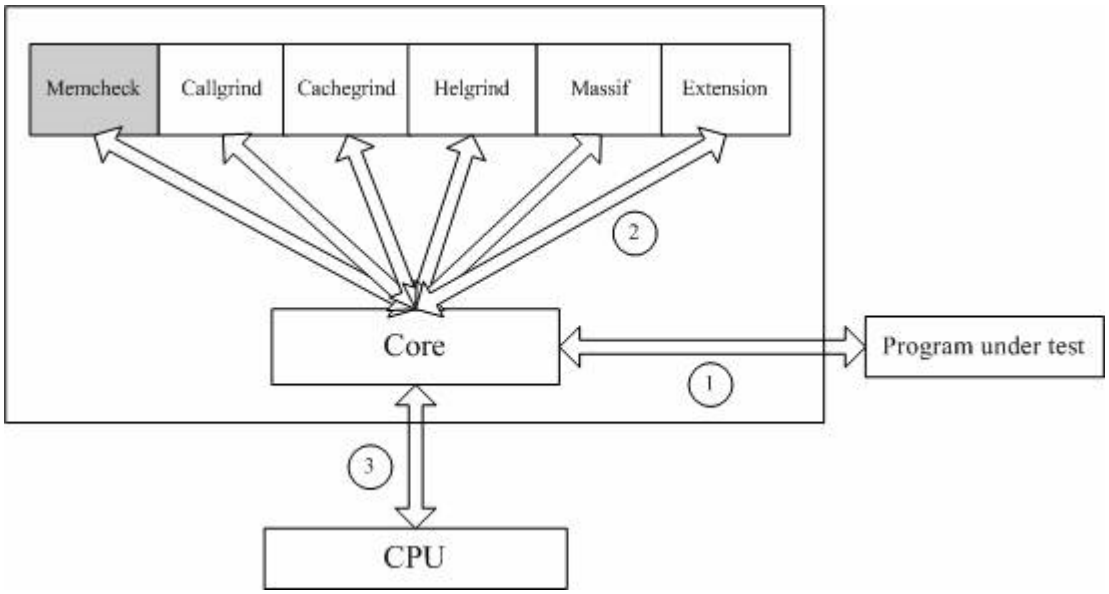
内存检测概述

体系结构

Valgrind 是一套 Linux 下，开放源代码（GPL V2）的仿真调试工具的集合。Valgrind 由内核（core）以及基于内核的其他调试工具组成。内核类似于一个框架（framework），它模拟了一个 CPU 环境，并提供服务给其他工具；而其他工具则类似于插件 (plug-in)，利用内核提供的服务完成各种特定的内存调试任务。Valgrind 的体系结构如下图所示：

图 1 Valgrind 体系结构



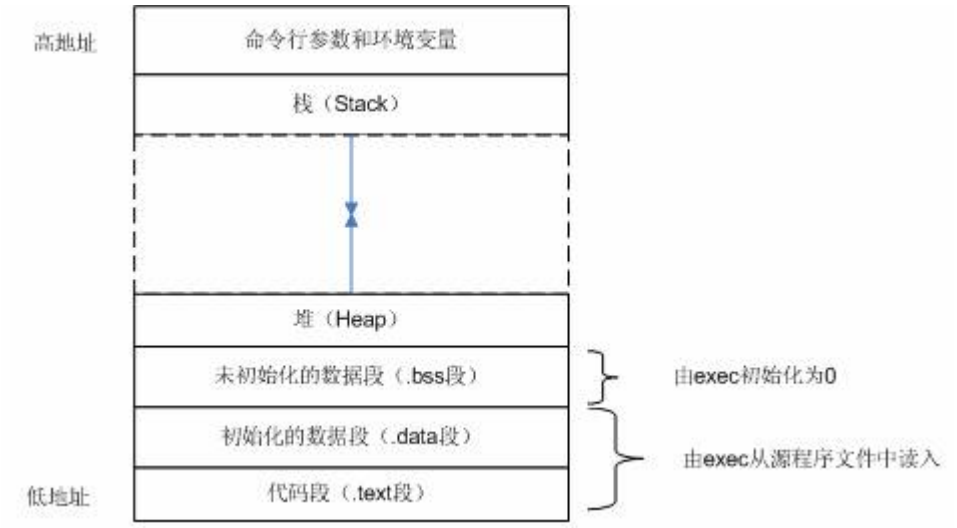
Valgrind 包括如下一些工具：

1. **Memcheck。**这是 valgrind 应用最广泛的工具，一个重量级的内存检查器，能够发现开发中绝大多数内存错误使用情况，比如：使用未初始化的内存，使用已经释放了的内存，内存访问越界等。这也是本文将重点介绍的部分。
2. **Callgrind**。它主要用来检查程序中函数调用过程中出现的问题。
3. **Cachegrind**。它主要用来检查程序中缓存使用出现的问题。
4. **Helgrind**。它主要用来检查多线程程序中出现的竞争问题。
5. **Massif**。它主要用来检查程序中堆栈使用中出现的问题。
6. **Extension。**可以利用 core 提供的功能，自己编写特定的内存调试工具。

Linux 程序内存空间布局

要发现 Linux 下的内存问题，首先一定要知道在 Linux 下，内存是如何被分配的？下图展示了一个典型的 Linux C 程序内存空间布局：

图 2： 典型内存空间布局



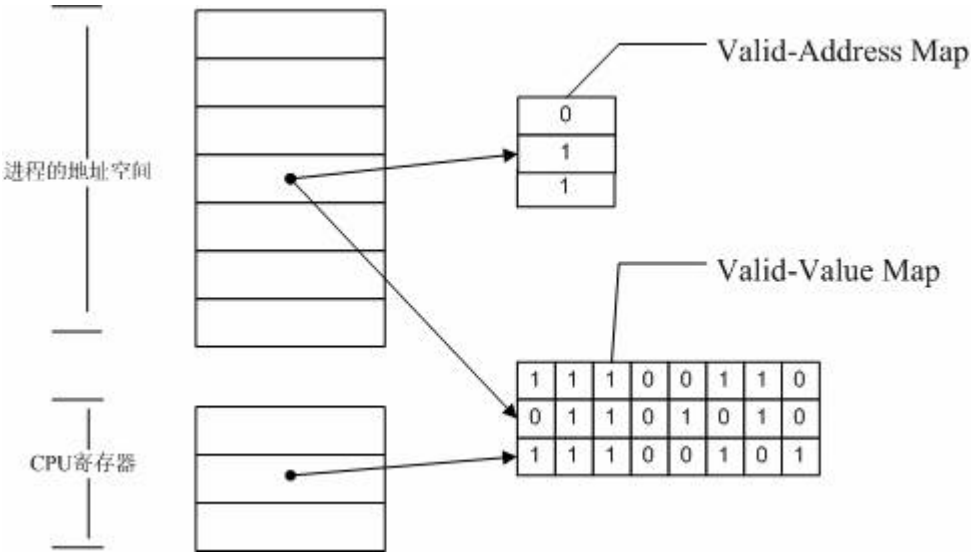
一个典型的 Linux C 程序内存空间由如下几部分组成：

- **代码段（.text）。**这里存放的是 CPU 要执行的指令。代码段是可共享的，相同的代码在内存中只会有一个拷贝，同时这个段是只读的，防止程序由于错误而修改自身的指令。
- **初始化数据段（.data）。**这里存放的是程序中需要明确赋初始值的变量，例如位于所有函数之外的全局变量：int val=100。需要强调的是，以上两段都是位于程序的可执行文件中，内核在调用 exec 函数启动该程序时从源程序文件中读入。
- **未初始化数据段（.bss）。**位于这一段中的数据，内核在执行该程序前，将其初始化为 0 或者 null。例如出现在任何函数之外的全局变量：int sum;
- **堆（Heap）。**这个段用于在程序中进行动态内存申请，例如经常用到的 malloc，new 系列函数就是从这个段中申请内存。
- **栈（Stack）。**函数中的局部变量以及在函数调用过程中产生的临时变量都保存在此段中。

内存检查原理

Memcheck 检测内存问题的原理如下图所示：

图 3 内存检查原理



Memcheck 能够检测出内存问题，关键在于其建立了两个全局表。

1. **Valid-Value 表：**

对于进程的整个地址空间中的每一个字节(byte)，都有与之对应的 8 个 bits；对于 CPU 的每个寄存器，也有一个与之对应的 bit 向量。这些 bits 负责记录该字节或者寄存器值是否具有有效的、已初始化的值。

1. **Valid-Address 表**

对于进程整个地址空间中的每一个字节(byte)，还有与之对应的 1 个 bit，负责记录该地址是否能够被读写。

检测原理：

- 当要读写内存中某个字节时，首先检查这个字节对应的 A bit。如果该 A bit 显示该位置是无效位置，memcheck 则报告读写错误。
- 内核（core）类似于一个虚拟的 CPU 环境，这样当内存中的某个字节被加载到真实的 CPU 中时，该字节对应的 V bit 也被加载到虚拟的 CPU 环境中。一旦寄存器中的值，被用来产生内存地址，或者该值能够影响程序输出，则 memcheck 会检查对应的 V bits，如果该值尚未初始化，则会报告使用未初始化内存错误。

回页首

Valgrind 使用

第一步：准备好程序

为了使 valgrind 发现的错误更精确，如能够定位到源代码行，建议在编译时加上-g 参数，编译优化选项请选择 O0，虽然这会降低程序的执行效率。

这里用到的示例程序文件名为：sample.c（如下所示）,选用的编译器为 gcc。

生成可执行程序 **gcc –g –O0 sample.c –o sample**

清单 1

```
1  #include <stdlib.h>
2
3  void fun()
4  {
5          int *p=(int*)malloc(10*sizeof(int));
6          p[10]=0;
7  }
8
9  int main(int argc, char* argv[])
10 {
11         fun();
12         return 0;
13 }
```

第二步：在 valgrind 下，运行可执行程序。

利用 valgrind 调试内存问题，不需要重新编译源程序，它的输入就是二进制的可执行程序。调用 Valgrind 的通用格式是：**valgrind [valgrind-options] your-prog [your-prog-options]**

Valgrind 的参数分为两类，一类是 core 的参数，它对所有的工具都适用；另外一类就是具体某个工具如 memcheck 的参数。Valgrind 默认的工具就是 memcheck，也可以通过“--tool=*tool name*”指定其他的工具。Valgrind 提供了大量的参数满足你特定的调试需求，具体可参考其用户手册。

这个例子将使用 memcheck，于是可以输入命令入下：**valgrind** <*Path*>**/sample.**

第三步：分析 valgrind 的输出信息。

以下是运行上述命令后的输出。

清单 2

```
cdlyangj:/home/workspace/intro # valgrind ./sample
==32372== Memcheck, a memory error detector.
==32372== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==32372== Using LibVEX rev 1854, a library for dynamic binary translation.
==32372== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==32372== Using valgrind-3.3.1, a dynamic binary instrumentation framework.
==32372== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==32372== For more details, rerun with: -v
==32372==
==32372== Invalid write of size 4
==32372==    at 0x80483CF: fun (sample.c:6)
==32372==    by 0x80483EC: main (sample.c:11)
==32372==  Address 0x416f050 is 0 bytes after a block of size 40 alloc'd
==32372==    at 0x4023888: malloc (vg_replace_malloc.c:207)
==32372==    by 0x80483C5: fun (sample.c:5)
==32372==    by 0x80483EC: main (sample.c:11)
==32372==
==32372== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 3 from 1)
==32372== malloc/free: in use at exit: 40 bytes in 1 blocks.
==32372== malloc/free: 1 allocs, 0 frees, 40 bytes allocated.
==32372== For counts of detected errors, rerun with: -v
==32372== searching for pointers to 1 not-freed blocks.
==32372== checked 56,780 bytes.
==32372==
==32372== LEAK SUMMARY:
==32372==    definitely lost: 40 bytes in 1 blocks.
==32372==      possibly lost: 0 bytes in 0 blocks.
==32372==    still reachable: 0 bytes in 0 blocks.
==32372==         suppressed: 0 bytes in 0 blocks.
==32372== Rerun with --leak-check=full to see details of leaked memory.
```

- 左边显示类似行号的数字（32372）表示的是 Process ID。
- 最上面的红色方框表示的是 valgrind 的版本信息。
- 中间的红色方框表示 valgrind 通过运行被测试程序，发现的内存问题。通过阅读这些信息，可以发现：

    1. 这是一个对内存的非法写操作，非法写操作的内存是 4 bytes。
    2. 发生错误时的函数堆栈，以及具体的源代码行号。
    3. 非法写操作的具体地址空间。

- 最下面的红色方框是对发现的内存问题和内存泄露问题的总结。内存泄露的大小（40 bytes）也能够被检测出来。

示例程序显然有两个问题，一是 fun 函数中动态申请的堆内存没有释放；二是对堆内存的访问越界。这两个问题均被 valgrind 发现。

回页首

利用 Memcheck 发现常见的内存问题

在 Linux 平台开发应用程序时，最常遇见的问题就是错误的使用内存，我们总结了常见了内存错误使用情况，并说明了如何用 valgrind 将其检测出来。

使用未初始化的内存

问题分析：

对于位于程序中不同段的变量，其初始值是不同的，全局变量和静态变量初始值为 0，而局部变量和动态申请的变量，其初始值为随机值。如果程序使用了为随机值的变量，那么程序的行为就变得不可预期。

下面的程序就是一种常见的，使用了未初始化的变量的情况。数组 a 是局部变量，其初始值为随机值，而在初始化时并没有给其所有数组成员初始化，如此在接下来使用这个数组时就潜在有内存问题。

清单 3

```
1  #include <stdio.h>
2
3  int main ( void )
4  {
5      int a[5];
6      int i, s;
7      a[0] = a[1] = a[3] = a[4] = 0;
8      s = 0;
9      for (i = 0; i < 5; i++)
10         s += a[i];
11     if (s == 377)
12         printf("sum is %d\n", s);
13     return 0;
14 }
```

结果分析：

假设这个文件名为：**badloop.c**，生成的可执行程序为 **badloop**。用 memcheck 对其进行测试，输出如下。

清单 4

```
Conditional jump or move depends on uninitialised value(s)
   at 0x8048409: main (badloop.c:11)

ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 3 from 1)
malloc/free: in use at exit: 0 bytes in 0 blocks.
malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
For counts of detected errors, rerun with: -v
All heap blocks were freed -- no leaks are possible.
```

输出结果显示，在该程序第 11 行中，程序的跳转依赖于一个未初始化的变量。准确的发现了上述程序中存在的问题。

内存读写越界

问题分析：

这种情况是指：访问了你不应该/没有权限访问的内存地址空间，比如访问数组时越界；对动态内存访问时超出了申请的内存大小范围。下面的程序就是一个典型的数组越界问题。pt 是一个局部数组变量，其大小为 4，p 初始指向 pt 数组的起始地址，但在对 p 循环叠加后，p 超出了 pt 数组的范围，如果此时再对 p 进行写操作，那么后果将不可预期。

清单 5

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6          int len=4;
7          int* pt=(int*)malloc(len*sizeof(int));
8          int* p=pt;
9
10         for(int i=0;i<len;i++)
11         {
12                 p++;
13         }
14
15         *p=5;
16         printf("the value of p equal:%d",*p);
17         return 0;
18 }
```

结果分析：

假设这个文件名为 badacc.cpp，生成的可执行程序为 badacc，用 memcheck 对其进行测试，输出如下。

清单 6

```
==5064== Invalid write of size 4
==5064==    at 0x804850F: main (badacc.cpp:15)
==5064==  Address 0x4286038 is 0 bytes after a block of size 16 alloc'd
==5064==    at 0x4023888: malloc (vg_replace_malloc.c:207)
==5064==    by 0x80484E9: main (badacc.cpp:7)
==5064==
==5064== Invalid read of size 4
==5064==    at 0x8048518: main (badacc.cpp:16)
==5064==  Address 0x4286038 is 0 bytes after a block of size 16 alloc'd
==5064==    at 0x4023888: malloc (vg_replace_malloc.c:207)
==5064==    by 0x80484E9: main (badacc.cpp:7)
the value of p equal:5==5064==
==5064== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 3 from 1)
==5064== malloc/free: in use at exit: 16 bytes in 1 blocks.
==5064== malloc/free: 1 allocs, 0 frees, 16 bytes allocated.
==5064== For counts of detected errors, rerun with: -v
==5064== searching for pointers to 1 not-freed blocks.
==5064== checked 110,028 bytes.
```

输出结果显示，在该程序的第 15 行，进行了非法的写操作；在第 16 行，进行了非法读操作。准确地发现了上述问题。

内存覆盖

问题分析：

C 语言的强大和可怕之处在于其可以直接操作内存，C 标准库中提供了大量这样的函数，比如 strcpy, strncpy, memcpy, strcat 等，这些函数有一个共同的特点就是需要设置源地址 (src)，和目标地址(dst)，src 和 dst 指向的地址不能发生重叠，否则结果将不可预期。

下面就是一个 src 和 dst 发生重叠的例子。在 15 与 17 行中，src 和 dst 所指向的地址相差 20，但指定的拷贝长度却是 21，这样就会把之前的拷贝值覆盖。第 24 行程序类似，src(x+20) 与 dst(x) 所指向的地址相差 20，但 dst 的长度却为 21，这样也会发生内存覆盖。

清单 7

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[])
6  {
7          char x[50];
8          int i;
9          for(i=0;i<50;i++)
10         {
11                 x[i]=i+1;
12         }
13
14         strncpy(x+20, x, 20);     // ok
15         strncpy(x+20, x, 21);     // overlap
16         strncpy(x, x+20, 20);     // ok
17         strncpy(x, x+20, 21);     // overlap
18
19         x[39]='\0';
20         strcpy(x,x+20);           //ok
21
22         x[39]=39;
23         x[40]='\0';
24         strcpy(x,x+20);           //overlap
25
26         return 0;
27 }
```

结果分析：

假设这个文件名为 badlap.cpp，生成的可执行程序为 badlap，用 memcheck 对其进行测试，输出如下。

清单 8

```
==26612== Source and destination overlap in strncpy(0xBEBCC237, 0xBEBCC223, 21)
==26612==    at 0x4025C44: strncpy (mc_replace_strmem.c:291)
==26612==    by 0x804844E: main (badlap.c:15)
==26612==
==26612== Source and destination overlap in strncpy(0xBEBCC223, 0xBEBCC237, 21)
==26612==    at 0x4025C44: strncpy (mc_replace_strmem.c:291)
==26612==    by 0x8048488: main (badlap.c:17)
==26612==
==26612== Source and destination overlap in strcpy(0xBEBCC20E, 0xBEBCC222)
==26612==    at 0x4025D2E: strcpy (mc_replace_strmem.c:268)
==26612==    by 0x80484BE: main (badlap.c:24)
```

输出结果显示上述程序中第 15，17，24 行，源地址和目标地址设置出现重叠。准确的发现了上述问题。

动态内存管理错误

问题分析：

常见的内存分配方式分三种：静态存储，栈上分配，堆上分配。全局变量属于静态存储，它们是在编译时就被分配了存储空间，函数内的局部变量属于栈上分配，而最灵活的内存使用方式当属堆上分配，也叫做内存动态分配了。常用的内存动态分配函数包括：malloc, alloc, realloc, new 等，动态释放函数包括 free, delete。

一旦成功申请了动态内存，我们就需要自己对其进行内存管理，而这又是最容易犯错误的。下面的一段程序，就包括了内存动态管理中常见的错误。

清单 9

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6      int i;
7      char* p = (char*)malloc(10);
8      char* pt=p;
9
10     for (i = 0; i < 10; i++)
11     {
12         p[i] = 'z';
13     }
14     delete p;
15
16     pt[1] = 'x';
17     free(pt);
18     return 0;
19 }
```

常见的内存动态管理错误包括：

- 申请和释放不一致

由于 C++ 兼容 C，而 C 与 C++ 的内存申请和释放函数是不同的，因此在 C++ 程序中，就有两套动态内存管理函数。一条不变的规则就是采用 C 方式申请的内存就用 C 方式释放；用 C++ 方式申请的内存，用 C++ 方式释放。也就是用 malloc/alloc/realloc 方式申请的内存，用 free 释放；用 new 方式申请的内存用 delete 释放。在上述程序中，用 malloc 方式申请了内存却用 delete 来释放，虽然这在很多情况下不会有问题，但这绝对是潜在的问题。

- 申请和释放不匹配

申请了多少内存，在使用完成后就要释放多少。如果没有释放，或者少释放了就是内存泄露；多释放了也会产生问题。上述程序中，指针 p 和 pt 指向的是同一块内存，却被先后释放两次。

- 释放后仍然读写

本质上说，系统会在堆上维护一个动态内存链表，如果被释放，就意味着该块内存可以继续被分配给其他部分，如果内存被释放后再访问，就可能覆盖其他部分的信息，这是一种严重的错误，上述程序第 16 行中就在释放后仍然写这块内存。

结果分析：

假设这个文件名为 badmac.cpp，生成的可执行程序为 badmac，用 memcheck 对其进行测试，输出如下。

清单 10

```
Mismatched free() / delete / delete []
   at 0x40230BC: operator delete(void*) (vg_replace_malloc.c:342)
   by 0x8048530: main (badmac.cpp:14)
 Address 0x4286028 is 0 bytes inside a block of size 10 alloc'd
   at 0x4023888: malloc (vg_replace_malloc.c:207)
   by 0x8048500: main (badmac.cpp:7)

Invalid write of size 1
   at 0x8048537: main (badmac.cpp:16)
 Address 0x4286029 is 1 bytes inside a block of size 10 free'd
   at 0x40230BC: operator delete(void*) (vg_replace_malloc.c:342)
   by 0x8048530: main (badmac.cpp:14)

Invalid free() / delete / delete[]
   at 0x402342C: free (vg_replace_malloc.c:323)
   by 0x8048544: main (badmac.cpp:17)
 Address 0x4286028 is 0 bytes inside a block of size 10 free'd
   at 0x40230BC: operator delete(void*) (vg_replace_malloc.c:342)
   by 0x8048530: main (badmac.cpp:14)
```

输出结果显示，第 14 行分配和释放函数不一致；第 16 行发生非法写操作，也就是往释放后的内存地址写值；第 17 行释放内存函数无效。准确地发现了上述三个问题。

内存泄露

问题描述：

内存泄露（Memory leak）指的是，在程序中动态申请的内存，在使用完后既没有释放，又无法被程序的其他部分访问。内存泄露是在开发大型程序中最令人

头疼的问题，以至于有人说，内存泄露是无法避免的。其实不然，防止内存泄露要从良好的编程习惯做起，另外重要的一点就是要加强单元测试（Unit Test），而 memcheck 就是这样一款优秀的工具。

下面是一个比较典型的内存泄露案例。main 函数调用了 mk 函数生成树结点，可是在调用完成之后，却没有相应的函数：nodefr 释放内存，这样内存中的这个树结构就无法被其他部分访问，造成了内存泄露。

在一个单独的函数中，每个人的内存泄露意识都是比较强的。但很多情况下，我们都会对 malloc/free 或 new/delete 做一些包装，以符合我们特定的需要，无法做到在一个函数中既使用又释放。这个例子也说明了内存泄露最容易发生的地方：即两个部分的接口部分，一个函数申请内存，一个函数释放内存。并且这些函数由不同的人开发、使用，这样造成内存泄露的可能性就比较大了。这需要养成良好的单元测试习惯，将内存泄露消灭在初始阶段。

清单 11

```
1  #ifndef _BADLEAK_
2  #define  _BADLEAK_
3
4  typedef struct _node {
5          struct _node *l;
6          struct _node *r;
7          char v;
8  }node;
9
10 node *mk( node *l,  node *r, char val);
11 void nodefr(node * n);
12
13 #endif
```

清单 11.2

```c
#include <stdlib.h>
#include "tree.h"

node *mk( node *l,  node *r, char val)
{
        node *f =(node *)malloc(sizeof(*f));
        f->l = l;
        f->r = r;
        f->v=val;
        return f;
}

void nodefr(node * n)
{
        if(n)
        {
                nodefr(n->l);
                nodefr(n->r);
                free(n);
        }
}
```

清单 11.3

```c
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

int main()
{
         node* tree1,*tree2,*tree3;

         tree1=mk(mk(mk(0,0,'3'),0,'2'),0,'1');

         tree2=mk(0,mk(0,mk(0,0,'6'),'5'),'4');

         tree3=mk(mk(tree1,tree2,'8'),0,'7');

         return 0;
}
```

结果分析：

假设上述文件名位 tree.h, tree.cpp, badleak.cpp，生成的可执行程序为 badleak，用 memcheck 对其进行测试，输出如下。

清单 12

```
96 (12 direct, 84 indirect) bytes in 1 blocks are definitely lost in loss record 2 of
   at 0x4023888: malloc (vg_replace_malloc.c:207)
   by 0x804850F: mk(_node*, _node*, char) (tree.cpp:6)
   by 0x8048614: main (badleak.cpp:13)

LEAK SUMMARY:
   definitely lost: 12 bytes in 1 blocks.
   indirectly lost: 84 bytes in 7 blocks.
     possibly lost: 0 bytes in 0 blocks.
   still reachable: 0 bytes in 0 blocks.
        suppressed: 0 bytes in 0 blocks.
```

该示例程序是生成一棵树的过程，每个树节点的大小为 12（考虑内存对齐），共 8 个节点。从上述输出可以看出，所有的内存泄露都被发现。Memcheck 将内存泄露分为两种，一种是可能的内存泄露（Possibly lost），另外一种是确定的内存泄露（Definitely lost）。Possibly lost 是指仍然存在某个指针能够访问某块内存，但该指针指向的已经不是该内存首地址。Definitely lost 是指已经不能够访问这块内存。而 Definitely lost 又分为两种：直接的（direct）和间接的（indirect）。直接和间接的区别就是，直接是没有任何指针指向该内存，间接是指指向该内存的指针都位于内存泄露处。在上述的例子中，根节点是 directly lost，而其他节点是 indirectly lost。