A Technique for Tracing Memory Leaks in C++

Steven J. Beaty
NCR Microelectronics
2057 Vermont
Fort Collins, Colorado 80525
(303) 226-9622
Steve.Beaty@FtCollinsCO.ncr.com

Abstract

Dynamic memory usage in C++ can lead to unrecovered allocations during long executions. A method of overloading the builtin `new` and `delete` functions is introduced that can ameliorate this problem.

1 Introduction

This approach was originally implemented in half a day a number of years ago (at a company that no longer exists.) It has evolved and matured over the intervening years into something that is only mildly better than the original.

A need locate memory leaks was identified in a long-running program (a highly optimizing, retargetable compiler cite.) This sense of a "leak" was dynamic memory being `new`ed without being `delete`d. The executable size quickly outgrew the size of semiconductor memory in the machine, requiring an undesirable amount of swapping activity. The program was written in C++ and was running on a Unix machine. A first step to attacking the problem was to find where memory was being requested, used, and not returned.

2 Approach

In C++, it is easy to overload the builtin `new` and `delete` operators with user-supplied versions and thereby determine *when* memory is requested and returned. While this is useful information, it is more desirable to know *where* the memory requests are coming from. A solution to this desire is to find where `new` or `delete` routine will return to and thereby finding a address in the calling function. With this datum, and knowledge of the symbol table, the routine that called `new` or `delete` can be ascertained.

With the calling information available, a useful form of output is needed. It was decided that other tools could perform the cull of good versus bad memory requests, if the following information was present:

- whether `new` or `delete` was called,
- from which routine it was called,
- what address was requested or released, and
- an overall order for requesting/releasing.

The last item is needed to correctly match up `new` and `delete` pairs. This memory tracing activity was seen as a development issue, deemphasizing the need for rapid run times. A quick-and-dirty, minimalist method was chosen and implemented.

3 The Code

The code for this approach will be placed inline (using a monospaced font for readability). The comments have been stripped; a running commentary will take their place. It is hoped that this presentation approach will benefit both the casual reader who is interested only in the method, and the reader who wishes to use the code as the basis of an implementation. First, then, the top of the file "Space.C":

```
#include <a.out.h>
#include <string.h>
#include <stream.h>
#include <malloc.h>
#include <new.h>
#include <stdlib.h>

int space_debug = 0;
char *executable_name = "a.out";

static int number_of_calls;
static int first_time = 1;
static FILE *space_out;
```

The `space_debug` variable allows the programmer to decide whether or not to trace a particular portion of code. When tracing is desired, two statements are needed:

```
extern int space_debug;
space_debug = 1;
```

These may be littered throughout a program, wherever memory is thought to be escaping. The name of the executing program must be supplied by the programmer, as there is no portable way in Unix to deduce this name from a running program. The `number_of_calls` variable is used to keep the memory requests and releases in the order they occurred in the running program. The importance of this order will become apparent. `First_time` simply keeps track of whether the symbol table has been read in or not. `Space_out` is the file that will contain the raw memory trace information.

3.1 The Symbol Table

Some of the data required are the name and address of the routine that called either `new` or `delete`. This information is available in the executable file in the form of the symbol table that contains name/type/value tuples. Our concern is only for the name and address (value) fields. We will only pay attention to those entries that have types corresponding to function and file names. An array of these values is built using the following structure:

```
struct symlist
{
        char *name;
        int address;
};

#define START_SIZE 1000
static  int     symlist_count;
static  struct  symlist *symlist;
```

Functions have type text in the symbol table and the corresponding value is the address of the routine. A difficulty for this application present in the UNIX operating system is that the symbol table is stripped just prior to execution. Therefore, this information must be read in from the file resident on the disk. Function calls exist (e.g. `nlist`) that can read this raw information into memory however, for simplicity it was decided to let existing system-level programs do the work. The following function `read_sym()` creates a file called `symtab` and then reads the pertinent information into an array of `struct symlist`s. First, local declarations are made and space is allocated to hold the information.

```
void read_sym ()
{
        int     symlist_size = START_SIZE;

        symlist = (struct symlist *)
                malloc (sizeof (struct symlist) * symlist_size);
```

Then, a call to `system` creates the `symtab` file. The only interesting entries are those with either a `t` or a `T` (i.e. those that are function addresses or file names.) The addresses are also sorted in ascending order so that subsequent searching is easier. The arguments to `sort` specify sorting based on the first field (the address) and the reverse order of the second (so that file names appear in the list before the functions that appear in them.)

```
        char    name[512];

        fprintf (stderr, "creating symtab...");
        sprintf (name, "nm %s | egrep ' t | T ' | sort +0 -1 +1r > symtab",
                executable_name);
        system (name);
        fprintf (stderr, "done\n");
```

Note the traditional C I/O routines (e.g. `fprintf`) are used here. This allows for debugging to take place before the C++ streams are constructed. Then, the information contained in the file is read in. If this information overflows the allocated array, the array is reallocated.

```
        FILE *infile = fopen ("symtab", "r");
        if (infile == NULL)
        {
                fprintf (stderr, "error opening symtab\n");
                exit (1);
        }

        int     address;
        char    type;

        symlist_count = 0;
        while (fscanf (infile, "%x %c %s", &address, &type, name) != EOF)
        {
                symlist[symlist_count].address = address;
                symlist[symlist_count].name = malloc (strlen (name) + 1);
                strcpy (symlist[symlist_count].name, name);

                if (++symlist_count >= symlist_size)
                {
                        symlist_size += START_SIZE;
                        symlist = (struct symlist *)
                                realloc ((char *) symlist,
                                  sizeof (struct symlist) * symlist_size);
                }
        }
}
```

With the information now in the symlist array, a function's name may be determined if its address is known. The following function, print_sym, accomplishes this feat by performing a binary search on the array. It is possible that the address is less than any present in the program. This condition occurs when a stack trace goes past the address for main.

```
void print_sym (unsigned address)
{
        if (address < symlist[0].address)
        {
                fprintf (space_out, " the operating system ");
                return;
        }
```

Then print_sym performs a binary search for the correct address.

```
        int lower = 0;
        int middle;
        int upper = symlist_count;

        while (1)
        {
                middle = (upper + lower) / 2;
```

If address is between the current address and the one immediately below, middle points to the correct symbol. That is, address is in the range of addresses contained in the symbol (function name) pointed to by middle. [1]

```
                if (address > symlist[middle].address &&
                        address < symlist[middle + 1].address)
                {
                        fprintf (space_out, "%s", symlist[middle].name);
```

A search up symlist from middle is then made to find the name of the source file where the function is defined. File names do not start with a _; if the Gnu compiler was used, gcc_compiled can be ignored.

```
                        int i = middle;
                        do
                        {
                                i--;
                        }
                        while (symlist[i].name[0] == '_' ||
                                !strcmp (symlist[i].name, "gcc_compiled."));

                        fprintf (space_out, " in %s ", symlist[i].name);
                        break;
                }
```

If the address was not found, continue the search.

[1] If middle points to a file name, it cannot be true that the next address is less than address, resolving any confusion.

```
                else if (lower >= upper)
                {
                        fprintf (space_out, " an unknown address ");
                        return;
                }
                else
                {
                        address > symlist[middle].address ? upper : lower = middle;
                }
        }
}
```

3.2 New and delete

With the ability to print the name of a routine, it is desirable to report this from the `new` and `delete` functions. Propitiously, these two functions may be overloaded. The real action takes place in these routines, and some slight-of-hand is necessary to retrieve the return address.

```
void* operator new(size_t size)
{
```

In 680x0 architectures, a stack frame is as follows:

return address
frame pointer
locals

Therefore, if we know where a local variable is exactly, we can find the return address. With the assumption that locals grow downward on the stack, the first declared local will be just below the frame pointer which is just below the return address. With the further assumption that each of these are the size of an `integer`, the following code places the return address in the variable `address`.

```
#if sun && (mc68000 || mc68020)
        int i;
        int address = (*(&i+2));
#endif
```

In the SPARC architecture, the both return address and locals are placed in registers. An `asm` statement is therefore used to place the return address in a variable.

```
#if sun && sparc
        register int address;
#endif
        char *p;

#if sun && sparc && __GNUC__
        asm ("mov %i7,%l0");
#endif

#if sun && sparc && !__GNUC__
        asm ("mov %i7,%i5");
#endif
```

If the space_debug variable is on, print out the relevant information for this call to new.

```
        if (space_debug)
        {
```

If this is the first time to print out information, open the space.raw file and read in the symbol table information.

```
                if (first_time)
                {
                        if ((space_out = fopen ("space.raw", "w")) == NULL)
                        {
                                fprintf (stderr, "error opening space.raw\n");
                                exit (1);
                        }
                        first_time = 0;
                        read_sym ();
                }
                fprintf (space_out, "new(%d) called in: ", size);
                print_sym (address);
        }
```

Once the calling information is printed, memory is actually set aside.

```
        if ((p = malloc (unsigned (size))) == 0)
        {
                fprintf (stderr, "out of malloc space\n");
                exit (1);
        }
```

The actual location and the sequence number are also printed to enable later matching of request/release pairs.

```
        if (space_debug)
        {
                fprintf (space_out, "returning: %08x %08d\n",
                        p, number_of_calls);
                ++number_of_calls;
        }

        return (void*)p;
}
```

The delete operator is very similar. Once again, the address of the calling routine is found and printed if space_debug is turned on. A delete might be called before the first traced new, so the symbol table might need to be read here also.

```
void operator delete(void* p)
{
#if sun && (mc68000 || mc68020)
        int i;
        int address = (*(&i+2));
#endif

#if sun && sparc
        register int address;
#endif

#if sun && sparc && __GNUC__
        asm ("mov %i7,%l0");
#endif

#if sun && sparc && !__GNUC__
        asm ("mov %i7,%i5");
#endif

        if (space_debug)
        {
                if (first_time)
                {
                        if ((space_out = fopen ("space.raw", "w")) == NULL)
                        {
                                fprintf (stderr, "error opening space.raw\n");
                                exit (1);
                        }
                        first_time = 0;
                        read_sym ();
                }
                fprintf (space_out, "delete called in: ");
                print_sym (address);

                fprintf (space_out, "with: %08x %08d\n", p, number_of_calls);
                ++number_of_calls;
        }
```

[?] defines releasing the memory if the pointer is non-null.

```
        if (p) free ((char *)p);
}
```

3.3 Wheat and chaff

After program execution is complete, there is a file `space_raw` that contains the raw `new` and `delete` information. An example `new` line from this file is:

```
new(40944) called in: _build_a_declarator__FP6symboli in symbolic.o
returning: 002af688 00000008
```

The name of the function is obviously mangled, but it is certainly possible to apply some de-mangling to achieve a more human-readable format. An example `delete` line is:

```
delete called in: __$_Set_iterator in Set.o with: 0029b730 00000055
```

The UNIX routine `sort` is invoked to pair the requests up with their associated releases. Sorting on the eighth field and beyond uses both the address and the calling sequence information for the sort.

```
        cat space.raw | sort +7 | ./throwout
```

After the sorting, the results are piped into a program `throwout`. This is an `awk` script that removes matching (on adjacent lines) `new` and `delete` pairs. There are several notables.

1. `Throwout` does not check to see if the `new` and `delete` are called within the same file so as to allow for the splitting of class functionality across multiple files.
2. Two files are produced by `throwout`: `space.recovered` and `space.bugs`. The former is the pairs of matching request/release pairs and may be used to double check the results. The later is the non-matching pairs and should be used to find and remove memory allocation errors.
3. The overall scheme involves recognizing a `delete` line and seeing if the previous line is a `new`. If it is, and the previous `new`'s address is equivalent, then that space is considered recovered.

So, the code. First, create two empty output files.

```
#! /bin/awk -f

BEGIN   {
        print "" > "space.recovered";
        print "" > "space.bugs";
}
```

When a `delete` line is recognized, the program determines if the previous line is non-null and the address matches. If so, this space is recovered. When `previous_line` is null the previous line was a `delete` line. This convention was decided upon because it does not matter what follows a `delete` line; either it matched a `new`, or it is a bug (either not matching the previous `new`'s address, or having two `delete`s in a row.) In either case the line can be correctly characterized and placed in the proper file.

```
/^delete/ {
        if (previous_line != "" && previous_address == $8)
        {
                print previous_line >> "space.recovered";
                print $0 >> "space.recovered";
        }
        else
        {
                if (previous_line)
                {
                        print previous_line >> "space.bugs";
                }
                print $0 >> "space.bugs";
        }
        previous_line = "";
        previous_address = $8;
}
```

When a `new` line is recognized and the previous line is non-null (it was also a `new` line), the previous line is placed into the bugs file. This line and address is then saved for comparison.

```
/^new/ {
        if (previous_line)
        {
                print previous_line >> "space.bugs";
        }
        previous_line = $0;
        previous_address = $8;
}
```

There is obviously great reliance upon UNIX commands with this method. This reliance was chosen to speed up the implementation. Certainly other methods of filtering the data exist; bundling the sorting and matching routines into the program certainly is one of these.

4 Porting

In porting these routines to different machine/compiler combinations, the greatest difficulty has been retrieving the return address information. Most machines/operating systems enforce some calling conventions while leaving others to the discretion of the language translator. For example, a particular machine might enforce where the return address is placed, but does not specify where locals variables may be stored. This allows compilers the flexibility of choosing from the many types of storage that may hold changing values for a given routine.

In the `new` and `delete` functions above, there are numerous `#if`s to control where the return address is placed. The machine/compiler combinations covered are:

1. Sun-3's (680x0 based) with any compiler,

2. Sun SPARC's with the AT&T 2.0 compiler, and

3. Sun SPARC's with the GNU 1.39 compiler.

A difference between the GNU and the AT&T compiler is where the local "register" variable `i` is placed on a SPARC machine. In the AT&T compiler, it is placed in register `i5`, whereas the GNU compiler places it in `l0`. Another difference is the AT&T compiler does not allow variables to be declared after an `asm` statement (probably a reasonable stipulation) while the GNU compiler does.

The optimization level being used for compilation may also affect the placement of locals. No effort to that end was made in this implementation to determine optimization level.

On machines where function return values reside in memory a trace is easily made for as many levels as desired. The following is an example for retrieving the next calling level for a 680x0-based machine.

```
#if sun && (mc68000 || mc68020)
                fprintf (space_out, "from: ");
                print_sym (*(int *)((*(&i+1))+4));
#endif
```

On machines with register windowing (such as the SPARC []), only one return address is guaranteed to be present in the registers. Any others may have been spilled to memory because of a window overflow. For simplicity it was chosen to print only one level; one can certainly regenerate more levels, either by forcing a window spill, or by recognizing the spill condition and examining the spill memory.

5 Strengths and Weaknesses

There are several benefits to using this method. First and foremost, it produces information that facilitates the removal of memory request/release errors in C++ programs. This has proven useful in many different programming circumstances. Secondly, it was easy to program and is easy to use. Reliance on other utilities made it both easier to implement and easier to comprehend. Also, the only change in the executable is the addition of the `Space.o` module; no change to source code is required, a major benefit. Thirdly, no special compilation methods are required. For example, the `-g` is unneeded; however an un`strip`ped executable is required. Finally, the ability to selectively trace different parts of code has shown to be very handy. This can keep the size of the output manageable and allows for an iterative approach to memory tracing.

There are certainly shortcomings with this method.

- There is a need to port the `new` and `delete` functions to every new machine/compiler combination. This must be done by someone fairly familiar with the workings of the pair.
- There is an associated run-time speed penalty. The method has some overhead during initialization of the symbol table information, and during `new` and `delete` calls, although the major penalty is accrued during the output to the `space.raw` file. If debugging is not turned on, there is very little overhead (note that the symbol table is created and read only if debugging actually takes place.)
- If the executable is dynamically linked to a library, the names of the routines supplied by that library are not available in the executable. This difficulty can be solved several ways, the easiest probably being simply statically linking the entire executable for memory debugging purposes.
- Inline functions can create some confusion about where a `new` or `delete` is called from. It will report that the routine was called from a C++ source file, when in fact the actual function reference may be in an include file.