end(), back\_inserter(v2),  bind2nd(minus<int>(), 1)); | functions defined in <functional> |
| transform(v.begin(),v.end(), back\_inserter(v2),  [](const int &i){ return i-1;}); | lambda functions is kind of function object |
| int x =1;  transform(v.begin(),v.end(), back\_inserter(v2),  [x](const int &i){ return i-x;}); | same as the above |
| int Minus(const int x, const int y){  return x-y;  }  int main(void) {  vector<int> v={1,2,3,4,5};  vector<int> v2;  **transform(v.begin(),v.end(), back\_inserter(v2),**  **bind2nd(ptr\_fun(Minus), 1));**  for(int &i: v2) cout << i;  return 1;  } | the function return value. bind2nd(Minus,1) is not working.  ptr\_fun() convert function to function objects. |
| **struct Cal{**  **int operator()(const int &x){ return x-1; }**  **};**  int main(void) {  vector<int> v={1,2,3,4,5};  vector<int> v2;  **transform(v.begin(),v.end(), back\_inserter(v2), Cal());**  for(int &i: v2) cout << i;  return 1;  } | apply struct objects and overload operator() |
|  |  |
|  |  |
|  |  |

## errors

### syntax errors

|  |  |
| --- | --- |
| int t[]={10,5,9,6,2,4,7,8,3,1};  vector<int> v(t,t+10);  sort(v.begin(),v.end());  cout << min\_element(v.begin(), v.end()); | compiling error.  return iterator which should be dereferenced.  correct: cout << \*min\_element(v.begin(), v.end()); |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### Invalidation

Iterator becomes invalid if the container is changed internally, such insertion, deletion, appending, or moving.

for map or set, "c.erase(iter)" cause invalidation of current iterator. but "iter=c.erase(iter)" or "c.erase(iter++)" will increment the iterator, which is ok.

for string, vector, "c.erase(iter)" or "c.insert(iter)" will invalidate iterator and those after the current.

vector

|  |  |
| --- | --- |
| vector<int> v {1,4,6,8,9,0};  vector<int>::iterator iter;  for(iter = v.begin(); iter!=v.end(); iter++){  **if(\*iter>4) v.erase(iter);**  }  for(int& i: v) cout << i; | v=1480  v.erase(iter) delete current iterator, and then the iter will move to the next one. So 8 will not be deleted. |
| vector<int> v {1,4,6,8,9,0};  vector<int>::iterator iter;  for(iter = v.begin(); iter!=v.end();){  **if(\*iter>4) { v.erase(iter); }**  **else{ iter++; }**  }  for(int& i: v) cout << i; | v=140  Compared with the above, here is the correct one. |
| vector<int> v {1,4,6,8,9,0};  vector<int>::iterator iter;  for(iter = v.begin(); iter!=v.end();){  if(\*iter>4) v.push\_back(4);  } | compiling error: std::bad\_alloc  push\_back() cause changes of capacity, therefore begin() and end() is invalid. |
| vector<int> v {1,4,6,8,9,0};  vector<int>::iterator iter;  for(iter = v.begin(); iter!=v.end();){  if(\*iter>4) v.pop\_back();  } | compiling error |
| vector<int> v {1,4,6,8,9,0};  vector<int>::iterator iter;  for(iter = v.begin(); iter!=v.end();){  if(\*iter>4) v.insert(iter, 20);  } | runtime error |
|  |  |
|  |  |
|  |  |

map

|  |  |
| --- | --- |
| map <char, int> m = {{'c',2}, {'a',2},{'b',2}, {'d',3}};  map <char, int>::iterator iter;  for(iter=m.begin(); iter != m.end(); iter++)  if(iter->second==2)  iter=m.erase(iter); | b2d3 |
| map <char, int> m = {{'c',2}, {'a',2},{'b',2}, {'d',3}};  map <char, int>::iterator iter;  for(iter=m.begin(); iter != m.end();)  if(iter->second==2)  iter=m.erase(iter);  else  iter++; | d3 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# STL: algorithm

**declare algorithm**

**#include <algorithm>**

rules for methods

1. algorithm::<method>(first, last, …). Both first and last are iterators namely c.begin() and c.end().

2. outputIterator:

Sequential container: "back\_inserter(c2)" OR "c2.resize(size)" should be defined if "c2.begin()" is used.

associative container: "inserter(c2, c2.begin())"

3. <method>(first, last, val) and <method>\_if(first, last, func)

|  |  |
| --- | --- |
|  |  |
| new\_first\_iter=copy(first, last, c2.begin()) | new\_last\_iter=copy\_if(first, last, c2.begin(), func) |
| size=count(first, last, val) | size=count\_if(first, last, func) |
| first\_iter=find(first, last, val) | first\_iter=find\_if(first, last, func)  first\_iter = find\_if\_not(first, last, func) |
| last\_iter=remove(first, last, val) | last\_iter= remove\_if(first, last, func) |
| new\_last\_iter=remove\_copy(first, last, c2.begin(), val) | new\_last\_iter=remove\_copy\_if(first, last, c2.begin(), func) |
| void replace(first, last, val, new-val) | void replace\_if(first, last, func, new-val) |
| new\_last\_iter=replace\_copy(first, last, c2.begin(), val, new-val) | new\_last\_iter=replace\_copy\_if(first, last, c2.begin(), func, new-val) |

4. <method>\_n(first, last, size, val): generate container with repeated elements

* end\_iter=copy\_n(first, n, c2.begin()/back\_insert(c2)/inserter(c2, c2.begin()): copy n elements from c to c2. Here, "end\_iter==c2.end()"
* void fill\_n(first, last, n, val): fill sequence with n values
* void generate\_n(first, last, n, val): generate n values for sequence with function
* begin\_iter=search\_n(first, last, n, val)/ search\_n(first, last, n, val, func): search range for elements

5. <method>\_of(first, last, …): test condition

all\_of()

any\_of()

none\_of()

find\_first\_of()

## math

### max/min

min

max

minmax

min\_element(iter.begin(), iter.end(), func())

max\_element(iter.begin(), iter.end(), func())

minmax\_element

|  |  |  |
| --- | --- | --- |
| **action** | **code** | **Note** |
| string | string str = "hello";  cout << "max=" << \*(max\_element(str.begin(), str.end()));  cout << "min=" << \*(min\_element(str.begin(), str.end())); | output: oe |
| vector | vector<int> v={1,2,3};  cout << "max=" << \*(max\_element(v.begin(), v.end()));  cout << "min=" << \*(min\_element(v.begin(), v.end())); |  |
| linked list |  |  |
| deque |  |  |
|  |  |  |
| map | map<char, int> m = {{'a',1}, {'b',2}, {'c',3}};  typedef pair<char,int> my;  my a = \*max\_element(m.begin(), m.end(), \  [](const my& x, const my& y){  return x.second < y.second;});  cout << "max=" << a.second;  my b = \*min\_element(m.begin(), m.end(), \  [](const my& x, const my& y){  return x.second < y.second; });  cout << "min=" << b.second; | output: max=3min=1 |
| multi-map |  |  |
| set |  |  |
| multi-set |  |  |
|  |  |  |
| stack |  |  |
| queue |  |  |
| priority-queue |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### counting

**count(iter.begin(), iter.end(), <value> )**.

**count\_if(iter.begin(), iter.end(), func() )**.

|  |  |  |
| --- | --- | --- |
| **action** | **code** | **Note** |
| string | int f = count(str.begin(), str.end(), 'l');  cout << f; | output:3  count times of letter 'l'. |
|  | string str = "hello world";  int f = count\_if(str.begin(), str.end(),      [x='l'](const char& c){return c>x;});  cout << f; | output: 4 (2o+1w+1r)  count times of letters greater than 'l'. |
| vector | vector<int> v {1,2,3,4,3,1,0,5};  int f = count(v.begin(), v.end(), 3);  cout << f; | output:2  counts times of integer=3; |
|  | vector<int> v {1,2,3,4,3,1,0,5};  int f = count\_if(v.begin(), v.end(),      [x=3](const int& i){ return i>=x; });  cout << f; | output:4  counts integers which is greater than or equal to 3. |
| linked list |  |  |
| deque |  |  |
|  |  |  |
| map |  |  |
| multi-map |  |  |
| set |  |  |
| multi-set |  |  |
|  |  |  |
| stack |  |  |
| queue |  |  |
| priority-queue |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## merge

### concatenate

algorithm::merge(c1.begin(), c1.end(), c2.begin(), c2.end(),c.begin())

algorithm::inplace\_merge(c1.begin(), c1.end(), c2.begin(), c2.end(),c.begin())

string

accumulate

|  |  |
| --- | --- |
| string s1("GAA"), s2("TTC");  string ss; ss.resize(s1.size()+s2.size());  merge(s1.begin(), s1.end(), s2.begin(), s2.end(), \      ss.begin());  cout << ss; | output:GAATTC |
| string s1("GAA"), s2("TTC");  string ss;  merge(s1.begin(), s1.end(), s2.begin(), s2.end(), \      back\_inserter(ss));  cout << ss; | output:GAATTC |
| string s1("GAA");  **merge(s1.begin(), s1.end(), s1.begin(), s1.end(), \**  **back\_inserter(s1));**  cout << s1; | output:GAAGAAGAA  repeat 3 times |
|  |  |
|  |  |
|  |  |

vector

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### combine/sum/product

operator+ means concatenation for string, and sum for integer or float.

**1.algorithm::accumulate()**

#include <numeric>

accumulate(c.begin(), c.end(), <initial value>).

accumulate(c.begin(), c.end(), <initial object>, <lambda function>).

function of accumulate() like sum() or join()

**2. algorithm::inner\_product()**

inner\_product(v.begin(), v.end(), v2.begin(), <initial value>)

product matching elements of two containers one on one then return sum

|  |  |  |
| --- | --- | --- |
| **action** | **code** | **Note** |
| string | string str = "abc";  string f = accumulate(str.begin(), str.end(), string(""),      [](string ss, const char& c){  return  ss=="" ? (ss+c) : (ss+','+c);});  cout << f; | output:a,b,c |
|  |  |  |
|  |  |  |
| vector | vector<int> v {1,2,3,4,3,1,0,5};  int f = accumulate(v.begin(), v.end(),0);  cout << "sum=" << f; | output: sum=19 |
| vector<int> v {1,2,3,4};  int f= accumulate(v.begin(),v.end(),1,      [](int& s, const int& i){return s\*i; });  cout << f; | output: 24  accumulative product |
| vector<string> vs {"abc", "def"};  string ss = accumulate( vs.begin(), vs.end(), string(""));  cout << ss; | output: abcdef  convert string vector to string. |
| vector<string> vs {"ab", "cd", "ef"};  string f = accumulate(vs.begin(), vs.end(), string("str:"), [](string ss, string& s) { return ss+','+s; }); cout << f; | output: str:,ab,cd,ef  convert string vector to string with separator. |
| vector<int> v {1,2,3,4}; vector<int> v2 {1,2,3,4};  int f = inner\_product(v.begin(), v.end(), v2.begin(), 0);  cout<< f; | output:30  0+1x1+2x2+3x3+4x4=  1+4+9+16=30 |
| vector<int> v {1,2,3,4};  int f = inner\_product(v.begin(), v.end(), v.rbegin(), 0);  cout<< f; | output:20  0+1x4+2x3+3x2+4x1  =4+6+6+4=20 |
| linked list |  |  |
| deque |  |  |
|  |  |  |
| map |  |  |
| multi-map |  |  |
| set |  |  |
| multi-set |  |  |
|  |  |  |
| stack |  |  |
| queue |  |  |
| priority-queue |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### union

iter=set\_union(c1.begin(),c1.end(), c2.begin(),c2.end(), c.begin()). elements of c1 and those of c2 which are not included in c1.

iter=set\_difference(c1.begin(),c1.end(), c2.begin(),c2.end(), c.begin()). elements of c2 which are not included in c1.

set\_symmetric\_difference(c1.begin(),c1.end(), c2.begin(),c2.end(), c.begin()). elements exists only in c1 or c2.

set\_intersection(c1.begin(),c1.end(), c2.begin(),c2.end(), c.begin()). elements co-exist in c1 and c2.

Note:

1. compare elements by position. The two containers should be sorted in advance.

2. size of the union container should be large enough.

3. return iterator to the end of union range. Resizing could be done for removing extra elements.

|  |  |
| --- | --- |
| vector<int> v1 {8,4,4,1}; sort(v1.begin(),v1.end());  vector<int> v2 {4,8,9,0}; sort(v2.begin(),v2.end());  vector<int> v;  v.resize(v1.size()+v2.size());  vector<int>::iterator iter;  **iter = set\_union(v1.begin(),v1.end(), \**  **v2.begin(), v2.end(), v.begin());**  v.resize(iter-v.begin());  for(int& i: v) cout << i; | v=014489  v1=1448, not v1=09, |
| vector<int> v1 {8,4,4,1}; sort(v1.begin(),v1.end());  vector<int> v2 {4,8,9,0}; sort(v2.begin(),v2.end());  vector<int> v;  v.resize(v1.size()+v2.size());  vector<int>::iterator iter;  **iter = set\_difference(v1.begin(),v1.end(), \**  **v2.begin(), v2.end(), v.begin());**  v.resize(iter-v.begin());  for(int& i: v) cout << i; | v=14  v1=1448, v2=0489,  difference 14 |
| vector<int> v1 {8,3,4,4,1}; sort(v1.begin(),v1.end());  vector<int> v2 {4,8,9,0}; sort(v2.begin(),v2.end());  vector<int> v;  v.resize(v1.size()+v2.size());  vector<int>::iterator iter;  iter = set\_symmetric\_difference(v1.begin(),\  v1.end(), v2.begin(), v2.end(), v.begin());  v.resize(iter-v.begin());  for(int& i: v) cout << i; | output: 01349  v1=13448, v2=0489  symmetric\_difference 1349 |
| vector<int> v1 {8,4,4,1}; sort(v1.begin(),v1.end());  vector<int> v2 {4,8,9,0}; sort(v2.begin(),v2.end());  vector<int> v;  v.resize(v1.size()+v2.size());  vector<int>::iterator iter;  iter = set\_intersection(v1.begin(),v1.end(), \  v2.begin(), v2.end(), v.begin());  v.resize(iter-v.begin());  for(int& i: v) cout << i | v=48  v1=1448, v2=0489, intersection 48 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## partition

### partition

**iter=algorithm::partition(c.begin(), c.end(), <function>)**. split container into two parts by returns of function.

**algorithm::stable\_partition(c.begin(), c.end(), <function>)**.

**algorithm::partition\_copy(c.begin(), c.end(), back\_inserter(c2), back\_inserter(c3), <function>)**.

vector

|  |  |
| --- | --- |
| vector<int> v {8,10,4,2,3,1};  vector<int>::iterator iter;  iter = partition(v.begin(), v.end(), \      [](const int& i){  return i%2==1;});  **vector<int> v1(distance(v.begin(), iter));**  **vector<int> v2(distance(iter, v.end()));**  copy(v.begin(), iter, v1.begin());  for(int& i: v1) cout << i;  copy(iter, v.end(), v2.begin());  for(int& i: v2) cout << i; | v1=13, v2=2108  partition elements into two parts and the order of elements may differ from the original order. |
| vector<int> v {8,10,4,2,3,1};  vector<int>::iterator iter;  iter = stable\_partition(v.begin(), v.end(), \      [](const int& i){ return i%2==1; });  vector<int> v1(distance(v.begin(), iter));  vector<int> v2(distance(iter, v.end()));  copy(v.begin(), iter, v1.begin());  for(int& i: v1) cout << i;  copy(iter, v.end(), v2.begin());  for(int& i: v2) cout << i; | v1=31  v2=81042  partition into two parts and no sorting |
| vector<int> v {8,10,4,2,3,1};  vector<int> v1, v2;  partition\_copy(v.begin(), v.end(),  back\_inserter(v1), back\_inserter(v2),\  [](const int& i){ return i%2==1;});  for(int& i: v1) cout << i;  for(int& i: v2) cout << i; | v1=31  v2=81042 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### split

split string

|  |  |
| --- | --- |
| vector<string> split(const string& str, string sep=","){      static vector<string> items;  **int i = str.find(sep);**      // cout << i << " " << str << endl;      if(i>=0){          items.push\_back(str.substr(0,i));          i += sep.size();          string sub = str.substr(i);          return split(sub,sep);      }      if(!str.empty())          items.push\_back(str);      return items;  } | use methods defined in <string>  string s = "abc,123,a";  vector<string> items = split(s, ",");  for(string& i: items) cout << i << "\t"; |
| vector<string> split(string& str, string sep=","){      static vector<string> items;      string::iterator iter;  **iter = search(str.begin(), str.end(), \**  **sep.begin(), sep.end());**      items.push\_back(string(str.begin(), iter));      if(iter != str.end()){          iter += sep.size();          string sub(iter, str.end());          split(sub, sep);      }      return items;  } | use methods of <algorithm> |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## Search

### find/search

1. Return an iterator to the first detected element, or the iterator to "last".

2. Type of elements (int, float, string) should support operator==, or overload operator==() if type of elements is object.

3. return index: "algorithm::distance(c.begin(), iter);"

**1. find single element**

**first\_detect\_iter/last\_iter = algorithm::find(first, last, const val)**:

**first\_detect\_iter/last\_iter = algorithm::find\_if(first, last, unary func)**

**first\_detect\_iter/last\_iter = algorithm::find\_if\_not(first, last, unary func)**

**first2\_detect\_iter/last\_iter = algorithm::find\_first\_of(first, last, first2, last2, <binary func>)**: find any one of elements of one container matched with element in another container. That is could work like regular expression.

**2. find two adjacent identical elements**

**first\_detect\_iter/last\_iter = algorithm::adjacent\_find(first, last, <binary func>)** work on two adjacent elements of container.

**3. find the subset of a container**

**first2\_detect\_iter/last\_iter = algorithm::search(first, last, first2, last2, <binary func>)**. find the first occurrence of container c2 in container c.

**first2\_right-detect\_iter/last\_iter = algorithm::find\_end(first, last, first2, last2, <binary func>).** find the last occurrence of container c2 in container c

**first\_detect\_iter/last\_iter = algorithm::search\_n(first, last, size, const val, <binary func>)**. find the first occurrence of n continuous elements in container

string

|  |  |
| --- | --- |
| string s = "ATCGAT";  string::iterator iter;  **iter = find(s.begin(), s.end(), 'G');**  **cout << \*iter;** | ouput:G  the 3rd argument must be same data type of elements of the container. |
| string s = "AcTCGAT";  string::iterator iter;  **iter = find\_if(s.begin(), s.end(), \**  **[](const char& c){ return c=='c' || c=='C';});**  cout << \*iter; | output:c |
| string s = "ABC3DG40";  string digits="0123456789";  string::iterator iter;  **iter = find\_if(s.begin(), s.end(), \**  **[&digits](const char& c){**  **return digits.find(c)!=string::npos; });**  cout << \*iter; | output: 3  find the digits: find characters which could be one of characters. |
| string s = "ATXGCNGTI";  string sub = "INX";  string::iterator iter;  **iter = find\_first\_of(s.begin(), s.end(), \**  **sub.begin(), sub.end());**  cout << \*iter; | output:X  return iterator if any character "INX" is matched. |
| string s = "ATxGCNGTI";  string sub = "INX";  string::iterator iter;  **iter = find\_first\_of**(s.begin(), s.end(), sub.begin(), sub.end(), [](const char& a, const char& b){  return toupper(a)==toupper(b);  });  cout << \*iter; | output:x |
|  |  |
|  |  |
|  |  |
| string s = "ATTTGCCGTI";  string::iterator iter;  **iter = adjacent\_find(s.begin(), s.end());**  cout << \*iter; | output:T  find two continuous characters like "TTT" or "CC" |
| string s = "ATTTGCCGTI";  string::iterator iter;  **iter = adjacent\_find(s.begin(), s.end(), \**  **[](const char& a, const char& b){**  **return a=='G'&&b=='C';**  **});**  cout << \*iter;  cout << distance(s.begin(), iter); | output: G4  find "GC" in string. |
|  |  |
|  |  |
| string s = "AGCCTCATTGCCGAT";  string sub = "GCC";  string::iterator iter;  **iter = find\_end(s.begin(), s.end(), sub.begin(), sub.end());**  cout << distance(s.begin(), iter); | output: 9  find matched sub string. two currency of GCC at 1 and 9. |
| string s = "AGCCTCATTGCCGAT";  string sub = "GCC";  string::iterator iter;  **iter = search(s.begin(), s.end(), sub.begin(), sub.end());**  cout << distance(s.begin(), iter); | output: 1  find matched sub string. two currency of GCC at 1 and 9. |
| string s = "AGCTCATTgccGAT";  string sub = "GCC";  string::iterator iter;  iter = search(s.begin(), s.end(), sub.begin(), sub.end(), [](char& a, char& b){  return toupper(a) == toupper(b);  });  cout << distance(s.begin(), iter); | 8 |
| string s = "AGCCCTCATTGCCGAT";  string sub = "CC";  string::iterator iter;  **iter = search\_n(s.begin(), s.end(), 2, 'C');**  cout << distance(s.begin(), iter); | output:2 |
|  |  |
|  |  |

vector

|  |  |
| --- | --- |
| vector<int> v {1,2,3,2,4};  vector<int>::iterator iter;  iter = find(v.begin(), v.end(), 2);  cout << distance(v.begin(), iter); | output:1  2 is at 1 and 3. return the first occurrence. |
| vector<int> v {1,3,10,2,4};  vector<int>::iterator iter;  iter = find\_if(v.begin(), v.end(), \      [](const int& i){ return i%2==0;});  cout << distance(v.begin(), iter); | output:2  find first even number |
| vector<int> v {3,10,20,1,4,2};  vector<int> v2 {1,2};  vector<int>::iterator iter;  iter = find\_first\_of(v.begin(), v.end(), \  v2.begin(), v2.end());  cout << distance(v.begin(), iter); | output:3 |
| vector<int> v {3,1,2,1,4,1,2};  vector<int> v2 {1,2};  vector<int>::iterator iter;  iter = find\_end(v.begin(), v.end(), v2.begin(), v2.end());  cout << distance(v.begin(), iter); | output:5 |
| vector<int> v {3,1,2,1,4,1,2};  vector<int> v2 {1,2};  vector<int>::iterator iter;  iter = search(v.begin(), v.end(), v2.begin(), v2.end());  cout << distance(v.begin(), iter); | output:1 |
| vector<int> v {3,1,1,1,4,1,2};  vector<int>::iterator iter;  **iter = search\_n(v.begin(), v.end(), 3, 1);**  cout << distance(v.begin(), iter);  iter = search\_n(v.begin(), v.end(), 1, 1);  cout << iter - v.begin();  iter = search\_n(v.begin(), v.end(), 7, 1);  cout << iter - v.begin(); | output:117  find {1,1,1}, or {1,1}  Note: if size=0, don't move iterator. Here \*iter=0. |
| vector<int> v {3,1,3,3,4,1,2};  vector<int>::iterator iter;  iter = adjacent\_find(v.begin(), v.end());  cout << distance(v.begin(), iter); | output:2  find first occurrence of two continuous identical elements. Here is {3,3} |
| vector<int> v {0,1,3,6,4,1,2};  vector<int>::iterator iter;  iter = adjacent\_find(v.begin(), v.end(),\      [](const int& a, const int& b){          return a!=0 && b/a==2;  });  cout << distance(v.begin(), iter); | output:2  find first occurrence of two continuous elements with specific associations. Here is {3,6} |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

map

|  |  |
| --- | --- |
| int t[]={10,5,9,6,2,4,7,8,3,1};  map<int, int> m;  for(int i=0; i<10; i++)  m[i] = i;  map<int, int>::iterator iter;  **pair<const int, int> p(5, 5);**  **iter = find(m.begin(), m.end(), p);**  if (iter != m.end())  cout << iter->second; | 5  Note: "**pair<int, int> p(5, 5);" will cause error.** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### Compare

**bool=algorithm::equal(v.begin(), v.end(), v2.begin())**. compare one by one. Return true if v is prefix of v2 **or** v==v2.

**pair<iter1, iter2>=algorithm::mismatch(v.begin(), v.end(), v2.begin()).** compare one by one. Return pair type of iterator which the first occurrence of mismatching.

**bool=algorithm::lexicographical\_compare(v.begin(), v.end(), v2.begin(), v2.end())**. check if v is prefix of v2 **and** v is shorter than v2.

bool=algorithm::includes(c.begin(), c.end(), c2.begin(), c2.end(), <binary function>). check if c contain c2. binary operator is < if no binary function.

|  |  |
| --- | --- |
| string s("GAATTC");  string ss("GAATTC");  bool res = equal(s.begin(), s.end(), ss.begin());  cout << res; | output:1 |
| string s("GAATT");  string ss("GAATTC");  bool res = equal(s.begin(), s.end(), ss.begin());  cout << res; | ouput:1 |
| string s("GAATTCG");  string ss("GAATTC");  bool res = equal(s.begin(), s.end(), ss.begin());  cout << res; | output:0 |
| string s("GAATTCCG");  string ss("GAATTC");  bool res = equal(s.begin(), s.end(), ss.begin());  cout << res; | ouput:0 |
| string s("GAATT");  string ss("GAATTC");  **pair<string::iterator, string::iterator> mypair;**  **mypair = mismatch(s.begin(), s.end(), ss.begin());**  cout << distance(s.begin(), mypair.first);  cout << distance(ss.begin(), mypair.second); | output:55  if s=ss="GAATTC", return 66 |
| string s("GAATTC");  string ss("GAATTC");  bool res = lexicographical\_compare( \  s.begin(), s.end(), ss.begin(), ss.end());  cout << res; | output:0 |
| string s("GAATT");  string ss("GAATTC");  bool res = lexicographical\_compare( \  s.begin(), s.end(), ss.begin(), ss.end());  cout << res; | output:1 |
| string a="eqrtyw", b="ert";  cout << includes(a.begin(),a.begin()+6,  b.begin(), b.begin()+3); | output:1  (!(a<b) && !(b<a)) both should be sorted.  e q r t y w  e r t |
| string a="eqrtyw", b="ret";  cout << includes(a.begin(),a.begin()+1,  b.begin(), b.begin()+1); | output:0  (!(a<b) && !(b<a))  e q r t y w  ret  b is not sorted. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**vector**

|  |  |
| --- | --- |
| vector<int> v {1,2}; vector<int> v2 {1,2};  bool f = equal(v.begin(), v.end(), v2.begin()); | output: f=1 |
| vector<int> v {1,2};  vector<int> v2 {1,2, 3};  bool f = equal(v.begin(), v.end(), v2.begin()); | output:f=1  v is prefix of v2. |
| vector<int> v {1,2,3};  vector<int> v2 {1,2};  bool f = equal(v.begin(), v.end(), v2.begin()); | ouput:f=0 |
| vector<int> v {1,2};  vector<int> v2 {1,2};  bool f = lexicographical\_compare( \      v.begin(), v.end(), v2.begin(), v2.end()); | outpu:f=0 |
| vector<int> v {1,2};  vector<int> v2 {1,2,3};  bool f = lexicographical\_compare( \      v.begin(), v.end(), v2.begin(), v2.end()); | output:f=1  v is shorter than v2. |
|  |  |
| vector<int> v {1,2,3}; vector<int> v2 {1,2};  pair<vector<int>::iterator, vector<int>::iterator> mypair;  mypair = mismatch( v.begin(), v.end(), v2.begin(), v2.end());  cout << distance(v.begin(), mypair.first);  cout << distance(v2.begin(), mypair.second); | output:22 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**integer**

|  |  |
| --- | --- |
| cout<< max(12, 3);  cout<< min(12, 3); | output:123 |
| cout<< max('A', 'Z');  cout<< min('1', '2'); | output:Z1 |

### binary search

**Note: 1. elements in containers should already be sorted. or they return wrong result though no error is reported.**

**iter=algorithm::lower\_bound(c.begin(), c.end(), <bound value>)**. \*iter = the smallest bound value or neighboring larger value. Get all elements smaller than the bound.

**iter=algorithm::upper\_bound(c.begin(), c.end(), <bound value>)**. \*iter = the neighboring larger value. Get all elements greater than the bound.

**pairt<iter1, iter2>=algorithm::equal\_range(c.begin(), c.end(), <bound value>)**. iter1 is lower\_bound, iter2 is upper\_bound. iter1 and iter2 could pointer to the same iterator.

|  |  |
| --- | --- |
| vector<int> v {1,2,3,4,5,6,7,8,9};  vector<int>::iterator iter;  **iter = lower\_bound(v.begin(), v.end(), 6);**  cout << \*iter << distance(v.begin(), iter); | output:65  index of value=6 is 5. |
| vector<int> v {1,2,3,4,5,8,9};  vector<int>::iterator iter;  iter = lower\_bound(v.begin(), v.end(), 6);  cout << \*iter << distance(v.begin(), iter); | output: 85 |
| vector<int> v {1,2,3,6,6,8,9};  vector<int>::iterator iter;  iter = lower\_bound(v.begin(), v.end(), 6);  cout << \*iter << distance(v.begin(), iter); | 63 |
| vector<int> v {1,2,3,4,5,6,7,8,9};  vector<int>::iterator iter;  **iter = upper\_bound(v.begin(), v.end(), 6);**  cout << \*iter << distance(iter, v.end()); | output:73 |
| vector<int> v {1,2,3,4,5,7,8,9};  vector<int>::iterator iter;  **iter = upper\_bound**(v.begin(), v.end(), 6);  cout << \*iter << distance(iter, v.end()); | output: 73 |
| vector<int> v {1,2,3,4,5,6,7,8,9};  pair<vector<int>::iterator, vector<int>::iterator> iter;  **iter = equal\_range(v.begin(), v.end(), 6);**  cout << \*iter.first << \*iter.second; | output:67 |
| vector<int> v {1,2,5,6,6,6,7,8,9};  pair<vector<int>::iterator, vector<int>::iterator> iter;  iter = equal\_range(v.begin(), v.end(), 6);  cout << distance(v.begin(), iter.first);  cout << distance(iter.second, v.end()); | 33 |
| vector<int> v {1,2,3,4,5,5,7,8,9};  pair<vector<int>::iterator, vector<int>::iterator> iter;  iter = equal\_range(v.begin(), v.end(), 6);  cout << \*iter.first << \*iter.second; | 77 |
|  |  |
|  |  |
|  |  |

### check consistency

bool=algorithm::all\_of(c.begin(), c.end(), <function>). test condition on all elements in range.

bool=algorithm::any\_of(c.begin(), c.end(), <function>). test condition on any one of elements in range.

bool=algorithm::none\_of(c.begin(), c.end(), <function>). test condition on all elements in range.

|  |  |
| --- | --- |
| vector<int> v {2,4,0};  cout << all\_of(v.begin(), v.end(),  [](const int& i){ return i%2==0; }); | 1  check if all elements are even. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## random

random\_shuffle(c.begin(),c.end()). shuffling

shuffle(c.begin(),c.end()).

|  |  |
| --- | --- |
| vector<int> v {1,2,3,4,5,6,7,8,9,0};  **random\_shuffle**(v.begin(),v.end());  for(int& i: v) cout << i; | 9203168457 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## Sort

**sort all elements**

**algorithm::sort(c.begin(), c.end())**. sort elements by ASCII code(0-255) ascendingly.

**algorithm::sort(c.rbegin(), c.rend())**. sort descendingly.

**algorithm::sort(c.begin(), c.end(), <function>)**. sort elements determined by the bool returns of function

**algorithm::stable\_sort(c.begin(), c.end())**. upgrade sort(). don't change relative order if there are some equal elements.

**algorithm::stable\_sort(c.begin(), c.end(), <function>)**.

**partial sort:**

**algorithm::partial\_sort(<iter1>, <iter2>, <iter3>)**. sort elements of iter2-iter3 ascending, and move the sorted part the beginning. Left elements of iter1-iter2 unsorted.

**algorithm::partial\_sort\_copy(<iter1>, <iter2>, <iter3>, back\_inserter(c2))**.

algorithm::is\_sorted(c.begin(), c.end())

**algorithm::is\_sorted\_until(c.begin(), c.end())**. return iterator of which element is not ascending sorted.

algorithm::nth\_element(c.begin(), c.begin()+<nth by index>, c.end())

string

|  |  |
| --- | --- |
| string str = "0asdfHXhjk 24l.";  sort(str.begin(), str.end());  cout << str; | output: .024HXadfhjkls |

vector

|  |  |
| --- | --- |
| vector<int> v {10,-2,0,13,4};  **sort(v.begin(), v.end());**  for(int& i: v) cout << i << ','; | output: -2,0,4,10,13, |
| vector<int> v {10,-2,0,13,4};  **sort(v.rbegin(), v.rend());**  for(int& i: v) cout << i << ','; | output: 13,10,4,0,-2, |
| vector<string> vs {"0ab", " a", "zcd", "ef"};  **sort(vs.begin(), vs.end());**  for(string& i: vs) cout << i << ','; | output: a,0ab,ef,zcd, |
| vector<string> vs {"chr10", "chr3", "chr20", "chr19", \  "chr1", "chrX", "chrY"};  **sort(vs.begin(), vs.end(),**  **[](const string& a, const string& b){**      try{          int x = stoi(a.substr(3));          int y = stoi(b.substr(3));          return x < y;      }catch(...){ return a < b; }  });  for(string& i: vs) cout << i << ','; | output: chr1,chr3,chr10,chr19,  chr20,chrX,chrY,  sort chromosome |
| vector<int> v {9,8,7,6,5,4,3,2,1};  **partial\_sort**(v.begin(), v.begin()+5, v.end()); | v=1,2,3,4,5,9,8,7,6, |
| vector<int> v {9,1,2,3,4,5,6,7,8,0};  **nth\_element(v.begin(), v.begin()+3, v.end());**  for(int& i: v) cout << i << ','; | 1,0,2,3,4,5,6,7,8,9,  the 3th of index by ascending is 3. All element before 3 are smaller than 3, and those after 3 are greater. but the order of two parts may vary. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## update

### replace/fill

algorithm::replace(c.begin(), c.end(), <old value>, <new value>).

algorithm::replace\_if(c.begin(), c.end(), <function>, <new value>).

algorithm::replace\_copy(c.begin(), c.end(), back\_inserter(c2), <old value>, <new value>).

algorithm::replace\_copy\_if(c.begin(), c.end(), back\_inserter(c2), <function>, <new value)>).

**algorithm::fill(c.begin(), c.end(),<value>)**. fill part of containers with a certain value

**algorithm::fill\_n(c.begin(),<n times>, <value>)**. fill part of containers with a certain value

**algorithm::generate(c.begin(), c.end(), <function>)**. build a container of which elements is returned by the function

**algorithm::generate\_n(c.begin(), <n times>, <function>)**. build container with n times of repeated elements which is determined by the function

string

|  |  |
| --- | --- |
| string s = "ATCGTCAT";  **replace**(s.begin(),s.end(), 'T', 'C'); | s=ACCGCCAC |
| string s = "ATcGTCAT";  **replace\_if**(s.begin(),s.end(), \  [](const char& c){ return c=='c'; }, 'C'); | s=ATCGTCAT |
| string s = "ATGTCAT"; string ss;  **replace\_copy**(s.begin(),s.end(), back\_inserter(ss), 'T', 'C'); | ss=ACGCCAC |
| string s = "ATcGTCAT"; string ss;  **replace\_copy\_if**(s.begin(),s.end(), back\_inserter(ss), \  [](const char& c){ return c=='c';}, 'C'); | ss=ATCGTCAT |
| string s = "AITGTCZ", nt="ATGC", ss;  **replace\_copy\_if**(s.begin(),s.end(), back\_inserter(ss), [&nt](const char& c){  return nt.find(c)==nt.npos;  }, 'N');  cout << ss; | ss=ANTGTCN  format DNA sequence |
| string s;  s.resize(5);  **fill(s.begin(), s.end(),'N');** | NNNNN |
| string s; s.resize(5);  **fill\_n(s.begin(), 5,'N');** | NNNNN |
| string s;s.resize(5);  **generate(s.begin(), s.end(), [](){return 'A';});** | AAAAA |
| string s;s.resize(5);  **generate\_n(s.begin(), 2, [](){return 'A';});**  cout << s; | AA |
|  |  |
|  |  |

vector

|  |  |
| --- | --- |
| vector<int> v {1,2,2,3,3,4};  **replace(v.begin(),v.end(), 2, 5);**  for(int& i: v) cout << i; | 155334 |
| vector<int> v {1,2,2,3,3,4};  **replace\_if**(v.begin(),v.end(), [](const int& i){  return i%2==0;},0);  for(int& i: v) cout << i; | 1000330 |
| vector<int> v {1,2,2,3,3,4};  vector<int> v2;  **replace\_copy**(v.begin(),v.end(), back\_inserter(v2), 2, 0);  for(int& i: v2) cout << i; | 100334 |
| vector<int> v {1,2,2,3,3,4};  vector<int> v2;  **replace\_copy\_if**(v.begin(),v.end(), back\_inserter(v2), \  [](const int& i){ return i%2==0;},0);  for(int& i: v2) cout << i; | 1000330 |
|  |  |
| vector<int> v(5);  **fill**(v.begin(),v.end(),2); | 22222 |
| vector<int> v(5);  **fill\_n**(v.begin(), 2, 5); | 55000 |
| vector<int> v(5);  **generate**(v.begin(), v.end(), [](){      static int i=0; return ++i;  });  for(int& i: v) cout << i; | 12345 |
| vector<int> v(5);  **generate**(v.begin(), v.end(), [](){      static int i=0; return i += 2;  });  for(int& i: v) cout << i; | 246810 |
| vector<int> v(5);  **generate**(v.begin(), v.end(), [](){return 2;});  for(int& i: v) cout << i; | 22222 |
| vector<int> v(5);  **generate\_n**(v.begin(), 2, [](){return 2;});  for(int& i: v) cout << i; | 22000 |
|  |  |
|  |  |

### remove duplicates

it is better to use sort() before unique().

**algorithm::unique(c.begin(), c.end())**. remove all consecutive duplicates in range. Non-consecutive duplicates will not be removed.

algorithm::unique(c.begin(), c.end(), <function>).

algorithm::unique\_copy(c.begin(), c.end(), back\_inserter(c2)).

algorithm::unique\_copy(c.begin(), c.end(), back\_inserter(c2), <function>).

|  |  |
| --- | --- |
| vector<int> v {1,2,2,3,4};  vector<int>::iterator iter;  iter=unique(v.begin(), v.end());  for(int& i: v) cout << i; | output:12344 |
| vector<int> v {1,1,2,2,3,3,4,2};  vector<int>::iterator iter;  iter=unique(v.begin(), v.end());  v.erase(iter, v.end());  for(int& i: v) cout << i; | output:12342  first sort then remove duplicates |
| vector<int> v {1,1,2,2,3,3,4,2};  vector<int> v2;  sort(v.begin(), v.end());  unique\_copy(v.begin(), v.end(), back\_inserter(v2));  for(int& i: v2) cout << i; | output:1234  first sort, then remove duplicates, copy unique to new container |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## Modify

### Copy

result\_last\_iter = algorithm::**copy**(first, last, c2.begin()/back\_inserter(c2)/inserter(c2, c2.begin())

result\_last\_iter = algorithm::**copy\_if**(first, last, c2.begin()/back\_inserter(c2), func)

result\_last\_iter = algorithm::**copy\_n**(first, n, c2.begin()/back\_inserter(c2)). copy n elements from the [first] iterator

**result\_last\_iter = algorithm::copy\_backward(first, last, c2.end())**

result\_last\_iter = algorithm::**reverse\_copy**(first, last, c2.begin()/back\_inserter(c2))

result\_last\_iter = algorithm::**rotate\_copy**(first, middle, last, c2.begin()/back\_inserter(c2))

result\_last\_iter = algorithm::**unique\_copy**(first, last, c2.begin()/back\_inserter(c2))

pair<last\_iter2, last\_iter3> range = **algorithm::partition\_copy**(first, last, c2.begin(), c3.begin(), func), partition range of sorted container c into two containers into c2 and c3.

pair<last\_iter2, last\_iter3> range = **algorithm::partition\_sort\_copy**(first, last, c2.begin(), c3.begin(), func): sort container c and then partition range of c into two containers c2 and c3.

last\_iter2 = algorithm::**remove\_copy**(first, last, c2.begin(), val)

last\_iter2 = algorithm::**remove\_copy\_if**(first, last, c2.begin(), func)

last\_iter2 = algorithm::**replace\_copy**(first, last, c2.begin(), val, new-val)

last\_iter2 = algorithm::**replace\_copy\_if**(first, last, c2.begin(), func, new-val)

|  |  |  |
| --- | --- | --- |
| **action** | **code** | **Note** |
| string | string str = "abc"; string ss;  **for\_each(str.begin(), str.end(), \**  **[&](const char& c){ ss.push\_back(c);});**  cout << ss; |  |
| string str = "abc"; string ss;  **copy(str.begin(), str.end(), back\_inserter(ss));**  cout << ss; | the same as  string ss; ss.resize(3);  copy(str.begin(), str.end(), ss.begin()); |
| string str = "abcdef";  string ss;  **copy\_n(str.begin()+1, 2, back\_inserter(ss));**  cout << ss; | output: bc  the same as :  string ss; ss.resize(3);  copy\_n(str.begin()+1, 2, ss.begin()); |
| string str = "ANXTGTBA";  string nt="ATGC"; string ss;  **copy\_if(str.begin(), str.end(), back\_inserter(ss),\**  **[&nt](const char& c){**  **string::size\_type pos = nt.find(c);**  **return (pos==nt.npos) ? false : true;**  **});** cout << ss; | output:ATGTA |
| string str = "abcdef";  str.resize(str.size()+3);  copy\_backward(str.begin(), str.begin()+6, str.end());  cout << str; | abcabcdef  Note: |
| string str = "ATGTA";  int s = str.size(); str.resize(s\*2-1);  **copy\_backward(str.rbegin()+s, str.rbegin()+(s\*2-1), str.end());**  cout << str; | ATGTATGTA  mirror sequence |
|  |  |
|  |  |
|  |  |
|  |  |
| vector | vector<int> v {1,2,3,4};  vector<int> v2;  **copy(v.begin(), v.end(), back\_inserter(v2));**  for(int& i: v2) cout << i << ','; | 1,2,3,4,  The same as  vector<int> v2; v2.resize(v.size());  copy(v.begin(), v.end(), v2.begin()); |
| vector<int> v {1,2,3,4};  vector<int> v2;  **copy(v.rbegin()+1, v.rbegin()+3, back\_inserter(v2));**  for(int& i: v2) cout << i << ','; | 2,3, |
| vector<int> v {1,2,3,4};  vector<int> v2;  vector<int>::iterator iter;  **iter=copy\_n(v.begin(), 3, back\_inserter(v2));**  **cout << (iter=v2.end());** | 1  v2=123 |
| vector<int> v {1,2,3,4,5,6};  vector<int> v2;  **copy\_if(v.begin(), v.end(), back\_inserter(v2),**  **[](const int& i){**  **return i%2==0;;**  **});**  for(int& i: v2) cout << i; | 246 |
| vector<int> v {1,2,3};  **v.resize(v.size()\*2);**  **copy\_backward(v.begin(), v.begin()+4, v.end());**  for(int& i: v) cout << i; | 123123 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| linked list |  |  |
| deque |  |  |
|  |  |  |
| map | map <char, int> m = {{'a',1}};  map <char, int> m2 = m; |  |
| map <char, int> m = {{'a',1}};  map <char, int> m2(m.begin(), m.end()); |  |
|  | map<char, int> m = {{'a',1}};  map<char, int> m2;  **map<char, int>::iterator iter = m2.begin();**  **copy(m.begin(), m.end(), inserter(m2,iter));**  for(pair<char,int> el: m2)  cout << el.first << el.second; | a1  Note: back\_inserter() is not working for associative containers. |
|  | map<char, int> m {{'b',2}, {'c',-2}, {'e',4}};  map<char, int> m2;  // map<char, int>::iterator iter;  **copy\_n(m.begin(), 2, inserter(m2, m2.begin()));**  for(pair<char, int> el:m2)  cout << el.first << el.second; | b2c-2 |
|  |  |  |
|  |  |  |
|  |  |  |
| multi-map |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| set |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| multi-set |  |  |
|  |  |  |
| stack |  |  |
| queue |  |  |
| priority-queue |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Note: . size\_t copy(char \*p, size\_t length, size\_t pos=0). strcopy() only works for C, where string is char array.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### Reverse

c(c.rbegin(), c.rend())

algorithm::transform(c.rbegin(), c.rend(), back\_inserter(c2), <lambda func>)

algorithm::accumulate(c.rbegin(), c.rend(), <initial value>, <lambda func>)

algorithm::for\_each(c.rbegin(), c.rend(), <lambda func>)

algorithm::copy(c.rbegin(), c.rend(), c2.begin())

algorithm::copy\_backward(c.begin(), c.end(), c2.end())

algorithm::reverse(c.begin(), c.end()).

algorithm::reverse\_copy(c.begin(), c.end(), back\_inserter(c2))

|  |  |  |
| --- | --- | --- |
| **action** | **code** | **Note** |
| string | string ss = string(str.rbegin(), str.rend()); cout << ss; | string str = "ATCGAT";  output: ss==TAGCTA or str==TAGCTA |
| string ss;  **transform(str.rbegin(), str.rend(), back\_inserter(ss),**      [](const char& c){ return c; }); cout << ss; |
| **string f = accumulate(str.rbegin(), str.rend(), string(""),**  **[](string f, const char& c){ return  f+c;});**  cout << f; |
| string ss;  **for\_each(str.rbegin(), str.rend(), \**  **[&](const char& c){ ss.push\_back(c); });** |
| string ss; ss.resize(str.size());  **copy(str.rbegin(), str.rend(), ss.begin());** |
| string ss; ss.resize(str.size());  **copy\_backward(str.rbegin(), str.rend(), ss.end());** |
| string ss;  **reverse\_copy(s.begin(), s.end(), back\_inserter(ss));** |
| vector | **vector<int> v2(v.rbegin(),v.rend());** | vector<int> v {1,2,3,4}; |
| vector<int> v2;  **transform(v.rbegin(),v.rend(), back\_inserter(v2),**  **[](const int& i){ return i; });**  for(int& i: v2) cout << i << ','; |
| vector<int> v2;  **for\_each(v.rbegin(),v.rend(), [&](int& i){ v2.push\_back(i); });** |
| **reverse(v.begin(), v.end()); //reverse on-site** |
| vector<int> v2;  **reverse\_copy(v.begin(), v.end(), back\_inserter(v2));** |
| vector<int> v2;  copy(v.rbegin(),v.rend(), back\_inserter(v2)); |
|  |
| linked list |  |  |
| deque |  |  |
|  |  |  |
| map | map<char, int> m {{'b',2}, {'a',1},{'c',3}, {'e',4}};  map<int, char> m2;  for\_each(m.begin(),m.end(), [&m2](const pair<char, int>& el){  cout << el.first << el.second;  m2.insert(pair<int, char>(el.second, el.first) );  }); | switch key-value |
| multi-map |  |  |
| set |  |  |
| multi-set |  |  |
|  |  |  |
| stack |  |  |
| queue |  |  |
| priority-queue |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### swap/rotate

**algorithm::swap(<index 1>, <index 2>)**. swap two elements

**algorithm::iter\_swap(&<index 1>, &<index 2>)**. swap two elements of one containers

**algorithm::iter\_swap(iter1, iter2)**. swap two elements of two containers

**iter2=algorithm::swap\_ranges(first, last, c2.begin())**. swap middle part of one container with tail part of another container

**iter=algorithm::swap\_ranges(c.begin(), c.begin()+2, c.begin()+4)**. swap middle part and tail part within one container.

**algorithm::rotate(<iter 1>, <iter 2>, <iter 3>)**. set two neighboring parts encompassed by three bounds, and swap the two parts in one container. iter3>iter2>iter1. or cause runtime error.

|  |  |  |
| --- | --- | --- |
| **action** | **code** | **Note** |
| string | string s("GAATTC");  swap(s[0],s[5]); | s=CAATTG |
| string s("GAATTC");  iter\_swap(&s[0], &s[5]); | s=CAATTG |
| string s("GAATTC");  **swap\_ranges(s.begin(), s.begin()+3,**  **s.begin()+3);** | s=TTCGAA |
| string s("GAATTC");  string ss("GGG");  swap\_ranges(s.begin(), s.begin()+3,  ss.begin()); | ssGGGTTC |
| string s("GAATTC");  string ss("GGG");  swap\_ranges(s.rbegin(), s.rbegin()+3,  ss.begin()); | ss=GAAGGG |
| string s("GAATTC");  rotate(s.begin(), s.begin()+3, s.begin()+6); | s=TTCGAA  swap GAA with TTC |
| string s("GAATTC");  rotate(s.begin(), s.begin()+2, s.end()); | s=ATTCGA  swap GA with ATTC |
| string s("GAATTC");  rotate(s.begin(), s.begin()+2, s.begin()+4); | s=ATGATC  swap GA with AT |
| vector | vector<int> v {1,2,3,4,5,6};  swap(v[2], v[3]);  for(int& i: v) cout << i; | output:124356 |
| vector<int> v {1,2,3,4,5,6};  **iter\_swap(&v[2], &v[3]);**  for(int& i: v) cout << i; | output:124356  swap reference of the same container |
| int t[]={10,5,9,6,2,4,7,8,3,1};  vector<int> v(t, t+10);  **iter\_swap(v.begin(), t+9);**  for(int& i: v) cout << i; | v=10596247831->  1596247831  swap 10 in vector v and 1 in array t. |
| vector<int> v {1,2,3,4,5,6};  **swap\_ranges(v.begin(), v.begin()+2,**  **v.begin()+2);** | v=341256  swap first two with 3-4th elements |
| vector<int> v {1,2,3,4,5,6};  vector<int> v2 {7,8};  swap\_ranges(v.begin()+1, v.begin()+3,  v2.begin()); | v2=178456 |
| vector<int> v {1,2,3,4,5,6};  rotate(v.begin(), v.begin()+3, v.end()); | v=456123 |
| void swap(vector<int>& v, int a1, int a2, \  int b1, int b2){  rotate(v.begin()+a1, v.begin()+b1,\  v.begin()+b2+1);  int n = a1 + b2 - b1 + 1;  rotate(v.begin()+n, v.begin()+n+a2-a1+1,\  v.begin()+n+a2-a1+b2-b1+1);  }  int main(){  vector<int> v {1,2,3,4,5,6,7,8,9,0};  swap(v, 2, 3, 6, 8);  for(int& i: v) cout << i;  return 1;  } | v=1234567890  swap 3-4 (index 2-3) with 7-9 (index 6-8).  v=1278956340 |
| linked list |  |  |
| deque |  |  |
|  |  |  |
| map |  |  |
| multi-map |  |  |
| set |  |  |
| multi-set |  |  |
|  |  |  |
| stack |  |  |
| queue |  |  |
| priority-queue |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### move

**iter = algorithm::remove(c.begin(), c.end(), <remove value>)**. Move unremoved elements to the begin.

**iter = algorithm::remove\_if(c.begin(), c.end(), <function>)**.

**algorithm::remove\_copy(c.begin(), c.end(), c2.begin(), <remove value>)**.

**algorithm::remove\_copy\_if(c.begin(), c.end(), c2.begin(), <function>)**.

**move(c.begin(), c.end(), back\_inserter(c2) )**. copy elements from one container to another container

**mov\_backward(<iter1>, <iter2>, <iter3>)**.

string

|  |  |
| --- | --- |
| string s = "ATATA";  **remove**(s.begin(), s.end(),'A'); | s=TTATA |
| string s = "ATATA";  string::iterator iter;  **iter = remove(s.begin(), s.end(), 'A');**  string ss;  ss.resize(distance(s.begin(), iter));  copy(s.begin(), iter, ss.begin()); | TT  remove all 'A'. Here is complete code compared with the above |
| string s = "ATGTATA";  string ss; ss.resize(s.size());  **remove\_copy(s.begin(), s.end(), ss.begin(), 'A');** | TGTT |
| string s = "ATGCNXCI"; string nt="ATGC";  string::iterator iter;  **iter = remove\_if(s.begin(), s.end(), \**  **[&nt](const char& c){**  **return nt.find(c)==nt.npos;**  **});**  string ss;  ss.resize(distance(s.begin(), iter));  copy(s.begin(), iter, ss.begin()); | ATGCC  remove characters except "ATGC" |
| string s = "ATGCNXCI"; string nt="ATGC";  string ss;  ss.resize(s.size());  **remove\_copy\_if(s.begin(), s.end(), ss.begin(),**  **[&nt](const char& c){**  **return nt.find(c)==nt.npos;**  **});** | ATGCC |
|  |  |
| string s = "ATATA"; string ss;  **move**(s.begin(), s.begin()+2, back\_inserter(ss));  cout << s << ',' << ss; | ATATA,AT |
| string s = "ATAT";  s.resize(s.size()+4);  **move\_backward**(s.begin(), s.begin()+4, s.end()); | s=ATATATAT |

vector

|  |  |
| --- | --- |
| vector<int> v {1,2,3,4,1,1};  vector<int>::iterator iter;  **iter = remove(v.begin(), v.end(), 1);**  vector<int> v2(distance(v.begin(), iter));  copy(v.begin(), iter, v2.begin());  for(int& i: v2) cout << i; | 234 |
| vector<int> v {1,2,3,4,6,20};  vector<int>::iterator iter;  **iter = remove\_if(v.begin(), v.end(), \**  **[](const int i){**  **return i%2==0;**  **});**  vector<int> v2(distance(v.begin(), iter));  copy(v.begin(), iter, v2.begin());  for(int& i: v2) cout << i; | 13 |
| vector<int> v {1,2,3,1};  vector<int> v2(v.size());  **remove\_copy(v.begin(), v.end(), v2.begin(), 1);**  for(int& i: v2) cout << i; | 2300 |
| vector<int> v {1,2,3,4,6,20};  vector<int> v2(v.size());  **remove\_copy\_if(v.begin(), v.end(), v2.begin(),\**  **[](const int i){        return i%2==0;    });**  for(int& i: v2) cout << i; | 130000 |
|  |  |
|  |  |
|  |  |
|  |  |

### Convert

convert string to integer

1. string::stoi(str)

2. sstream::istringstream(str)

3. algorithm::transform()

convert string to integer

|  |  |
| --- | --- |
| cout << stoi("12");  cout << stoi("34abc12");  cout << stoi("5\n6"); | output:12345  wrong cout << stoi("");  wrong cout << stoi("abc12"); |
| int i=0;  istringstream("12")>>i; cout << i;  istringstream("34abc12")>>i;cout << i;  istringstream("5\n6")>>i;cout << i;  istringstream("")>>i;cout << i; // nothing assigned  istringstream("abc12")>>i;cout << i; // return 0 | output:1234550 |
|  |  |
|  |  |
|  |  |

convert vector

|  |  |
| --- | --- |
|  |  |
|  |  |
| int arr[] = {1, 2, 3};  int n = sizeof(arr) / sizeof(arr[0]);  **vector<int> v(arr, arr + n);** | convert array to vector |
| int arr[] = {1, 2, 3, 4, 5, 6};  int n = sizeof(arr) / sizeof(arr[0]);  **vector<int> v(arr+1, arr + n);** | copy array of 2nd-end into a vector |
| vector<int> vect1{ 10, 20, 30 };  vector<int> vect2(vect1.begin(), vect1.end()); | copy vector |
| vector<string> v {"a","b", "c"};  string ss= ""; string sep=",";  **ss=accumulate(v.begin(),v.end(), string(""),\**  **[&sep](string ss, string& s){**  **return ss=="" ? s : ss+sep+s;**  **});**  cout << ss; | output:a,b,c  join string vector to string. |
| vector<int> v {1,2,3,4};  string ss= ""; string sep=",";  **ss=accumulate(v.begin(),v.end(), string(""),\**  **[&sep](string ss, int& i){**  **string s = to\_string(i);**  **return ss=="" ? s : ss+sep+s;**  **});**  cout << ss; | output:1,2,3,4 |
|  |  |

array to containers

int t[]={10,5,9,6,2,4,7,8,3,1};

|  |  |
| --- | --- |
| set<int> s(t, t+10);  for\_each(s.begin(),s.end(),  [](const int& i){ cout<<i;}); | 12345678910 |
| int t[]={10,5,9,6,2,4,7,8,3,1};  set<int> s1(t, t+10);  vector<int> v1(s1.rbegin(), s1.rend());  swap\_ranges(s1.begin(), s1.end(), v1.begin()); | compiling error  set and vector can' be switched. |
|  |  |
|  |  |
|  |  |

# Template

## define templates

|  |  |
| --- | --- |
| template <typename T>  int func(const T &a, const T &b){  …  return 1;  } | <typename T> is template parameter list, which can't be empty. There is type T, of which actual type is determined at compile time depending on how the template function is used. |
| template <typename T>  int compare(const T& a, const T& b){  if(a<b) return -1;  if(a>b) return 1;  return 0;  }  int main(){  int a=3, b=6;  cout << compare(a,b);  float c=3.3;  cout << compare(float(b),c);  string s1="abc", s2="bcd";  cout << compare(s1,s2);  return 1;  } | -11-1  type T could be int, float, string, etc based on how the function is called. |
|  |  |
| template<typename T>  struct RGB {  T r, g, b;  }; | struct template |
|  |  |
|  |  |
|  |  |
|  |  |

## STL sequential container

## STL associative containers

## non-modifying STL algorithms

## modifying STL algorithms

## sorting STL operations

## STL merge operations

## STL utilities an functional libary

# function

function declaration, definition, and call

## basic

### declare, define, and call

Function declaration, is done to tell the compiler about the existence of the function. Function's return type, its name & parameter list is mentioned.

|  |  |
| --- | --- |
| int main(void){  int f();// declare the function  f();// call the function  return 0;  }  // define a function  int f(){ cout<<"func"; return 1; } | 1. should declare the function before call it, because the definition of this function is behind the main().  2. put f() preceeds main(). it is ok to ignore declaration step |
| int main(void){  int f(double); // declare the function  float i=2.3;  f(i); // call the function  return 0;  }  // define a function  int f(double x){ cout<<"func"; return 1; } |  |
| int main(void){  int f(int, int); // declare the function  int a=1,b=2;  f(a,b); // call the function  return 0;  }  // define a function  int f(int x, int y){ cout<<"func"; return 1; } |  |
| int main(void){  int f(int, void \*);  int a=1;  void \*b=nullptr;  f(a,b);  return 0;  }  int f(int x, void \*y){ cout<<"func"; return 1; } |  |
| int main()  int main(){ return 1; } | No needs to declare main() in C++ |
| int main(int argc, char \*argv[]){  cout<< "arguments:" << argc << endl;  cout<< "file name: " << argv[0] << endl;  return 0;  } | output:  arguments:1  file name: F:\cpp\template.exe |
| int main(void){  int func(void)  {return 2;}  int res = func();  cout << res << endl;  return 0;  } | compiling error  The func() can't be placed within the function main() in C++. But it is ok in C  one approach is to move the func() out of main() |
|  |  |

configure default arguments

|  |  |
| --- | --- |
| int fun(int p1 =1, int p2=1){  return p2<<p1;  }  int main(void){  cout<<fun(); //=2  cout<<fun(2); //=4  cout<<fun(2,3); //12  cout<<fun(fun(), fun(2)); //16  return 0;  } | output:2412  1<<1=2  1<<2=4  3<<2=3\*4=12  default arguments |
| bool f(bool x, bool y){  return x<y;  }  int main(void) {  cout << f(true,false);  return 0;  } | output: 0  bool func() return 0 or 1 |
|  |  |
|  |  |

### return statements

|  |  |
| --- | --- |
| int f(void){  int a=3;  }  int main(){  cout<< f();  return 0;  } | output: 1  Note: return statement is missing. return 1 or unpredictable value |
| int f(void){  int a;  return a=4;  }  int main(){  cout<< f();  return 0;  } | output: 4  Note: return a; |
| char f1(char c){return c=='z' ? 'a': c+1;}  char f2(char &c){c=f1(c);return c;}  int main(void){  char x='x';  cout<<f2(x);  cout<<f2(x);  cout<<f2(x)<<endl;  return 0;  } | output: yza  In f2(char &c), the variable c and the arguments x share the same address. |

### pass arguments

Pass value, pointer, or reference as arguments to function

A pointer is a variable that hold memory address of another variable

A reference is same object, just with a different name and reference must refer to an object. Since references can’t be NULL, they are safer to use. It is useful to change values of variables within functions.

1. A pointer can be re-assigned while reference cannot, and must be assigned at initialization only.

2. Pointer can be assigned NULL directly, whereas reference cannot.

3. Pointers can iterate over an array, we can use increment/decrement operators to go to the next/previous item that a pointer is pointing to.

4. A pointer is a variable that holds a memory address. A reference has the same memory address as the item it references.

5. A pointer to a class/struct uses ‘->'(arrow operator) to access it’s members whereas a reference uses a ‘.'(dot operator)

6. A pointer needs to be dereferenced with \* to access the memory location it points to, whereas a reference can be used directly.

Some equal patterns are showed as below:

output is test

|  |  |
| --- | --- |
| void print(char c[]){cout<<c; }  int main(void) {  print("test");  return 0;  } | pass char array  c pattern |
| void print(char \*c){cout<<c;}  int main(void) {  print("test");  return 0;  } | pass char pointer  c pattern |
| **void print(string c){cout<<c; }**  int main(void) {  string s="test"; print(s);  return 0;  } | pass string value  c++ |
| **void print(string& c){cout<<c; }**  int main(void) {  string s="test";print(s);  return 0;  } | pass string reference  c++ |
| **void print(string \*c){cout<<\*c;}**  int main(void) {  string s="test"; string \*p=&s; print(p);  return 0;  } | pass string pointer  c++ |

For char, char[] or string type, compare the approaches below:

|  |  |
| --- | --- |
| void f(char \*p){ cout<< \*p;}  int main(void) {  string s="good"; f(&s[0]);  return 0;  } | output: g  pass char pointer |
| void f(char \*p){ cout<< p; }  int main(void) {  string s="good";f("good");  return 0;  } | output: good  pass string value. In this case f(s); is wrong.  compare with the above |
| void f(string \*p){ cout<< \*p; }  int main(void) {  string s="good";f(&s);  return 0;  } | output: good  pass string pointer  compared with the above |
| string func(string a){return "No." + a;}  string n = "001"; string user = func(n);  cout << user << endl; | pass value of a string, return value of new string. |
| string f(string s){  return s.substr(0,1) + s.substr(1,1) + s.substr(1,1) + s.substr(0,1);  }  int main(void) {  string \*s = new string("ab");  cout << f("ab") << endl;  cout << f("abba") << endl;  cout << f(s) << endl;  return 0;  } | output:abba  three approaches are equal |
| **void func(string &a){ a = "No." + a; }**  int main(void){  string n = "001"; func(n);  cout << n << endl;  return 0;  } | pass reference. No return is needed.  c++ style is recommended. |
| **string func(char \*p){**  string str = p; str = "No." + str; return str;  }  int main(void){  string n = "001";string user = func(&n[0]);  cout << user << endl;  return 0;  } | pass pointer, return string  not recommended |
| char \*func(char \*p){  string str = p; str = "No." + str; return &str[0];  }  int main(void){  string n = "001"; char \*p = &n[0];  p = func(p); cout << p << endl;  return 0;  } | pass pointer, return pointer  not recommended |
|  |  |
|  |  |

For int, float type, compare the approaches below

|  |  |
| --- | --- |
| void swap(float \*x, float \*y){  float z = \*x; x = y; y = z;  }  int main(void){  vector <float> t = {3.,2.,1.};  swap(&t[0],&t[2]);  cout << t[1];  return 0;  } | pass pointer |
| void swap(float \*x, float \*y){  float \*z = x; x = y; y = z;  }  int main(void){  vector <float> t = {3.,2.,1.};  float \*p = &t[0];  swap(p,p+2);  cout << t[0] << t[1] << t[2];  return 0;  } | pass pointer |
| void swap(float &x, float &y){  float z = x; x = y; y = z;  }  int main(void){  vector <float> t = {3.,2.,1.};  swap(t[0],t[2]);  cout << t[0] << t[1] << t[2];  return 0;  } | pass reference |
| int f(int &a){ return ++a;}  int main(void){  int t=3;  cout<< f(t) << endl;  return 0;  } | output:4  reference &a of variable t. a is alias name of t. |
| int \*f(int &a){ return &++a; }  int main(void){  int t=3;  cout<< \*f(t) << endl;  return 0;  } | output: 4;  return pointer of reference, then dereference the pointer in main() |
|  |  |
|  |  |
|  |  |

array to reference

Pass array to function in C is to pass pointer to function. It is not recommended to pass array to reference. The reference approach is clumsy compared with pointer approach.

|  |  |
| --- | --- |
| int f(int (&a)[4]){ return \*a; }  int main(void){  int t[]={1,2,3,4};  cout<< f(t) << endl;  return 0;  } | output:1  Note pass array by reference, the size should be explicitly declared |
| int f(int (&a)[4]){ return \*a++;} | wrong: a is reference, can't move or initialized as pointer |
| int f(int (&a)[]){ return \*a; } | wrong: size of array to reference should be declared. |
| int \*f(int &a){  //move pointer  **return &a+1;**  }  int main(void){  int t[]={1,2,3,4};  cout<< \*f(\*t) << endl;  return 0;  } | output: 2  pass reference of 1st value to function, and reference the reference as pointer before moving and returns |
| int f(int &a){  //move pointer and return value  return \*(&a+1);  }  int main(void){  int t[]={1,2,3,4};  cout<< f(\*t) << endl;  return 0;  } | The pattern is equal to the above. |
| int f1(int p){ ++p; return p++; }  **int f2(int &p){**  **++p; return p++; }**  int main(void){  int a=1, b, c;  b=f1(a); c=f2(b);  cout << a+b+c << endl;  return 0;  } | output: 8  pass value: b=f1(1)=2  pass pointer: c=f2(b=2)=3, b=4  a=1, b=4, c=3  define f2(int &p), and reference: f2(b)  so pass address of b into function, p and b share the same address, or p and b are the same. |
| // pass value: copy value, different address  char f1(char c){ return c++; }  // pass reference: same value, same address  char f2(char &c){ c--; return c++; }  int main(void){  char x='x'; // reference  cout<<f1(x); cout<<f2(x);  cout<<x<<endl;  return 0;  } | output:xwx |
| // pass pointer itself has its own address  // point a memory address,  //which may be allocated to a variable  char f3(char \*c){  (\*c)--;  // return \*c firstly  // then move pointer c rather than x;  return \*c++;  }  char f4(char \*c){  // return \*c firstly, then add value  return (\*c)++;  }  int main(void){  x='x';  cout<<f3(&x); cout<<f4(&x);  cout<<x<<endl;  return 0;  } | output:wwx |
| int f1(int \*a){ return \*a+1; }  int \*f2(int \*a){ return a+1; }  int \*f3(int &a){ return &a+1; }  int main(void){  int t[]={0,1,2,3};  cout<< f1(f3(\*f2(t)));  return 0;  } | output:3 |
| string fun(string &t, string s="", int r=2){  while(--r) s +=s;  t +=s;  return s;  }  int main(void){  string name ="x";  cout << fun(name, name) <<name;  return 0;  } | output: xxxxx  while loop execute one time:  s=s+s="xx"=>fun()="xx"  t=t+s="xxx" => name="xxx" |
|  |  |

## <functional>

This head file define some common functions, which could be used in vector-style operations, namely algorithm::for\_each() or algorithm::transform().

#include <functional>

**plus(x,y) +, minus(x,y) -, divides(x,y) /, multiplies(x,y) \*, modulus(x,y) %**

|  |  |
| --- | --- |
| vector<int> v={1,2,3,4,5};  vector<int> v2;  transform(v.begin(),v.end(), v.begin(), \  back\_inserter(v2), **plus<int>()**);  for(int &i: v2) cout << i; | v2=2,4,6,8,10 |
| vector<int> v={1,2,3,4,5};  vector<int> v2;  transform(v.begin(),v.end(), back\_inserter(v2), \  **bind2nd(plus<int>(), 10)**);  for(int &i: v2) cout << i; | v2=11,12,13,14,15 |
| vector<int> v={1,2,3,4,5};  vector<int> v2;  transform(v.begin(),v.end(), back\_inserter(v2),  **bind2nd(minus<int>(), 1)**);  for(int &i: v2) cout << i; | output: 01234 |
| vector<int> v={1,2,3,4,5};  vector<int> v2;  **transform(v.begin(),v.end(), back\_inserter(v2),**  **bind1st(minus<int>(), 1));**  for(int &i: v2) cout << i; | output: 0-1-2-3-4  compared with the above. bind1st: 1-el |
|  |  |
|  |  |
|  |  |
|  |  |

**equal\_to(x,y), not\_equal\_to(x,y), greater(x,y), greater\_equal(x,y), less(x,y), less\_equal()**

|  |  |
| --- | --- |
| vector<int> v={1,2,3,4,5};  vector<int> v2;  transform(v.begin(),v.end(),  back\_inserter(v2), bind2nd(equal\_to<int>(), 1));  for(int &i: v2) cout << i; | v2=10000 |
| vector<int> v={1,2,3,4,5};  vector<int> v1={1,3,3,1,10};  vector<int> v2;  transform(v.begin(),v.end(), v1.begin(),  back\_inserter(v2), **greater<int>()**);  for(int &i: v2) cout << i; | v2=00010 |
|  |  |
|  |  |

**logical\_and(x,y), logical\_not(x), logical\_or(x,y)**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

## other

### inline function

If the compiler performs function inlining, it replaces a call of that function with its body, avoiding the overhead of a function call (placing data on stack and retrieving the result), which may result in a larger executable as the code for the function has to be repeated multiple times. The result is similar to [function-like macros](https://en.cppreference.com/w/c/preprocessor/replace), except that identifiers and macros used in the function refer to the definitions visible at the point of definition, not at the point of call.

inline function should be quickly executable. inline function could work like MACRO but address more functions.

|  |  |
| --- | --- |
| #include <iostream>  #include <cmath>  #include <iomanip>  using namespace std;  inline int func(void)  {  return 2;  }  int main(void){  int res = func();  cout << res << endl;  return 0;  } |  |
| using namespace std;  **inline int**  **function (int parameter) {**  **return parameter \* 2;**  **}**  int main(void) {  int  var = 1;  var = function (var);  var = function (var);  var = function (var);  cout << var << endl;  return 0;  } | split them into line with "inline int" and function()  Or "int inline func()" |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### overloading

functions are denoted as the same name, but have different arguments. Overloading is kind of polymorphism of OOP

Here are overloading rules:

1. Prerequisite: Function name must be identical.

2.. There is exact match: Type and number of parameters match with those of arguments.

3. If a not exact match is found: Char, Unsigned char, and short are promoted to an int. Or Float is promoted to double

4. If still no match is found: C++ tries to find a match through the standard conversion.

5. compiling error: could not convert ... from ... to 'std::\_\_cxx11::...'

|  |  |
| --- | --- |
|  |  |
| **float func(int a){ return ++a;}**  int main(void) {  double i=20.5;cout << func(i);  return 0;  } | output: int:21  type\_casting:  pass arguments: convert double to int  return: convert int to float |
| int func(int a){return ++a; }  int main(void) {  long i=1234567890; cout << func(i);  return 0;  } | output: 1234567891 |
| int func(int a){ return ++a; }  int main(void) {  long i=12345678900;cout << func(i);  return 0;  } | output: -539222987 depending platform  long int is bigger than int |
|  |  |

### syntax errors

|  |  |
| --- | --- |
| **int func(int a){**  cout<<"int a"<<endl;  return ++a;  }  **int func(int &a){**  cout<<"reference a"<<endl;  return a++;  }  int main(void) {  int i=2;**func(i);**  return 0;  } | error: overloaded is ambiguous  calling func() with value or reference is unknown. |
| void f(string s){}  int main(void) {  int i=2; f(i);  return 0;  } | error: could not convert 'i' from 'int' to 'std::\_\_cxx11::string  {aka std::\_\_cxx11::basic\_string<char>}' |

# Input/Output

functions

|  |  |
| --- | --- |
| string line\_of\_types;  getline(cin, line\_of\_types);  cout << line\_of\_types << endl; | getline() |
|  |  |
|  |  |

## <iostream>

std:cout

The stream is stack type- "last one, first out". the order of printing is left to right, but the executed order is from right to left.

cout.setf(oct/dec/hex/scientific): configure format of ouput in octal, decimal, or heximal, or scientifica

|  |  |
| --- | --- |
| class A{  int \*val;  public: **A() {val = new int; \*val=0;}**  int get(){return ++(\*val);}  };  int main(){  **A a, b=a;**  cout << a.get() << b.get();  return 0;  } | output: 21  Note: objects a and b use same constructor. the attribute the pointer sharing this same value. |
| std::cout << 5; | writes integer 5 to standard output  if declare namespace std, std:: could be ignored.  cout<<5 |
| int bytes=255  cout<<bytes; | decimal |
| cout << "hexadecimal=" << hex << bytes; | std::hex |
| cout<< oct<<bytes; | std:oct |
| cout<<hex<<bytes | std::hex |
| #include <iostream> // std::cout, std::endl  #include <iomanip> // std::setbase  int main () {  std::cout << std::setbase(16);  std::cout << 110 << std::endl;  return 0;  } | default is decimal  std::cout << std::setbase(16); force hexadecimal |
| cout << 100<<',';  cout.setf(ios::hex);  cout << 100; | output:100,100  execution is good but not hex is printed. |
| cout << 100<<',';  **cout.setf(ios::hex, ios::basefield);**  cout << 100; | output:100,64 |
| cout << 100 << ',';  cout.setf(ios::hex, ios::basefield);  cout << 100 << ',';  cout.setf(ios::dec);  cout << 100; | output:100,64,100  mask oct/dec/hex operations |
| cout.setf(ios::hex, ios::basefield);  cout << 100 << ',';  cout.setf(ios::showbase);  cout << 100; | output: 64,0x64 |
| cout.setf(ios::hex, ios::basefield);  cout << 100 << ',';  cout.flags(ios::showbase);  cout << 100; | output:100,100  ios::flags() get/sets format.  showbase shows all base of numerics. |
| char Char = 'X', Minus = '-';  float Float = 2.5;  cout << Char << Minus << Float; | output: X-2.5 |
| char Char = 'X';  int Int = Char;  cout << Char << " " << (int)Char << " " << Int << " " << (char)Int; | output: X 88 88 X |
| int i=3,j=++i,k=++i;  cin>>i;  cout<<k+i<<j-i<<i\*i; | output: 10-125 |
| float x=2.5;  cout << scientific << x; | output: 2.500000e+000 |
| float x=2.5;  cout << scientific << "x"; | output: x  Note compare it with the above |
| int i=8;  **cout << i << hex << i+i << oct << i;** | output: 81010 |
| double b = 1e15;  double s = 1e-15;  cout << fixed << b+s; | output: 1000000000000000.000000  std.fixed fix float point notation |
| double b = 1e15; double s = 1e-15;  cout << scientific << b+s; | output: 1.000000e+015 |
| #include <iostream>  #include <iomanip> //setprecission  using namespace std;  int main(void){  double b = 1e15, s = 1e-15;  cout << fixed << setprecision(2) << b+s;  return 0;  } | output: 1000000000000000.00 |

std::cin

1. trim the input by white space/tab space.

2. if date type is wrong, nothing is assigned.

bool cin.eof()

bool cin.bad(): detect bad bit namely memory overflow

bool cin.fail(): except bad bit, detect fail bit namely end character

bool cin.good()

bool cin.clear()

|  |  |
| --- | --- |
| int i;  std::cin >> i; | reads integer from the standard input  enter "3" or "3.4", i=3  enter "abc" or "abc3" or "3abc", i=0,  enter "1 2 3", i=1 |
| string s;  std::cin >> s; | enter 456, s="456"  enter "one two three", s="one" |
| vector<int> v(2);  vector<int>::iterator iter = v.begin();  for(iter; iter != v.end(); iter++){  std::cin >> \*iter;  cout << \*iter;  } | generate a vector  "1<enter>2<enter>" -> v=12  "1<whitespace>2<whitespace>3<enter>" -> v=12 |
| list<int> I;  **for(;!cin.fail();){**  int i; cin>>i;  I.push\_back(i);  }  for\_each(I.begin(), I.end(),\  [](const int&i){ cout<<i;}); | enter"1 2 3 end<enter>", then I=1230  "end" indicate end of stream.  Note: if cin.bad() is used, runs forever till runtime error caused by "St9bad\_alloc" |
|  |  |

## text file: <fstream>

<fstream>

|  |  |
| --- | --- |
| #include <iostream> // cerr, cout, endl, ios::in  #include <fstream> //ifstream  #include <cstdlib>  // exit  using namespace std;  int main(void) {  **ifstream infile("F:\\cpp\\read.txt", ios::in);**      if( !infile ){          cerr << "file can't be opened" << endl;          exit(1);      }      string line;      while(infile >> line) {          cout << line << endl;      }      return 0;  } | print a text file and print them |
| #include <iostream> // cerr, cout, endl, ios::out  #include <fstream> //ofstream  #include <cstdlib>  // exit  #include <vector>  #include <string>  using namespace std;  int main(void) {      ofstream outfile("F:\\cpp\\write.txt", ios::out);      if(!outfile){          cerr << "file can't be opened" << endl;          exit(1);      }      vector<string> lines {"abc", "123"};      for(string line: lines) {          outfile << line << endl;          // cout<< line;      }      return 1;  } | write strings in vector into a text file |
| //split lines into columns  void split(vector<string>& items, string& str, string sep=","){      string::iterator iter;      iter = search(str.begin(), str.end(), \          sep.begin(), sep.end());      items.push\_back(string(str.begin(), iter));      if(iter != str.end()){          iter += sep.size();          string sub(iter, str.end());          split(items, sub, sep);      }  }  int main(){      string file\_path = "F:\\cpp\\tab.txt";      string sep = "\t";      //write to text file separated by tab space      ofstream outfile; //file object      outfile.open(file\_path, ios::out);      if(!outfile){          cerr << "file can't be opened!" << endl;          exit(1);      }      map<string, int> st {{"John",34},{"Mary",12},{"Cary",77}};      outfile << "Name" << sep << "Age" << endl;      for(pair<string, int> el: st){          outfile << el.first << sep << el.second << endl;      }      outfile.close();      // read from text file      ifstream infile;      infile.open(file\_path, ios::in);      string line;      while(getline(infile, line)){          // cout << ">>" <<  line << endl;          vector<string> items;          split(items, line, sep);          cout << items[0] << "\t" << items[1] <<endl;      }      infile.close();      return 1;  } | write/read text which stores table data separated by table space |
| string ss="he";      ostringstream oss;    oss.str(ss);     oss.put(65);      ss = oss.str();    cout << ss;  oss.str("") // clear memory | output:Ae  oss.put() insert character from the front |
|  |  |
|  |  |
|  |  |
|  |  |

## <sstream>

#include <sstream>

ostringstream os;

istringstream is(str); read substring from string separated by whitespace or tab space

stringstream ss(str); bind stream with string

|  |  |
| --- | --- |
| string s = "abc 123";  **istringstream is(s);**  do{      string sub;  **is >> sub;**      cout << sub << endl;  **}while(!is.eof());** | output:  abc  123 |
| string s = "abc 123";  istringstream is(s);  string sub;  **while(is >> sub)**      cout << sub << endl; | output:  abc  123  same as the above |
| string s = "abc 123 def";  **stringstream ss(s);**  vector<string> items;  do{      string sub;      ss >> sub;      items.push\_back(sub);  }while(!ss.eof());  for(string& i: items) cout << i << "\t"; | abc 123 def  "stringstream" could split string |
| string s = "abc 123 def";  stringstream ss(s);  vector<string> items;  string sub;  while(ss >> sub)      items.push\_back(sub);  for(string& i: items) cout << i << "\t"; | abc 123 def  same as the above |
| template<class T>  string toString(const T& t){      ostringstream oss;      oss << t;      return oss.str();  }  int main(){      string s = toString(12.3f);      cout << s;      return 1;  } | convert integer, char, bool, float or double to string. |
|  |  |
|  |  |
|  |  |
|  |  |

# Basic data type

## digits and character

c++ don't support "unsigned double float".

### digits

|  |  |
| --- | --- |
| 0x10 | output: 16  correct: 0X16 |
| x10 | invalid |
| int i=1,j=2;  if(i>j&&j>i) i++;  if(i>j||j>1) j++;  if(i|j>1) i++;  if(i&j) j++;  cout << i\*j << endl; | output:8 |
| int i=2;  float a=3;  bool f1=a/i>i/a;  float f2=a/i>i/a;  int f3=a/i>i/a;  cout << f1 << f2 << f3 << endl; | output:111 |
| #include <iostream>  using namespace std;  int main(void) {  cout << "This computing environment uses:" << endl;  cout << sizeof(char) << " bytes for chars" << endl;  cout << sizeof(short int) << " bytes for shorts" << endl;  cout << sizeof(int) << " bytes for ints" << endl;  cout << sizeof(long int) << " bytes for longs" << endl;  cout << sizeof(float) << " bytes for floats" << endl;  cout << sizeof(double) << " bytes for doubles" << endl;  cout << sizeof(bool) << " byte for bools" << endl;  cout << sizeof(int \* ) << " bytes for pointers" << endl;  return 0;  } | 1 bytes for chars  2 bytes for shorts  4 bytes for ints  8 bytes for longs  4 bytes for floats  8 bytes for doubles  1 byte for bools  **8 bytes for pointers ??** |
| int a, b;  a=b=1; | it is ok to combining assignment |
| int a=b=1; | compiling error |
| float a(3); | It is C++ style. Being equal to "float a= 3;" in C |
| int i=3;  **i=(i--);**  cout << i; | output:3  The execution order: decrement i=3 to i=2 firstly, then return previous value i=3 to i, so i=3. |
| const a =3.14;  const b= a\*a;  cout << b; |  |

### type conversion

short int (2 bytes, (2^4)^2=2^8=256) , long int(8 bytes, (2^4)^8=42^32=,297,967,296), int(4bytes, (2^4)^4)=2^16=65536)

float, double float

|  |  |
| --- | --- |
| int a = 1; float i = 2;   cout << a/i; | output: 0.5  **Note: implicit convert to float** |
| int a=1; float i=2;  cout<< (float) a/i; | the same as the above. That is explicit conversion. |
| short s=1;int i=2;  float f=4.4;double d=6.6;  cout<< s/float(i) + int(f)/i + long(d)/s; | output:8.5  s/float(i) = 1/2.0=0.5  int(f)/i =4/2=2  long(d)/s=6/1=6 |
| short s=1;int i=2;  float f=4.4;double d=6.6;  cout<< s/i + f/i + d/s; | output:8.8  0+2.2+6.6 |
| short s=1;int i=2;  float f=4.4;double d=6.6;  int x = s/i + f/i + d/s;  cout<< x; | output:8  0+2+6 |
| float f = 3.21; double d = 1.23;  int k = int(f) + (int) d; cout << k << endl; | output: 4 |
| float f = 1234.5678;  double d = f;  if (d == f)cout << "equal" << endl;  elsecout << "not equal" << endl; | output: equal  Note: double is 8 bytes, and float is 4 bytes. double has wider range than float. |
| **double d = 123456.789012; float f = d;**  **if (d == f) cout << "equal" << endl;**  **else cout << "not equal" << endl;** | **output: not equal** |
| int i=10;  float f=3.5;  cout << float(i)/int(f); | **output: 3.33333** |
| int \*i = new int;  \*i=1.0/2\*2/1\*2/4\*4;  cout << \*i; | **output: 2**  1.0/2\*2/1\*2/4\*4 = 0.5\*2/1\*2/4\*4 =1.0/1\*2/4\*4=1.0\*2/4\*4=2.0/4\*4=0.5\*4=2.0->2 |
| int a=2; float b=1;  cout << (float) a % b; | compiling error  Note: % is not working for float type |
| float func(double x){ return x/(.5\*x); }  int main(void){  int i=2;  int v = func(i);  cout << v;  return 0;  } | though argument is double type, pass int is ok. |

explicit conversion

|  |  |
| --- | --- |
| double scores = 95.5;  int n = static\_cast<int> (scores); | n=95  static\_cast convert data type |
| float a=1.2; a = (int) a;  cout<<a; | output: 1 |
| double i = -5.55; cout << (int) i; | output:-5 |
|  |  |
|  |  |

## enum

|  |  |
| --- | --- |
| enum e {a, b,c,d};  cout << a << e(2) << e(a); | output: 020  #define a 0 |
| enum e {a, b,c,d};  cout << e(b+1); | output:2  #define b 1 |
| enum e {a, b,c,d};  e f = e(a+c);  cout <<f <<endl; | output:2  e f = e(0+2) =e(2)=2 |
| enum e {a, b,c,d=-1};  cout <<e(b) <<endl; | output:2  a=0,b=1,c=2,d=-1 |
| enum e {a=1, b,c,d};  **e f = c;**  cout << f <<endl; | output:3 |
| enum e {a=1, b,c,d};  **e f = e(a+1);**  cout << f <<endl; | output:2 |
| enum e {a=1, b,c,d};  e& operator++(e &x){x=a; return x; }  int main(void) {  e f = c; cout << int(++f) <<endl;  return 0;  } | output:1  f=3 -> ++f -> x=1  Note: x=e(a+1) is right. x=++a is wrong. a is enum |
| enum answer {yes, no, na};  enum answer a[3];  a[0] = no; a[1] = yes; a[2] =na;  for(int i = 0; i < 3; i++)  cout << a[i]; | output:102  instantiate enum type as a |
| enum state{ok, error, warning};  enum state s1, s2, s3;  s1=ok; s2=warning; s3=error;  cout << s1 << s2 << s3; | output:021 |
|  |  |
|  |  |
|  |  |
|  |  |

### syntax errors

|  |  |
| --- | --- |
| enum e {a=1, b,c,d};  e f = a+1;  cout <<f <<endl; | compiling error: invalid conversion from int to main()::e  correct: e f = e(a+1)  Note: use overloading operator+ |
| class Int{  public: T v;  Int(T a){ v = a; }  Int& operator++(){ v += 2; return \*this;}  };  ostream& operator<<(ostream &o, Int &a){  ++a; return o << a.v;  }  int main(void) {  Int i = B;cout << i;  return 0;  } | compiling error  v+=2 is wrong. v is enum type  correct v= T(v+2) |
| enum e {a=1, b,c,d};  e d = e(a+1);  cout << d <<endl; | compiling error: previous declaration 'main()::e d'  can't update d because d is declared. |

# exceptions

logic error:

run time error:

|  |
| --- |
| #include <exception>  try{  …..  }  catch (string &problem) {  cout << problem << endl;  }  catch(…){ } |
| int problem -> cout << problem  string &problem -> cout << problem  class &exc -> cout << exc.msg |
|  |
|  |
|  |

## throw exceptions

### throw various data type:

#include <exception>

throw exception() -> catch(exception &e) -> e.what()

|  |  |
| --- | --- |
| int main(void) {  try { throw exception(); }  catch(exception &e){ cout << e.what(); }  return 0;  } | output: std::exception  Note: "throw exception("xxx");" is wrong |
| void f(void){**throw exception();** }  int main(void) {  int a=4;  **try { f(); }**  **catch (exception &e){ a--;}**  **catch(...){ a++;}**  cout << a << endl;  return 0;  } | output: 3  "throw exception();" |

throw instance of a class

throw cls() -> catch(cls &c) -> cout<< c.msg

|  |  |
| --- | --- |
| class Class {  public:string msg;  Class(string txt): msg(txt) { }  };  **void function(int i) { throw Class("object"); }**  int main(void) {  try {function(1);}  catch(Class &exc) {cout << exc.msg;}  return 0;  } | output: object  "throw Class("object");" |
| class A{  public: static int a;  A(){ a++; }  };  int A::a=0;  **void f(void){ A a; throw A(); }**  int main(void) {  A a;  **try { f(); }**  **catch(...){ }**  cout << A::a << endl;  return 0;  } | output: 3  catch(…) other exception  "throw A();" |

throw pointers

E e; -> throw &e -> catch(E\*)

|  |  |
| --- | --- |
| class E{  public: string msg;  E() {msg="bad";}  };  void f(int i){  E e;  switch(i){  case 0: throw e;  case 1: throw &e;  }  cout <<0;  };  int main(void) {  try { f(0); }  catch(E e){ cout << "msg=" << e.msg; }  return 0;  } | output: msg=bad |
| class E{  public:  string msg;  E() {msg="bad";}  };  void f(int i){  E e;  switch(i){  case 0: throw e;  case 1: throw &e;  }  cout <<0;  };  int main(void) {  try { f(1); }  catch(E\* e){ cout << "msg=" << e->msg; }  return 0;  } |  |
| class E{};  void f(int i){  E e;  switch(i){  case 0: throw e;  case 1: throw &e;  }  cout <<0;  };  int main(void) {  try { f(1); }  catch(void \*){ cout << 2; }  catch(E\*){ cout <<1; }  return 0;  } | output:2  catch(void \*) will override catch(E\*) |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

throw digits

throw 1 -> catch(int &i)

|  |  |
| --- | --- |
| int i=1;  **void f(void){ throw 1;i++; }**  void g(void){  i++;  **try{ f(); }**  **catch( int &i){ throw ++i; }**  }  int main(void) {  try { g(); i++; }  catch(...){ i++; }  cout << i << endl;  return 0;  } | output:3  throw <int>; |
| class X{  public:  X() { cout << 0; };  ~X() { cout << 2; };  };  int main(void) {  try {  X \*x = new X();  **throw true;**  delete x;  }  **catch(bool i){ cout << i; }**  return 0;  } | output: 01  1. throw bool type  2. destructor will not run |

throw string("0") -> catch(string &s)

|  |  |
| --- | --- |
| class X{  public:  X() { cout << 1; };  ~X() { cout << 2; };  };  void exec(){  **{ X x;}**  **throw string("0");**  }  int main(void) {  **try { exec();}**  **catch(string &s){ cout << s; }**  return 0;  } | output: 120  1. throw string("")  2. once throw exception, destructor will be run. |
|  |  |

### throw in function:

"double func (char param) throw (int);" This function func() return double float, only throw int type exception. try cause will not catch it if throw other type.

|  |  |
| --- | --- |
| class X{  public:  **X() throw(int) { cout << 1; };**  **~X() throw(int) { cout << 2; };**  void exec() { throw 0; }  };  void exec(X &x){ x.exec(); }  int main(void) {  **X x;**  **try { exec(x);}**  **catch(int &i){ cout << 0; }**  return 0;  } | output:102  1. define throw in constructor  2. destructor will be run even if exception is thrown when define class using "X x;" |
| class X{  public:  **X() throw(int) { cout << 1; };**  **~X() throw(int) { cout << 2; };**  void exec() { throw 0; }  };  void exec(X\* x){ x->exec(); }  int main(void) {  **X \*x = new X();**  **try { exec(x);}**  catch(int &i){ cout << 0; }  return 0;  } | output: 10  compared with the above  1. use class pointer, destructor will not run if exception is thrown |
|  |  |

## hierarchy of exceptions

|  |  |
| --- | --- |
| class Class {  public:  string msg;  Class(string txt) : msg(txt) {}  };  void function(int i) throw (string,Class) {  switch(i) {  case 0 : throw string("string");  case 1 : throw Class("object");  default: cout << "OK" << endl;  }  }  void level(int i) throw(Class) {  try {  function(i);  } catch(string &exc) {  cout << "String [" << exc << "] caught in level()" << endl;  }  }  int main(void) {  for(int i = 0; i < 2; i++) {  cout << "-------" << endl;  try {  level(i);  } catch(Class &exc) {  cout << "Object [" << exc.msg << "] caught in main()" << endl;  }  }  return 0;  } | output:  -------  String [string] caught in level()  -------  Object [object] caught in main() |
| int c=3;  try{  switch(c){  case 1: throw 20;  case 2: throw 5.2f;  case 3: throw 'a';  }  }  catch(int e){cout << "int exception";}  catch(float e){cout << "float exception";}  catch(...){cout << "other exception";} | output: other exception |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## customary exceptions

|  |  |
| --- | --- |
| class X: public logic\_error{  public: X(): logic\_error("0") {};  };  void exec(X &x) throw(X) { throw x; }  int main(void) {  X x;  try { exec(x); }  catch(X &i){ cout << i.what(); }  return 0;  } | output:0 |
| class X: public logic\_error{  public:  X(string s): logic\_error(s) {};  ~X() { cout << 2; }  };  void exec() throw(logic\_error) {  X x("0"); throw x; cout << 3;  }  int main(void) {  // X x("4");  try { exec(); }  catch(X &i){ cout << i.what(); }  return 0;  } | output: 202  ????? should be 02 ?? why destructor run twice |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## syntax errors

|  |  |
| --- | --- |
| int main(void) {  int a =8, b=0, c=6;  try{c=a/b;}  cout << c << endl;  return 0;  } | miss catch block   expected 'catch' before 'cout' |
| **void f(void){ throw exception("?");}**  int main(void) {  int a=4;  try { f(); }  catch(...){ a++; }  catch (exception &e){ a--; }  cout << a << endl;  return 0;  } | two errors:  1 should be "throw exception();" can't pass "?".  2.'...' handler must be the last handler for its try block |
| class X: public runtime\_error {  public: X(): domain\_error("0") {};  };  void z() throw(X){  X x;  throw x;  cout << 2;  }  int main(void) {  X x;  try { z();}  catch(X &i){ cout << 1;}  catch(domain\_error &i){ cout<<0;}  return 0;  } | wrong inheritance of exceptions  std:stdexcept -> logic\_error -> domain\_error |
| class X: public logic\_error{  public:  X(): logic\_error("0") {};  };  void exec() throw(X) {  throw new logic\_error("0");  }  int main(void) {  X x;  try { exec(); }  catch(X &i){ cout << i.what(); }  return 0;  } | terminate called after throwing an instance of 'std::logic\_error\*'  ??????????????? |
| int main(void) {  try { throw 3.14; }  catch(double x){ x\*=2; }  cout <<x;  return 0;  } | out of scope. x is only defined in catch(){}. |
| class X{  public:  X(void) throw(int) {cout<<1;}  ~X(void) throw(int) {cout<<2;}  **void exec() { throw string("0");** }  };  void exec(X &x) {  x.exec();  }  int main(void) {  X x;  try { exec(x); }  catch(int &i){ cout << i; }  return 1;  } | correct: catch(string &i) |
|  |  |
| int main(void) {  try { throw 2./4; }  catch(int i){ cout << i; }  return 0;  } | terminate called after throwing an instance of 'double'  correct: catch(double i) |
|  |  |
|  |  |

# memory

## dynamic memory: <new>

new expression: 1. launch operator new() and allocate memory. Throw std::bad\_alloc. 2.Initial object and return address. 3. operator delete() release memory.

|  |  |
| --- | --- |
| **<data\_type>\* p\_var = new <data\_type>(<initial expression list>);** | **allocate memory for this data type** |
| int \*p = **new int;**  \*p=2; cout << \*p; delete p; | =2  allocate memory to a integer, assign address to the pointer p. |
| int \*p = **new int(2);**  cout << \*p; delete p; | =2 |
| float \*f = **new float(10.36);**  cout << \*f; delete f; | =10.36 |
| char \*f = **new char('a');**  cout << \*f; delete f; | =a |
| double \*p = **new double[2] {1.2, 2.4};**  cout << \*p; delete [] p; | =1.2  allocate memory to double array with size of 2, assign address to the pointer p. |
| int n=10;  int \*p = new int[n][10]; | allocate memory to 2-D integer array of 10x10. |
| char \*f = **new char[4] {'a','b','c','\0'};**  cout << \*f; delete f; | =a |
| string \*f = **new string("abc");**  cout << \*f; delete f; | =abc |
| **vector<int> \*f = new vector<int> {1,2,3};**  **cout << f->at(1); delete f;** | =2 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| int \*p = new int+1; | (new int) +1. pointer move to the next address. |

placement new

|  |  |
| --- | --- |
| p\_var = new (placements) <data\_type> | pass placements as parameters to the allocation function as additional arguments. The allocation function is known as "placement new" |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## memory leak

## The new and delete keyword

the new keyword create a new block address for variable and create object. But the variable should be initialized in the next.

The new will return a pointer. Here is the usage:

new <data-type>

In C, they could be malloc() and free() only ask for the memory. the variable should be created in advance. It is ok to use them in c++ but not suggested.

|  |  |
| --- | --- |
|  | output |
| int \*ptr = new int[10];  cout << sizeof(ptr) << endl;  cout << ptr << endl; | 4  0xf27f48 |
| float \*ptr = new float[10];  cout << sizeof(ptr) << endl;  cout << ptr << endl; |  |
|  |  |
|  |  |
|  |  |
|  |  |

the delete word

|  |  |
| --- | --- |
| int \*ptr = new int;  cout << sizeof(ptr) << endl;  cout << ptr << endl;  delete ptr; |  |
| char \*ptr = new char[10];  cout << sizeof(ptr) << endl;  cout << ptr << endl;  **delete [] ptr;** | "delete [] ptr;" will delete the array of pointers. That is entire array  "delete ptr;" will only delete the first cell.  "delete ptr[1]" will delete the 2nd object allocations |
| // Creating int pointer  int\* ptr1 = new int;    // Initializing pointer with value 20  int\* ptr2 = new int(20);    cout << "Value of ptr1 = " << \*ptr1 << "\n";  cout << "Value of ptr2 = " << \*ptr2 << "\n";    delete ptr1; // Destroying ptr1  delete ptr2; // Detroying ptr2 | delete variable no matter whether that is initialized or not. |
| void \*ptr;  delete ptr; | delete void pointer |
| int \*ptr = (int \*)malloc(10\*sizeof(int))  delete [] ptr; | delete array allocated by malloc() |
| int x;  delete x; | compiling error.  can't delete nonpointer |
| int i=3;  int \*ptr=&i;  cout << i << endl;  delete ptr;  cout << \*ptr << endl; | **though no compiling error, the delete will not work. delete only delete dynamically allocated memory, but the variable is static variable.** |