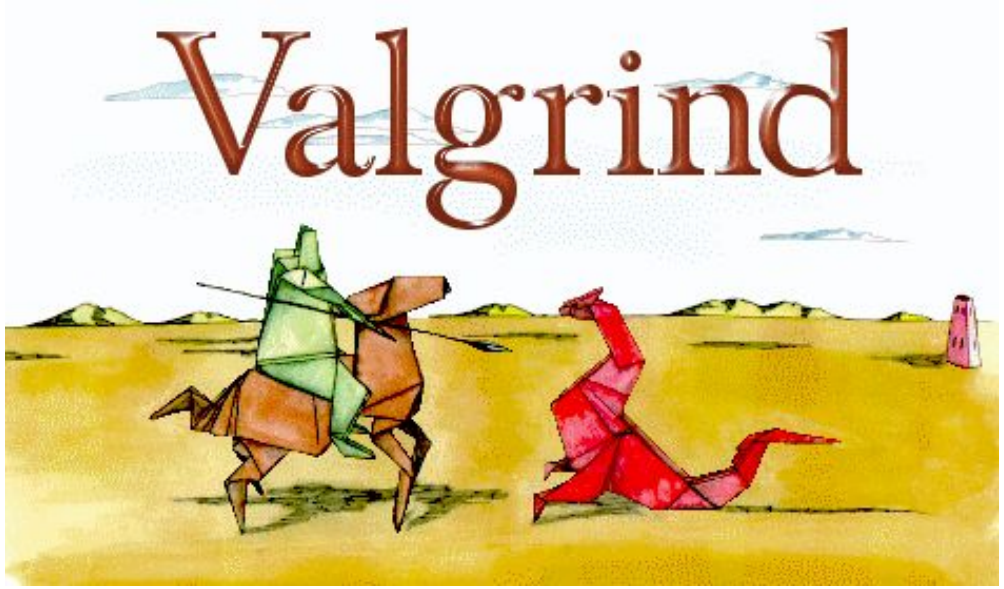
Lab_memory

Valgrind - a programming tool for memory debugging, memory leak detection, and profiling.



We will always check for memory errors and leaks on your assignments unless specified

Example 1

```
1  int main(){
2  int * arr = new int[100];
3  return arr[100];
4  }
```

What is wrong with this code?

```
1  int main(){
2  int * arr = new int[100];
3  return arr[100];
4  }
```

If we run the program it will complete without errors.

```
[mariamv2@linux-24 Lab_memory]$ ./main
[mariamv2@linux-24 Lab_memory]$
```

What happens if we run valgrind?

valgrind ./main

```
==22601== Memcheck, a memory error detector
==22601== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==22601== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==22601== Command: ./a.out
==22601==
==22601== Invalid read of size 4
==22601==    at 0x400587: main (main.cpp:3)
==22601==  Address 0x5a221d0 is 0 bytes after a block of size 400 alloc'd
==22601==    at 0x4C2A8E8: operator new[](unsigned long) (vg_replace_malloc.c:423)
==22601==    by 0x40057E: main (main.cpp:2)
==22601==
==22601==
==22601== HEAP SUMMARY:
==22601==     in use at exit: 400 bytes in 1 blocks
==22601==   total heap usage: 1 allocs, 0 frees, 400 bytes allocated
==22601==
==22601== LEAK SUMMARY:
==22601==    definitely lost: 400 bytes in 1 blocks
==22601==    indirectly lost: 0 bytes in 0 blocks
==22601==      possibly lost: 0 bytes in 0 blocks
==22601==    still reachable: 0 bytes in 0 blocks
==22601==         suppressed: 0 bytes in 0 blocks
==22601== Rerun with --leak-check=full to see details of leaked memory
==22601==
==22601== For counts of detected and suppressed errors, rerun with: -v
==22601== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
[mariamv2@linux-24 Lab_memory]$
```

```
==22601== Memcheck, a memory error detector
==22601== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==22601== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==22601== Command: ./a.out
==22601==
==22601== Invalid read of size 4
==22601==    at 0x400587: main (main.cpp:3)
==22601==  Address 0x5a221d0 is 0 bytes after a block of size 400 alloc'd
==22601==    at 0x4C2A8E8: operator new[](unsigned long) (vg_replace_malloc.c:423)
==22601==    by 0x40057E: main (main.cpp:2)
==22601==
==22601==
==22601== HEAP SUMMARY:
==22601==     in use at exit: 400 bytes in 1 blocks
==22601==   total heap usage: 1 allocs, 0 frees, 400 bytes allocated
==22601==
==22601== LEAK
==22601==    de
==22601==    ir
==22601==
==22601==    st
==22601==
==22601== Rerur
==22601==
==22601== For counts of detected and suppressed errors, rerun with: -v
==22601== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
[mariamv2@linux-24 Lab_memory]$
```

```
1  int main(){
2  int * arr = new int[100];
3  return arr[100];
4  }
```

```
1  int main(){
2  int * arr = new int[100];
3  return arr[100];
4  }
```

Out-of-bounds access to heap, stack, and globals. This error occurs when you allocate some memory and then try to access a region outside your allocated space.

```
==22601== Memcheck, a memory error detector
==22601== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==22601== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==22601== Command: ./a.out
==22601==
==22601== Invalid read of size 4
==22601==    at 0x400587: main (main.cpp:3)
==22601==  Address 0x5a221d0 is 0 bytes after a block of size 400 alloc'd
==22601==
==22601==
==22601==
==22601==
==22601==
==22601==
==22601==
==22601==
==22601== LEAK SUMMARY:
==22601==    definitely lost: 400 bytes in 1 blocks
==22601==    indirectly lost: 0 bytes in 0 blocks
==22601==      possibly lost: 0 bytes in 0 blocks
==22601==    still reachable: 0 bytes in 0 blocks
==22601==         suppressed: 0 bytes in 0 blocks
==22601== Rerun with --leak-check=full to see details of leaked memory
==22601==
==22601== For counts of detected and suppressed errors, rerun with: -v
==22601== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
[mariamv2@linux-24 Lab_memory]$
```

```
1  int main(){
2  int * arr = new int[100];
3  return arr[100];
4  }
```

Example 2 - Use of an uninitialized value.

```
1  int main(){
2     int x;
3     cout << x << endl;
4  }
```

Example 3 - Invalid free error

```
1  int main(){
2     int * x = new int;
3     delete x;
4     delete x;
5     }
```

Destructor

- Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed.
- A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

Animal.h

```
class Animal{
  Animal();
  ~Animal();
}
```

Animal.cpp

```
Animal::Animal(){}

Animal::~Animal(){}
```

```
class Animal{
private:
    int *num;

public:
    Animal(int n1);
    ~Animal();
};
```

```
Animals::Animal(int n1){
    num = new int(n1);
}

Animal::~Animal(){
    delete num;
}
```

How will you change destructor if num was array of integers?

```
class Animal{
private:
    int *num;

public:
    Animal(int n1);
    ~Animal();
};
```

```
Animals::Animal(int n1){
    num = new int(n1);
}

Animal::~Animal(){
    delete num;
}
```

How will you change destructor if num was array of integers?

delete []

Example 4 - Mismatched free() / delete / delete []

```
1  int main(){
2    int * x = new int[6];
3    delete x;
4    }
```

Example 5 - Find the errors

```
1   int main(){
2       int * arr = new int[10];
3       int * x = new int;
4       int * y;
5       arr[0] = *y;
6       delete arr;
7       delete x;
8       delete y;
9       return 0;
10  }
```

Example 5 - Find the errors

```
1   int main(){
2       int * arr = new int[10];
3       int * x = new int;
4       int * y;
5       arr[0] = *y; // y not initialized
6       delete arr; // should be delete[] arr
7       delete x;
8       delete y; // Should not delete, not on heap
9       return 0;
10  }
```

Tip: Valgrind output can get long

If necessary pipe the output of Valgrind to a file

valgrind ./exec &> log.txt

See lab handout for other valgrind options

Lab Memory

- Debugging similar to last lab
- Future labs we will provide working code that you will add to
- PraireLearn Worksheet

Tip: Read the Doxygen before beginning!

Make sure you understand what the code base is supposed to do!

The **Allocator** class takes as input **Students** and **Rooms**

```
Allocator::Allocator ( const string &  studentFile,
                       const string &  roomFile
                     )
```

Creates an **Allocator** object based on a list of students and a list of rooms.

Parameters

studentFile Path to roster file

roomFile Path to room list file

Reminders

Labs are collaborative

MPs are not

Assignment is due through PriareLearn

- Do the worksheet first