**基于内存查看STL常用容器内容**

DEC 3RD, 2014 12:00 AM

有时候在线上使用gdb调试程序core问题时，可能没有符号文件，拿到的仅是一个内存地址，如果这个指向的是一个STL对象，那么如何查看这个对象的内容呢？

只需要知道STL各个容器的数据结构实现，就可以查看其内容。本文描述了SGI STL实现中常用容器的数据结构，以及如何在gdb中查看其内容。

**string**

string，即basic\_string bits/basic\_string.h：

mutable \_Alloc\_hider \_M\_dataplus;

...

const \_CharT\*

c\_str() const

{ return \_M\_data(); }

...

\_CharT\*

\_M\_data() const

{ return \_M\_dataplus.\_M\_p; }

...

struct \_Alloc\_hider : \_Alloc

{

\_Alloc\_hider(\_CharT\* \_\_dat, const \_Alloc& \_\_a)

: \_Alloc(\_\_a), \_M\_p(\_\_dat) { }

\_CharT\* \_M\_p; // The actual data.

};

size\_type

length() const

{ return \_M\_rep()->\_M\_length; }

\_Rep\*

\_M\_rep() const

{ return &((reinterpret\_cast<\_Rep\*> (\_M\_data()))[-1]); }

...

struct \_Rep\_base

{

size\_type \_M\_length;

size\_type \_M\_capacity;

\_Atomic\_word \_M\_refcount;

};

struct \_Rep : \_Rep\_base

即，string内有一个指针，指向实际的字符串位置，这个位置前面有一个\_Rep结构，其内保存了字符串的长度、可用内存以及引用计数。当我们拿到一个string对象的地址时，可以通过以下代码获取相关值：

void ds\_str\_i(void \*p) {

char \*\*raw = (char\*\*)p;

char \*s = \*raw;

size\_t len = \*(size\_t\*)(s - sizeof(size\_t) \* 3);

printf("str: %s (%zd)\n", s, len);

}

size\_t ds\_str() {

std::string s = "hello";

ds\_str\_i(&s);

return s.size();

}

在gdb中拿到一个string的地址时，可以以下打印出该字符串及长度：

(gdb) x/1a p

0x7fffffffe3a0: 0x606028

(gdb) p (char\*)0x606028

$2 = 0x606028 "hello"

(gdb) x/1dg 0x606028-24

0x606010: 5

**vector**

众所周知vector实现就是一块连续的内存，bits/stl\_vector.h。

template<typename \_Tp, typename \_Alloc = std::allocator<\_Tp> >

class vector : protected \_Vector\_base<\_Tp, \_Alloc>

...

template<typename \_Tp, typename \_Alloc>

struct \_Vector\_base

{

typedef typename \_Alloc::template rebind<\_Tp>::other \_Tp\_alloc\_type;

struct \_Vector\_impl

: public \_Tp\_alloc\_type

{

\_Tp\* \_M\_start;

\_Tp\* \_M\_finish;

\_Tp\* \_M\_end\_of\_storage;

\_Vector\_impl(\_Tp\_alloc\_type const& \_\_a)

: \_Tp\_alloc\_type(\_\_a), \_M\_start(0), \_M\_finish(0), \_M\_end\_of\_storage(0)

{ }

};

\_Vector\_impl \_M\_impl;

可以看出sizeof(vector<xxx>)=24，其内也就是3个指针，\_M\_start指向首元素地址，\_M\_finish指向最后一个节点+1，\_M\_end\_of\_storage是可用空间最后的位置。

iterator

end()

{ return iterator (this->\_M\_impl.\_M\_finish); }

const\_iterator

...

begin() const

{ return const\_iterator (this->\_M\_impl.\_M\_start); }

...

size\_type

capacity() const

{ return size\_type(const\_iterator(this->\_M\_impl.\_M\_end\_of\_storage)

- begin()); }

可以通过代码从一个vector对象地址输出其信息：

template <typename T>

void ds\_vec\_i(void \*p) {

T \*start = \*(T\*\*)p;

T \*finish = \*(T\*\*)((char\*)p + sizeof(void\*));

T \*end\_storage = \*(T\*\*)((char\*)p + 2 \* sizeof(void\*));

printf("vec size: %ld, avaiable size: %ld\n", finish - start, end\_storage - start);

}

size\_t ds\_vec() {

std::vector<int> vec;

vec.push\_back(0x11);

vec.push\_back(0x22);

vec.push\_back(0x33);

ds\_vec\_i<int>(&vec);

return vec.size();

}

使用gdb输出一个vector中的内容：

(gdb) p p

$3 = (void \*) 0x7fffffffe380

(gdb) x/1a p

0x7fffffffe380: 0x606080

(gdb) x/3xw 0x606080

0x606080: 0x00000011 0x00000022 0x00000033

**list**

众所周知list被实现为一个链表。准确来说是一个双向链表。list本身是一个特殊节点，其代表end，其指向的下一个元素才是list真正的第一个节点：

bits/stl\_list.h

bool

empty() const

{ return this->\_M\_impl.\_M\_node.\_M\_next == &this->\_M\_impl.\_M\_node; }

const\_iterator

begin() const

{ return const\_iterator(this->\_M\_impl.\_M\_node.\_M\_next); }

iterator

end()

{ return iterator(&this->\_M\_impl.\_M\_node); }

...

struct \_List\_node\_base

{

\_List\_node\_base\* \_M\_next; ///< Self-explanatory

\_List\_node\_base\* \_M\_prev; ///< Self-explanatory

...

};

template<typename \_Tp>

struct \_List\_node : public \_List\_node\_base

{

\_Tp \_M\_data; ///< User's data.

};

template<typename \_Tp, typename \_Alloc>

class \_List\_base

{

...

struct \_List\_impl

: public \_Node\_alloc\_type

{

\_List\_node\_base \_M\_node;

...

};

\_List\_impl \_M\_impl;

template<typename \_Tp, typename \_Alloc = std::allocator<\_Tp> >

class list : protected \_List\_base<\_Tp, \_Alloc>

所以sizeof(list<xx>)=16，两个指针。每一个真正的节点首先是包含两个指针，然后是元素内容(\_List\_node)。

通过代码输出list的内容：

#define NEXT(ptr, T) do { \

void \*n = \*(char\*\*)ptr; \

T val = \*(T\*)((char\*\*)ptr + 2); \

printf("list item %p val: 0x%x\n", ptr, val); \

ptr = n; \

} while (0)

template <typename T>

void ds\_list\_i(void \*p) {

void \*ptr = \*(char\*\*)p;

NEXT(ptr, T);

NEXT(ptr, T);

NEXT(ptr, T);

}

size\_t ds\_list() {

std::list<int> lst;

lst.push\_back(0x11);

lst.push\_back(0x22);

lst.push\_back(0x33);

ds\_list\_i<int>(&lst);

return lst.size();

}

在gdb中可以以下方式遍历该list：

(gdb) p p

$4 = (void \*) 0x7fffffffe390

(gdb) x/1a p

0x7fffffffe390: 0x606080

(gdb) x/1xw 0x606080+16 # 元素1

0x606090: 0x00000011

(gdb) x/1a 0x606080

0x606080: 0x6060a0

(gdb) x/1xw 0x6060a0+16 # 元素2

0x6060b0: 0x00000022

**map**

map使用的是红黑树实现，实际使用的是stl\_tree.h实现：

bits/stl\_map.h

typedef \_Rb\_tree<key\_type, value\_type, \_Select1st<value\_type>,

key\_compare, \_Pair\_alloc\_type> \_Rep\_type;

...

\_Rep\_type \_M\_t;

...

iterator

begin()

{ return \_M\_t.begin(); }

bits/stl\_tree.h

struct \_Rb\_tree\_node\_base

{

typedef \_Rb\_tree\_node\_base\* \_Base\_ptr;

typedef const \_Rb\_tree\_node\_base\* \_Const\_Base\_ptr;

\_Rb\_tree\_color \_M\_color;

\_Base\_ptr \_M\_parent;

\_Base\_ptr \_M\_left;

\_Base\_ptr \_M\_right;

...

};

template<typename \_Val>

struct \_Rb\_tree\_node : public \_Rb\_tree\_node\_base

{

typedef \_Rb\_tree\_node<\_Val>\* \_Link\_type;

\_Val \_M\_value\_field;

};

template<typename \_Key\_compare,

bool \_Is\_pod\_comparator = std::\_\_is\_pod<\_Key\_compare>::\_\_value>

struct \_Rb\_tree\_impl : public \_Node\_allocator

{

\_Key\_compare \_M\_key\_compare;

\_Rb\_tree\_node\_base \_M\_header;

size\_type \_M\_node\_count; // Keeps track of size of tree.

...

}

\_Rb\_tree\_impl<\_Compare> \_M\_impl;

...

iterator

begin()

{

return iterator(static\_cast<\_Link\_type>

(this->\_M\_impl.\_M\_header.\_M\_left));

}

所以可以看出，大部分时候(取决于\_M\_key\_compare) sizeof(map<xx>)=48，主要的元素是：

\_Rb\_tree\_color \_M\_color; // 节点颜色

\_Base\_ptr \_M\_parent; // 父节点

\_Base\_ptr \_M\_left; // 左节点

\_Base\_ptr \_M\_right; // 右节点

\_Val \_M\_value\_field // 同list中节点技巧一致，后面是实际的元素

同list中的实现一致，map本身作为一个节点，其不是一个存储数据的节点，

\_Rb\_tree::end

iterator

end()

{ return iterator(static\_cast<\_Link\_type>(&this->\_M\_impl.\_M\_header)); }

由于节点值在\_Rb\_tree\_node\_base后，所以任意时候拿到节点就可以偏移这个结构体拿到节点值，节点的值是一个pair，包含了key和value。

在gdb中打印以下map的内容：

size\_t ds\_map() {

std::map<std::string, int> imap;

imap["abc"] = 0xbbb;

return imap.size();

}

(gdb) p/x &imap

$7 = 0x7fffffffe370

(gdb) x/1a (char\*)&imap+24 # \_M\_left 真正的节点

0x7fffffffe388: 0x606040

(gdb) x/1xw 0x606040+32+8 # 偏移32字节是节点值的地址，再偏移8则是value的地址

0x606068: 0x00000bbb

(gdb) p \*(char\*\*)(0x606040+32) # 偏移32字节是string的地址

$8 = 0x606028 "abc"

或者很多时候没有必要这么装逼+蛋疼：

(gdb) p \*(char\*\*)(imap.\_M\_t.\_M\_impl.\_M\_header.\_M\_left+1)

$9 = 0x606028 "abc"

(gdb) x/1xw (char\*)(imap.\_M\_t.\_M\_impl.\_M\_header.\_M\_left+1)+8

0x606068: 0x00000bbb

*完*

原文地址： <http://codemacro.com/2014/12/03/gdb_stl/>  
written by [Kevin Lynx](http://codemacro.com/)  posted at [http://codemacro.com](http://codemacro.com/)