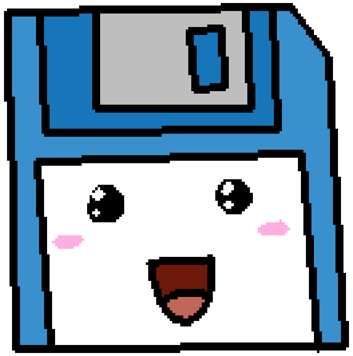
Memory Management

Written by Rachel J. Morris, last updated 2016-02-15

Break all the things!



Types of Memory Errors

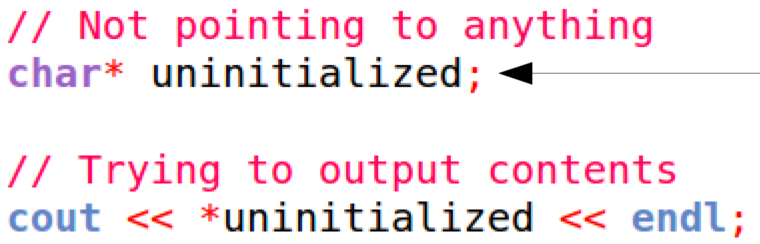
Once we're actually working with a system's memory, there are quite a few chances to create problems, such as...

- Invalid Memory Address
- Memory Leaks
- Missing Allocation
- Uninitialized Memory Access
- And More...

Types of Memory Errors

Invalid Memory Access
Usually occurs when accessing unallocated (or freed up) memory.
This might happen when...

De-referencing a pointer to an invalid address

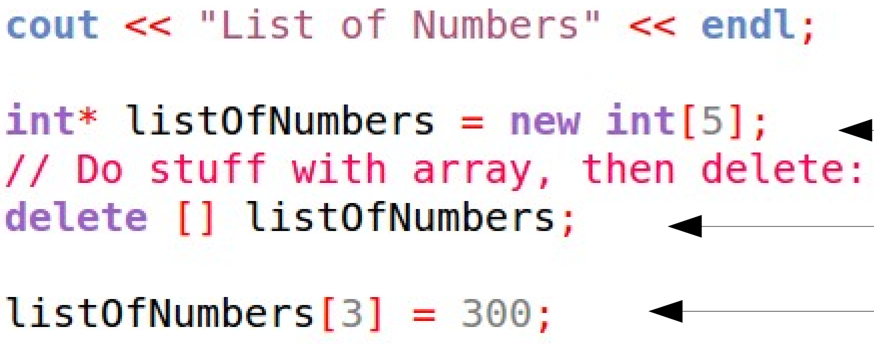


```
// Not pointing to anything
char* uninitialized;

// Trying to output contents
cout << *uninitialized << endl;
```

Pointer without an address

Trying to dereference, no (or garbage) address!

Trying to assign data at an unallocated address



```
cout << "List of Numbers" << endl;

int* listOfNumbers = new int[5];
// Do stuff with array, then delete:
delete [] listOfNumbers;

listOfNumbers[3] = 300;
```

New dynamic array

Deleting dynamic array

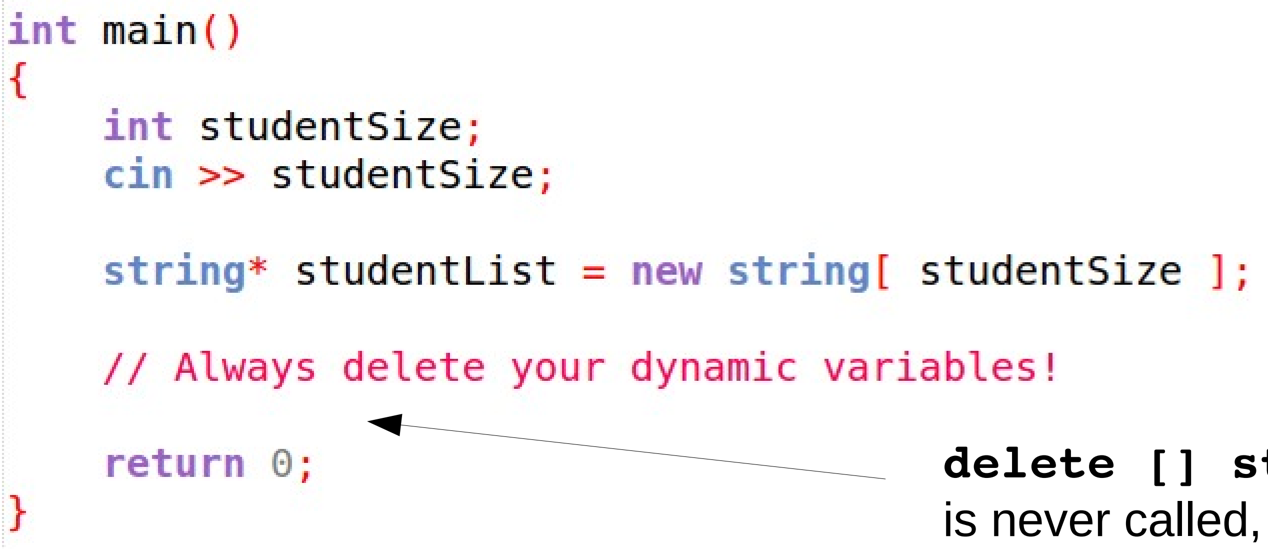
Setting value in deleted array

Types of Memory Errors

Memory Leak

Occurs when memory is allocated (via **new**), but never deallocated (via **delete**).

Always free any memory that you allocate!



```
int main()
{
    int studentSize;
    cin >> studentSize;

    string* studentList = new string[ studentSize ];

    // Always delete your dynamic variables!

    return 0;
}
```

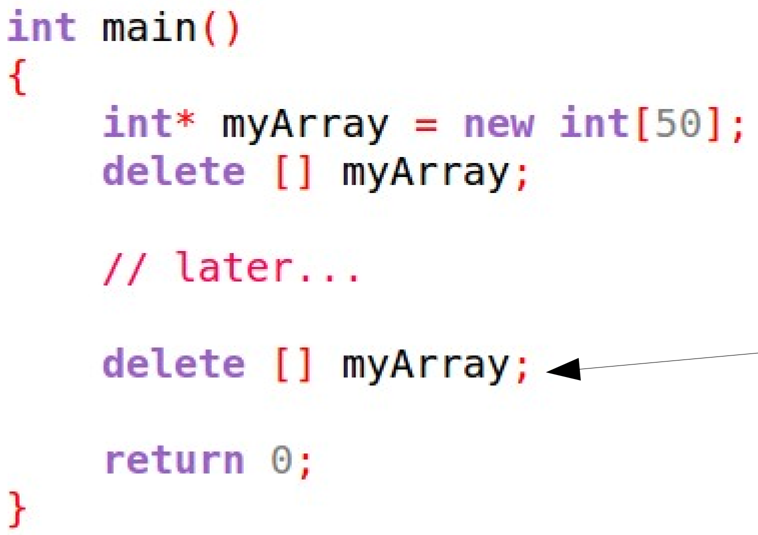
`delete [] studentList`

is never called, so that memory is never freed!

Types of Memory Errors

Missing Allocation

Occurs when you try to free memory that has already been freed!



```
int main()
{
    int* myArray = new int[50];
    delete [] myArray;

    // later...

    delete [] myArray;

    return 0;
}
```

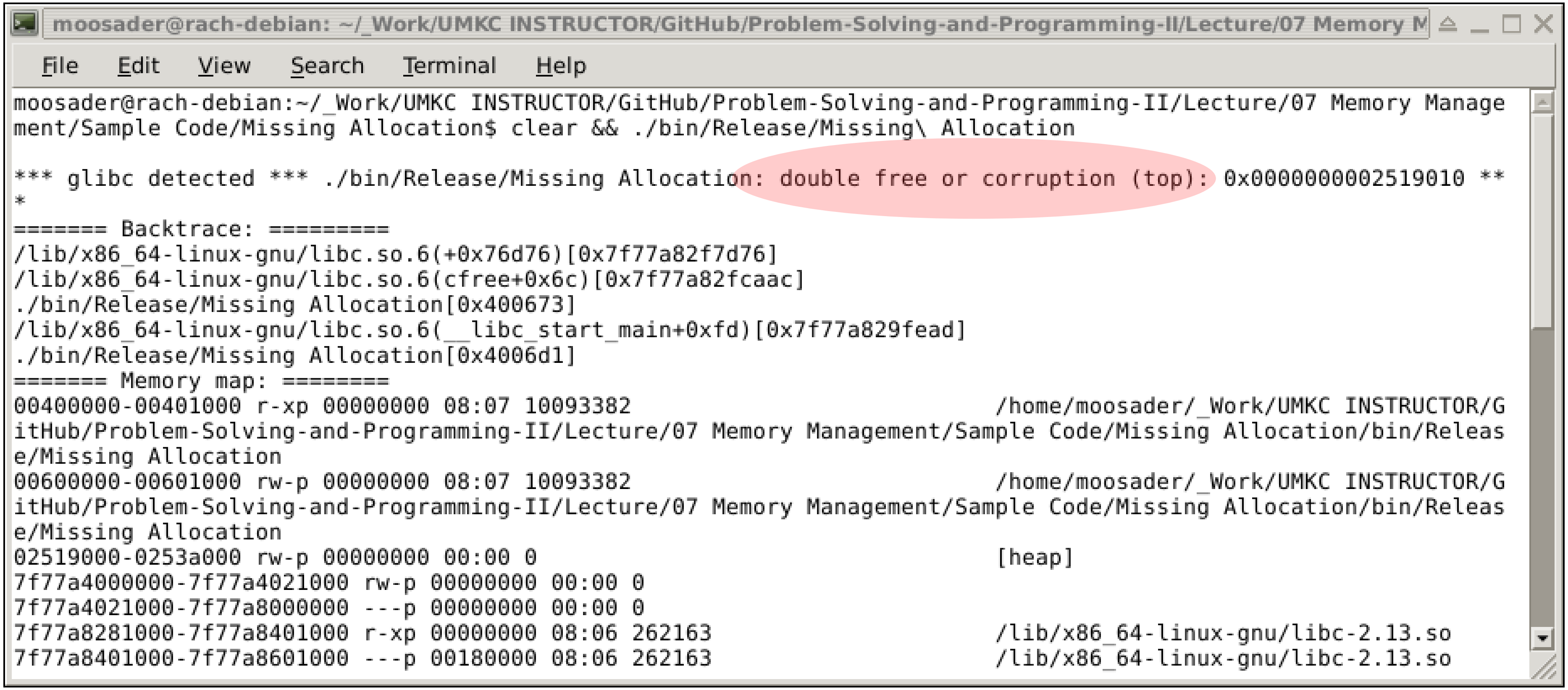
We don't “own” this chunk of memory after we've freed it, so any data this pointer is pointing to might be corrupted or deleted, affecting other programs running on the system!

Types of Memory Errors

Missing Allocation

Occurs when you try to free memory that has already been freed!

You'll get a big error message!

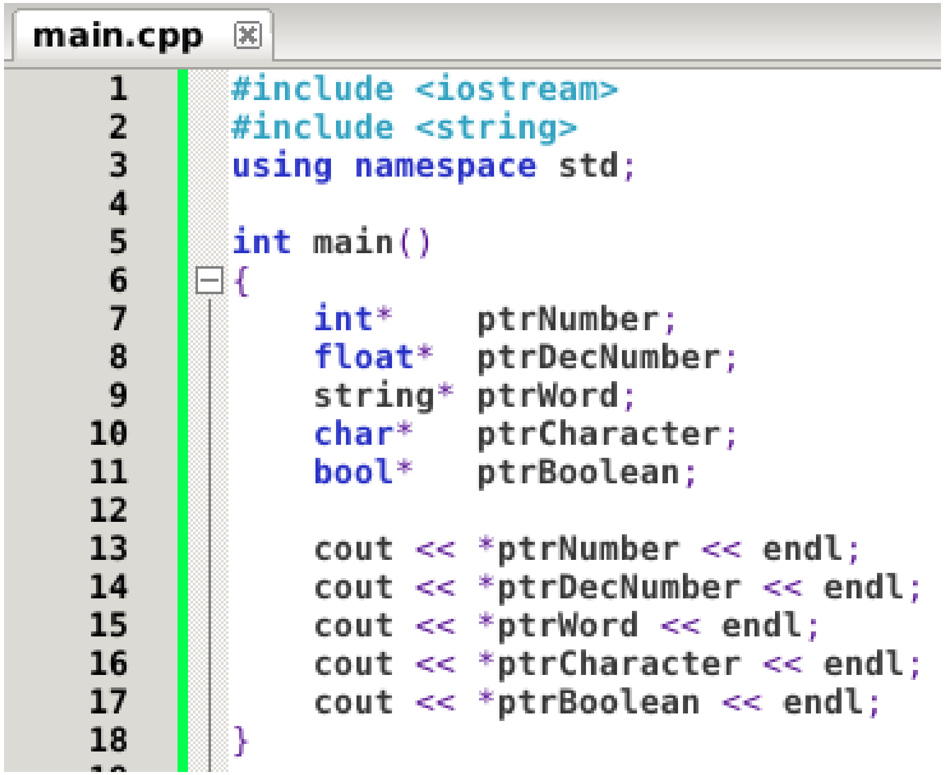

moosader@rach-debian: ~/_Work/UMKC INSTRUCTOR/GitHub/Problem-Solving-and-Programming-II/Lecture/07 Memory M
File
Edit
View
Search
Terminal
Help

```
moosader@rach-debian:~/_Work/UMKC INSTRUCTOR/GitHub/Problem-Solving-and-Programming-II/Lecture/07 Memory Manage
ment/Sample Code/Missing Allocation$ clear && ./bin/Release/Missing\ Allocation

*** glibc detected *** ./bin/Release/Missing Allocation: double free or corruption (top): 0x0000000002519010 **
*
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x76d76)[0x7f77a82f7d76]
/lib/x86_64-linux-gnu/libc.so.6(cfree+0x6c)[0x7f77a82fcaac]
./bin/Release/Missing Allocation[0x400673]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xfd)[0x7f77a829fead]
./bin/Release/Missing Allocation[0x4006d1]
======= Memory map: ========
00400000-00401000 r-xp 00000000 08:07 10093382                           /home/moosader/_Work/UMKC INSTRUCTOR/G
itHub/Problem-Solving-and-Programming-II/Lecture/07 Memory Management/Sample Code/Missing Allocation/bin/Releas
e/Missing Allocation
00600000-00601000 rw-p 00000000 08:07 10093382                           /home/moosader/_Work/UMKC INSTRUCTOR/G
itHub/Problem-Solving-and-Programming-II/Lecture/07 Memory Management/Sample Code/Missing Allocation/bin/Releas
e/Missing Allocation
02519000-0253a000 rw-p 00000000 00:00 0                                  [heap]
7f77a4000000-7f77a4021000 rw-p 00000000 00:00 0
7f77a4021000-7f77a8000000 ---p 00000000 00:00 0
7f77a8281000-7f77a8401000 r-xp 00000000 08:06 262163                     /lib/x86_64-linux-gnu/libc-2.13.so
7f77a8401000-7f77a8601000 ---p 00180000 08:06 262163                     /lib/x86_64-linux-gnu/libc-2.13.so
```

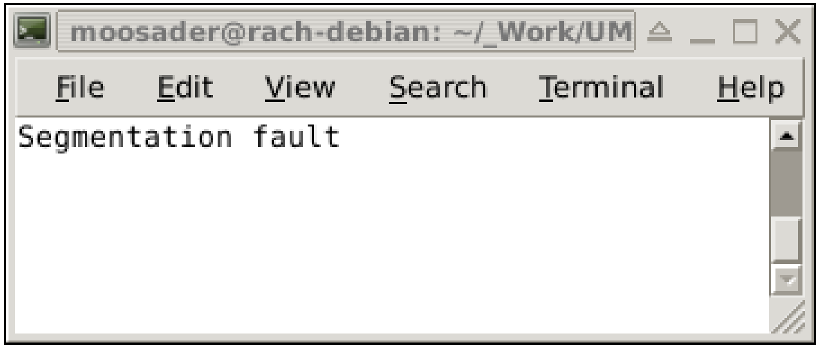
Types of Memory Errors

Uninitialized Memory Access

Occurs when you try try to read a value of an uninitialized variable.

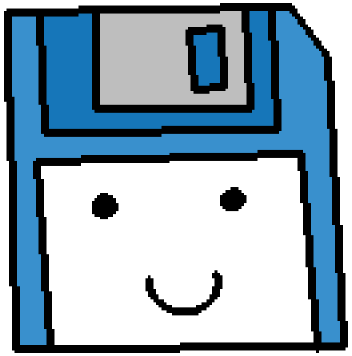


```cpp
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int*    ptrNumber;
    float*  ptrDecNumber;
    string* ptrWord;
    char*   ptrCharacter;
    bool*   ptrBoolean;

    cout << *ptrNumber << endl;
    cout << *ptrDecNumber << endl;
    cout << *ptrWord << endl;
    cout << *ptrCharacter << endl;
    cout << *ptrBoolean << endl;
}
```



```
Segmentation fault
```

The Stack and the Heap

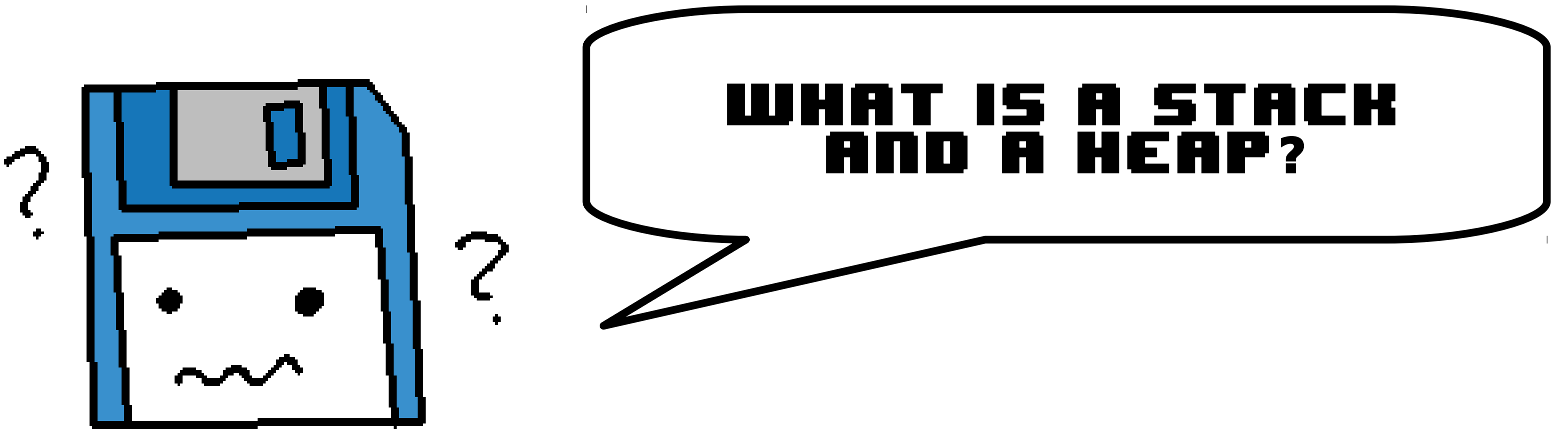


The Stack and the Heap

Different types of variables are stored in different types of memory.

Dynamic arrays are allocated in the **heap.**

Local variables and parameters are allocated in the **stack.**



The Stack and the Heap

The Stack

The **Stack** has a more complex structure than the **Heap**.

The size of the stack is fixed,
containing a sequence of memory addresses.

There is no need to manually erase memory;
it is “lazy deleted”, and will be overwritten when
that space is needed.

When we declare variables, parameters, or make a
function call, this data is pushed onto the Stack.

The Stack and the Heap

The Stack

The stack is optimized,
and the CPU can manage the memory quickly,
dealing with reads and writes.

When a function is called,
its variables are **pushed onto the stack**.

When the function ends,
its variables are **popped off the stack**
(in other words, freed).

The Stack and the Heap

The Stack

The stack grows and shrinks as its variables are **pushed** on and **popped** off.

We don't manage the memory ourself – It is handled for us automatically.

Variables on the stack only exist while the function that they belong to is running.

Remember that there is also a limit on the size of the stack.

The Stack and the Heap

The Heap...

Handles dynamically allocated memory (dynamic arrays)

Does not automatically deallocated memory; we must do this manually.

Dynamically allocated memory has to be accessed via a **pointer**.

Large objects, such as arrays, classes, etc., should be allocated here.

There is no size restriction on the heap.

Variables created on the heap are accessible anywhere in the program, by passing pointers around.

The Stack and the Heap

Think of the Stack as dealing with functions
(local variables = variables inside a function)

And think of the Heap as dealing with
dynamic memory and pointers!

The Stack and the Heap

STACK

- Optimized and fast
- Has a limited size
- Automatically handles memory allocation & deallocation
- Variables cannot be resized
- Relates to local variables & functions.

HEAP

- No size limit
- Variables can be accessed anywhere
- Manual memory management
- You can resize your variables