面向对象程序设计

类模板与 STL

2020 年 春

耿楠

计算机科学系
信息工程学院

西北农林科技大学
NORTHWEST A&F UNIVERSITY
中国 · 杨凌

- **函数模板** (function templates or generic functions)

```
template <class 类型名1, class 类型名2, ...>
返回类型 函数名 (形参表)
{
  ...
}
```

- 利用函数模板进行求和运算

```cpp
//例 07-01; ex07-01.cpp
//求不同类型的两个数之和，演示 c++ 的简单函数模版的定义与使用
#include <iostream>

using namespace std;

template <class T1, class T2>
T1 add(T1 x, T2 y)
{
   return x + y;
}

int main()
{
   cout << add(9, 'A') << endl;           //隐式实例化
   cout << add<int, char>(9, 'A') << endl; //显式实例化
}
```

- 类模板 (class templates or generic classes) 定义

```
        或typename ···,   或typename ···
template <class 参数化类型1, class 参数化类型2, ···,
          普通类型1, 普通类型2, ···>
class 类名
{
  成员名;
}
```

- 关于**class**和**typename**，请参阅：

http://dev.yesky.com/13/2221013.shtml

http://blogs.msdn.com/b/slippman/archive/2004/08/11/212768.aspx

- 安全数组类模板的定义

```
//例 07-02; ex07-02.cpp
//模板类的写法，演示 c++ 的简单模板类的写法
template <class T, int size>
class CSafeArray
{
   T a[size];
public:
   CSafeArray()
   {
      for(int i = 0; i < size; i++) a[i] = i;
   }
   T &operator[](int i)
   {
      if(i < 0 || i > size - 1)
      {
         cout << "Index value of " << i << " is out-of-bounds.\n";
         exit(1);
      }
      return a[i];
   }
};
```

参数化类型

- 安全数组类模板的定义

```
//例 07-02; ex07-02.cpp
//模板类的写法，演示 c++ 的简单模板类的写法
template <class T,int size>
class CSafeArray
{
   T a[size];
public:
   CSafeArray()
   {
      for(int i = 0; i < size; i++) a[i] = i;
   }
   T &operator[](int i)
   {
      if(i < 0 || i > size - 1)
      {
         cout << "Index value of " << i << " is out-of-bounds.\n";
         exit(1);
      }
      return a[i];
   }
};
```

普通类型

- 类模板中的成员函数 → 函数模板

```
//例 07-02; ex07-02.cpp
//模板类的写法，演示 c++ 的简单模板类的写法
template <class T, int size>
class CSafeArray
{
   T a[size];
public:
   CSafeArray()
   {
      for(int i = 0; i < size; i++) a[i] = i;
   }
   T &operator[](int i)
   {
      if(i < 0 || i > size - 1)
      {
         cout << "Index value of " << i << " is out-of-bounds.\n";
         exit(1);
      }
      return a[i];
   }
};
```

函数模板

- 类外定义类模板成员函数

```
//例 07-03; ex07-03.cpp
//在模板类的基础上写符号重载函数，演示 c++ 的符号重载函数的书写方法
template <class T, int size>
class CSafeArray
{
   T a[size];
public:
   CSafeArray()
   {
      for(int i = 0; i < size; i++)
         a[i] = i;
   }
   T &operator[](int i);
};
```

模板参数表

```
//例 07-04; ex07-04.cpp
//符号重载，演示 c++ 的符号简单重载方法
template <class T, int size>
T &CSafeArray<T, size>::operator[](int i)
{
   if(i < 0 || i > size - 1)
   {
      cout << "Index value of ";
      cout << i << " is out-of-bounds.\n";
      exit(1);
   }
   return a[i];
}
```

类模板的实例化

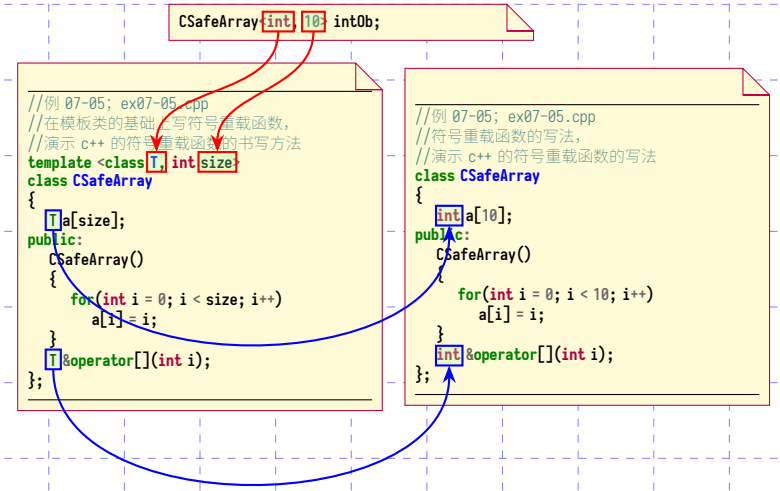
- 类模板 ⟶ 具体类 ⟶ 对象
- 语法

```
类模板名<基本数据类型名或构造数据类型名或常数表达式> 对象1，对象2，··· ，对象 n;
```

- 示例：

```
CSafeArray <int, 10>  intOb;
CSafeArray <double, 10>  doubleOb;
```

- 类模板的实例化和函数的实参与形参的结合类似



- 类模板的实例化和函数的实参与形参的结合类似

```
//例 07-06；ex07-06.cpp
//模版类的使用，演示 c++ 的简单模板类的使用
int main()
{
   const int SIZE = 10;
   CSafeArray<int, SIZE> intOb;
   CSafeArray<double, SIZE> doubleOb;

   cout << "Integer array: ";
   for(int i = 0; i < SIZE; i++) intOb[i] = i;
   for(int i = 0; i < SIZE; i++)
      cout << intOb[i] << " ";
   cout << '\n';

   cout << "Double array: ";
   for(int i = 0; i < SIZE; i++) doubleOb[i] = (double) i / 3;
   for(int i = 0; i < SIZE; i++)
      cout << doubleOb[i] << " ";
   cout << '\n';

   intOb[SIZE + 1] = 100;

   return 0;
}
```

整型安全数组

浮点型安全数组

- 对模板参数表中的类型赋默认的数据类型或数值（默认类属形参值）

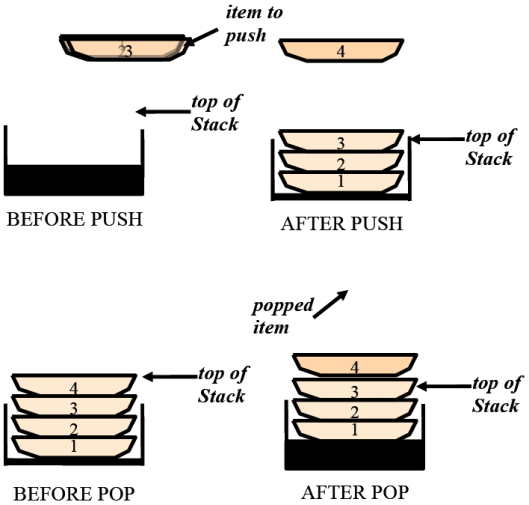
```
//例 07-07: ex07-07.cpp
//在模板类的基础上写符号重载函数，演示 c++ 的符号重载函数的书写方法
template <class T = int, int size = 10>
class CSafeArray
{
   T a[size];
public:
   CSafeArray()
   {
      int i;
      for(i = 0; i < size; i++)
         a[i] = i;
   }
   T &operator[](int i);
};
```

模板参数默认值

```
//例 07-07: ex07-07.cpp
//定义模板类的对象，演示 c++ 定义模板类对象的方法
int main()
{
   CSafeArray <>intOb;
   CSafeArray <double> doubleOb1;
   CSafeArray<double, 100> doubleOb2;
}
```

- 顺序堆栈类 (FILO)
  - 数据成员：栈顶索引 (top)，数据存储空间 (s)
  - 成员函数：(Push) 和 (Pop)

- 顺序堆栈类的基本操作 (FILO)



类模板的定义

```
//例 07-08; ex07-08.cpp
//定义一个栈类
const int SIZE = 10;

template <class ST>
class stack
{
   ST s[SIZE]; // holds the stack
   int top; // index of top-of-stack
public:
   stack()
   {
      top = 0;   // initialize stack
   }

   void Push(ST ob); // push object on stack
   ST Pop(); // pop object from stack
};
```

类模板的函数模板

```
//例 07-08; ex07-08.cpp
//定义一个入栈函数
// Push an object.
template <class ST>
void stack<ST>::Push(ST ob)
{
   if(top == SIZE)
   {
      cout << "Stack is full.\n";
      return;
   }
   s[top] = ob;
   top++;
}
```

```
//例 07-08; ex07-08.cpp
//定义一个出栈函数
// Pop an object.
template <class ST>
ST stack<ST>::Pop()
{
   if(top == 0)
   {
      cout << "Stack is empty.\n";
      return 0;
   }
   top--;
   return s[top];
}
```

类模板的使用

```
//例 07-08; ex07-08.cpp
//使用定义的栈类模板来完成不同类型栈的入栈出栈操作
int main()
{
   int i;

   stack<char> cs;
   cs.Push('a');
   cs.Push('b');
   cs.Push('c');
   for(i = 0; i < 3; i++)
      cout << "Pop cs: " << cs.Pop() << "\n";

   stack<double> ds;
   ds.Push(1.1);
   ds.Push(2.2);
   ds.Push(3.3);
   for(i = 0; i < 3; i++)
      cout << "Pop ds: " << ds.Pop() << "\n";

   return 0;
}
```

字符栈

浮点数栈

- 类模板成员函数声明与定义 (实现) 需处于同一文件。
- 分离编译模式：**export**关键字
  - VC 与 GCC 编译器不支持
- 其它解决方案

`http://www.codeproject.com/KB/cpp/Template_implementaion.aspx`

- 优势
  - 通过数据类型参数化实现代码重用（设计一个类模板，可用于多种数据类型场合）
  - STL 的基础
- 缺点
  - 类层次上再次抽象，对初学者来说可读性较差，不易使用

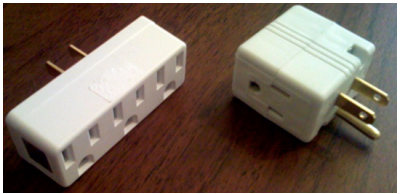
- 使用函数模板和类模板实现（类型参数化）
- 提供标准的数据结构和算法
- 可提高程序开发效率

- 对初学者来说，语法晦涩难懂
- STL 函数较多，不易记忆
- 建议：耐心学习参考例程
- 多看多练：

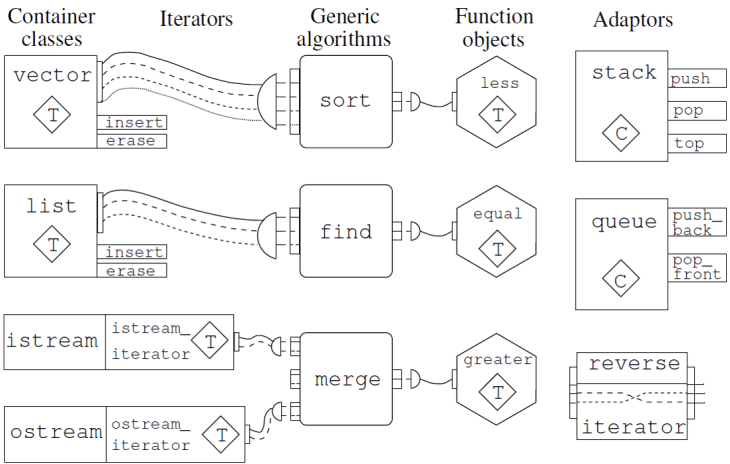
http://msdn.microsoft.com/en-us/library/c191tb28.aspx

http://www.cplusplus.com/reference/stl/

- 组成（六部分）
  - 容器 (Containers)
  - 迭代器 (Iterators)
  - 算法 (Algorithms)
  - 适配器 (Adapters)
  - 函数对象 (Function Objects)
  - 分配器 (Allocators)

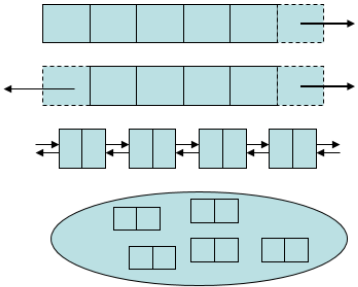


- 组成（六部分）

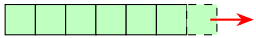

Container classes
Iterators
Generic algorithms
Function objects
Adaptors
vector
T
insert
erase
sort
less
T
stack
C
push
pop
top
list
T
insert
erase
find
equal
T
queue
C
push_back
pop_front
istream
istream_iterator
T
ostream
ostream_iterator
T
merge
greater
T
reverse
iterator

- 一个通用的数据结构
  - 可以处理不同数据类型
  - 包含基本的数据结构，如链表、堆栈、队列等

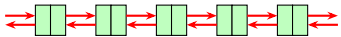
- 分类
  - 顺序容器 (sequence)
  - 关联容器 (sorted associative)
  - 容器适配器 (adaptors)
  - 特殊容器 (special)



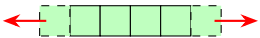
- 特点：添加或插入位置与元素值无关 (无自动排序)
  - 向量 (动态数组 vector)



  - 列表 (双向链表 list)



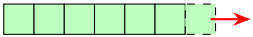
  - 双端队列 (deque: double-ended queue)



- 注意：不同容器的插入、删除和存取特性不同，需根据任务选择合适容器
- 评价指标之一：时间复杂度

Typical Values of Complexity		
Type	**Notation**	**Meaning**
Constant	$O(1)$	The runtime is independent of the number of elements.
Logarithmic	$O(log(n))$	The runtime grows logarithmically with respect to the number of elements.
Linear	$O(n)$	The runtime grows linearly (with the same factor) as the number of elements grows.
n-log-n	$O(n\,log(n))$	The runtime grows as a product of linear and logarithmic complexity.
Quadratic	$O(n^2)$	The runtime grows quadratically with respect to the number of elements.

- 特点



  - 在内存中占有一块连续的空间 (动态数组)
  - 可自动扩充，且提供越界检查
  - 适合在向量末尾插入或删除数据
  - 可用 [] 运算符直接存取数据 (随机访问 random access)

- 例 1

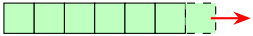
```cpp
//例 07-09; ex07-09.cpp
//容器 vector 的使用，演示 c++ 中容器 vector 的简单使用
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector<int> v;

   for (int i = 0; i < 6; ++i)
      v.push_back(i);

   for (int i = 0; i < 3; ++i)
      v.pop_back();

   for (int i = 0; i < (int)v.size(); ++i)
      cout <<v[i]<< ' ';
   cout << endl;

   return 0;
}
```

- 例 2

```cpp
//例 07-09; ex07-09.cpp
//容器 vector 的使用，演示 c++ 中容器 vector 的简单使用
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector<int> v(6,1);

   for (int i=0; i<v.size( ); ++i)
      cout << v[i] << ' ';
   cout << endl;

   for (int i=0; i<v.size( ); ++i)
      v[i]=i;

   for (int i=0; i<v.size( ); ++i)
      cout << v[i] << ' ';
   cout << endl;
}
```

容量为 6，用 1 初始化

- 复杂度

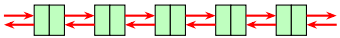


  - 随机存取：$O(1)$
  - 向量末尾插入或删除：$O(1)$

- 使用

  - `#include <vector>`
  - 适于快速存取数据但非尾部频繁插入删除的场合

- 特点



  - 双向链表（前驱和后继）
  - 可在任意位置插入或删除数据
  - 不能用 [] 运算符直接存取数据 (不支持随机访问 random access)

▶ 例 3

```
//例 07-10; ex07-10.cpp
//容器 list 的插入函数、迭代器的使用，演示 c++ 中容器 list 的简单使用
#include <iostream>
#include <list>
#include <vector>
using namespace std;
int main (){
   list<int> mylist;

   list<int>::iterator it;

   for (int i = 1; i <= 5; i++) mylist.push_back(i);

   it = mylist.begin();
   ++it;

   mylist.insert (it, 10);
   mylist.insert (it, 2, 20);

   --it;

   vector<int> myvector (2, 30);
   mylist.insert (it, myvector.begin(), myvector.end());

   for (it = mylist.begin(); it != mylist.end(); it++)
      cout << *it << " ";
   cout << endl;

   return 0;
}
```

迭代器：类似于指针

- 例 4

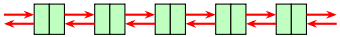
```
//例 07-11; ex07-11.cpp
//容器 list 中的排序、合并等函数的使用，演示 c++ 中容器 list 的高级函数使用
#include <iostream>
#include <list>
using namespace std;
bool mycomparison (double first, double second){
   return ( int(first) < int(second) );
}
int main (){
   list<double> first, second;

   first.push_back (3.1);
   first.push_back (2.2);
   first.push_back (2.9);

   second.push_back (3.7);
   second.push_back (7.1);
   second.push_back (1.4);

   first.sort();
   second.sort();

   first.merge(second);

   second.push_back (2.1);

   first.merge(second, mycomparison);

   for (list<double>::iterator it = first.begin(); it != first.end(); ++it)
      cout << *it << " ";
   cout << endl;

   return 0;
}
```

声明对象

排序

合并

- 复杂度

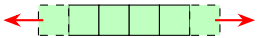


  - 访问前驱或后继：$O(1)$
  - 根据索引值访问节点数据：$O(n)$
  - 任意位置插入或删除：$O(1)$

- 使用

  - `#include <list>`
  - 适于数据频繁插入删除而不在意查找速度的场合
  - 排序、合并操作效率高

- 特点



  - 以多内存块 (chunks of storage) 形式存储数据
  - 可自动扩充，且提供越界检查
  - 适合在向量头尾插入或删除数据
  - 可用 [] 运算符直接存取数据 (随机访问 random access)

▶ 例 5

```cpp
//例 07-12; ex07-12.cpp
//容器 deque 的使用，演示 c++ 中容器 deque 的简单函数使用
#include <iostream>
#include <deque>
using namespace std;

int main()
{
  deque<float> dv;

  for (int i = 0; i < 6; ++i)
  {
    dv.push_front(i * 1.1);
  }

  for (int i = 0; i < dv.size(); ++i)
    cout << dv[i] << ' ';
  cout << endl;

  return 0;
}
```

▶ 例 6

```cpp
//例 07-13; ex07-13.cpp
//容器 deque 的 assign 函数的使用，演示 c++ 中容器 deque 的高级函数使用
#include <iostream>
#include <deque>
#include <string>

using namespace std;

int main()
{
   deque<string> ds;

   ds.assign (3, string("Hello"));
   ds.push_back ("] last");
   ds.push_front ("first [");

   for (int i = 0; i < ds.size(); ++i)
      cout << ds[i] << " ";
   cout << endl;

   ds.pop_front();
   ds.pop_back();

   for (int i = 1; i < ds.size(); ++i)
      ds[i] = "another " + ds [i];

   ds.resize (4, "Hello C++");

   for (int i = 0; i < ds.size(); ++i)
      cout << ds[i] << " ";
   cout << endl;

   return 0;
}
```

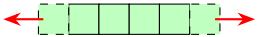
声明对象

添加 3 个数据：Hello 字符串

下标访问

扩展为 4 个元素，扩展后的元素赋值为 Hello C++

- 复杂度



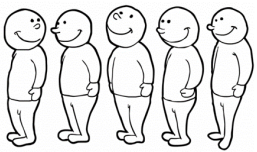
  - 随机存取：$O(1)$
  - 队列头尾插入或删除：$O(1)$

- 使用

  - `#include <deque>`
  - 适于快速在队列头尾存取数据但非频繁插入删除的场合
  - 若非头尾插入删除数据，效率比 list 低

- 特点：元素的添加或插入位置与元素的值相关
  - 集合 (set) 和多集 (multiset, 允许重复元素)
  - 映射 (map) 和多映射 (multimap, 允许重复元素键值)
- 复杂度：插入、删除和查找 $O(log(n))$

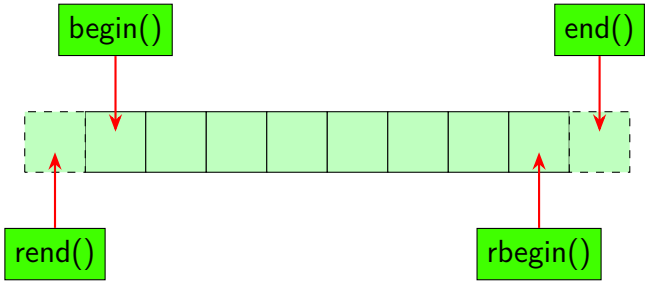
- 由顺序容器转换出的新容器 (list→queue, vector→stack)，不支持迭代器
  - 队列 (queue)
  - 优先队列 (priority queue)
  - 堆栈 (stack)



- 位集 bitset：灵活对二进制位进行操作
  - 整型到二进制的转换
  - 位的直接访问和设置等

- 能对顺序容器或关联容器中的每个元素进行连续存取的对象 (一个特殊的指针)
- 容器名<数据类型>::iterator 迭代器名
- 非标准迭代器
  - const_iterator
  - reverse_iterator
  - const_reverse_iterator

- 容器名.begin()，容器名.end()
- 容器名.size()



▶ 例 7

```cpp
//例 07-14; ex07-14.cpp
//容器 vector 的反向迭代器的使用，演示 c++ 中容器 vector 中反向迭代器的使用
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector <int>v;
   vector <int>::reverse_iterator p;

   for(int i = 0; i < 10; i++)
      v.push_back(i);

   for(p = v.rbegin(); p != v.rend(); p++)
      cout << *p << " ";
   cout << endl;
}
```

非 p−

分类

category				characteristic	valid expressions
all categories				Can be copied and copy-constructed	X b(a); b = a;
				Can be incremented	++a; a++; *a++
Random Access	Bidirectional	Forward	Input	Accepts equality/inequality comparisons	a == b; a != b
				Can be dereferenced as an *rvalue*	*a; a->m
			Output	Can be dereferenced to be the left side of an assignment operation	*a = t; *a++ = t
				Can be default-constructed	X a; X()
				Can be decremented	--a; a--; *a--
				Supports arithmetic operators + and -	a + n; n + a; n – a; a - b
				Supports inequality comparisons (< and >) between iterators	a < b; a > b
				Supports compound assignment operations +=, -=, <= and >=	a += n; a -= n a <= b; a >= b
				Supports offset dereference operator ([])	a[n]

- 在 vector 和 deque 中实现跳跃式访问

```
vector <int>::iterator p;
for(p=v.begin(); p!=v.end(); p+=2)
```

- list 中使用“+=”无法实现
  - 使用函数 advance

```
list <int>::iterator p;
for(p=v.begin(); p!=v.end(); advance(p,2))
```

- 提供一种一般化方法 (generic method)对不同类型容器中的元素进行访问

- 迭代器示例

```
//例 07-15; ex07-15.cpp
//在容器 vector 上定义寻找函数 FIND 来
//获得数 value 在容器 vector 中的位置
template <class ITER, class T>
ITER Find(ITER first, ITER last, T value)
{
   while(first != last && *first != value)
      ++first;
   return first;
}
```

```
//例 07-15; ex07-15.cpp
//容器 vector 上定义输出函数
template <class ITER>
void Print(ITER first, ITER last)
{
   while(first != last)
   {
      cout << *first << " ";
      ++first;
   }
   cout << endl;
}
```

- 迭代器示例

```cpp
//例 07-15; ex07-15.cpp
//容器 vector 的使用实例，演示 c++ 中容器 vector 的简单使用
typedef vector <int> container;
typedef vector <int>::iterator iterCon;

int main()
{
   container v;
   iterCon where;
   int key = 10;

   for(int i = 0; i < 10; i++)
      v.push_back(i);

   where = Find(v.begin(), v.end(), key);

   if(where != v.end())
      cout << *where << endl;
   else
      cout << "Fail to find the value!" << endl;

   Print(v.begin(), v.end());

   return 0;
}
```

- **算法实现机理**
- **非修改操作** Non-modifying sequence operations
- **修改操作** Modifying sequence operations
- **排序** Sorting
- **堆** Heap
- **二分查找** Binary search
- **合并** Merge
- **最小最大值** Min/max

- 基于迭代器和函数模板
- 算法 <algorithm> 是用来处理一个数据序列区间 (range) 的函数集合
- 区间中的元素通过迭代器进行访问
- 独立于所操作的容器

`for_each`	Apply function to range
`find`	Find value in range
`find_if`	Find element in range
`find_end`	Find last subsequence in range
`find_first_of`	Find element from set in range
`adjacent_find`	Find equal adjacent elements in range
`count`	Count appearances of value in range
`count_if`	Return number of elements in range satisfying condition
`mismatch`	Return first position where two ranges differ
`equal`	Test whether the elements in two ranges are equal
`search`	Find subsequence in range
`search_n`	Find succession of equal values in range

- find 函数模板

```cpp
//例 07-16; ex07-16.cpp
//定义寻找函数 find 来获得数 value 所在的位置
template<class InputIterator, class T>
InputIterator find( InputIterator first, InputIterator last, const T& value )
{
   for ( ; first != last; first++) if ( *first == value ) break;
   return first;
}
```

- find 函数模板

```cpp
//例 07-15; ex07-15.cpp
//容器 vector 的综合使用实例，演示 c++ 中容器 vector 的使用
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main ()
{
  int myints[] = { 10, 20, 30 , 40 };
  int * p;

  // pointer to array element:
  p = find(myints, myints + 4, 30);
  ++p;
  cout << "The element following 30 is " << *p << endl;

  vector<int> v (myints, myints + 4);
  vector<int>::iterator it;

  // iterator to vector element:
  it = find (v.begin(), v.end(), 30);
  ++it;
  cout << "The element following 30 is " << *it << endl;

  return 0;
}
```

for_each 函数模板

```
//例 07-15; ex07-15.cpp
//定义 for_each 函数，使得在 first 与 last 之间的元素都做 f 处理
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f)
{
   for ( ; first != last; ++first ) f(*first);
   return f;
}
```

函数指针

函数对象

- for_each 函数模板

```cpp
//例 07-17; ex07-17.cpp
//容器 vector 的综合使用，演示 c++ 中容器 vector 的综合使用实例
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

void myfunction (int i)
{
  cout << " " << i;
}

struct myclass
{
  void operator() (int i)
  {
    cout << " " << i;
  }
} myobject;

int main ()
{
  vector<int> v;
  v.push_back(10);
  v.push_back(20);
  v.push_back(30);

  cout << "v contains:";
  for_each (v.begin(), v.end(), myfunction);

  // or:
  cout << "\nv contains:";
  for_each (v.begin(), v.end(), myobject);

  cout << endl;

  return 0;
}
```

count_if 函数模板

```cpp
//例 07-18; ex07-18.cpp
//返回指定区间内的奇数个数，演示 c++ 中 count_if 的使用
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool IsOdd (int i)
{
   return ((i % 2) == 1);
}

int main ()
{
   int mycount;

   vector<int> v;
   for (int i = 1; i < 10; i++) v.push_back(i); // v: 1 2 3 4 5 6 7 8 9

   mycount = (int) count_if (v.begin(), v.end(), IsOdd);
   cout << "v contains " << mycount  << " odd values.\n";

   return 0;
}
```

`copy`	Copy range of elements
`copy_backward`	Copy range of elements backwards
`swap`	Exchange values of two objects
`swap_ranges`	Exchange values of two ranges
`iter_swap`	Exchange values of objects pointed by two iterators
`transform`	Apply function to range
`replace`	Replace value in range
`replace_if`	Replace values in range
`replace_copy`	Copy range replacing value
`replace_copy_if`	Copy range replacing value
`fill`	Fill range with value
`fill_n`	Fill sequence with value
`generate`	Generate values for range with function

generate_n	Generate values for sequence with function
remove	Remove value from range
remove_if	Remove elements from range
remove_copy	Copy range removing value
remove_copy_if	Copy range removing values
unique	Remove consecutive duplicates in range
unique_copy	Copy range removing duplicates
reverse	Reverse range
reverse_copy	Copy range reversed
rotate	Rotate elements in range
rotate_copy	Copy rotated range
random_shuffle	Rearrange elements in range randomly
partition	Partition range in two
stable_partition	Partition range in two - stable ordering

替换函数 replace

```cpp
//例 07-19; ex07-19.cpp
//容器 vector 中 replace 函数的使用，演示 c++ 中容器 vector 的高级函数使用
// replace algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main ()
{
  int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
  // 10 20 30 30 20 10 10 20
  vector<int> v (myints, myints + 8);

  // 10 99 30 30 99 10 10 99
  replace (v.begin(), v.end(), 20, 99);

  cout << "v contains:";
  for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    cout << " " << *it;

  cout << endl;

  return 0;
}
```

删除函数 remove

```cpp
//例 07-20; ex07-20.cpp
//容器 vector 中 remove 函数的使用，演示 c++ 中容器 vector 的高级函数使用
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main ()
{
   int myints[] = {10, 20, 30, 30, 20, 10, 10, 20};
   // 10 20 30 30 20 10 10 20
   vector <int> v(myints, myints + 8);
   vector <int>::iterator p, pEnd;

   pEnd = remove (v.begin(), v.end(), 20);

   for (p = v.begin(); p != pEnd; ++p)
      cout << " " << *p;
   cout << endl;

   return 0;
}
```

▶ 排序算法

`sort`	Sort elements in range
`stable_sort`	Sort elements preserving order of equivalents
`partial_sort`	Partially Sort elements in range
`partial_sort_copy`	Copy and partially sort range
`nth_element`	Sort element in range

```cpp
//例 07-21, ex07-21.cpp
//对各类容器的排序进行比较, 演示 c++ 中各类容器的使用
#include <iostream>
#include <list>
#include <vector>
#include <set>
#include <algorithm>
#include <ctime>
#include <iterator>

using namespace std;

bool compare(int i, int j)
{
  return (i > j);
}

int main()
{
  const int sz = 10000000;
  const int top100 = 100;

  vector<int> origin;
  for(int i = 0; i < sz; i++)
    origin.push_back(i);

  random_shuffle (origin.begin(), origin.end());
  //copy(origin.begin(), origin.end(), ostream_iterator<int>(cout, "\t"));

  list<int> lo;
  for(int i = 0; i < sz; i++)
    lo.push_back(origin[i]);
  clock_t ticks = clock();
  lo.sort();
  //copy(lo.begin(), lo.end(), ostream_iterator<int>(cout, "\t"));
  cout << "list sort:" << clock() - ticks << "ms" << endl;

  multiset<int> so;
  ticks = clock();
  for(int i = 0; i < sz; i++)
    so.insert(origin[i]);
  //copy(so.begin(), so.end(), ostream_iterator<int>(cout, "\t"));
  cout << "set sort:" << clock() - ticks << "ms" << endl;

  vector<int>vo;
  for(int i = 0; i < sz; i++) vo.push_back(origin[i]);
  ticks = clock();
  sort(vo.begin(), vo.end());
  //copy(vo.begin(), vo.end(), ostream_iterator<int>(cout, "\t"));
  cout << "vector sort:" << clock() - ticks << "ms" << endl;

  for(int i = 0; i < sz; i++)  vo[i] = origin[i];
  ticks = clock();
  make_heap (vo.begin(), vo.end());
  sort_heap(vo.begin(), vo.end());
  //copy(vo.begin(), vo.end(), ostream_iterator<int>(cout, "\t"));
  cout << "heap sort:" << clock() - ticks << "ms" << endl;

  for(int i = 0; i < sz; i++) vo[i] = origin[i];
  ticks = clock();
  partial_sort (vo.begin(), vo.begin() + top100, vo.end());
  copy(vo.begin(), vo.begin() + top100, ostream_iterator<int>(cout, "\t"));
  cout << "partial_sort top100:" << clock() - ticks << "ms" << endl;
}
```

排序算法

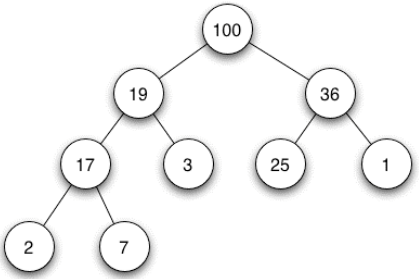
Name	Best	Average	Worst	Memory	Stable	Method
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	Depends	Partitioning
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends	Yes	Merging
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection
Insertion sort	n	n^2	n^2	1	Yes	Insertion
Selection sort	n^2	n^2	n^2	1	Depends	Selection
Shell sort	n	$n(\log n)^2$ or $n^{3/2}$	$O(n\log^2 n)$	1	No	Insertion
Bubble sort	n	n^2	n^2	1	Yes	Exchanging
Bucket sort		$n+k$			Yes	

排序算法

```cpp
//例 07-22; ex07-22.cpp
//容器 vector 中排序函数的使用，演示 C++ 中容器 vector 中排序函数的高级使用
// sort algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i, int j)
{
  return (i < j);
}

struct myclass
{
  bool operator() (int i, int j)
  {
    return (i < j);
  }
} myobject;

int main ()
{
  int myints[] = {32, 71, 12, 45, 26, 80, 53, 33};
  vector<int> v (myints, myints + 8);          // 32 71 12 45 26 80 53 33
  vector<int>::iterator it;

  // using default comparison (operator <):
  sort (v.begin(), v.begin() + 4);        //(12 32 45 71)26 80 53 33

  // using function as comp
  sort (v.begin() + 4, v.end(), myfunction); // 12 32 45 71(26 33 53 80)

  // using object as comp
  sort (v.begin(), v.end(), myobject);    //(12 26 32 33 45 53 71 80)

  // print out content:
  cout << "v contains:";
  for (it = v.begin(); it != v.end(); ++it)
    cout << " " << *it;

  cout << endl;

  return 0;
}
```

▶ 堆算法

push_heap	Push element into heap range
pop_heap	Pop element from heap range
make_heap	Make heap from range
sort_heap	Sort elements of heap



▶ 堆排序 sort_heap

```cpp
//例 07-23; ex07-23.cpp
//堆的建立，插入元素到堆，从堆中移出元素，演示 c++ 中关于堆的函数使用
// range heap example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main ()
{
  int myints[] = {10, 20, 30, 5, 15};
  vector<int> v(myints, myints + 5);
  vector<int>::iterator it;

  make_heap (v.begin(), v.end());
  cout << "initial max heap   : " << v.front() << endl;

  pop_heap (v.begin(), v.end());
  v.pop_back();
  cout << "max heap after pop : " << v.front() << endl;

  v.push_back(99);
  push_heap (v.begin(), v.end());
  cout << "max heap after push: " << v.front() << endl;

  sort_heap (v.begin(), v.end());

  cout << "final sorted range :";
  for (unsigned i = 0; i < v.size(); i++) cout << " " << v[i];

  cout << endl;

  return 0;
}
```

▶ 二分查找

`lower_bound`	Return iterator to lower bound
`upper_bound`	Return iterator to upper bound
`equal_range`	Get subrange of equal elements
`binary_search`	Test if value exists in sorted array

前提: 操作的对象序列已经排序

二分查找 binary_search

```cpp
//例 07-24: ex07-24.cpp
//二分查找函数的使用，演示 c++ 中 binary_search 函数的使用
// binary_search example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i, int j)
{
  return (i < j);
}

int main ()
{
  int myints[] = {1, 2, 3, 4, 5, 4, 3, 2, 1};
  vector<int> v(myints, myints + 9);              // 1 2 3 4 5 4 3 2 1

  // using default comparison:
  sort (v.begin(), v.end());

  cout << "looking for a 3... ";
  if (binary_search (v.begin(), v.end(), 3))
    cout << "found!\n";
  else cout << "not found.\n";

  // using myfunction as comp:
  sort (v.begin(), v.end(), myfunction);

  cout << "looking for a 6... ";
  if (binary_search (v.begin(), v.end(), 6, myfunction))
    cout << "found!\n";
  else cout << "not found.\n";

  return 0;
}
```

▶ 合并

`merge`	Merge sorted ranges
`inplace_merge`	Merge consecutive sorted ranges
`includes`	Whether sorted range includes another range
`set_union`	Union of two sorted ranges
`set_intersection`	Intersection of two sorted ranges
`set_difference`	Difference of two sorted ranges
`set_symmetric_difference`	Symmetric difference of two sorted ranges

前提: 操作的对象序列已经排序

求交集 set_intersection

```cpp
//例 07-25: ex07-25.cpp
//求两个集合的交集，演示 c++ 中 set_intersection 函数的使用
// set_intersection example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main ()
{
  int first[] = {5, 10, 15, 20, 25};
  int second[] = {50, 40, 30, 20, 10};
  vector<int> v(10);                      // 0 0 0 0 0 0 0 0 0 0
  vector<int>::iterator it;

  sort (first, first + 5);  //  5 10 15 20 25
  sort (second, second + 5); // 10 20 30 40 50

  it = set_intersection (first, first + 5, second, second + 5, v.begin());
  // 10 20 0 0 0 0 0 0 0 0

  cout << "intersection has " << int(it - v.begin()) << " elements.\n";

  return 0;
}
```

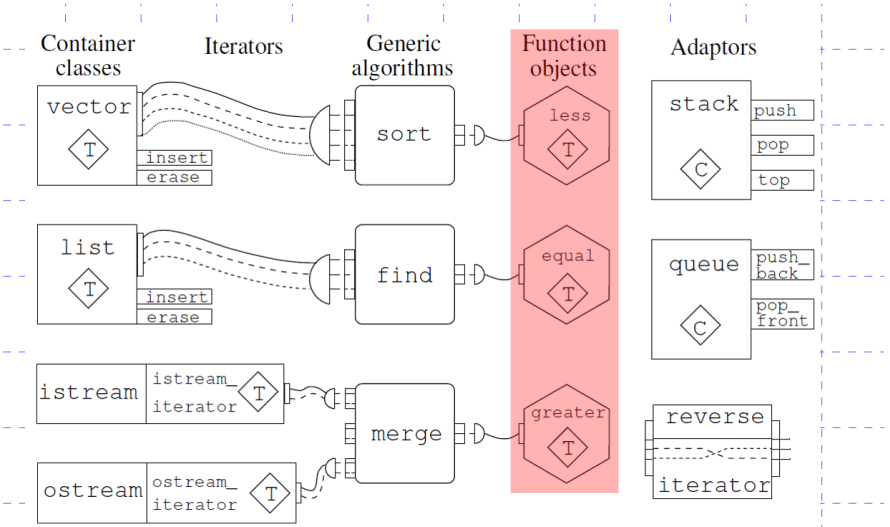
最小最大值

`min`	Return the lesser of two arguments
`max`	Return the greater of two arguments
`min_element`	Return smallest element in range
`max_element`	Return largest element in range
`lexicographical_compare`	Lexicographical less-than comparison
`next_permutation`	Transform range to next permutation
`prev_permutation`	Transform range to previous permutation

- 求最大值 max_element

```cpp
//例 07-26: ex07-26.cpp
//求数组中的最大最小值，演示 c++ 中 min_element、max_element 函数的使用
// min_element/max_element
#include <iostream>
#include <algorithm>
using namespace std;

bool myfn(int i, int j)
{
  return i < j;
}

struct myclass
{
  bool operator() (int i, int j)
  {
    return i < j;
  }
} myobj;

int main ()
{
  int myints[] = {3, 7, 2, 5, 6, 4, 9};

  // using default comparison:
  cout << "The smallest element is " << *min_element(myints, myints + 7) << endl;
  cout << "The largest element is " << *max_element(myints, myints + 7) << endl;

  // using function myfn as comp:
  cout << "The smallest element is " << *min_element(myints, myints + 7, myfn) << endl;
  cout << "The largest element is " << *max_element(myints, myints + 7, myfn) << endl;

  // using object myobj as comp:
  cout << "The smallest element is " << *min_element(myints, myints + 7, myobj) << endl;
  cout << "The largest element is " << *max_element(myints, myints + 7, myobj) << endl;

  return 0;
}
```

函数对象


Container classes
Iterators
Generic algorithms
Function objects
Adaptors
vector
T
insert
erase
sort
less
T
stack
C
push
pop
top
list
T
insert
erase
find
equal
T
queue
C
push_back
pop_front
istream
istream_iterator
T
merge
greater
T
reverse
iterator
ostream
ostream_iterator
T

- 定义
  - 函数调用运算符 ()
  - 如果某个类重载了 ()，该类的实例就是一个函数对象

```
struct MyClass
{
  返回值 operator()(参数列表) {}
};

MyClass myObj;
myObj(实参);
```

- 优点
  - 使用灵活，可包含状态信息
  - 可当作数据类型作为模板参数传递

- 例 1

```
//例 07-27; ex07-27.cpp
//模板类的定义
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

template <class T>
class AddClass
{
private:
   T theValue;
public:
   AddClass (const T& v) : theValue(v) {}
   void operator() (T& elem) const
   {
      elem += theValue;
   }
};

template <class T, int theValue>
void AddFun(T& elem)
{
   elem += theValue;
}
```

- 例 1

```cpp
//例 07-27; ex07-27.cpp
//输出 AddFun 处理过后的容器中的值，演示 c++ 中 for_each 的使用
int main()
{
   vector<int> v(10, 0);
   const int n = 10;

   for_each (v.begin(), v.end(), AddFun<int, 10>);
   //for_each (v.begin(), v.end(), AddClass<int>(10));
   for_each (v.begin(), v.end(), Print);
   cout << endl;
   for_each (v.begin(), v.end(), AddFun < int, 11 > );
   //for_each (v.begin(),v.end(), AddClass<int>(11));
   for_each (v.begin(), v.end(), Print);
   cout << endl;
}
```

- 例 2

```
//例 07-27; ex07-27.cpp
//输出 AddFun 处理过后的容器中的值，演示 c++ 中 for_each 的使用
int main()
{
   vector<int> v(10, 0);
   const int n = 10;

   for_each (v.begin(), v.end(), AddFun<int, n>);
   //for_each (v.begin(), v.end(), AddClass<int>(n));
   for_each (v.begin(), v.end(), Print);
   cout << endl;
   for_each (v.begin(), v.end(), AddFun < int, n + 1 > );
   //for_each (v.begin(),v.end(), AddClass<int>(n+1));
   for_each (v.begin(), v.end(), Print);
   cout << endl;
}
```

- 例 2

```cpp
//例 07-28; ex07-28.cpp
//定义 PersonSortClass 类
#include <iostream>
#include <string>
#include <set>

using namespace std;
struct Person
{
   string fName;
   string lName;
   Person(string gName = "", string fName = "");
      fName(gName), lName(fName) {}
};

class PersonSortClass
{
public:
   bool operator() (const Person& p1, const Person& p2)
   {
      if (p1.lName < p2.lName)
         return true;
      else if(p1.lName == p2.lName)
         return (p1.fName < p2.fName);
      else
         return false;
   }
};

bool PersonSortFun(const Person& p1, const Person& p2)
{
   if (p1.lName < p2.lName)
      return true;
   else if(p1.lName == p2.lName)
      return (p1.fName < p2.fName);
   else
      return false;
}
```

- 例 2

```
//例 07-28; ex07-28.cpp
//在容器 set 中插入元素，演示 c++ 中 set 容器的高级用法
int main()
{
   typedef set<Person, PersonSortClass> PersonSet;
   PersonSet s;
   PersonSet::iterator p;

   s.insert(Person("Thomas  ", "Edison"));
   s.insert(Person("Helen   ", "Keller"));
   s.insert(Person("James   ", "Taylor"));
   s.insert(Person("Elizabeth", "Taylor"));
   s.insert(Person("Isaac   ", "Newton"));
   s.insert(Person("Lewis   ", "Carrol"));

   for (p = s.begin(); p != s.end(); ++p)
      cout << (*p).fName << "\t" << (*p).lName << endl;
}
```

- 算术操作 Arithmetic operations
- 比较操作 Comparison operations
- 逻辑操作 Logical operations
- 其它

- 算术操作

plus	Addition function object class
minus	Subtraction function object class
multiplies	Multiplication function object class
divides	Division function object class
modulus	Modulus function object class
negate	Negative function object class

- 算术操作

```
template <class T>
struct plus : binary_function <T,T,T>
{
  T operator() (const T& x, const T& y) const
  {
    return x+y;
  }
};
```

- plus 类模板

```cpp
//例 07-29; ex07-29.cpp
//first、second 数组中的对应元素相加，演示 c++ 中 transform 函数的使用
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

int main ()
{
   int first[] = {1, 2, 3, 4, 5};
   int second[] = {10, 20, 30, 40, 50};
   int results[5];
   transform(first, first + 5, second, results, plus<int>());
   for (int i = 0; i < 5; i++)
      cout << results[i] << " ";

   cout << endl;

   return 0;
}
```

- 比较操作

`equal_to`	Function object class for equality comparison
`not_equal_to`	Non-equality comparison
`greater`	Greater-than inequality comparison
`less`	Less-than inequality comparison
`greater_equal`	Greater-than-or-equal-to comparison
`less_equal`	Less-than-or-equal-to comparison

- 大于操作 greater

```cpp
//例 07-30; ex07-30.cpp
//数组排序，演示 c++ 中 sort 函数的使用
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

int main ()
{
   int numbers[] = {20, 40, 50, 10, 30};
   sort (numbers, numbers + 5, greater<int>() );
   for (int i = 0; i < 5; i++)
      cout << numbers[i] << " ";
   cout << endl;
   return 0;
}
```

- 逻辑操作

`logical_and`	Logical AND function object class
`logical_or`	Logical OR function object class
`logical_not`	Logical NOT function object class

- 逻辑非操作 logical_not

```
template <class T>
struct logical_not : unary_function <T,bool>
{
  bool operator() (const T& x) const
  {
    return !x;
  }
};
```

▶ 逻辑非操作 logical_not

```cpp
//例 07-31: ex07-31.cpp
//对数组 value 中的布尔值取反，演示 c++ 中 transform 函数的使用
#include <functional>
#include <algorithm>

int main ()
{
   bool values[] = {true, false};
   bool result[2];

   transform (values, values + 2, result, logical_not<bool>() );
   cout << boolalpha << "Logical NOT:\n";

   for (int i = 0; i < 2; i++)
      cout << "NOT " << values[i] << " = " << result[i] << "\n";

   return 0;
}
```

- 树
- 图
- 其它
  - 矩阵运算
  - 正则表达式
  - 智能指针
  - 高斯分布随机数
  - 多线程操作等

- Core::tree(适合在 VS 下编译)

  http://archive.gamedev.net/archive/reference/programming/features/coretree2/page5.html

- Tree.hh: an STL-like C++ tree class

  http://tree.phi-sci.com/documentation.html

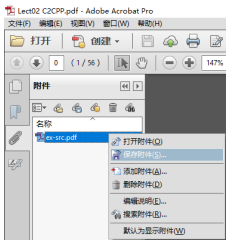
- 一组扩充 C++ 功能性的经过同行评审（Peer-reviewed）且开放源代码程序库。
- 开源代码。 **http://www.boost.org/**
- 许多 Boost 的开发人员来自 C++ 标准委员会，而部份的 Boost 库成为 C++ 的 TR1 标准之一。

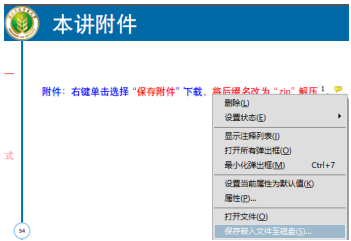
http://mail.ustc.edu.cn/~jxw95216/boost_doc/libs/libraries.htm http://www.kmonos.net/alang/boost/

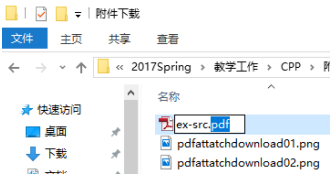
- 分类库列表
  - String and text processing 字符串与文本处理
  - Containers 容器
  - Iterators 迭代器
  - Algorithms 算法
  - Function Objects and higher-order programming 函数对象与高阶编程
  - Generic Programming 泛型编程
  - Template Metaprogramming 模板元编程
  - Preprocessor Metaprogramming 预处理元编程
  - Concurrent Programming 并发编程
  - Math and numerics 数学与数字
  - Correctness and testing 正确性与测试
  - Data structures 数据结构
  - Image processing 图像处理
  - Input/Output 输入/输出
  - Inter-language support 交叉语言支持
  - Memory 内存
  - Parsing 语法分析
  - Programming Interfaces 编程接口
  - Miscellaneous 杂项
  - Broken compiler workarounds 不合标准的编译器支持

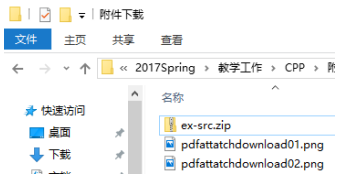
- 安装与编译
  - **date_time, regex, thread, python, signals, test, filesystem, serialization, program_options** 需要 bjam 进行编译生成 *.lib 文件;
    VS: **http://www.boostpro.com/download/**
  - 其余组件只需包含相应头文件即可。

附件：右键单击该链接，选择“保存附件”下载，将后缀名改为“.zip”解压[1] [2]。









[1] 请退出全屏模式后点击该链接。

[2] 以 Adobe Acrobat Reader 为例。

本讲结束，谢谢!
欢迎多提宝贵意见和建议

西北农林科技大学
NORTHWEST A&F UNIVERSITY
中国 · 杨凌