使用 GDB 调试 RB-tree 的几个问题

作者 ： 余祖波(livelylittlefish@gmail.com)
Blog: http://blog.csdn.net/livelylittlefish, http://www.abo321.org

Content

0. 引子

在 Linux 平台上开发 C/C++程序，就免不了要使用 GDB，当然你也可以使用 DDD(Data Display Debugger)，关于 DDD 可以参考 Appendix 和官方网站，非本文重点，不做讨论。本文就使用 GDB 调试的过程中碰到的几个问题进行简单介绍。

1. 第 1 个例子

我们先看一个使用 GDB 调试的例子。

```
(gdb)
operator new (__p=0x8246018)
    at /usr/lib/gcc/i386-redhat-linux/4.1.2/../../../../include/c++/4.1.2/new:94
94      inline void* operator new(std::size_t, void* __p) throw() { return __p; }
(gdb)
0x08049059 in __gnu_cxx::new_allocator<int>::construct (this=0xbfca1b9f, __p=0x8246018,
    __val=@0xbfca1cec)
    at /usr/lib/gcc/i386-redhat-linux/4.1.2/../../../../include/c++/4.1.2/ext/new_allocator.h:104
104         { ::new(__p) _Tp(__val); }
(gdb)
~allocator (this=0xbfca1b9f)
    at /usr/lib/gcc/i386-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/allocator.h:105
105         ~allocator() throw() { }
(gdb)
```

本文讨论的问题如下。

(1) at 提示前半部分代表什么？

表示当前调用的代码所在的库的路径，即某个 symbol 存在于该路径下的某个库中。该例子指出当前调用的函数均存在于/usr/lib/gcc/i386-redhat-linux/4.1.2 目录下的库文件中。例如，

```
# cd /usr/lib/gcc/i386-redhat-linux/4.1.2
# ls
crtbegin.o    finclude     libgcj.so        libgfortran.so  libstdc++.so
crtbeginS.o   include      libgcj.spec      libgij.so       libsupc++.a
crtbeginT.o   libgcc.a     libgcj-tools.so   libgomp.a      SYSCALLS.c.X
crtend.o      libgcc_eh.a  libgcov.a        libgomp.so
crtendS.o     libgcc_s.so  libgfortran.a     libgomp.spec
crtfastmath.o  libgcj_bc.so  libgfortranbegin.a  libstdc++.a

# objdump -x libsupc++.a | grep new
new_handler.o:     file format elf32-i386
```

```
rw-r--r-- 102/102   3564 Oct 21 23:42 2007 new_handler.o
  8 .text._ZSt15set_new_handlerPFvvE 0000001f  00000000  00000000  00000060  2**4
 13 .bss.__new_handler 00000004  00000000  00000000  00000120  2**2
00000000 l    d  .text._ZSt15set_new_handlerPFvvE      00000000 .text._ZSt15set_new_handlerPFvvE
00000000 l    d  .bss.__new_handler    00000000 .bss.__new_handler
00000000 g     F .text._ZSt15set_new_handlerPFvvE      0000001f _ZSt15set_new_handlerPFvvE
00000000 g     O .bss.__new_handler    00000004 __new_handler
RELOCATION RECORDS FOR [.text._ZSt15set_new_handlerPFvvE]:
00000014 R_386_GOT32      __new_handler
00000024 R_386_PC32       .text._ZSt15set_new_handlerPFvvE

new_op.o:     file format elf32-i386
rw-r--r-- 102/102   2432 Oct 21 23:42 2007 new_op.o
00000000        *UND*  00000000 __new_handler
0000002d R_386_GOT32      __new_handler

new_opnt.o:     file format elf32-i386
rw-r--r-- 102/102   2360 Oct 21 23:42 2007 new_opnt.o
00000000        *UND*  00000000 __new_handler
00000030 R_386_GOT32      __new_handler

new_opv.o:     file format elf32-i386
rw-r--r-- 102/102   2100 Oct 21 23:42 2007 new_opv.o

new_opvnt.o:     file format elf32-i386
rw-r--r-- 102/102   1608 Oct 21 23:42 2007 new_opvnt.o
 14 .text.__cxa_vec_new2 000000c5  00000000  00000000  00000630  2**4
 15 .text.__cxa_vec_new 00000053  00000000  00000000  00000700  2**4
 16 .text.__cxa_vec_new3 000000cc  00000000  00000000  00000760  2**4
00000000 l    d  .text.__cxa_vec_new2   00000000 .text.__cxa_vec_new2
00000000 l    d  .text.__cxa_vec_new    00000000 .text.__cxa_vec_new
00000000 l    d  .text.__cxa_vec_new3   00000000 .text.__cxa_vec_new3
00000000 g     F .text.__cxa_vec_new2   000000c5 __cxa_vec_new2
00000000 g     F .text.__cxa_vec_new    00000053 __cxa_vec_new
00000000 g     F .text.__cxa_vec_new3   000000cc __cxa_vec_new3
RELOCATION RECORDS FOR [.text.__cxa_vec_new2]:
```

```
RELOCATION RECORDS FOR [.text.__cxa_vec_new]:
00000049 R_386_PLT32      __cxa_vec_new2
RELOCATION RECORDS FOR [.text.__cxa_vec_new3]:
0000014c R_386_PC32       .text.__cxa_vec_new2
00000174 R_386_PC32       .text.__cxa_vec_new
00000194 R_386_PC32       .text.__cxa_vec_new3

# nm libsupc++.a | grep new
nm: eh_arm.o: no symbols
new_handler.o:
00000000 T _ZSt15set_new_handlerPFvvE
00000000 B __new_handler
new_op.o:
       U __new_handler
new_opnt.o:
       U __new_handler
new_opv.o:
new_opvnt.o:
00000000 T __cxa_vec_new
00000000 T __cxa_vec_new2
00000000 T __cxa_vec_new3
```

解析 new_handler.o 中的一个符号，

```
# c++filt _ZSt15set_new_handlerPFvvE
std::set_new_handler(void (*)())
```

该函数在 new_handler.cc 文件中的定义如下：

```
using std::new_handler;
new_handler __new_handler;

new_handler
```

```
std::set_new_handler (new_handler handler) throw()
{
  new_handler prev_handler = __new_handler;
  __new_handler = handler;
  return prev_handler;
}
```

其中，new_handler 在 new 文件中被定义，如下。

```
typedef void (*new_handler)();
```

另外，我们从 makefile 文件中也可以看到的确是以下 5 个关于"new"的.cc 文件被编译。

```
new_handler.cc \
new_op.cc \
new_opnt.cc \
new_opv.cc \
new_opvnt.cc \
```

(2) at 提示后半部分代表什么？

后半部分的提示表示当前代码执行到该文件，文件后面的数字表示行号。

../include 只是表示在 INCLUDE 路径下，并没有指明是哪个 include 目录，可能是/usr/include，可能是/usr/local/include，也可能是第三方的某个 include。

那么../include/c++/4.1.2/new 文件具体在哪里？

笔者实验的 Linux 平台上，文件如下。

```
/usr/include/c++/4.1.2/new
/usr/include/c++/4.1.2/ext/new_allocator.h
/usr/include/c++/4.1.2/bits/stl_tree.h
```

(3) 如果要阅读 gcc 的源代码，那么(2)中的文件在哪里？

以上 3 个文件对应 gcc 的源代码，如下。

```
.\libstdc++-v3\libsupc++\new
.\libstdc++-v3\include\ext\new_allocator.h
```

.\libstdc++-v3\include\bits\stl_tree.h (红黑树的实现就在这个文件中，该目录下以 stl_开头的文件就是 GCC 使用的 STL)

.表示 gcc 源代码目录，本文为 E:\opensource**gcc-4.1.2**。

2．第 2 个例子

再看一个使用 GDB 单步运行的例子。

```
(gdb) step
std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_M_insert (this=0xbfd33574, __x=0x0,
__p=0xbfd33578, __v=@0xbfd3358c)
    at /usr/lib/gcc/i386-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/stl_tree.h:817
817                                                 _S_key(__p)));
(gdb) n
819         _Link_type __z = _M_create_node(__v);
(gdb)
821         _Rb_tree_insert_and_rebalance(__insert_left, __z, __p,
(gdb) step
823         ++_M_impl._M_node_count;
(gdb) l
818
819         _Link_type __z = _M_create_node(__v);
820
821         _Rb_tree_insert_and_rebalance(__insert_left, __z, __p,
822                                       this->_M_impl._M_header);
823         ++_M_impl._M_node_count;
824         return iterator(__z);
825       }
826
(gdb)
```

该例子要讨论的问题如下。

stl_tree.h 第 821 行的_Rb_tree_insert_and_rebalance 调用为什么没有 step in(进入函数内部)？在 Linux 平台，该函数的实现在哪里？在 gcc 源代码中该函数在哪里？

我们先看该函数在 gcc 源代码中的位置。

(1) 在 gcc 源代码中该函数在哪里？

仔细查看 gcc 的源代码，我们发现，_Rb_tree_insert_and_rebalance 函数的定义并不在 str_tree.h 中，而是在 **tree.cc** 文件中，位于.\libstdc++-v3\src 目录下，该目录下还有其他的 pool_allocator.cc、mt_allocator.cc、list.cc 等，其对应的头文件分别为：

.\libstdc++-v3\include\ext\pool_allocator.h
.\libstdc++-v3\include\ext\mt_allocator.h
.\libstdc++-v3\include\std\std_list.h

对于 list.cc 需要解释一下。在 list.cc 文件中，直接#include <list>，但实际上，list 文件在 gcc 的源代码包里是不存在的。那么包含 list 实际上是包含谁？我们在 Linux 系统中查看，list 文件在/usr/include/c++/4.1.2 目录中，如下。

```
# pwd
/usr/include/c++/4.1.2
# ls
algorithm  clocale  ctime             functional      java    ostream    typeinfo
backward   cmath    cwchar            gcj             javax   queue      utility
bits       complex  cwctype           gnu             limits  set        valarray
bitset     csetjmp  cxxabi.h          i386-redhat-linux  list    sstream    vector
...
```

打开 **list** 文件，其内容如下，这个标准的头文件实际上就是包含 stl_list 的定义文件 stl_list.h 及其需要的其他头文件，如 allocator.h 等。

```
 60 #ifndef _GLIBCXX_LIST
 61 #define _GLIBCXX_LIST 1
 62
 63 #pragma GCC system_header
 64
```

```
65 #include <bits/functexcept.h>
66 #include <bits/stl_algobase.h>
67 #include <bits/allocator.h>
68 #include <bits/stl_construct.h>
69 #include <bits/stl_uninitialized.h>
70 #include <bits/stl_list.h>   //stl list 的定义文件
71
72 #ifndef _GLIBCXX_EXPORT_TEMPLATE
73 # include <bits/list.tcc>    //stl list 的某些模板成员函数的实现文件
74 #endif
75
76 #ifdef _GLIBCXX_DEBUG
77 # include <debug/list>
78 #endif
```

对应于 gcc 源代码，安装到 Linux 系统中的 list 文件实际上就是.\libstdc++-v3\include\std\std_list.h 文件，只是在安装到时候将其改名并拷贝到 /usr/include/c++/4.1.2 目录。同样的道理也适用于 STL 其他的 container，如 vector，set，deque，map，stack，queue 等。所以我们使用 STL 编写代码时，包含的文件是<list>，<map>，<vector>等，而非<std_list.h>，<std_map.h>，<std_vector.h>。

以下是 readme 文件中对.\libstdc++-v3\include\std 和.\libstdc++-v3\src 目录下的文件的解释。

```
   include/std
     Files meant to be found by #include <name> directives in
     standard-conforming user programs.

  src
    Files that are used in constructing the library, but are not
    installed.
```

问题扯远了，我们回到要讨论的问题。

(2) 为什么没有单步进入(step in)_Rb_tree_insert_and_rebalance 函数？

要想 step in 某个函数，一定要有相应的源代码的文件，很显然，该函数所在的文件 tree.cc 在 Linux 系统里并不存在。那如果我们希望单步进入该函数，可能需要重新编译 gcc 源代码并将所有.h 和.c/.cc/.cpp 的文件都安装到相应的目录下。例如在笔者的虚拟机里，调试 ACE 的程序，都可以进入 ACE 源代码内部进行。但从上述对.\libstdc++-v3\src 目录下的文件的解释可以看出，在 Linux 系统上并不安装这些.cc 文件，因此，要单步调试这些程序，貌似很困难。但也不是没有方法，我们稍后讨论。

那么，

(3) 该函数的实现在什么地方？即被编译进了哪个库？能否看到其信息？

摘_Rb_tree_increment 函数的代码，如下。

```
namespace std
{
  _Rb_tree_node_base*
  _Rb_tree_increment(_Rb_tree_node_base* __x)
  {
    if (__x->_M_right != 0)
      {
        __x = __x->_M_right;
        while (__x->_M_left != 0)
          __x = __x->_M_left;
      }
    else
      {
        _Rb_tree_node_base* __y = __x->_M_parent;
        while (__x == __y->_M_right)
          {
            __x = __y;
            __y = __y->_M_parent;
          }
        if (__x->_M_right != __y)
          __x = __y;
```

```
      }
    return __x;
  }
...
}
```

其参数_Rb_tree_node_base 定义如下，

```
  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;
    ...
  };
```

很显然，该函数是属于 std 空间的一个全局函数，不是类的成员函数，也不是模板函数，其参数_Rb_tree_node_base 也不是模板类。tree.cc 中的 9 个函数均是这样，且在每个函数内部，只是对指针的操作，不涉及定义一个对象(编译时需要分配逻辑地址，占虚拟空间)，或者使用 new/delete 操作符，因此所有这些函数(tree.cc 文件)在编译期间不需要确定一些数据类型，gcc 在编译时便将其编译到了静态库 libstdc++.a 中(tree.cc 的目标文件为 tree.o)，使用时再链接到目标可执行文件。

实际上，可以认为 tree.cc 文件(9 个函数)是一个独立的小模块，而且是一个独立性很强的模块，_Rb_tree_node_base 的改变(即 stl_tree.h 的改变)不会影响 tree.cc。像这样的模块，应该做成静态库供用户使用，从而减少编译依赖和时间。

可以通过如下命令查看静态库 libstdc++.a 中的关于 RBtree 的操作信息。

```
# cd /usr/lib/gcc/i386-redhat-linux/4.1.2  //以下要解析的库在该目录
# nm libstdc++.so                          //libstdc++.so 是动态库，其中没有 symbols
nm: libstdc++.so: no symbols
```

```
# nm libstdc++.a | grep _Rb_tree_
00000000 T _ZSt18_Rb_tree_decrementPKSt18_Rb_tree_node_base
00000000 T _ZSt18_Rb_tree_decrementPSt18_Rb_tree_node_base
00000000 T _ZSt18_Rb_tree_incrementPKSt18_Rb_tree_node_base
00000000 T _ZSt18_Rb_tree_incrementPSt18_Rb_tree_node_base
00000000 T _ZSt20_Rb_tree_black_countPKSt18_Rb_tree_node_baseS1_
00000000 T _ZSt20_Rb_tree_rotate_leftPSt18_Rb_tree_node_baseRS0_
00000000 T _ZSt21_Rb_tree_rotate_rightPSt18_Rb_tree_node_baseRS0_
00000000 T _ZSt28_Rb_tree_rebalance_for_erasePSt18_Rb_tree_node_baseRS_
00000000 T _ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_
nm: stubs.o: no symbols
nm: eh_arm.o: no symbols

# c++filt _ZSt18_Rb_tree_decrementPKSt18_Rb_tree_node_base
std::_Rb_tree_decrement(std::_Rb_tree_node_base const*)

# c++filt _ZSt18_Rb_tree_decrementPSt18_Rb_tree_node_base
std::_Rb_tree_decrement(std::_Rb_tree_node_base*)

# c++filt _ZSt28_Rb_tree_rebalance_for_erasePSt18_Rb_tree_node_baseRS_
std::_Rb_tree_rebalance_for_erase(std::_Rb_tree_node_base*, std::_Rb_tree_node_base&)
```

以上通过 c++filt 命令解析的符号便是.\libstdc++-v3\src\tree.cc 文件中的函数，其他的符号，读者可自行试验，也是 tree.cc 中的函数。

由以上信息可知，该函数被编译进了 libstdc++.a 这个静态库中。通过如下命令，可以得知 tree.cc 编译后的目标文件 tree.o 的确被打包进了静态库 libstdc++.a。

```
# cd /usr/lib/gcc/i386-redhat-linux/4.1.2
# mkdir test          //建立一个试验目录，试验完后容易删除
# cp libstdc++.a test/libstdc++.a
# cd test
# ls
libstdc++.a
```

```
# ar -x libstdc++.a    //extract file(s) from the archive
# ls                   //所有.o 文件便是从静态库 libstdc++.a 中释放出来的
...
atomicity.o         eh_call.o          istream-inst.o      pool_allocator.o
basic_file.o        eh_catch.o         istream.o           pure.o
bitmap_allocator.o  eh_exception.o     libstdc++.a         sstream-inst.o
c++locale.o         eh_globals.o       limits.o            stdexcept.o
codecvt_members.o   eh_personality.o   list.o              streambuf-inst.o
...
ctype.o             functexcept.o      misc-inst.o         tree.o
debug_list.o        globals_io.o       monetary_members.o  valarray-inst.o
debug.o             globals_locale.o   mt_allocator.o      vec.o
del_opnt.o          guard.o            new_handler.o       vterminate.o
...
eh_arm.o            ios.o              numeric_members.o
```

我们甚至可以直接使用从 libstdc++.a 中释放出来的 tree.o，在生成可执行文件时，直接连接之。不像.\libstdc++-v3\include\bits\list.tcc 和 vector.tcc 中的函数，他们不仅是 list 类和 vector 类的成员函数，也是模板函数。因此在 std_list.h 文件的第 73 行，需要包含 list.tcc 文件，如上 list 文件(Linux 平台上安装 std_list.h 后的文件)的内容所示。同样的道理也适用于 std_vector.h 文件。

我们也可以使用 nm 命令直接查看 tree.o 的相关信息。

```
# nm -A tree.o
tree.o:00000000 T _ZSt18_Rb_tree_decrementPKSt18_Rb_tree_node_base
tree.o:00000000 T _ZSt18_Rb_tree_decrementPSt18_Rb_tree_node_base
tree.o:00000000 T _ZSt18_Rb_tree_incrementPKSt18_Rb_tree_node_base
tree.o:00000000 T _ZSt18_Rb_tree_incrementPSt18_Rb_tree_node_base
tree.o:00000000 T _ZSt20_Rb_tree_black_countPKSt18_Rb_tree_node_baseS1_
tree.o:00000000 T _ZSt20_Rb_tree_rotate_leftPSt18_Rb_tree_node_baseRS0_
tree.o:00000000 T _ZSt21_Rb_tree_rotate_rightPSt18_Rb_tree_node_baseRS0_
tree.o:00000000 T _ZSt28_Rb_tree_rebalance_for_erasePSt18_Rb_tree_node_baseRS_
tree.o:00000000 T _ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_
```

```
tree.o:         U __gxx_personality_v0
```

以上符号经 demangle 后分别与以下命令的结果相对应。

```
# nm -C tree.o
00000000 T std::_Rb_tree_decrement(std::_Rb_tree_node_base const*)
00000000 T std::_Rb_tree_decrement(std::_Rb_tree_node_base*)
00000000 T std::_Rb_tree_increment(std::_Rb_tree_node_base const*)
00000000 T std::_Rb_tree_increment(std::_Rb_tree_node_base*)
00000000 T std::_Rb_tree_black_count(std::_Rb_tree_node_base const*, std::_Rb_tree_node_base const*)
00000000 T std::_Rb_tree_rotate_left(std::_Rb_tree_node_base*, std::_Rb_tree_node_base*&)
00000000 T std::_Rb_tree_rotate_right(std::_Rb_tree_node_base*, std::_Rb_tree_node_base*&)
00000000 T std::_Rb_tree_rebalance_for_erase(std::_Rb_tree_node_base*, std::_Rb_tree_node_base&)
00000000 T std::_Rb_tree_insert_and_rebalance(bool, std::_Rb_tree_node_base*, std::_Rb_tree_node_base*,
std::_Rb_tree_node_base&)
         U __gxx_personality_v0
```

也可以试试以下命令。对 nm 命令输出的解释可参考 nm 的 manual 页。

```
# nm -a tree.o
# nm -A tree.o
# nm -B tree.o
# nm -C tree.o (upper case)
# nm -g tree.o
# nm -l tree.o
# nm -n tree.o
# nm -o tree.o
# nm -p tree.o
# nm -P tree.o (upper case)
# nm -s tree.o
# nm -S tree.o (upper case)
# nm -u tree.o
```

其余均小写。

(4) 如何单步调试关于红黑树的操作，例如左旋、右旋、平衡等(tree.cc 中的函数)？

回答该问题，需要调试 STL 的源代码。摘录如下。

```
(gdb) step
std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_M_insert (this=0xbf849884, __x=0x0,
    __p=0xbf849888, __v=@0xbf84989c)
    at /usr/lib/gcc/i386-redhat-linux/4.1.2/../../../../include/c++/4.1.2/bits/stl_tree.h:817
817                                                 _S_key(__p)));
(gdb) n
819         _Link_type __z = _M_create_node(__v);
(gdb)
821         _Rb_tree_insert_and_rebalance(__insert_left, __z, __p,
```

(4.1) 利用 disassemble 命令找到_Rb_tree_insert_and_rebalance 的符号。

```
(gdb) disassemble
Dump of assembler code for function _ZNSt8_Rb_treeIiiSt9_IdentityIiESt4lessIiESaIiEE9_M_insertEPSt18_Rb_tree_node_baseS7_RKi:
...
0x08049245 <_ZNSt8_Rb_treeIiiSt9_IdentityIiESt4lessIiESaIiEE9_M_insertEPSt18_Rb_tree_node_baseS7_RKi+149>:
mov    %ecx,(%esp)
0x08049248 <_ZNSt8_Rb_treeIiiSt9_IdentityIiESt4lessIiESaIiEE9_M_insertEPSt18_Rb_tree_node_baseS7_RKi+152>:     call
0x80487b8 <_ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_@plt>
...
End of assembler dump.
```

可以利用 c++filt 命令验证该符号。

```
# c++filt _ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_
std::_Rb_tree_insert_and_rebalance(bool, std::_Rb_tree_node_base*, std::_Rb_tree_node_base*, std::_Rb_tree_node_base&)
```

(4.2) 利用 disassemble 命令查看该函数的起始地址

```
(gdb) disassemble _ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_
Dump of assembler code for function _ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_:
0x00c08750 <_ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_+0>:      push   %ebp
0x00c08751 <_ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_+1>:      mov    %esp,%ebp
0x00c08753 <_ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_+3>:      push   %edi
0x00c08754 <_ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_+4>:      push   %esi
0x00c08755 <_ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_+5>:      push   %ebx
...
```

由上可以看出，该函数的起始地址为 0x00c08750，很明显，只是一个虚拟地址，而非物理地址。

为什么要这么做？——因为函数被编译时均被 mangled，成为 symbol。
从如下命令可以看到在目标可执行文件中已经不存在_Rb_tree_insert_and_rebalance 这样的符号了。

```
(gdb) disassemble _Rb_tree_insert_and_rebalance
No symbol "_Rb_tree_insert_and_rebalance" in current context.
(gdb) disassemble std::_Rb_tree_insert_and_rebalance
No symbol "_Rb_tree_insert_and_rebalance" in namespace "std".
```

(4.3) 在该函数中设置断点

```
(gdb) b *0x00c08750
Breakpoint 2 at 0xc08750
(gdb) c
Continuing.

Breakpoint 2, 0x00c08750 in std::_Rb_tree_insert_and_rebalance () from /usr/lib/libstdc++.so.6
(gdb) stepi
0x00c08751 in std::_Rb_tree_insert_and_rebalance () from /usr/lib/libstdc++.so.6
(gdb)
```

```
0x00c08753 in std::_Rb_tree_insert_and_rebalance () from /usr/lib/libstdc++.so.6
(gdb)
0x00c08754 in std::_Rb_tree_insert_and_rebalance () from /usr/lib/libstdc++.so.6
(gdb)
0x00c08755 in std::_Rb_tree_insert_and_rebalance () from /usr/lib/libstdc++.so.6
(gdb)
0x00c08756 in std::_Rb_tree_insert_and_rebalance () from /usr/lib/libstdc++.so.6
(gdb)
```

由此，使用 stepi 命令便可调试汇编代码了。

从以上调试信息可以看出，在运行过程中，_Rb_tree_insert_and_rebalance 函数是从/usr/lib/libstdc++.so.6 这个动态库中加载的。可以通过以下方式验证该函数在 libstdc++.so.6 中的位置。

```
# gdb /usr/lib/libstdc++.so.6.0.8
...
(gdb) disassemble _ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_
Dump of assembler code for function _ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_:
0x00c08750 <_ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_+0>:      push   %ebp
0x00c08751 <_ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_+1>:      mov    %esp,%ebp
0x00c08753 <_ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_+3>:      push   %edi
...
```

题外话。

还可以通过如下方式查看该函数的汇编代码，可以看到，在 tree.o 中只是一个相对地址，没有虚拟地址。

```
# gdb ./tree.o
...
(gdb) disassemble 0
Dump of assembler code for function _ZSt18_Rb_tree_incrementPSt18_Rb_tree_node_base:
```

```
0x00000000 <_ZSt18_Rb_tree_incrementPSt18_Rb_tree_node_base+0>: push   %ebp
0x00000001 <_ZSt18_Rb_tree_incrementPSt18_Rb_tree_node_base+1>: mov    %esp,%ebp
0x00000003 <_ZSt18_Rb_tree_incrementPSt18_Rb_tree_node_base+3>: mov    0x8(%ebp),%ecx
0x00000006 <_ZSt18_Rb_tree_incrementPSt18_Rb_tree_node_base+6>: mov    0xc(%ecx),%edx
0x00000009 <_ZSt18_Rb_tree_incrementPSt18_Rb_tree_node_base+9>: test   %edx,%edx
...
```

```
# objdump -d tree.o
...
Disassembly of section .text._ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_:

00000000 <_ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_>:
   0:   55                      push   %ebp
   1:   89 e5                    mov    %esp,%ebp
   3:   57                      push   %edi
   4:   56                      push   %esi
   5:   53                      push   %ebx
   6:   83 ec 0c                 sub    $0xc,%esp
   9:   8b 45 14                 mov    0x14(%ebp),%eax
   c:   8b 7d 0c                 mov    0xc(%ebp),%edi
   f:   8b 5d 10                 mov    0x10(%ebp),%ebx
...
```

也可以试试以下命令。对 objdump 命令输出的解释可参考其的 manual 页。

```
# objdump -x tree.o
# objdump -d tree.o
# objdump -h tree.o
# objdump -s tree.o
# objdump -S tree.o (upper case)
```

以下问题，将另文讨论。
静态库和动态库的区别与使用
动态库如何加载？加载时虚拟地址空间如何重定位？

3. 小结

(1) 本文使用的命令
nm
objdump
c++filt

(2) STL 源代码位置
GCC 源代码中 STL 位置：.\libstdc++-v3\include\bits (.表示 gcc 源代码目录，本文为 E:\opensource\gcc-4.1.2)
Linux 系统中 STL 位置：/usr/include/c++/4.1.2/bits

Reference

.\libstdc++-v3\readme
man nm (nm 的 manual 页)
nm -h (nm help)

Appendix: About DDD

GNU DDD is a graphical front-end for command-line debuggers such as GDB, DBX, WDB, Ladebug, JDB, XDB, the Perl debugger, the bash debugger bashdb, the GNU Make debugger remake, or the Python debugger pydb. Besides ``usual'' front-end features such as viewing source texts, DDD has become famous through its interactive graphical data display, where data structures are displayed as graphs.
是 Linux 平台上的一种可视化调试工具，其最大的特点是数据结构以图形方式显示，非常直观。
http://www.gnu.org/software/ddd