Memory Management

Memory Management

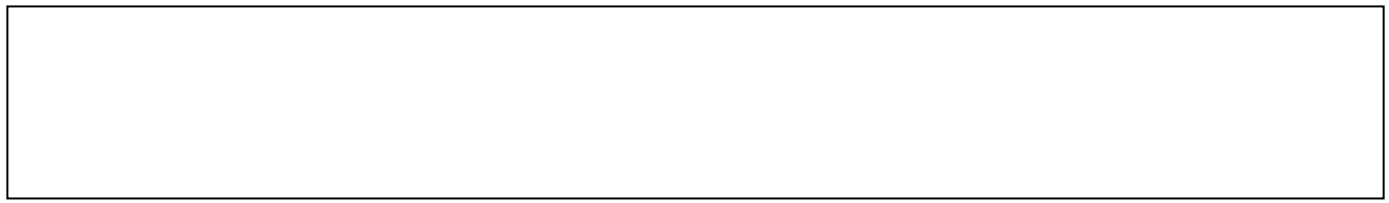
- Most of the time spent debugging C++ programs will be in finding and correcting memory bugs
- Can be very difficult to find
  - The bug may not manifest itself until much later in the code
- Types of memory errors:
  - Memory Leaks
  - Invalid Pointers
    - Uninitialized pointers
    - Dangling pointers
    - No value returned by a function
  - Out of Bounds Errors

Malloc

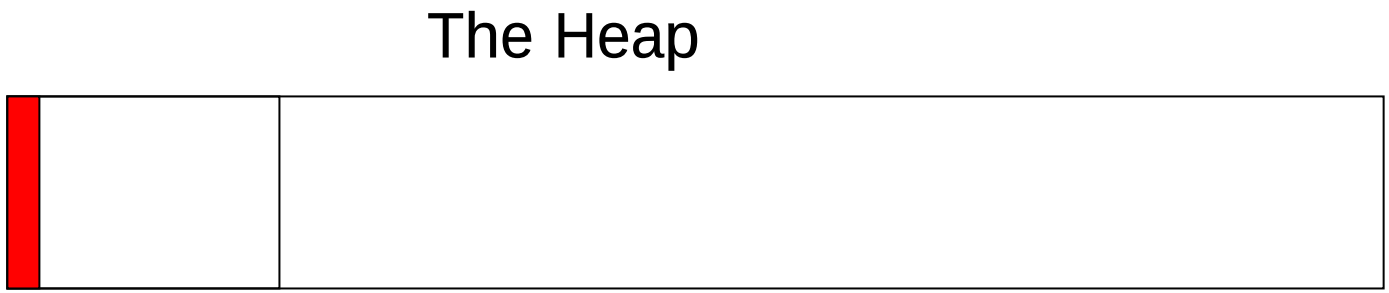
- To understand the memory problems that may occur, you need to understand the underlying memory management system
- `malloc` – the basic C routine for allocating memory
  - The user gives the actual number of bytes
  - The system goes to the heap and allocates the memory requested (if possible), plus additional memory for a "header" that stores bookkeeping information
  - Typical parameters
    - Header size of 4 bytes
    - Block size a non-zero multiple of 8 bytes
    - Minimum block size of 8 bytes
  - The header information contains the size of the memory plus a bit indicating if the memory is allocated or not

Malloc example

The Heap

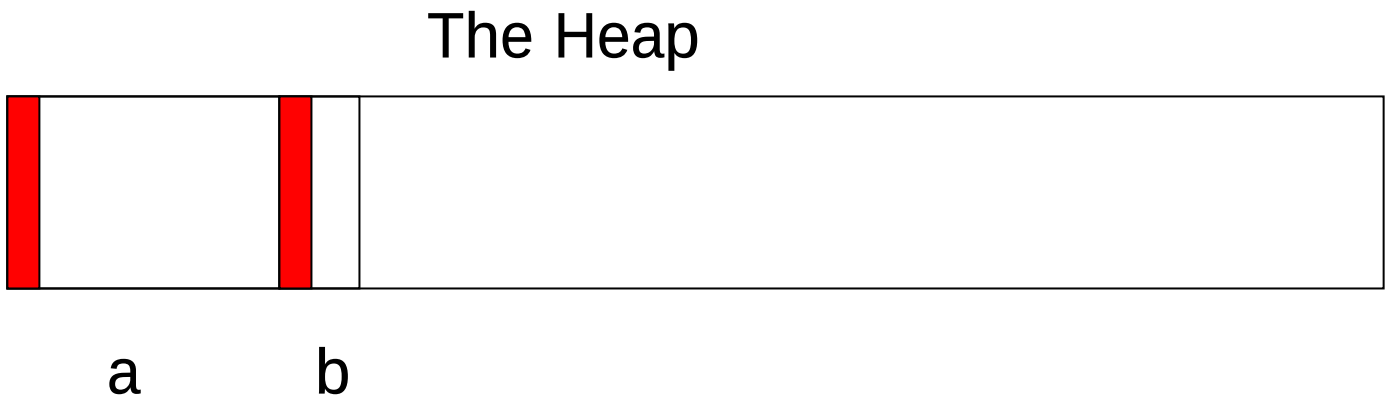


Malloc example



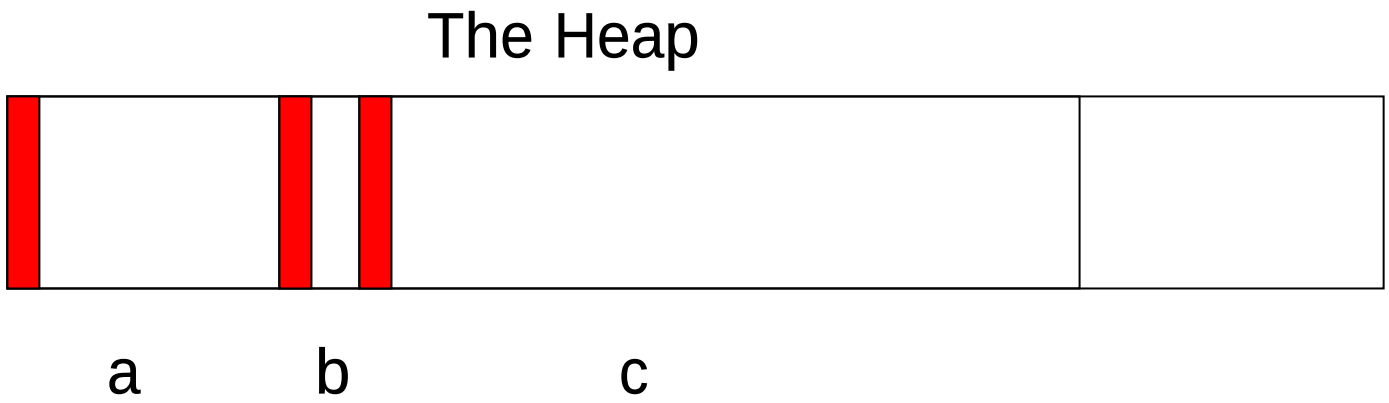

```
a = malloc(80);
```

Malloc example



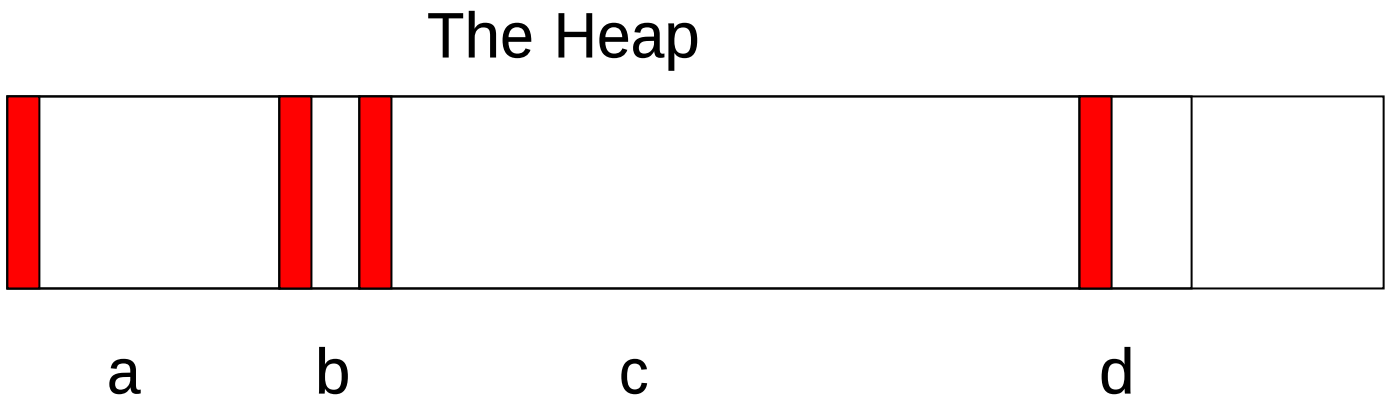

```
a = malloc(80);
b = malloc(5);
```

Malloc example




```
a = malloc(80);
b = malloc(5);
c = malloc(500);
```

Malloc example

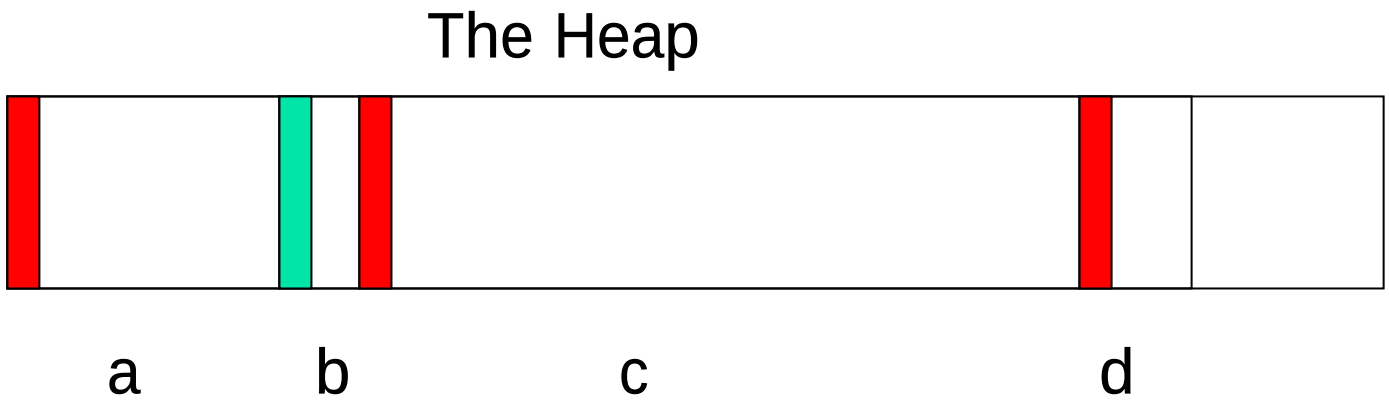



```
a = malloc(80);
b = malloc(5);
c = malloc(500);
d = malloc(20);
```

Free

- Used to deallocate memory
- Sets the bit in the header to indicate that the memory is available
- The deallocated memory can then be allocated to another malloc call

Free example




```
free(b);
```

Free

- You don’t need to tell the system how much to free – it gets it from the header
- When memory is freed, it is placed on a list of available memory
- `free` creates fragmented memory
- Can reclaim the fragmented memory by looking at the headers and coalescing contiguous memory that is not currently allocated

Allocating memory

- After several `malloc`'s and `free`'s, how should the system allocate memory
  - The open memory on the end of the heap
  - Blocks of space previously freed
- Finding the "first fit" is fast, but may cause a lot of fragmentation
- Finding the "best fit" is slow, but reduces fragmentation
- The system keeps a "free space" list
- Small chunks of memory are difficult to find a use for

Freeing storage

- How do we collapse small chunks into larger ones?
  - Merge adjacent free blocks
    - Check the blocks' headers. If two adjacent blocks are free, join them and create a new header
    - Can be done by storing a second copy of the header at the end of the block (called a "footer"). This makes it very easy to merge adjacent blocks

C++ Memory Management

- `new` – calls `malloc` to allocate the memory for the object, calls the appropriate constructor, and returns a pointer to the memory that was allocated
  - Do NOT use `malloc` in C++ - it bypasses the constructor
- `delete` – calls the destructor for each object being destructed, then calls `free`
  - Do NOT use `free` in C++ - it bypasses the destructor
- If you have allocated an array and call `delete` without the [], only the first object has its destructor called

Debugging Memory Problems

- Can be VERY time consuming
- VERY difficult to find at times
- Three types of memory errors
  - Memory leak
  - Dangling pointer
  - Out of Bounds Error

Memory Leaks

- A memory Leak is when you allocate memory, stop using it, but do not free it up.
- The memory becomes unusable – the system thinks it is still allocated, but your program no longer accesses it
- Types of memory leaks
  - Reachable – the program still has a pointer to the memory, but does not use it (could deallocate, but doesn't)
  - Lost – the program has discarded all pointers to the memory (can't deallocate even if it wants to)
  - Possibly Lost – the program still has a pointer into the middle of the memory block (e.g., an array). Could deallocate the memory, but would need to move the pointer back to the beginning before calling delete

Reasons for Memory Leaks

- You allocate memory and just forget to free it
  - E.g., Destructor or operator = isn't correct
- You allocate an array and forget to deallocate with `delete[]`
- An error condition causes a routine to abort without properly releasing memory
  - Return statements
  - Throwing exceptions
- Misunderstanding of whose responsibility it is to free the memory
  - I think you will, you think I will
  - Ownership of memory must be well-defined
  - “Owner” of memory is responsible for freeing it

Avoiding memory leaks

- Whenever you call new to allocate some memory, decide where in the code that memory will be deleted, and also write the code to delete it
- Always deallocate arrays with delete []
- Make sure that all memory is freed by all possible paths through a routine, even if errors occur
- Clearly define whose responsibility it is to free heap memory

Invalid Pointers

- Always initialize pointer variables to something, even if it's just NULL
- A pointer variable should always point to something valid, or be NULL
  - Garbage pointers are dangerous
- Compile with warnings turned on
  - Catches functions that don't return values
  - g++ -Wall ...

Invalid Pointers

- Symptoms of invalid pointers
  - Data mysteriously changes value for no apparent reason
  - Heap data structures become corrupted
  - Runtime stack becomes corrupted
  - Segmentation Fault will often result

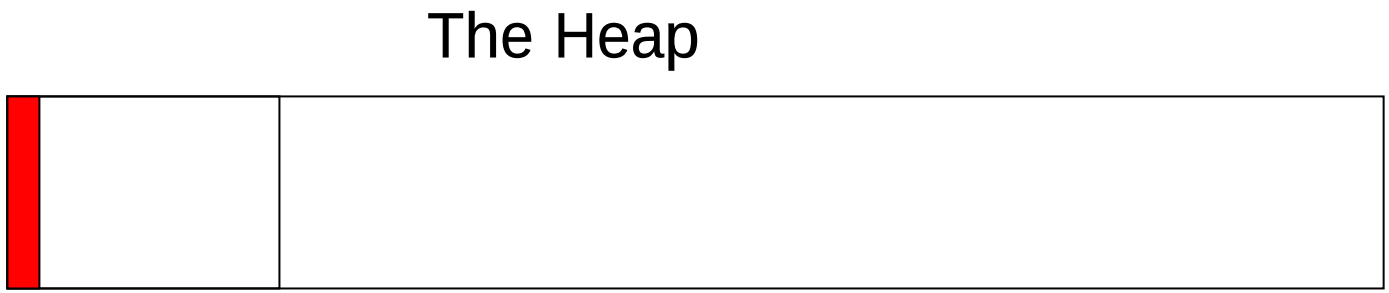
Dangling Pointers

- A dangling pointer occurs when you have a pointer that points to invalid memory
- This can occur when you have two pointers pointing to the same object and you call delete on one of the pointers but don't discard the other pointer
- The freed memory may be allocated to a different object
- The second original pointer will now point to this different object

Dangling Pointers

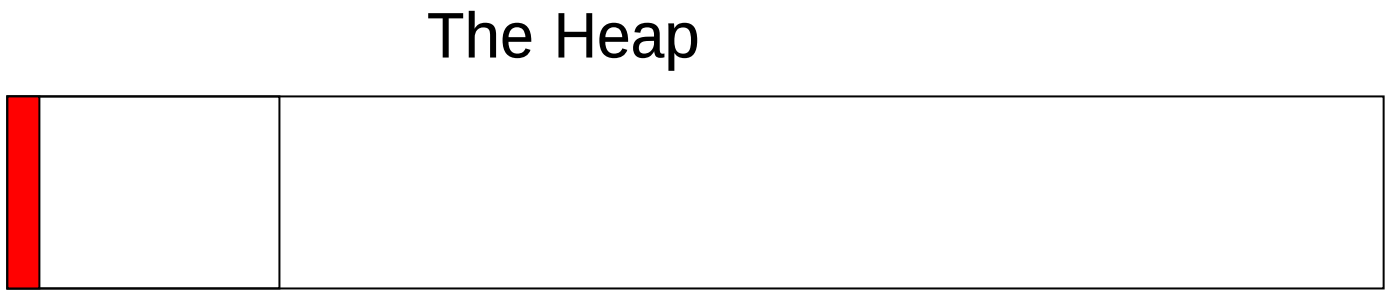
- May stomp on the other object
- May cross over the new object's boundary and stomp on the memory header info
- May cause strange behavior on deletion

Dangling Pointer example



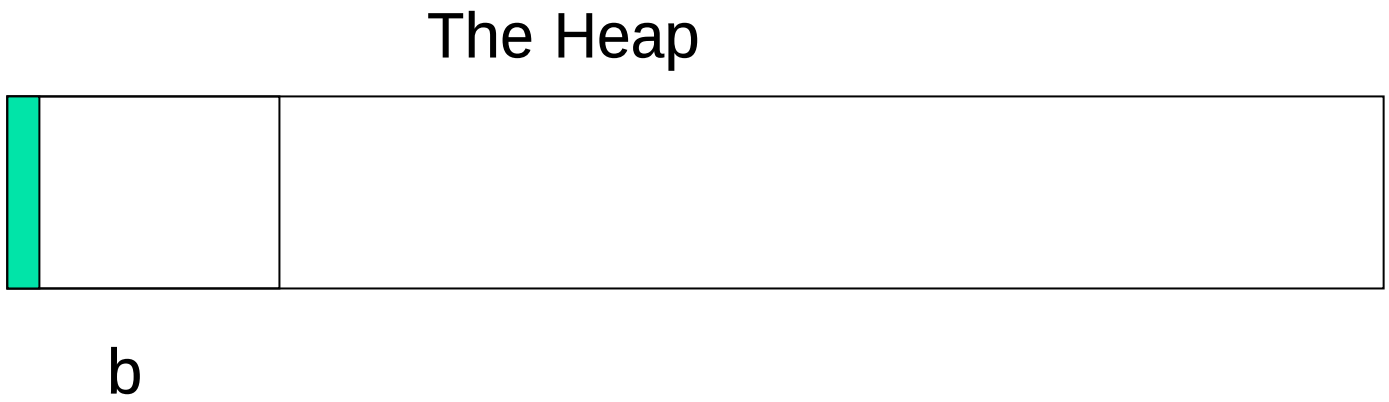

```
WebCrawler *a = new WebCrawler;
```

Dangling Pointer example



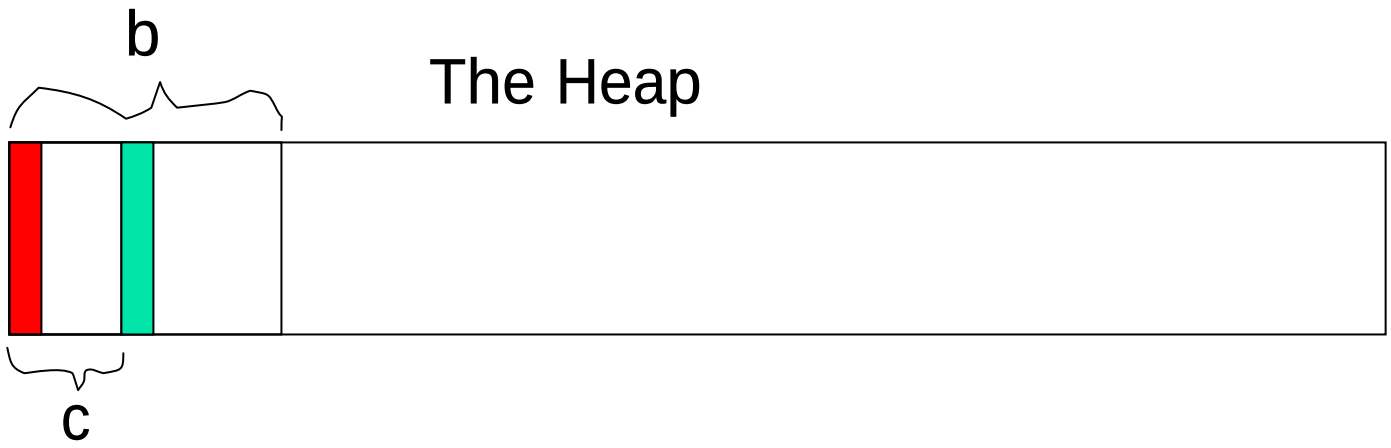

```
WebCrawler *a = new WebCrawler;
WebCrawler *b = a;
```

Dangling Pointer example



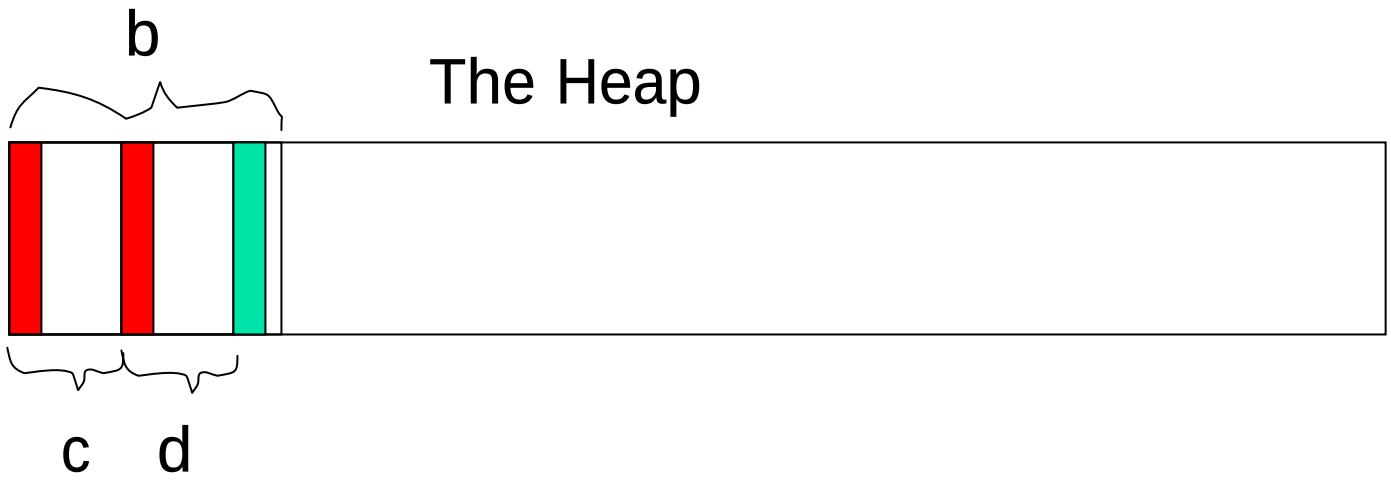

```
WebCrawler *a = new WebCrawler;
WebCrawler *b = a;
delete a;
```

Dangling Pointer example




```
WebCrawler *a = new WebCrawler;
WebCrawler *b = a;
delete a;
int *c = new int;
```

Dangling Pointer example




```
WebCrawler *a = new WebCrawler;
WebCrawler *b = a;
delete a;
int *c = new int;
int *d = new int;
```

Out of Bounds Error

- C++ does not do array bounds checking

- If you attempt to access memory beyond the memory you have allocated, it won't stop you
  - May stomp on another object
    - Data mysteriously changes
    - E.g., an adjacent variable on the runtime stack
  - May stomp on heap headers and footers

Memory Error Symptoms

- Memory consumed by program increases over time, even though it shouldn't
- Program crashes when you call new or delete because the heap is corrupted
- Variable mysteriously changes value for no apparent reason
  - Strange output
- Pointer variable mysteriously changes value, causing the program to crash when the pointer is dereferenced

Debugging Memory Errors

- Problem – the error may not manifest itself where the bug actually occurs in your code
  - You stomp on memory at one point in your code, the program crashes much later when you try to access the stomped on memory
- The key to debugging memory errors is in locating where the bug actually is

Debugging Memory Errors

- Step One: Get a hard symptom that you can identify
  - Hard crash
  - Some kind of test that shows when the error has occurred
  - Must be reproducible

Debugging Memory Errors

- Step Two: Pinpoint where the error manifests itself
  - Debugger stack trace
  - Trace statements

Debugging Memory Errors

- Step Three: Shrink the test data
  - Incrementally delete test data to get it as small as possible without eliminating the symptom

Debugging Memory Errors

- Step Four: Shrink the Code
  - Incrementally comment out code to get it as small as possible without eliminating the symptom
  - Make one change at a time, recompile, and run
  - Comment code until the symptom does not manifest itself
  - Look at the code that was most recently commented out

Debugging Memory Errors

- Step Five: Determine exactly where the error was caused

Tools for finding memory errors

- Debugging versions of new/delete
  - VC++ demo
- Static code analyzers
  - Compilers (e.g., g++ -Wall)
  - Commercial static analysis packages
- Runtime memory analyzers
  - Valgrind demo
  - Debugger watchpoints
- Homegrown tools

Memory Watcher

- When you allocate memory and initialize it, make copies of the memory's address and contents
- At various points in your code, check to see if the memory location still has its original value (if it doesn't, somebody stomped on it)

```
class MemoryWatcher {
   …
public:
   void Watch(void * addr, int bytes);
   void ReleaseWatch(void * addr);
   void Check();
};
```

Memory Watcher example

```
MemoryWatcher mw;
...
int *i = new int(15);
mw.Watch(i, sizeof(int));
...
mw.Check();
```

Memory Allocation Tracker

- An alternative method to the memory watcher is a memory allocation tracker
- Write a class that keeps track of all allocated memory
- Notify MemoryTracker object whenever memory is allocated or freed

```
class MemoryTracker {
   …
public:
   void Allocated(void * addr, int bytes, string where);
   void Freed(void * addr, string where);
   void PrintMemoryInfo();
};
```

Memory Allocation Tracker

- When MemoryTracker::Freed is called
  - If the address was not allocated, generate an error message
  - If the address was already deallocated, generate a different error message

Memory Allocation Tracker example

```
MemoryTracker mt;
...
int *i = new int(15);
mt.Allocated(i, sizeof(int), "Foo constructor");
...
delete i;
mt.Freed(i, "Foo destructor");
...
mt.PrintMemoryInfo();
```

Classes that contain Pointers

- You should always provide the following:
  - Copy Constructor
  - Destructor
  - Operator =

Trees of Pointers

- If you have a data structure that has pointers which point to other pointers, which in turn point to other pointers, etc. – how do you ensure that things get deleted corrctly?
- By using `delete`, the entire structure gets deleted
  - The parent destructor calls the children destructors, etc.
- What about DAGs?

Reference Counting

- Oftentimes a single piece of data is pointed to by multiple pointers
- One problem is knowing when to delete the object, and who should delete it.
  - The object must not be deleted until everyone using the object is done with it.
  - If everyone is done with it and it doesn’t get deleted, a memory leak occurs.

Reference Counting

- One approach is “reference counting”
- To reference count, we keep track of the number of pointers that point to each object
- To implement reference counting:
  - Whenever a new pointer to an object is created, the reference count is incremented
  - Whenever a pointer to the object is released, the counter is decremented
  - When the counter hits zero, the object is deleted

Reference Counting

- Example:

```
MyObject * p = new MyObject();
 // create a reference counted object
 p->AddRef();
 // increment the object's reference count
```

- The object’s reference count is initialized to zero, then the method AddRef increments it each time the pointer to the object is copied.

Reference Counting

- One problem – the system may make a copy of the pointer, e.g.,

```
void SomeFunction(MyObject * x) {
    x->AddRef(); // the system copies the pointer
         // x is used in the method
     . . .
         // as x is about to go out of scope, we
         // need to decrement the reference count
         // If the object's new reference count
         // becomes zero, we delete the object.
    if (x->ReleaseRef() == 0) {
        delete x;
    }
}
```

Reference Counting

- Forcing the user to remember when objects are copied by the system is too error prone
- To avoid this, “Smart Pointers” can be used
- A smart pointer is a C++ object that
  - stores a regular pointer to the reference counted object
  - automatically keeps track of the number of references to it
  - can be used like a normal C++ pointer
    - it overloads the C++pointer operators (e.g., *, ->, etc.).