Debugging Memory Errors

References:

The material in this handout is collected from the following references:

- Page 462 of the text book C++ Primer.
- Section $11.2.1$ of C++ Programming Language.
- Valgrind documentation.

Dynamic memory is error-prone!!!

Functions often operate in the following way:

1. Acquire free store memory by making calls to `new` or `new[]`.
2. Perform some operations on the dynamic objects bound to the acquired memory.
3. Return the previously acquired memory back to the free store using `delete` or `delete[]`.

A typical example of using pointers in this way is the use of `new` and `delete` to create and destroy an object:

```
struct some_struct {
  // the particular members of this structure are irrelevant ...
  std::string str;
};

void foo() {
  // create an object explicitly ...
  some_struct *p = new some_struct {};
  // perform some operations on object *p
  // now, dynamically deallocate heap memory
  delete p;
}
```

This function and other similar to it are a source of trouble.

Leaked or orphaned objects

People use `new` and then forget to call `delete` to return the acquired memory. The program is then said to have an *orphaned object* [orphaned because the object exists in free store memory but there is no way for the program to refer to it] and therefore a memory leak. The memory leak may or may not be a serious problem. Consider a *one-shot* program that acquires memory once, performs some actions on the acquired memory, forgets to return the memory, and then quits. In most cases, deallocating memory just before program exit is pointless - the operating system will reclaim the memory anyway.

Now, consider a program like a webserver that must run an indeterminate amount of time. This program regularly requires copies of a `some_struct` object:

```
// given the address of an object of type some_struct, make a deep copy
// of the object on the heap and return a pointer to the block of heap
// memory containing the deep copy
some_struct *SomeStructCopy(some_struct const *rhs) {
  // allocate memory in free store for an object of type some_struct
  some_struct *p { new some_struct {} };
  p->str = rhs->str; // make copies of members of *rhs
  return p;
}
```

`SomeStructCopy` allocates memory for a `some_struct` object on the free store, makes a deep copy to this object, and returns a pointer to the free store object to the caller. This introduces the problem that the caller of function `SomeStructCopy` must take ownership of the block of free store memory that was allocated by `SomeStructCopy`. If the caller forgets to deallocate the memory, a memory leak will arise. Since function `SomeStructCopy` is regularly called by the program which has to run for an indeterminate amount of time, there will arise a scenario where the memory leaks will gradually fill the free store until allocation requests cannot be satisfied, and the program crashes.

Premature deletion

People `delete` an object that they have some other pointer pointing to and later use that other pointer. The other pointer to the *freed object* is now a *dangling pointer* that no longer points to a valid object [so reading it may give bad results] and may indeed point to memory that has been reused for another object [so writing to it may corrupt an unrelated object].

```
void premature_deletion() {
  // create an object explicitly on the free store
  some_struct *p1 { new some_struct {} };
  // potential trouble - multiple pointers to the same object
  some_struct *p2 { p1 };
  // more trouble - p3 is uninitialized - it could be pointing anywhere
  some_struct *p3;
  // deleting p1 is lots of trouble - p2 doesn't point to valid object
  delete p1;
  // give false sense of safety by saying that p1 doesn't point anywhere
  p1 = nullptr;

  // p3 may now point to the memory pointed to by p2
  // that was previously used to store object *p1
  // so, p2 thinks it is pointing to *p1 but may now point to *p3
  p3 = new some_struct {};
  p3->str = "xyz";

  // more trouble - let's update (non-existent) "object" pointed to by p2
  p2->str = "abc"; // p2 thinks it is updating object ponted to by p1
  // this may have now updated p3->str to "abc" from "xyz"

  // may not print "xyz"
  std::cout << p3.str << "\n";

  //other code here - but runtime behavior of function is undefined
}
```

Double deletion

An object is freed twice. Double deletion is a problem because resource managers typically cannot track what code owns a resource. Consider:

```
// very bad code ...
void sloppy(int N) {
  // acquire memory from free store
  some_struct *p { new some_struct [N] };
  // use "array" of some_struct objects pointed to by p ...
  // return the memory back to free store
  delete [] p;

  // ... wait a while ...
  // somebody else has called new or new[] and gotten memory previously
  // use to store objects in array whose first element was pointed to by p

  // incorrect logic leads to second call to delete []
  delete [] p;
  // program behavior is undefined because sloppy() did not own memory
  // that was freed in 2nd delete []
}
```

By the second call to `delete`, the memory pointed to by `p` may have been reallocated for some other use and the allocator may get corrupted. In general, a double deletion is undefined behavior and the results are unpredictable and usually disastrous.

Dereferencing uninitialized pointers

Deleting a pointer that was never initialized and, therefore, contains a garbage value is a sure path to chaos. The system may try to recover the memory returned by the pointer, which is likely not associated with the free store.

```
// very bad code ...
void chaos() {
  some_struct *p; // define but not initialize pointer
  // no references to p ...
  delete p; // error!!! attempting to delete garbage
  // program behavior is too terrible to imagine because chaos() is
  // providing delete a pointer that contains some garbage value that
  // may point to some location on the heap (which is bad) or may point
  // to non free store memory (which is really bad) or may point to
  // some region complete outside the program memory (which is the worst)
  // all of these can lead to truly undefined behavior
}
```

On the other hand, operator `delete` recognizes the value `nullptr` and `delete` will do nothing when given a pointer having value `nullptr`.

```
void harmless() {
  some_struct *p {nullptr};
  // no references to p ...
  delete p; // harmless!!! this is a no operation
}
```

> ***Programming tip: Reinforce this rule every time you program - a variable should never be defined without being initialized at the same time. That is the advantage of C++ over C. C++ provides constructors that will be automatically invoked when a variable is defined.***

The reason people make any or all of these mistakes is typically not maliciousness and often not even simple sloppiness; it is genuinely hard to consistently de-allocate every allocated object in a large program once and at exactly the right point of computation.

Debugging programs that use free store

As seen earlier, manipulating free store memory is prone to the same bugs as manipulating stack memory:

- Dereferencing uninitialized pointers
- Dereferencing `nullptr`s
- Reading uninitialized memory
- Off-by-one array subscripting
- Getting hosed by a malformed C-style string that doesn't have a null terminating character

and some new ones:

- Failing to `delete` allocated memory causing a memory leak
- Accessing `delete`d memory
- Double-`delete`ing a pointer
- Exhausting the free store and failing to notice when `new` throws a `bad_alloc` exception or `nothrow` version of `new` returns `nullptr`

Memory issues come in two flavors: memory *errors* and memory *leaks*. When a program dynamically allocates memory and forgets to later free it, it creates a *memory leak*. A memory leak generally won't cause a program to misbehave, crash, or give wrong answers. Since memory leaks are not insidious, a memory leak is not an urgent situation but a detail that can be resolved at a later time. A *memory error*, on the other hand, is a red alert. Reading uninitialized memory, writing past the end of a piece of memory, accessing freed memory, and other memory errors are insidious activities with potentially catastrophic consequences. Memory errors should never be treated casually or ignored. Therefore, you must always prioritize identifying and fixing memory errors before memory leaks.

Checking a program with Valgrind

With so many ways of introducing memory bugs into our code, what are we to do? Fortunately, an amazing open-source debugging tool called Valgrind is available. According to Valgrind's website,

> Valgrind will save you hours of debugging time. With Valgrind tools you can automatically detect many memory management and threading bugs. This gives you confidence that your programs are free of many common bugs, some of which would take hours to find manually, or never be found at all. You can find and eliminate bugs before they become a problem.

Installing Valgrind is easy:

```
$ sudo apt-get install -y valgrind
```

Consider the following source code:

```
int main() {
  int *pi = new int [3] {1, 2, 3};
  std::cout << "pi[]: " << pi[0] << " | " << pi[1] << " | " << pi[2] << "\n";
  delete [] pi; // no problem: new[] followed by delete[]!!!
}
```

> ***Debugging tip: Compile your program using `g++` or `clang++` with `-g` option to include debugging information so that Valgrind's summary reports include exact line numbers.***

Suppose you've compiled and linked the above code [after adding necessary includes] into a program `main.out`. Test the program with Valgrind like this:

```
$ valgrind ./main.out
```

Valgrind will print a summary of `main.out`'s memory usage:

```
1 ==23528== HEAP SUMMARY:
2 ==23528==     in use at exit: 0 bytes in 0 blocks
3 ==23528==   total heap usage: 3 allocs, 3 frees, 73,740 bytes allocated
4 ==23528==
5 ==23528== All heap blocks were freed -- no leaks are possible
6 ==23528==
7 ==23528== For lists of detected and suppressed errors, rerun with: -s
8 ==23528== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Line 8 is the critical part of the summary and is what you should be aiming for: no memory leaks (line 5) and no memory errors (line 7).

Detecting memory leaks

The following code simulates a memory leak that occurs on a fairly common basis:

```
int* foo(int N) { return new int [N] {}; }

void boo() {
  int *pi = foo(3);
  std::cout << pi[0] << "\n";
}

int main() {
  std::cout << "Calling boo()\n";
  boo();
  // do other stuff after calling boo
}
```

Function `main` calls `boo` which in turn calls `foo`. Function `foo` returns to `boo` a pointer to the first element of a dynamically allocated array. Function `boo` fails in its responsibility of returning the memory back to the free store. To check for memory leaks, you need to include options `--leak-check=full` and `--show-leak-kinds=all` in the `valgrind` command:

```
$ valgrind --leak-check=full --show-leak-kinds=all ./main.out
```

Here's the report from Valgrind for the program:

```
1  ==16644== HEAP SUMMARY:
2  ==16644==     in use at exit: 12 bytes in 1 blocks
3  ==16644==   total heap usage: 3 allocs, 2 frees, 73,740 bytes allocated
4  ==16644==
5  ==16644== 12 bytes in 1 blocks are definitely lost in loss record 1 of 1
6  ==16644==    at 0x483C583: operator new[](unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
7  ==16644==    by 0x10921D: foo(int) (mem-exhaust.cpp:5)
8  ==16644==    by 0x109263: boo() (mem-exhaust.cpp:8)
9  ==16644==    by 0x1092AD: main (mem-exhaust.cpp:14)
10 ==16644==
11 ==16644== LEAK SUMMARY:
12 ==16644==    definitely lost: 12 bytes in 1 blocks
13 ==16644==    indirectly lost: 0 bytes in 0 blocks
14 ==16644==      possibly lost: 0 bytes in 0 blocks
15 ==16644==    still reachable: 0 bytes in 0 blocks
16 ==16644==         suppressed: 0 bytes in 0 blocks
17 ==16644==
18 ==16644== For lists of detected and suppressed errors, rerun with: -s
19 ==16644== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

It is easy to determine from the report that there is a memory leak: line 5 in `HEAP SUMMARY` indicates that 12 bytes are definitely lost; line 11 provides a `LEAK SUMMARY` that also indicates that 12 bytes were lost; and finally line 19 provides an `ERROR SUMMARY` indicating an error [although not necessarily a memory leak]. Lines 5 through 9 trace the reversed chain of calls that led to the memory leak: `foo` called operator `new[]` [at line 5 of source file containing definition of function `foo`] to allocate 12 bytes; `boo` called `foo`; and `main` called `boo`.

Valgrind categorizes leaks in the `LEAK SUMMARY` section using these terms:

- *definitely lost*: This is free store memory to which the program no longer has a pointer and therefore cannot be freed at program exit. Valgrind knows that you once had the pointer, but have since lost track of it at some earlier point in the program. This memory is definitely orphaned. Such cases can and should be fixed by the programmer.
- *indirectly lost*: This is free store memory that was never freed to which the only pointers to it also are lost. For example, if you orphan a linked list, the first node would be definitely lost, while subsequent nodes would be indirectly lost. Such cases can and should be fixed by the programmer.
- *possibly lost*: This is free store memory that was never freed to which Valgrind cannot be sure whether there is a pointer or not.
- *still reachable*: This is free memory that was never freed to which the program still has a pointer at exit (typically this means a global variable points to it). Such cases can and should be fixed by the programmer.

- *suppressed*: These are memory leaks potentially caused by libraries. Since programmers can't fix such leaks, they don't want to see them, and therefore Valgrind suppresses such leaks. Suppressed leaks are not a programmer's concern.

Detecting memory errors

While memory leaks are benign and don't cause undefined program behavior, memory errors are insidious that can cause undefined program behavior. Let's look at the list of programmer actions that are liable to cause memory errors:

1. Read from an uninitialized variable,
2. Dereferencing uninitialized pointers to read from or write to the free store object,
3. Dereferencing a `nullptr`,
4. Reading uninitialized free store memory,
5. Off-by-one array subscripting,
6. Dealing with a C-style string that doesn't have the null terminator,
7. Accessing `delete`d memory, and
8. Double-`delete`ing a pointer.

Memory errors are insidious and truly evil because a program with such an error seems to work correctly because you manage to get *lucky* much of the time. After several successful such lucky outcomes, you incorrectly feel confident that your program is definitely correct. Later if the program generates a catastrophic outcome, you'll tend to chalk it down to incorrect input or maybe an overheated computer. You might even release the software to a single customer or to millions of users.

> *"...the results are undefined, and we all know what "undefined" means: it means it works during development, it works during testing, and it blows up in your most important customers' faces." --* *Scott Meyers*.

Depending on luck is not a good strategy for developing robust software. Using Valgrind can help you track down the cause of visible memory errors as well as find errors lurking beneath the surface that you don't know about. The following subsections show examples of the most common error messages from Valgrind.

Invalid `free()` / `delete` / `delete[]` / `realloc()`

Consider this code that simulates a double-`delete`ion:

```
int main() {
  int *pi = new int [3] {1, 2, 3};
  std::cout << pi[2] << "\n";
  delete [] pi;
  int *pj = new int [3] {11, 22, 33};
  delete [] pi; // double-deletion
  std::cout << pj[2] << "\n";
  delete [] pj;
}
```

Neither `g++` nor `clang++` report any errors even with the full suite of warning options turned on. Nonetheless, Valgrind comes to the programmer's rescue:

```
1 ==17197== Invalid free() / delete / delete[] / realloc()
2 ==17197==    at 0x483D74F: operator delete[](void*) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
3 ==17197==    by 0x10929D: main (mem-exhaust.cpp:10)
4 ==17197==  Address 0x4da6c80 is 0 bytes inside a block of size 12 free'd
5 ==17197==    at 0x483D74F: operator delete[](void*) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
6 ==17197==    by 0x10925C: main (mem-exhaust.cpp:8)
7 ==17197==  Block was alloc'd at
8 ==17197==    at 0x483C583: operator new[](unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
9 ==17197==    by 0x1091FE: main (mem-exhaust.cpp:6)
10 ==17197==
11 ==17197== For lists of detected and suppressed errors, rerun with: -s
12 ==17197== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Valgrind reports on line 1 that there is a re-`delete` in function `main`. This error message is also produced when you try to free a memory block that was not returned by a heap or free store allocator.

Use of uninitialized value and Conditional jump or move depends on uninitialized value(s)

These error messages are generated by Valgrind when uninitialized memory is referenced. Consider code that reads uninitialized memory that was previously allocated by a `new` or `new[]` allocator:

```
int main() {
  int *pi = new int; // free store object *pi is uninitialized
  std::cout << *pi << std::endl;
  delete pi;
}
```

Valgrind provides this (abbreviated) report:

```
==17325== Conditional jump or move depends on uninitialised value(s)
==17325==    at 0x49781C2: std::ostreambuf_iterator<char, std::char_traits<char> > std::num_put<char, std::ostreambuf_iterator<char, std::char_traits<char> > >::_M_insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<char> >, std::ios_base&, char, long) const (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.28)
==17325==    by 0x4986ED9: std::ostream& std::ostream::_M_insert<long>(long) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.28)
==17325==    by 0x109216: main (mem-exhaust.cpp:7)
==17325==
==17325== Use --track-origins=yes to see where uninitialised values come from
==17325== For lists of detected and suppressed errors, rerun with: -s
==17325== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)
```

Next, consider code that is a bit different from the previous example - this code fragment contains references to uninitialized variables:

```
int main() {
  int ctr; // uninitialized variable - will contain garbage!!!
  for (int i{0}; i < ctr; ++i) {
    std::cout << ctr << "\n";
  }
}
```

References to uninitialized variables are reported by both `g++` and `clang++`. Let's look at the report from Valgrind:

```
1 ==17638== Conditional jump or move depends on uninitialised value(s)
2 ==17638==    at 0x1091C2: main (mem-exhaust.cpp:7)
3 ==17638==
4 ==17638== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

The message on line 1 indicates that the program contains references to uninitialized variables.

Invalid read of size n

Consider code that reads outside the bounds of allocated memory:

```
int main() {
  int *pi = new int [3];
  std::cout << pi[30] << std::endl;
  delete [] pi;
}
```

Neither `g++` nor `clang++` report any errors even with the full suite of warning options turned on. Valgrind again comes to the programmer's rescue:

```
==17700== Invalid read of size 4
==17700==    at 0x10920B: main (mem-exhaust.cpp:7)
==17700==  Address 0x4da6cf8 is 40 bytes inside an unallocated block of size 4,121,360 in arena "client"
==17700==
==17700== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Invalid write of size n

Consider code that writes outside the bounds of allocated memory:

```
int main() {
  int *pi = new int [3];
  pi[30] = 5;
  delete [] pi;
}
```

Neither `g++` nor `clang++` report any errors even with the full suite of warning options turned on. Valgrind detects the out-of-bounds write:

```
==17791== Invalid write of size 4
==17791==    at 0x1091CB: main (mem-exhaust.cpp:7)
==17791==  Address 0x4da6cf8 is 40 bytes inside an unallocated block of size 4,121,360 in arena "client"
==17791==
==17791== For lists of detected and suppressed errors, rerun with: -s
==17791== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Final words

The entire list of Valgrind error messages are listed here. As explained earlier, memory errors are insidious and can cause a variety of problems ranging from incorrect output and intermittent crashes. Use the examples provided in this document to gain experience in detecting and resolving memory errors by running Valgrind. And, after gaining experience, make sure to frequently use Valgrind to test your programs to avoid significant grading deductions.