<https://blog.csdn.net/hgy2011/article/details/8775994>

# libcstl简介

libcstl是一个应用于C语言编程的函数库，它将编程过程中经常使用的数据结构如向量、链表、集合、树等封 装成相应的数据结构并提供一系列的操作函数来操作保存在这些数据结构中的数据，同时它还将常用的算法如 排序、查找、划分等封装成相应的算法函数并提供迭代器来使两者之间建立联系方便使用。从libcstl的名字 就可以看出它于STL有一定的关系，是的libcstl的接口和实现都是模仿了STL。  
  
libcstl的产生主要是基于使用C语言编程过程中经常要使用向量、链表、集合等数据结构和排序、查找、划分 等算法，但是对于这些数据结构和算法每次都要重复的实现，如果有一个通用的类似于STL的数据结构和算法 的库就可以节约时间和成本，这样libcstl就诞生了。

# libcstl编译和安装

在libcstl-2.0.2中使用configure选项来支持2.0.的编译配置。  
  
  
关闭控制断言(--disable-assert)  
定义这个选项可以去掉库中的断言.当库函数接受到非法参数时,断言就会报错,但是有断言的版本执行效率低 (非断言版本效率大约是有断言版本的 20~40 倍).   
  
  
  
开启内存管理(--with-memory-management)  
这个选项可以开启内存管理，默认情况下内存管理是关闭的。   
  
  
  
配置 stack\_t 适配器的底层实现(--enable-stack-implementation=ARGUMENT)  
这个选项是配置 stack\_t 适配器的底层实现的。ARGUMENT 作为 stack\_t 的底层实现,ARGUMENT 可以是 vector(--enable-stack-implementation=vector)和 list(--enable-stack-implementation=list), 默认使用 deque\_t 作为 stack\_t的底层实现.  
  
  
  
配置 queue\_t 适配器的底层实现(--enable-queue-implementation=ARGUMENT)  
这个选项是配置 queue\_t 的底层实现,ARGUMENT 作为 queue\_t 的底层实现,ARGUMENT 是 list(--enable-queue-implementation=list),默认使用 deque\_t 为 queue\_t 的底层实现.  
  
  
  
配置 set\_t 的底层实现(--enable-set-implementation=ARGUMENT)  
  
  
  
配置 multiset\_t 的底层实现(--enable-multiset-implementation=ARGUMENT)  
  
  
  
配置 map\_t 的底层实现(--enable-map-implementation=ARGUMENT)  
  
  
配置 multimap\_t 的底层实现(--enable-multimap-implementation=ARGUMENT)  
  
以上四个选项是配置关联容器的底层实现的,ARGUMENT 的可选参数是 avl-tree,关联容器默认使用红黑树作为 底层实现(红黑树比 avl 树快 40%)。默认使用红黑树作为底层实现。

# libcstl基本概念

libcstl由多个分组成，容器，迭代器，算法和函数 容器: 容器以某种结构形式管理数据的集合,每一种容器都有各自的特点. 迭代器: 迭代器与容器相关联,应用与容器或容器的子集,它的主要好处在于为容器提供了一个统一的接口.迭代器是容器 和算法之间的桥梁,它使得算法独立于特定的容器. 算法: 算法是对数据集合进行某些操作,它可以排序,查找,修改或其他一些操作.算法使用迭代器作为输入,这样算法可 以独立于容器,实现在任意容器上的操作.同时算法还使用特定的函数对容器中的数据进行特定的操作. 函数: 函数只是规定了算法中使用的函数形式,并且定义了一些算法中常用的函数，可以作为算法的自定义规则.

## 容器

容器可以用来排序数据,管理和存储数据,所以为了不同的目的libcstl提供了不同的容器.容器分为: 序列容器: 序列容器主要用来存储和管理数据,容器中的数据的顺序与数据插入到容器中的次序有关,而与数据本身的值无关. libcstl提供的序列容器有:vector\_t, list\_t, deque\_t, slist\_t. 关联容器: 关联容器更关系容器中数据的排列顺序,容器中数据的顺序是排序的与数据本身有关而与数据插入到容器中的次 序无关.libcstl 提供的关联容器有:set\_t, map\_t, multiset\_t, multimap\_t, hash\_set\_t, hash\_map\_t, hash\_multiset\_t, hash\_multimap\_t. 容器适配器: 除了以上这些容器之外,libcstl为了特殊的目的还提供了容器适配器,它们都是基本的容器实现的. 容器适配器有:stack\_t,queue\_t,priority\_queue\_t. 字符串类型： string\_t类型可以像c-str一样拷贝,赋值,比较而不必考虑是否有足够的内存来保存字符串,会不会越界等等. 因为string\_t可以动态增长,并且易于使用,你可很方便的插入删除字符或子串,方便的替换等等.

## 迭代器

迭代器是对容器的特定范围的数据进行遍历的类型,这个范围可能是整个容器或者是容器的一部分.迭代器表示的 是容器中数据的位置的数据结构，它将各个容器的数据位置统一，使用同样的接口进行操作。各个容器都提供了迭代器结构， 同时各种算法都是通过迭代器来操作的，所以说迭代器是容器和算法之间联系的桥梁。

## 算法

libcstl为了处理数据提供了许多算法,例如查找,排序,拷贝,修改还有算术操作.算法不属于任何一种容器,它能 够处理任何容器中的数据,算法都是以迭代器作为输入

## 容器使用代码

以vector\_t例子介绍容器的使用，容器的使用可以分为四步：  
  
1.创建容器  
2.初始化容器  
3.对容器操作  
4.销毁容器  
  
以下是使用容器的代码，其中对容器遍历用了两种方法，一种是迭代器，一种是用下标，建议遍历的时候使用迭代器。

**[html]** [view plaincopy](http://blog.csdn.net/devilcash/article/details/7209374)

1. #include **<stdio.h>**
2. #include **<stdlib.h>**
3. #include **<time.h>**
5. #include **<cstl**/cvector.h**>**
6. /\*\*
7. \*bref 比较容器中两个节点值的大小
8. \*/
9. void value\_greater(const void \*cpv\_first, const void \*cpv\_second, void \*pv\_output)
10. {
11. if (\*(int \*)cpv\_first **>** \*(int \*)cpv\_second){
12. \*(bool\_t \*)pv\_output = true;
13. }else{
14. \*(bool\_t \*)pv\_output = false;
15. }
16. }
18. /\*\*
19. \*bref 用迭代器遍历容器
20. \*/
21. void vector\_travel\_by\_iter(vector\_t \*pt\_vec)
22. {
23. iterator\_t iter;
24. if (pt\_vec == NULL){
25. fprintf(stderr, "[travel\_inter]:vector is null.\n");
26. return;
27. }
28. printf("------------使用迭代器遍历 ------------\n");
29. for (iter = vector\_begin(pt\_vec); !iterator\_equal(iter, vector\_end(pt\_vec)); iter = iterator\_next(iter)){
30. printf("%d  ", \*(int \*)iterator\_get\_pointer(iter));
31. }
32. printf("\n");
33. }

36. /\*\*
37. \*bref 用下标遍历容器
38. \*/
40. void vector\_travel\_by\_index(vector\_t \*pt\_vec)
41. {
42. size\_t t\_index = 0;
43. if (pt\_vec == NULL){
44. fprintf(stderr, "[travel\_index]:vector is null.\n");
45. return;
46. }
47. printf("------------使用下标遍历------------\n");
48. for (t\_index = 0; t\_index **<** **10**; t\_index++){
49. printf("%d  ", \*(int \*)vector\_at(pt\_vec, t\_index));
50. }
51. printf("\n");
52. }
54. int main(int argc, char \*argv[])
55. {
56. vector\_t \*pt\_vec = create\_vector(int);
57. size\_t t\_index = 0;
58. iterator\_t iter;
59. if (pt\_vec == NULL){
60. fprintf(stderr, "create vector is failed.\n");
61. return -1;
62. }
64. vector\_init(pt\_vec);
65. srand((unsigned int)time(NULL));
67. ///循环赋值
68. for (t\_index = 0; t\_index **<** **10**; t\_index++){
69. vector\_push\_back(pt\_vec, rand()%64);
70. }
72. printf("begin sorting:\n");
73. vector\_travel\_by\_iter(pt\_vec);
74. vector\_travel\_by\_index(pt\_vec);
76. printf("\nafter order sorting:\n");
78. ///有条件排序，排序条件以回调函数的形式给出，如果回调函数为空，使用默认条件（递增）排序
79. algo\_sort\_if(vector\_begin(pt\_vec), vector\_end(pt\_vec), value\_greater);
80. vector\_travel\_by\_iter(pt\_vec);
81. printf("\nafter reverse sorting:\n");
83. ///默认条件（递增）排序
84. algo\_sort(vector\_begin(pt\_vec), vector\_end(pt\_vec));
85. vector\_travel\_by\_index(pt\_vec);
87. vector\_destroy(pt\_vec);
88. return 0;
89. }

**[html]** [view plaincopy](http://blog.csdn.net/devilcash/article/details/7209374)

1. 编译：
2. gcc -o test\_vector test\_vector.c -I/root/libcstl/lib/include/ -L/root/libcstl/lib/lib/ -lcstl

运行结果如下：

**[html]** [view plaincopy](http://blog.csdn.net/devilcash/article/details/7209374)

1. begin sorting:
2. ------------使用迭代器遍历 ------------
3. 27  7  16  58  49  24  53  14  34  22
4. ------------使用下标遍历------------
6. 27  7  16  58  49  24  53  14  34  22
8. after order sorting:
9. ------------使用迭代器遍历 ------------
10. 58  53  49  34  27  24  22  16  14  7
12. after reverse sorting:
13. ------------使用下标遍历------------
14. 7  14  16  22  24  27  34  49  53  58

## libcstl适用的数据类型

上面这些例子使用的都是基本的数据类型，但是在实际的编程过程中经常使用的是自定义的数据结构类型，libcstl-1.0不能够有效的 处理这些数据类型的，但是libcstl-2.0.0相对于1.0.1来说最大的提高就是增强了对各种数据类型的处理能力。libcstl-2.0.0有效的 处理绝大部分的数据类型。它将数据类型分为3类：  
  
1.C语言内建类型：如：int, long, double等。  
2.用户自定义类型：用户自己定义的数据结构。  
3.libcstl内建类型：如vector\_t, set\_t, hash\_multimap\_t等。   
  
其中libcstl内建类型是用户自定义类型的特例，只是libcstl对于libcstl内建类型进行了默认的处理

### 自定义数据类型使用例子

使用自定义数据类型步骤如下：  
  
1.定义数据类型  
2.定义与该数据类型相对应的初始函数，拷贝，比较函数，销毁函数  
3.使用type\_register(user\_t, user\_init, user\_copy, user\_less, user\_destroy);注册  
4.创建容器  
5.初始化容器  
6.操作容器  
7.销毁容器

**[html]** [view plaincopy](http://blog.csdn.net/devilcash/article/details/7209374)

1. #include **<stdlib.h>**
2. #include **<stdio.h>**
4. #include **<cstl**/clist.h**>**
6. #define N\_MAX  29
8. typedef unsigned char   uint8\_t;
9. typedef unsigned short  uint16\_t;
10. typedef unsigned int    uint32\_t;
12. typedef struct{
13. uint8\_t  id;
14. uint8\_t  age;
15. uint8\_t  sex;
16. char     name[N\_MAX];
18. }user\_t;

21. static void \_user\_init(const void \*cpv\_input, void \*pv\_output)
22. {
23. user\_t \* user\_input = (user\_t \*)cpv\_input;
24. memset(user\_input, 0, sizeof(user\_t));
25. \*(bool\_t \*)pv\_output = true;
26. }
28. static void \_user\_destroy(const void \*cpv\_input, void \*pv\_output)
29. {
30. user\_t \* user\_input = (user\_t \*)cpv\_input;
31. memset(user\_input, 0, sizeof(user\_t));
32. \*(bool\_t \*)pv\_output = true;
33. }
35. static void \_user\_copy(const void \*cpv\_first, const void \*cpv\_second, void \*pv\_output)
36. {
37. user\_t \*user\_1 = (user\_t \*)cpv\_first;
38. user\_t \*user\_2 = (user\_t \*)cpv\_second;
39. user\_1-**>**id = user\_2-**>**id;
40. user\_1-**>**age = user\_2-**>**age;
41. user\_1-**>**sex = user\_2-**>**sex;
42. strncpy(user\_1-**>**name, user\_2-**>**name, N\_MAX);
43. \*(bool\_t \*)pv\_output = true;
45. }
47. static void \_user\_less(const void \*cpv\_first, const void \*cpv\_second, void \*pv\_output)
48. {
49. if (((user\_t \*)cpv\_first)-**>**id **<** ((user\_t \*)cpv\_second)-**>**id){
50. \*(bool\_t \*)pv\_output = true;
51. }else{
52. \*(bool\_t \*)pv\_output = false;
53. }
54. }
56. void list\_travel\_by\_iter(list\_t \*pt\_list)
57. {
58. iterator\_t iter;
59. user\_t user;
60. if (pt\_list == NULL){
61. fprintf(stderr, "[travel\_inter]:list is null.\n");
62. return;
63. }
64. printf("------------使用迭代器遍历 ------------\n");
65. printf("Id\t   Name   \tAge\tSex\t\n");
66. for (iter = list\_begin(pt\_list); !iterator\_equal(iter, list\_end(pt\_list)); iter = iterator\_next(iter)){
67. iterator\_get\_value(iter, &user);
68. printf("%u\t %s \t% u\t %c\n", user.id, user.name, user.age, user.sex?'m':'f');
69. }
70. printf("\n");
71. }
73. int main(int argc, char \*argv[])
74. {
75. list\_t \*pt\_list = NULL;
76. size\_t index;
77. user\_t user\_obj[] = {{134, 57, 1, "steven jobs"},
78. {110, 50, 1, "bill gates"},
79. {85, 52, 0, "alex martelli"},
80. {220, 30, 0, "michel wood"},
81. {24, 72, 1, "bob green"},
82. {102, 29, 1, "devil cash"}};
84. type\_register(user\_t, \_user\_init, \_user\_copy, \_user\_less, \_user\_destroy);
86. pt\_list = create\_list(user\_t);
87. if (pt\_list == NULL){
88. fprintf(stderr, "create list is failed.\n");
89. return -1;
90. }
92. list\_init(pt\_list);
94. for (index = 0; index **<** **6**; index++){
95. list\_push\_back(pt\_list, &user\_obj[index]);
96. }
97. printf("before sorting\n");
98. list\_travel\_by\_iter(pt\_list);
100. list\_sort(pt\_list);
102. printf("after sorting\n");
103. list\_travel\_by\_iter(pt\_list);
105. list\_destroy(pt\_list);
107. return 0;
108. }

编译

**[html]** [view plaincopy](http://blog.csdn.net/devilcash/article/details/7209374)

1. gcc -o test\_list test\_list.c -I/root/libcstl/lib/include/ -L/root/libcstl/lib/lib/ -lcstl

运行结果如下：

**[html]** [view plaincopy](http://blog.csdn.net/devilcash/article/details/7209374)

1. before sorting
2. ------------使用迭代器遍历 ------------
3. Id         Name         Age     Sex
4. 134      steven jobs    57       m
5. 110      bill gates     50       m
6. 85       alex martelli  52       f
7. 220      michel wood    30       f
8. 24       bob green      72       m
9. 102      devil cash     29       m
11. after sorting
12. ------------使用迭代器遍历 ------------
13. Id         Name         Age     Sex
14. 24       bob green      72       m
15. 85       alex martelli  52       f
16. 102      devil cash     29       m
17. 110      bill gates     50       m
18. 134      steven jobs    57       m
19. 220      michel wood    30       f