# *Valgrind Tutorial*

## Why use Valgrind Debugger

Valgrind is a memory mismanagement detector. It shows you memory leaks, de-allocation errors, etc. Actually, Valgrind is a wrapper around a collection of tools that do many other things (e.g., cache profiling); however, here we focus on the default tool, Memcheck. Memcheck can detect:

- Use of uninitialized memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks -- where pointers to malloc'd blocks are lost forever
- Mismatched use of malloc/new/new [] vs. free/delete/delete []
- Overlapping src and dst pointers in memcpy() and related functions
- Some misuses of the POSIX pthreads API

## How to use Valgrind Debugger

To use this on our example program, test.c, try

```
#include <stdio.h>
#include<malloc.h>
int main()
{
        char *p;

        // Allocation #1 of 19 bytes
        p = (char *) malloc(19);
        free(p);

        // Allocation #2 of 12 bytes
        p = (char *) malloc(12);
        free(p);

        // Allocation #3 of 16 bytes
        p = (char *) malloc(16);
        free(p);
        return 0;
}
```

Compile the code with -g option (Where –g option enable built in debugging information)

```
gcc -g test.c –o test
```

This creates an executable named test. To check for memory leaks during the execution of test, try...

```
valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-callers=20 --track-fds=yes ./test
```

## Example of Detecting Memory Leak

This outputs a report to the terminal like

```
==9704== Memcheck, a memory error detector for x86-linux.
==9704== Copyright (C) 2002-2004, and GNU GPL'd, by Julian Seward et al.
==9704== Using valgrind-2.2.0, a program supervision framework for x86-linux.
==9704== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward et al.
==9704== for more details, rerun with: -v
==9704==
==9704==
==9704== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
==9704== malloc/free: in use at exit: 35 bytes in 2 blocks.
==9704== malloc/free: 3 allocs, 1 frees, 47 bytes allocated.
==9704== for counts of detected errors, rerun with: -v
==9704== searching for pointers to 2 not-freed blocks.
==9704== checked 1420940 bytes.
==9704==
==9704== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
==9704== at 0x1B903D38: malloc (vg_replace_malloc.c:131)
==9704== by 0x80483BF: main (test.c:15)
==9704==
==9704==
==9704== 19 bytes in 1 blocks are definitely lost in loss record 2 of 2
==9704== at 0x1B903D38: malloc (vg_replace_malloc.c:131)
==9704== by 0x8048391: main (test.c:8)
==9704==
==9704== LEAK SUMMARY:
==9704== definitely lost: 35 bytes in 2 blocks.
==9704== possibly lost:   0 bytes in 0 blocks.
==9704== still reachable: 0 bytes in 0 blocks.
==9704== suppressed: 0 bytes in 0 blocks.
```

Let's look at the code to see what happened. Allocation #1 (19 byte leak) is lost because p is pointed elsewhere before the memory from Allocation #1 is free'd.
To help us track it down, Valgrind gives us a stack trace showing where the bytes were allocated. In the 19 byte leak entry, the bytes were allocate in test.c, line 8. Allocation #2 (12 byte leak) doesn't show up in the list because it is free'd. Allocation #3 shows up in the list even though

there is still a reference to it (p) at program termination. This is still a memory leak! Again, Valgrind tells us where to look for the allocation (test.c line 15).

## Important option summary

Gnome applications tend to have deep stack traces, much of which comes from the glib main loop. So it tends to be important to specify a large value for --num-callers (say, 40 or so, just to be safe). Also, if checking for leaks, be sure to specify --leak-resolution=high.

### *Common valgrind options*

**Option:** --num-callers=number
**Purpose:** Determines the number of function calls (i.e. depth of stack trace) to display as part of showing where an error occurs within a program. The default is a measly 4.

**Option:** --leak-check=yes
**Purpose:** Enabling leak checking has valgrind search for memory leaks (i.e. allocated memory that has not been released) when the program finished.

**Option:** --leak-resolution=high
**Purpose:** An option that should be used when doing leak checking since all other options result in confusing reports.

**Option:** --show-reachable=yes
**Purpose:** An option that makes leak checking more helpful by requesting that Valgrind report whether pointers to unreleased blocks are still held by the program.

**Option:** -v
**Purpose:** Run in more verbose mode.

**Option:** -fno-inline
**Purpose:** An option for C++ programs which makes it easier to see the function-call chain.

**Option:** --gen-suppressions=yes
**Purpose:** A simple way to generate a suppressions file in order to facilitate ignoring certain errors in future runs of the same code.

**Option:** --skin=addrcheck
**Purpose:** (Note that the name of this option has become --tool and has become mandatory for the development release). This selects the specific tool of valgrind that will run. Memcheck (the only tool covered here) is the default.

**Option:** --logfile=file-basename
**Purpose:** Record all errors and warnings to file-basename.pidpid

Valgrind can detect many kinds of errors. Here's an explanation of the various error messages.

## Explanation of error messages from Memcheck

Despite considerable sophistication under the hood, Memcheck can only really detect two kinds of errors, use of illegal addresses, and use of undefined values. Nevertheless, this is enough to help you discover all sorts of memory-management nasties in your code. This section presents a quick summary of what error messages mean.

### *Illegal read / Illegal write errors*

For example:

```
Invalid read of size 4
   at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40F6B804: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40B07FF4: read_png_image__FP8QImageIO (kernel/qpngio.cpp:326)
   by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)
   Address 0xBFFFF0E0 is not stack'd, malloc'd or free'd
```

This happens when your program reads or writes memory at a place which Memcheck reckons it shouldn't. In this example, the program did a 4-byte read at address 0xBFFFF0E0, somewhere within the system-supplied library libpng.so.2.1.0.9, which was called from somewhere else in the same library, called from line 326 of qpngio.cpp, and so on.

Memcheck tries to establish what the illegal address might relate to, since that's often useful. So, if it points into a block of memory which has already been freed, you'll be informed of this, and also where the block was free'd at. Likewise, if it should turn out to be just off the end of a malloc'd block, a common result of off-by-one-errors in array subscripting, you'll be informed of this fact, and also where the block was malloc'd.

In this example, Memcheck can't identify the address. Actually the address is on the stack, but, for some reason, this is not a valid stack address -- it is below the stack pointer, %esp, and that isn't allowed. In this particular case it's probably caused by gcc generating invalid code, a known bug in various flavors of gcc.

Note that Memcheck only tells you that your program is about to access memory at an illegal address. It can't stop the access from happening. So, if your program makes an access which normally would result in a segmentation fault, you program will still suffer the same fate -- but you will get a message from Memcheck immediately prior to this. In this particular example, reading junk on the stack is non-fatal, and the program stays alive.

### *Use of uninitialized values*

For example:

```
Conditional jump or move depends on uninitialized value(s)
   at 0x402DFA94: _IO_vfprintf (_itoa.h:49)
   by 0x402E8476: _IO_printf (printf.c:36)
   by 0x8048472: main (tests/manuel1.c:8)
   by 0x402A6E5E: __libc_start_main (libc-start.c:129)
```

An uninitialized-value use error is reported when your program uses a value which hasn't been initialized -- in other words, is undefined. Here, the undefined value is used somewhere inside the printf() machinery of the C library. This error was reported when running the following small program:

```
int main()
{
      int x;
        printf("x = %d\n", x);
}
```

It is important to understand that your program can copy around junk (uninitialized) data to its heart's content. Memcheck observes this and keeps track of the data, but does not complain. A complaint is issued only when your program attempts to make use of uninitialized data. In this example, x is uninitialized. Memcheck observes the value being passed to _IO_printf and thence to _IO_vfprintf, but makes no comment.
However, _IO_vfprintf has to examine the value of x so it can turn it into the corresponding ASCII string, and it is at this point that Memcheck complains.

Sources of uninitialized data tend to be:

Local variables in procedures which have not been initialized, as in the example above.
The contents of malloc'd blocks, before you write something there. In C++, the new operator is a wrapper round malloc, so if you create an object with new, its fields will be uninitialized until you (or the constructor) fill them in, which is only Right and Proper.

***Illegal frees***

For example:

```
Invalid free()
   at 0x4004FFDF: free (vg_clientmalloc.c:577)
   by 0x80484C7: main (tests/doublefree.c:10)
   by 0x402A6E5E: __libc_start_main (libc-start.c:129)
   by 0x80483B1: (within tests/doublefree)
   Address 0x3807F7B4 is 0 bytes inside a block of size 177 free'd
   at 0x4004FFDF: free (vg_clientmalloc.c:577)
   by 0x80484C7: main (tests/doublefree.c:10)
```

```
by 0x402A6E5E: __libc_start_main (libc-start.c:129)
by 0x80483B1: (within tests/doublefree)
```

Memcheck keeps track of the blocks allocated by your program with malloc/new, so it can know exactly whether or not the argument to free/delete is legitimate or not. Here, this test program has freed the same block twice. As with the illegal read/write errors, Memcheck attempts to make sense of the address free'd. If, as here, the address is one which has previously been freed, you wil be told that -- making duplicate frees of the same block easy to spot.

***When a block is freed with an inappropriate de-allocation function***

In the following example, a block allocated with new[] has wrongly been de-allocated with free:

```
Mismatched free() / delete / delete []
   at 0x40043249: free (vg_clientfuncs.c:171)
   by 0x4102BB4E: QGArray::~QGArray(void) (tools/qgarray.cpp:149)
   by 0x4C261C41: PptDoc::~PptDoc(void) (include/qmemarray.h:60)
   by 0x4C261F0E: PptXml::~PptXml(void) (pptxml.cc:44)
   Address 0x4BB292A8 is 0 bytes inside a block of size 64 alloc'd
   at 0x4004318C: __builtin_vec_new (vg_clientfuncs.c:152)
   by 0x4C21BC15: KLaola::readSBStream(int) const (klaola.cc:314)
   by 0x4C21C155: KLaola::stream(KLaola::OLENode const *) (klaola.cc:416)
   by 0x4C21788F: OLEFilter::convert(QCString const &) (olefilter.cc:272)
```

The following was told to me be the KDE 3 developers. I didn't know any of it myself. They also implemented the check itself.
In C++ it's important to de-allocate memory in a way compatible with how it was allocated. The deal is:

If allocated with malloc, calloc, realloc, valloc or memalign, you must de-allocate with free.
If allocated with new[], you must de-allocate with delete[].
If allocated with new, you must de-allocate with delete.
The worst thing is that on Linux apparently it doesn't matter if you do muddle these up, and it all seems to work ok, but the same program may then crash on a different platform, Solaris for example. So it's best to fix it properly. According to the KDE folks "it's amazing how many C++ programmers don't know this".
Pascal Massimino adds the following clarification: delete[] must be called associated with a new[] because the compiler stores the size of the array and the pointer-to-member to the destructor of the array's content just before the pointer actually returned.
This implies a variable-sized overhead in what's returned by new or new[]. It rather surprising how compilers [Ed: runtime-support libraries?] are robust to mismatch in new/delete new[]/delete[].

### *Passing system call parameters with inadequate read/write permissions*

Memcheck checks all parameters to system calls. If a system call needs to read from a buffer provided by your program, Memcheck checks that the entire buffer is addressable and has valid data, i.e., it is readable. And if the system call needs to write to a user-supplied buffer, Memcheck checks that the buffer is addressable. After the system call, Memcheck updates its administrative Information to precisely reflect any changes in memory permissions caused by the system call. Here's an example of a system call with an invalid parameter:

```
#include <stdlib.h>
#include <unistd.h>
int main( void )
{
        char* arr = malloc(10);
          (void) write( 1 /* stdout */, arr, 10 );
          return 0;
}
```

You get this complaint...

```
Syscall param write(buf) contains uninitialized or un-addressable byte(s)
   at 0x4035E072: __libc_write
   by 0x402A6E5E: __libc_start_main (libc-start.c:129)
   by 0x80483B1: (within tests/badwrite)
   by <bogus frame pointer> ???
   Address 0x3807E6D0 is 0 bytes inside a block of size 10 alloc'd
   at 0x4004FEE6: malloc (ut_clientmalloc.c:539)
   by 0x80484A0: main (tests/badwrite.c:6)
   by 0x402A6E5E: __libc_start_main (libc-start.c:129)
   by 0x80483B1: (within tests/badwrite)
```

... Because the program has tried to write uninitialized junk from the malloc'd block to the standard output.

### *Overlapping source and destination blocks*

The following C library functions copy some data from one memory block to another (or something similar): memcpy(), strcpy(), strncpy(), strcat(), strncat(). The blocks pointed to by their src and dst pointers aren't allowed to overlap. Memcheck checks for this.

For example:

```
==27492== Source and destination overlap in memcpy(0xbffff294, 0xbffff280, 21)
==27492==    at 0x40026CDC: memcpy (mc_replace_strmem.c:71)
==27492==    by 0x804865A: main (overlap.c:40)
==27492==    by 0x40246335: __libc_start_main (../sysdeps/generic/libc-start.c:129)
==27492==                                  by 0x8048470: (within
/auto/homes/njn25/grind/head6/memcheck/tests/overlap)
==27492==
```

You don't want the two blocks to overlap because one of them could get partially trashed by the copying.

## Explanation of Error with example from Memcheck

***Finding Memory Leaks with Valgrind***

Memory leaks are among the most difficult bugs to detect because they don't cause any outward problems until you've run out of memory and your call to malloc suddenly fails. In fact, when working with a language like C or C++ that doesn't have garbage collection, almost half your time might be spent handling correctly freeing memory. And even one mistake can be costly if your program runs for long enough and follows that branch of code.

When you run your code, you'll need to specify the tool you want to use; simply running valgrind will give you the current list. We'll focus mainly on the memcheck tool for this tutorial as running valgrind with the memcheck tool will allow us to check correct memory usage. With no other arguments, Valgrind presents a summary of calls to free and malloc: (Note that 18490 is the process id on my system; it will differ between runs.)

```
% valgrind --tool=memcheck program_name

...
=18515== malloc/free: in use at exit: 0 bytes in 0 blocks.
==18515== malloc/free: 1 allocs, 1 frees, 10 bytes allocated.
==18515== for a detailed leak analysis, rerun with: --leak-check=yes
```

If you have a memory leak, then the number of allocs and the number of frees will differ (you can't use one free to release the memory belonging to more than one alloc). We'll come back to the error summary later, but for now, notice that some errors might be suppressed -- this is because some errors will be from standard library routines rather than your own code.

If the number of allocs differs from the number of frees, you'll want to rerun your program again with the leak-check option. This will show you all of the calls to malloc/new/etc that don't have a matching free.

For demonstration purposes, I'll use a really simple program that I'll compile to the executable called "example1"

```
#include <stdlib.h>
int main()
{
        char *x = malloc(100); /* or, in C++, "char *x = new char[100] */
        return 0;
}
```

```
% valgrind --tool=memcheck --leak-check=yes example1
```

This will result in some information about the program showing up, culminating in a list of calls to malloc that did not have subsequent calls to free:

```
==2116== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2116== at 0x1B900DD0: malloc (vg_replace_malloc.c:131)
==2116== by 0x804840F: main (in /home/cprogram/example1)
```

This doesn't tell us quite as much as we'd like, though -- we know that the memory leak was caused by a call to malloc in main, but we don't have the line number. The problem is that we didn't compile using the -g option of gcc, which adds debugging symbols. So if we recompile with debugging symbols, we get the following, more useful, output:

```
==2330== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2330== at 0x1B900DD0: malloc (vg_replace_malloc.c:131)
==2330== by 0x804840F: main (example1.c:5)
```

Now we know the exact line where the lost memory was allocated. Although it's still a question of tracking down exactly when you want to free that memory, at least you know where to start looking. And since for every call to malloc or new, you should have a plan for handling the memory, knowing where the memory is lost will help you figure out where to start looking.

There will be times when the --leak-check=yes option will not result in showing you all memory leaks. To find absolutely every unpaired call to free or new, you'll need to use the --show-reachable=yes option. Its output is almost exactly the same, but it will show more un-freed memory.

***Finding Invalid Pointer Use with Valgrind***

Valgrind can also find the use of invalid heap memory using the memcheck tool. For instance, if you allocate an array with malloc or new and then try to access a location past the end of the array:

```
char *x = malloc(10);
x[10] = 'a';
```

Valgrind will detect it. For instance, running the following program, example2, through Valgrind

```
#include <stdlib.h>
int main()
{
        char *x = malloc(10);
        x[10] = 'a';
        return 0;
}
```

With

```
valgrind --tool=memcheck --leak-check=yes example2
```

Results in the following warning

```
==9814== Invalid write of size 1
==9814== at 0x804841E: main (example2.c:6)
==9814== Address 0x1BA3607A is 0 bytes after a block of size 10 alloc'd
==9814== at 0x1B900DD0: malloc (vg_replace_malloc.c:131)
==9814== by 0x804840F: main (example2.c:5)
```

What this tell us is that we're using a pointer allocated room for 10 bytes, outside that range -- consequently, we have an 'Invalid write'. If we were to try to read from that memory, we'd be alerted to an 'Invalid read of size X', where X is the amount of memory we try to read. (For a char, it'll be one, and for an int, it would be either 2 or 4, depending on your system.) As usual, Valgrind prints the stack trace of function calls so that we know exactly where the error occurs.

### *Detecting the Use of Uninitialized Variables*

Another type of operation that Valgrind will detect is the use of an uninitialized value in a conditional statement. Although you should be in the habit of initializing all variables that you create, Valgrind will help find those cases where you don't. For instance, running the following code as example3:

```
#include <stdio.h>
int main()
{
        int x;
        if(x == 0)
        {
        printf("X is zero"); /* replace with cout and include
        iostream for C++ */
        }
        return 0;
}
```

through Valgrind will result in

```
==17943== Conditional jump or move depends on uninitialized value(s)
==17943== at 0x804840A: main (example3.c:6)
```

Valgrind is even smart enough to know that if a variable is assigned the value of an uninitialized variable, that that variable is still in an "uninitialized" state. For instance, running the following code:

```
#include <stdio.h>
int foo(int x)
{
        if(x < 10)
        {
                printf("x is less than 10\n");
        }
}
int main()
{
        Int y;
        foo(y);
}
```

in Valgrind as example4 results in the following warning:

```
==4827== Conditional jump or move depends on uninitialized value(s)
==4827== at 0x8048366: foo (example4.c:5)
==4827== by 0x8048394: main (example4.c:14)
```

You might think that the problem was in foo, and that the rest of the call stack probably isn't that important. But since main passes in an uninitialized value to foo (we never assign a value to y), it turns out that that's where we have to start looking and trace back the path of variable assignments until we find a variable that wasn't initialized.

This will only help you if you actually test that branch of code, and in particular, that conditional statement. Make sure to cover all execution paths during testing!

### *What else will Valgrind Find?*

Valgrind will detect a few other improper uses of memory: if you call free twice on the same pointer value, Valgrind will detect this for you; you'll get an error:

```
Invalid free()
```

along with the corresponding stack trace.

Valgrind also detects improperly chosen methods of freeing memory. For instance, in C++ there are three basic options for freeing dynamic memory: free, delete, and delete[]. The free function should only be matched with a call to malloc rather than a call to, say, delete -- on some systems, you might be able to get away with not doing this, but it's not very portable.
Moreover, the delete keyword should only be paired with the new keyword (for allocation of single objects), and the delete[] keyword should only be paired with the new[] keyword (for allocation of arrays). (Though some compilers will allow you to get away with using the wrong version of delete, there's no guarantee that all of them will. It's just not part of the standard.)

If you do trigger one of these problems, you'll get this error:

```
Mismatched free() / delete / delete []
```

Which really should be fixed even if your code happens to be working.

### *What Won't Valgrind Find?*

Valgrind doesn't perform bounds checking on static arrays (allocated on the stack). So if you declare an array inside your function:

```
int main()
{
        char x[10];
        x[11] = 'a';
}
```

Then Valgrind won't alert you! One possible solution for testing purposes is simply to change your static arrays into dynamically allocated memory taken from the heap, where you will get bounds-checking, though this could be a mess of un-freed memory.

## A Few More Caveats

What's the drawback of using Valgrind? It's going to consume more memory -- up to twice as much as your program normally does. If you're testing an absolutely huge memory hog, you might have issues. It's also going to take longer to run your code when you're using Valgrind to test it. This shouldn't be a problem most of the time, and it only affects you during testing. But if you're running an already slow program, this might affect you.

Finally, Valgrind isn't going to detect every error you have -- if you don't test for buffer overflows by using long input strings, Valgrind won't tell you that your code is capable of writing over memory that it shouldn't be touching. Valgrind, like another other tool, needs to be used intelligently as a way of illuminating problems.

## Summary

Valgrind is a tool for the x86 and AMD64 architectures and currently runs under Linux. Valgrind allows the programmer to run the executable inside its own environment in which it checks for unpaired calls to malloc and other uses of invalid memory (such as in initialized memory) or invalid memory operations (such as freeing a block of memory twice or calling the wrong de-allocator function). Valgrind does not check use of statically allocated arrays.

## Example:

***§ How to solve Memory Leak problem using Valgrind***

For this example, you're provided with a main.c file that dynamically allocates some vertex_t and adj_vertex_t structures to demonstrate the construction of an adjacency list. Problem is, the structures are never freed and so result in memory leaks.

Let's run the program through Valgrind to see how it can help use detect the leaks. After building the graph binary (make will do it), run the following command:

```
valgrind --leak-check=yes ./graph
```

This should produce results that look like the following:

```
==23076== Memcheck, a memory error detector
==23076== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==23076== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==23076== Command:/graph
==23076==
Adjacency list:
```

```
A: B(10)
B: A(10) C(5)
C: B(5)
==23076==
==23076== HEAP SUMMARY:
==23076== in use at exit: 168 bytes in 7 blocks
==23076== total heap usage: 7 allocs, 0 frees, 168 bytes allocated
==23076==
==23076== 168 (24 direct, 144 indirect) bytes in 1 blocks are definitely lost in loss record 7
 ==23076== at 0x4A0515D: malloc (vg_replace_malloc.c:195)
==23076== by 0x4005BC: main (main.c:17)
==23076==
==23076== LEAK SUMMARY:
==23076== definitely lost: 24 bytes in 1 blocks
==23076== indirectly lost: 144 bytes in 6 blocks
==23076== possibly lost: 0 bytes in 0 blocks
==23076== still reachable: 0 bytes in 0 blocks
==23076== suppressed: 0 bytes in 0 blocks
==23076==
==23076== for counts of detected and suppressed errors, rerun with: -v
==23076== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

The important bit here is that Valgrind reports that memory is "definitely lost". It also provides an estimate of how many bytes are leaked, and notes that there are 7 allocs and 0 frees (which is another indication that there is a memory leak). Finally, it also tells you that leaked memory was allocated by the main function (at line 17, to be specific) — if you leak memory in a deeply nested function, look for the top function in the stack trace.

To illustrate another common type of memory management bug Valgrind can detect for us, let us consider the following, faulty implementation of free_adj_list that a student implements in an attempt to free up the allocated vertices.

```
void free_adj_list (vertex_t *v) {
        adj_vertex_t *adjv = v->adj_list;
        while (adjv != NULL) {
                free(adjv);
                adjv = adjv->next;
        }
}
```

After implementing this function, a student then proceeds to free the three vertices allocated in main as follows:

```
free_adj_list(v1);
free(v1);
free_adj_list(v2);
free(v2);
free_adj_list(v3);
free(v3);
```

Before reading on, try to figure out on your own what's wrong with the implementation above. Here's the Valgrind report for the newly compiled binary:

```
==32242== Memcheck, a memory error detector
==32242== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==32242== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==32242== Command: ./graph
==32242==
Adjacency list:
A: B(10)
B: A(10) C(5)
C: B(5)
==32242== Invalid read of size 8
==32242== at 0x40060E: free_adj_list (main.c:9)
==32242== by 0x400844: main (main.c:65)
==32242== Address 0x4c1d170 is 16 bytes inside a block of size 24 free'd
==32242== at 0x4A04D72: free (vg_replace_malloc.c:325)
==32242== by 0x400609: free_adj_list (main.c:8)
==32242== by 0x400844: main (main.c:65)
==32242==
==32242== Invalid read of size 8
==32242== at 0x40060E: free_adj_list (main.c:9)
==32242== by 0x40085C: main (main.c:69)
==32242== Address 0x4c1d1d0 is 16 bytes inside a block of size 24 free'd
==32242== at 0x4A04D72: free (vg_replace_malloc.c:325)
==32242== by 0x400609: free_adj_list (main.c:8)
==32242== by 0x40085C: main (main.c:69)
==32242==
==32242== Invalid read of size 8
==32242== at 0x40060E: free_adj_list (main.c:9)
==32242== by 0x400874: main (main.c:71)
==32242== Address 0x4c1d290 is 16 bytes inside a block of size 24 free'd
==32242== at 0x4A04D72: free (vg_replace_malloc.c:325)
==32242== by 0x400609: free_adj_list (main.c:8)
==32242== by 0x400874: main (main.c:71)
==32242==
```

```
==32242==
==32242== HEAP SUMMARY:
==32242== in use at exit: 0 bytes in 0 blocks
==32242== total heap usage: 7 allocs, 7 frees, 168 bytes allocated
==32242==
==32242== All heap blocks were freed -- no leaks are possible
==32242==
==32242== for counts of detected and suppressed errors, rerun with: -v
==32242== ERROR SUMMARY: 4 errors from 3 contexts (suppressed: 6 from 6)
```

Note that there aren't any leaks ("no leaks are possible"). There are, however, 4 errors — one for each of the adj_vertex_t structures we freed. Turns out that line 9 in main.c (the line where Valgrind reports an "Invalid read") is the following one:

```
adjv = adjv->next;
```

We're trying to access a pointer in a region of memory that was already freed! Valgrind also tells us where it was freed — at line 8 in main.c (which happens to be the previous line of code). What you might find interesting is that the program doesn't crash — it's certainly not good practice to write code like this, though.

Here's a version of free_adj_list that uses a temporary pointer to fix the problem (make sure you understand how it works):

```
void free_adj_list (vertex_t *v) {
        adj_vertex_t *adjv, *tmp;
        adjv = v->adj_list;
        while (adjv != NULL) {
                tmp = adjv->next;
                free(adjv);
                adjv = tmp;
        }
}
```

Recompiling and running Valgrind again gives us the following report:

```
==32606== Memcheck, a memory error detector
==32606== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==32606== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==32606== Command: ./graph
==32606==
Adjacency list:
```

```
A: B(10)
B: A(10) C(5)
C: B(5)
==32606==
==32606== HEAP SUMMARY:
==32606== in use at exit: 0 bytes in 0 blocks
==32606== total heap usage: 7 allocs, 7 frees, 168 bytes allocated
==32606==
==32606== All heap blocks were freed -- no leaks are possible
==32606==
==32606== for counts of detected and suppressed errors, rerun with: -v
==32606== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Finally — 0 errors and no memory leaks.