Dynamic Memory

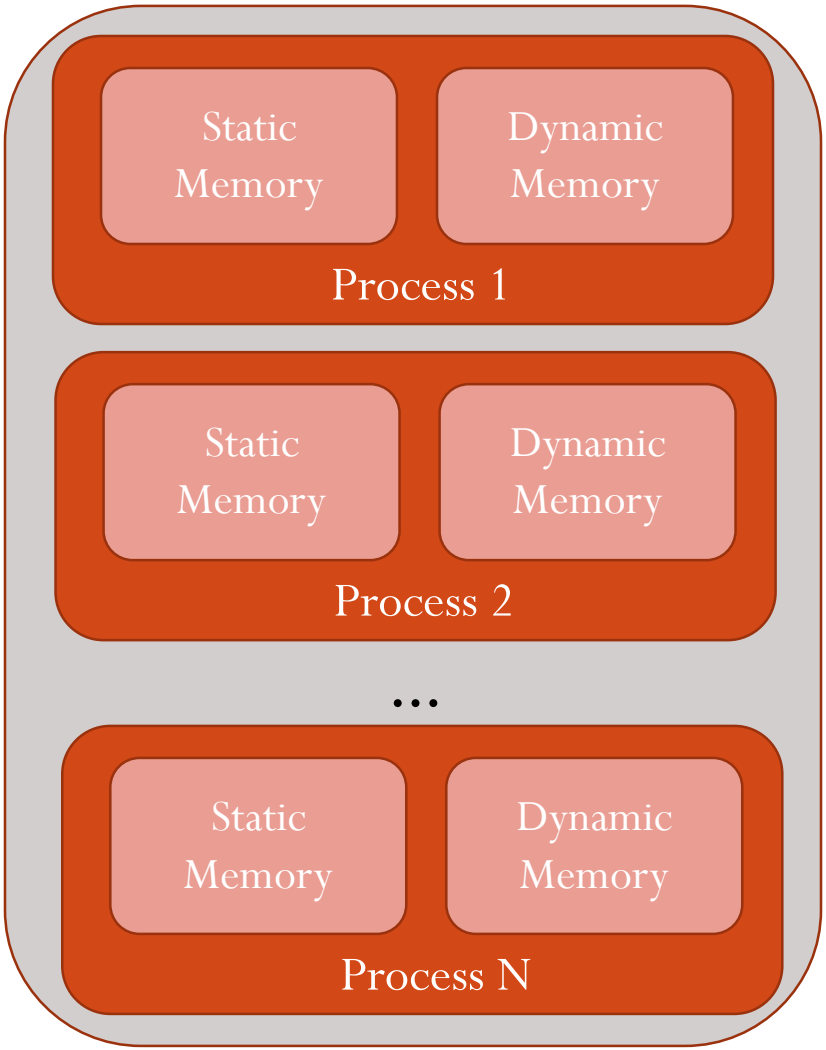
Chapter 4

Pages 153-167

Shallow Dive into Memory

- There are two memory pools for programs running in a modern OS

- Static Memory Pool:
  - Referred to as the stack
- Dynamic Memory Pool:
  - Referred to as the heap



OS Processes

The Stack (Static Memory)

- Stack Memory:
  - On each encounter of a function or new scope declared variables are placed on top of the stack.
  - On exiting the scope variables are popped off the stack
  - The top of the stack is tracked by the stack pointer which is a simple counter
  - Variables can only be popped from the top of the stack, not at any other location

The Stack (Static Memory)

- Lets investigate the stack allocation of the code below:
- Here we start with an empty stack

```
int main (){
    int a = 2, b;
    func1(a);
    func2();
    return 0;
}

void func1(int x){
    int y;
    func2();
}

void func2(){
    int z[2];

}
```

Mem Addr	Memory
0x1000	
0x1004	
0x1008	
0x100c	
0x1010	
0x1014	
0x1018	

The Stack (Static Memory)

- int a, b are allocated by placing them on the stack within a function frame
- The stack pointer decreases accordingly

```
int main (){
    int a = 2, b;
    func1(a);
    func2();
    return 0;
}

void func1(int x){
    int y;
    func2();
}

void func2(){
    int z[2];

}
```

Mem Addr	Memory
0x1000	a = 2
0x1004	b
0x1008	
0x100c	
0x1010	
0x1014	
0x1018	

main func frame

The Stack (Static Memory)

- Here we enter func1() through a call
- Parameters and declared variables of func1() are added to the stack

```
int main (){
    int a = 2, b;
    func1(a);
    func2();
    return 0;
}

void func1(int x){
    int y;
    func2();
}

void func2(){
    int z[2];

}
```

Mem Addr	Memory	
0x1000	a = 2	main frame
0x1004	b	
0x1008	x	func1 frame
0x100c	y	
0x1010		
0x1014		
0x1018		

The Stack (Static Memory)

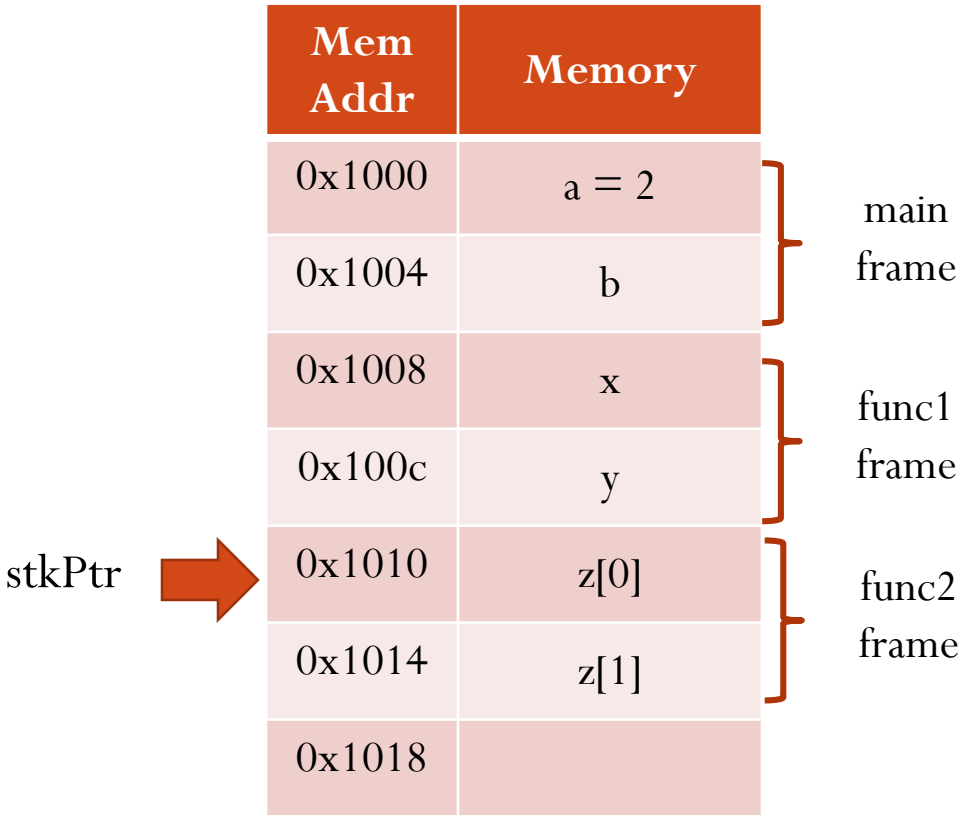
- Now we enter func2() through func1()
- Parameters and declared variables of func2() are added to the stack

```
int main (){
    int a = 2, b;
    func1(a);
    func2();
    return 0;
}

void func1(int x){
    int y;
    func2();
}

void func2(){
    int z[2];

}
```



The Stack (Static Memory)

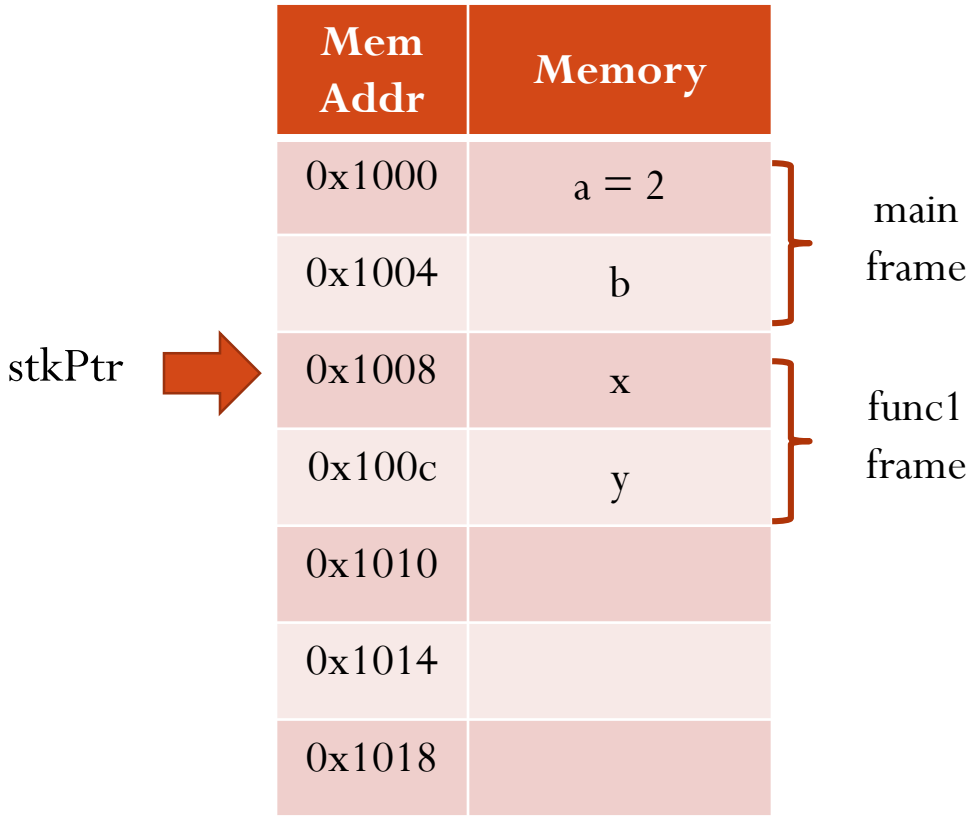
- Now func2() is left returning to func1()
- Variables on the stack are popped accordingly increasing our stkPtr

```
int main (){
    int a = 2, b;
    func1(a);
    func2();
    return 0;
}

void func1(int x){
    int y;
    func2();
}

void func2(){
    int z[2];

}
```



The Stack (Static Memory)

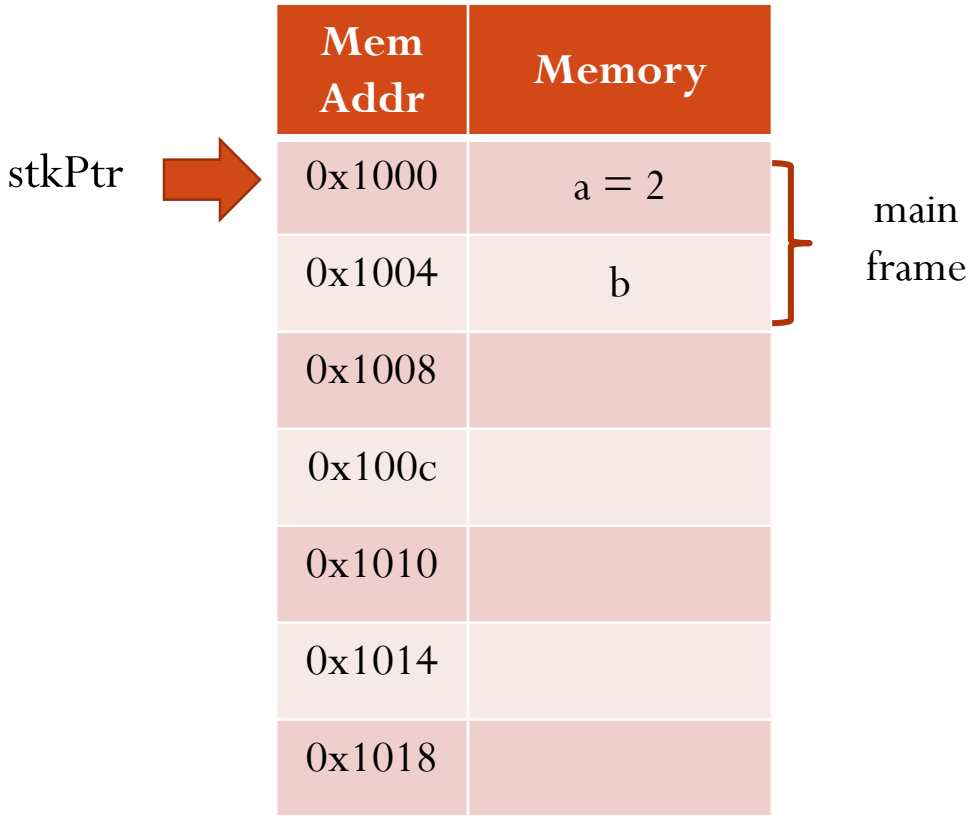
- Now func1() is left returning to main()
- This bring our stack to our original variables

```
int main (){
    int a = 2, b;
    func1(a);
    func2();
    return 0;
}

void func1(int x){
    int y;
    func2();
}

void func2(){
    int z[2];

}
```



The Stack (Static Memory)

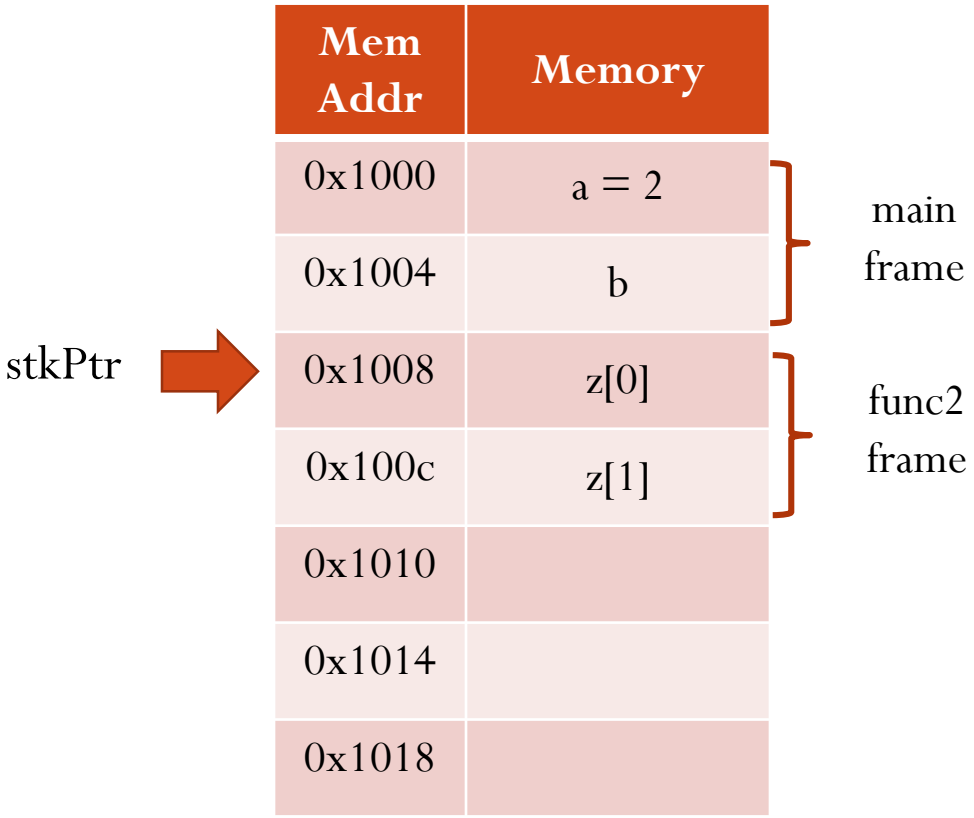
- The program enters func2() again through main()
- Add variables back in the stack

```
int main (){
    int a = 2, b;
    func1(a);
    func2();
    return 0;
}

void func1(int x){
    int y;
    func2();
}

void func2(){
    int z[2];

}
```



The Stack (Static Memory)

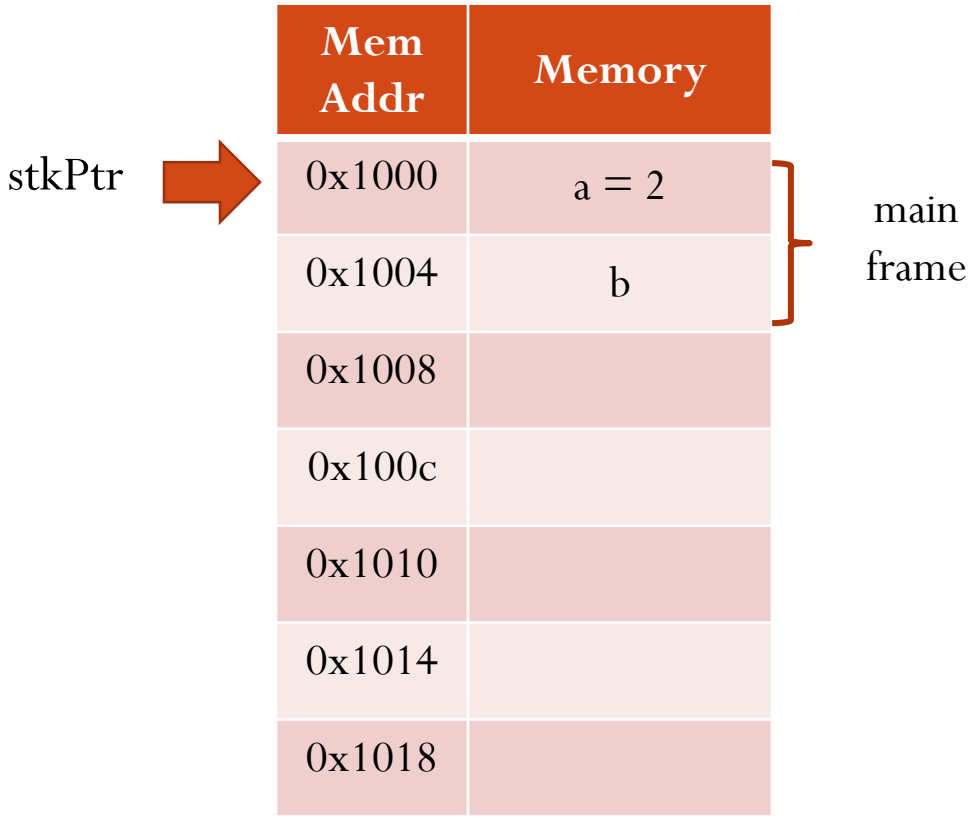
- Program exits func2()
- Data from func2() is removed from the stack

```
int main (){
    int a = 2, b;
    func1(a);
    func2();
    return 0;
}

void func1(int x){
    int y;
    func2();
}

void func2(){
    int z[2];

}
```



The Stack (Static Memory)

- Program exits main()
- Main data is removed leaving the stack empty

```
int main (){
   int a = 2, b;
   func1(a);
   func2();
   return 0;
}

void func1(int x){
   int y;
   func2();
}

void func2(){
   int z[2];

}
```

stkPtr 

Mem Addr	Memory
0x1000	
0x1004	
0x1008	
0x100c	
0x1010	
0x1014	
0x1018	

Some Fundamentals

- As functions are entered the variables are placed on the stack, the stack pointer decreases
- As functions are exited the variables are removed from the stack and the stack pointer increases

- **Variable locations are hardcoded into the assembly based on its relative location with respect to the stack pointer**
  - **Fixed at compile time!!!**

- What are the implications of this?

Implications of referring to stkPtr

- Lets jump to the first call of func1(2)
- What is the address of *z*?
  - stkPtr + 8

```
int main (){
int a = 2;
func1(a);
a = 3;
func1(a);
return 0;
}

void func1(int size){
   int y[size];
   int z;
}
```

Mem Addr	Memory
0x1000	a = 2
0x1004	y[0]
0x1008	y[1]
0x100c	z
0x1010	
0x1014	
0x1018	

main frame

func1 frame

Implications of referring to stkPtr

- Now the second call of func1(3)
- What is the address of *z*?
  - stkPtr + 12

```
int main (){
int a = 2;
func1(a);
a = 3;
func1(a);
return 0;
}

void func1(int size){
   int z;
   int y[size];
}
```

Mem Addr	Memory
0x1000	a = 2
0x1004	y[0]
0x1008	y[1]
0x100c	y[2]
0x1010	z
0x1014	
0x1018	

main frame

func1 frame

Implications of referring to stkPtr

- In the two examples the address of *z* was:
  - strPtr + 8
  - strPtr + 12

- If the size of the array is determined at runtime is it possible to hardcode the addresses of variables based on its relative position to the stack pointer?
  - No, because the relative position will also change at runtime

- Implication:
  - The size of the array must be known at compile time if it is to be placed on the stack

The Heap (Dynamic Memory)

- The heap is a less rigid memory storage structure
- Unlike the stack, individual memory allocations are **not** guaranteed to be in sequential order
- Bookkeeping is performed by a heap memory manager
  - A search must be performed to find a memory location large enough to store the information

- The heap offers greater flexibility since the size of the memory allocations only need to be known at runtime

A Heap of Terms

- Allocation:
  - A reservation of one or several consecutive memory locations in the heap for data storage
- Deallocation:
  - Freeing of a previous memory allocation (reservation) to allow for future reservations of the same location by different data
- Memory Leak:
  - An allocation of memory that is never deallocated throughout the life of the program
  - For every allocation, there should always be a deallocation!!!

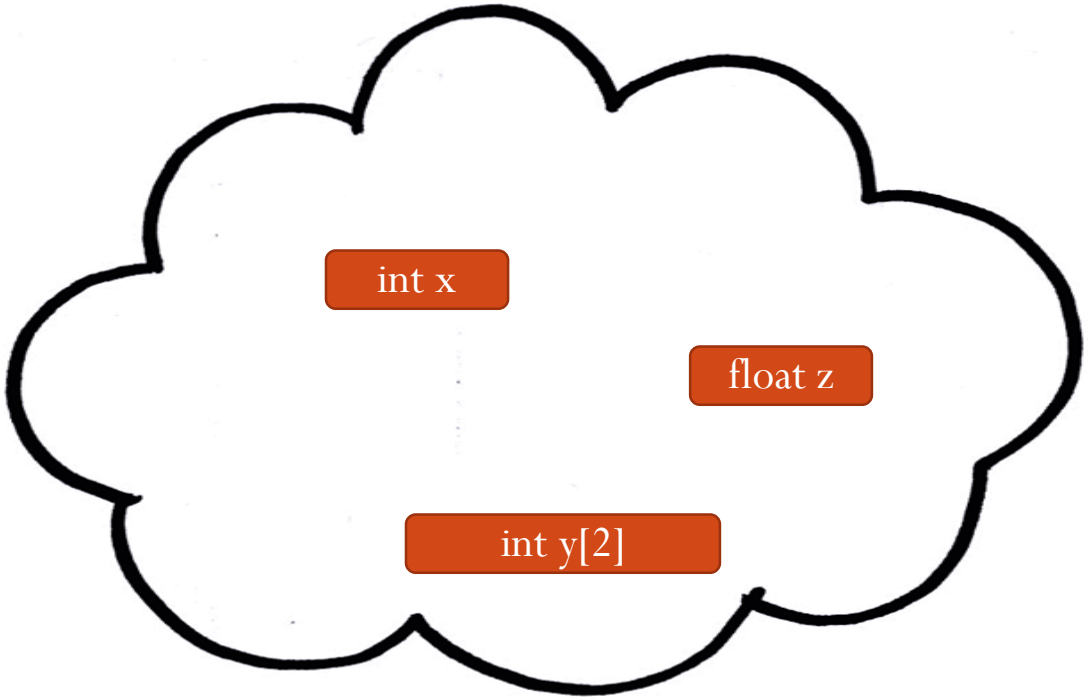
The Heap

- Up until now we have exclusively used the stack with some exceptions:
  - C++ strings
    - ever wondered why the library could get away with runtime sizing of character arrays but you couldn't?
  - Cout, Cin, fstreams
- Allocations to the heap can be performed using the following:
  - C++: keyword *new*
  - C: function malloc()
- Deallocations to the heap can be performed using:
  - C++: keyword *delete*
  - *C:* function free()

Heap Visualization

- Given the unstructured nature, the heap can be visualized as a cloud

- While there are complex algorithms to allocate memory we shall assume near random assignment of allocating memory



Using Dynamic Memory

```
int main ()
{
   int *num;
   num = new int;

   *num = 5;
   cout << *num << endl;
   delete num;

   return 0;
}
```

- To the right is an example of a dynamic allocation of an integer, and deallocation of an integer
- Notice we are working with a pointer
- Lets step through the process

Using Dynamic Memory

```
int *num;
```

- Start with a pointer on the stack and an empty heap

Mem Addr	Variable
0x1004	
0x1000	num = ?
The Stack	

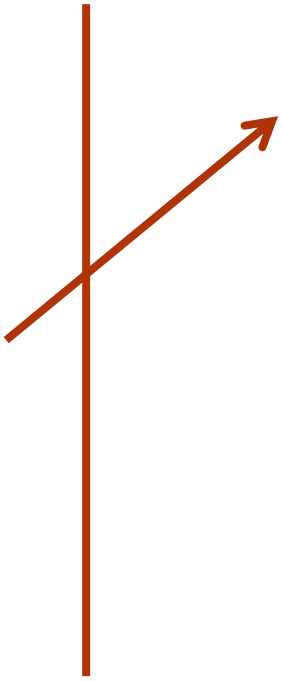
The Heap

Using Dynamic Memory

```
int *num;
num = new int;
```

- The keyword *new* returns the memory address of the allocated memory

Mem Addr	Variable
0x1004	
0x1000	num = 0x411C
The Stack	



0x411C	

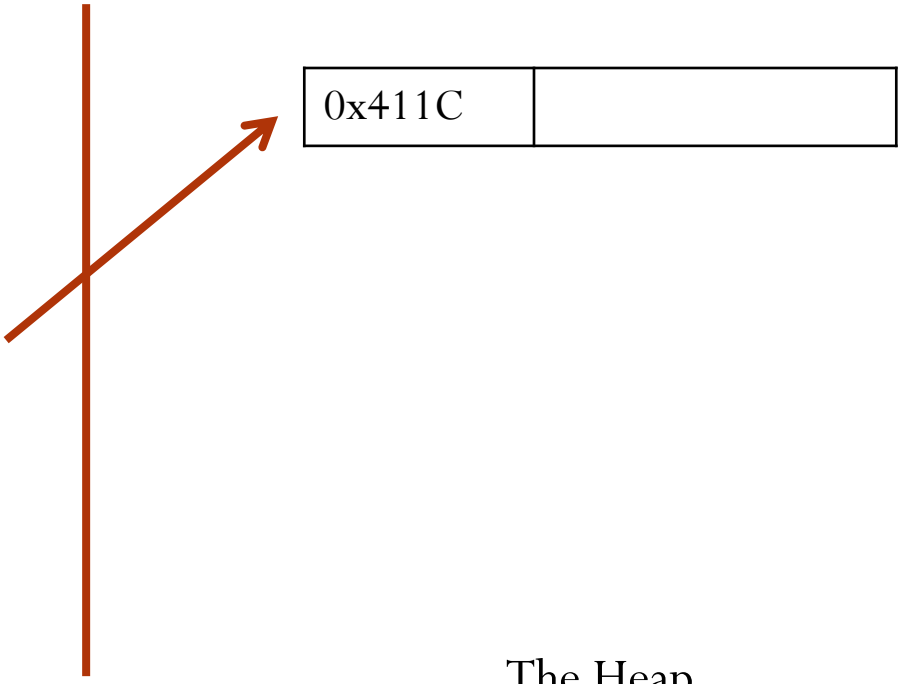
The Heap

Using Dynamic Memory

```
int *num;
num = new int;
```

- Thus we have a pointer on the stack pointing to our reserved memory

Mem Addr	Variable
0x1004	
0x1000	num = 0x411C
The Stack	

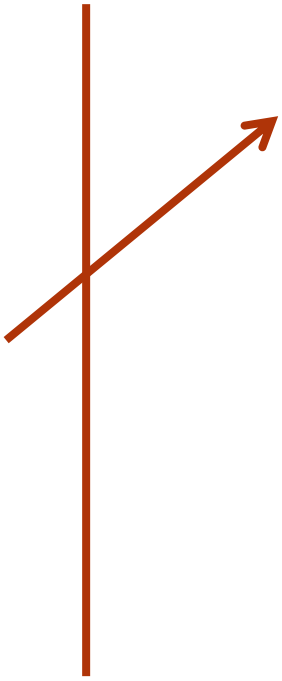


Using Dynamic Memory

```
int *num;
num = new int;
*num = 5;
```

- Now lets put some data in there using a dereference operator

Mem Addr	Variable
0x1004	
0x1000	num = 0x411C
The Stack	



0x411C	5

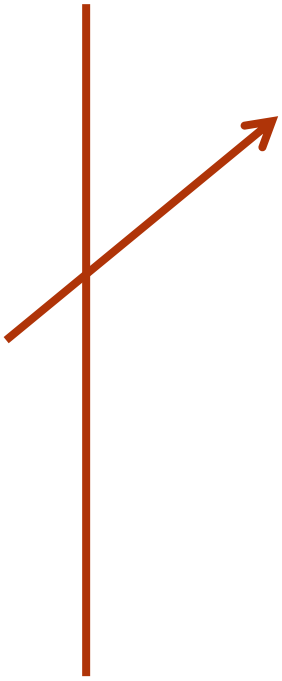
The Heap

Using Dynamic Memory

```
int *num;
num = new int;
*num = 5;
cout << *num << endl;
```

- The data can be fetched using the deref op.
- 5 will be printed to console

Mem Addr	Variable
0x1004	
0x1000	num = 0x411C
The Stack	



0x411C	5

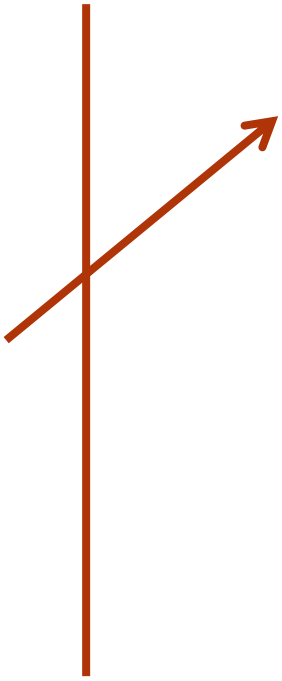
The Heap

Using Dynamic Memory

```
int *num;
num = new int;
*num = 5;
cout << *num << endl;
delete num;
```

- Finally the delete frees the memory in our heap
- Notice our pointer still points to 0x411C though

Mem Addr	Variable
0x1004	
0x1000	num = 0x411C
The Stack	



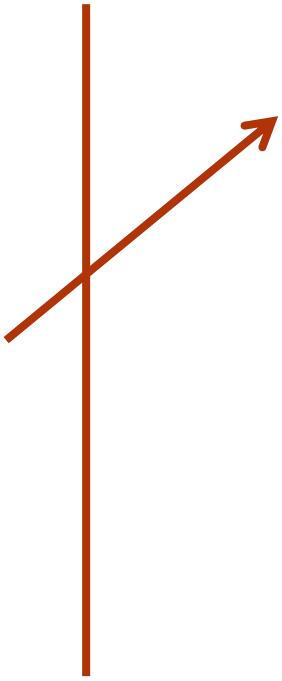
The Heap

Using Dynamic Memory

```
int *num;
num = new int;
*num = 5;
cout << *num << endl;
delete num;
```

- Once the memory is freed, it would be a terrible idea to continue to try and use it

Mem Addr	Variable
0x1004	
0x1000	num = 0x411C
The Stack	



The Heap

Dynamic Memory Assignment

- The new operator returns a pointer to the allocated memory
- Steps for Allocation

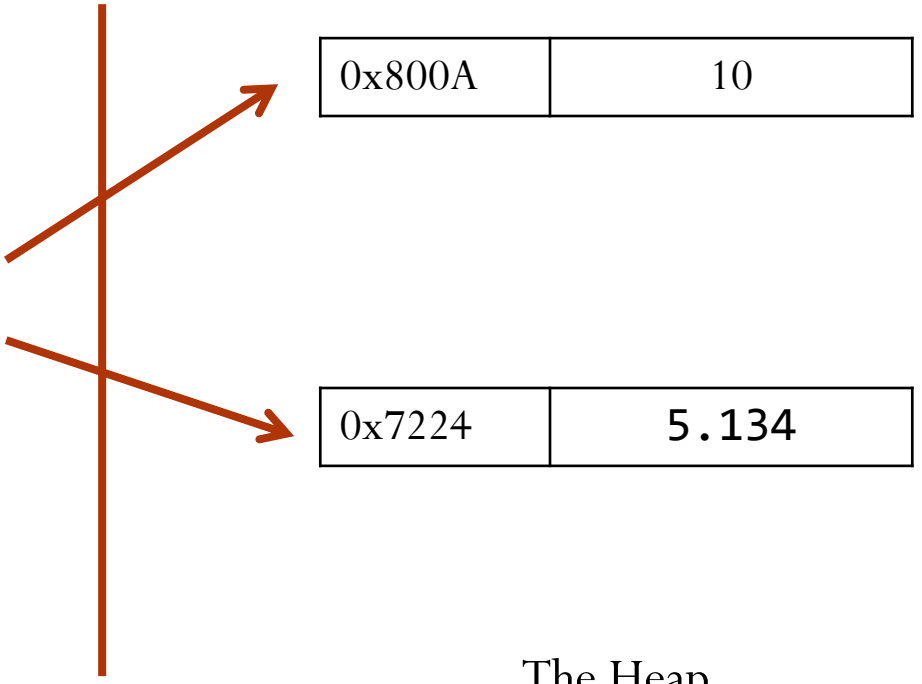
1. Declare pointer of data type of allocated memory
   - float *x;
2. Allocate memory using new keyword and the data type to allocate
   - x = new float;

Allocations

```
double *num1;
int *num2;
num1 = new double;
num2 = new int;
*num1 = 5.134;
*num2 = 10;
```

- Multiple allocations

Mem Addr	Variable
0x1004	num2 = 0x800A
0x1000	num1 = 0x7224
The Stack	

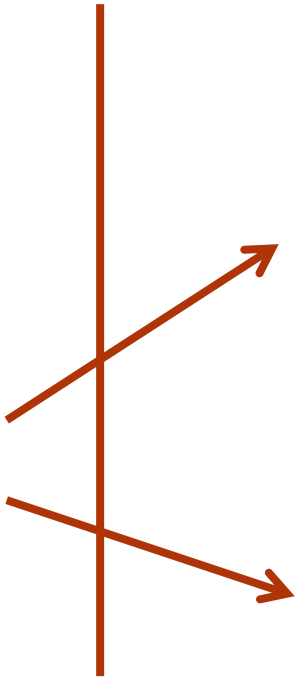


Deallocations

```
double *num1;
int *num2;
num1 = new double;
num2 = new int;
*num1 = 5.134;
*num2 = 10;
delete num1;
delete num2;
```

- Multiple deallocations

Mem Addr	Variable
0x1004	num2 = 0x800A
0x1000	num1 = 0x7224
The Stack	



The Heap

Dynamic Allocation of Arrays

- An array can be allocated on the heap by requesting a larger allocation
- This can be achieved by altering the syntax of the allocation

```
int *arr;
arr = new int[10];
```

- More generically

```
datatype *arr;
arr = new datatype[#_of_elements];
```

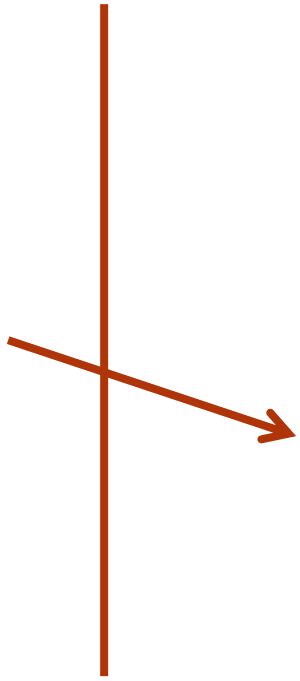
- Since the heap is dynamic, the number of elements can be a variable

Array Allocation

```
float *arr;
arr = new float[4];
```

- Here we have pointer to an allocation of 4 consecutive floats in our heap

Mem Addr	Variable
0x1004	
0x1000	arr = 0x7224
The Stack	



0x7230	
0x722C	
0x7228	
0x7224	

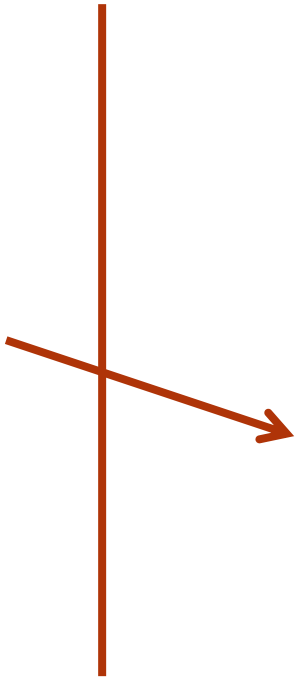
The Heap

Dynamic Array Assignment

```
float *arr;
arr = new float[4];
arr[0] = 1.337;
```

- Once the allocated, assigning values into a dynamic array is identical to that of a static

Mem Addr	Variable
0x1004	
0x1000	arr = 0x7224
The Stack	



0x7230	
0x722C	
0x7228	
0x7224	1.337

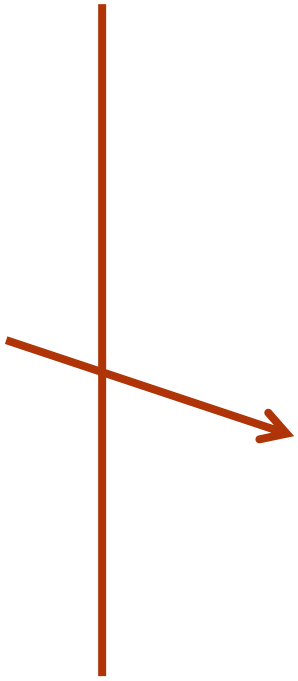
The Heap

Dynamic Array Access

```
float *arr;
arr = new float[4];
arr[0] = 1.337;
cout << arr[0] << endl;
```

- Again, fetching from a dynamic array does not differ to that of a static
- Here our code will print 1.337

Mem Addr	Variable
0x1004	
0x1000	arr = 0x7224
The Stack	



0x7230	
0x722C	
0x7228	
0x7224	1.337

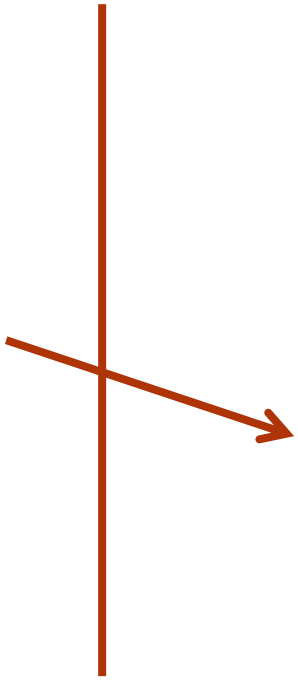
The Heap

Deallocation of Dynamic Array

```
float *arr;
arr = new float[4];
arr[0] = 1.337;
cout << arr[0] << endl;
delete [] arr;
```

- Empty brackets are used to indicate a deallocation of an array
- Brackets must be placed before the array name
- Do not place a size inside the brackets

Mem Addr	Variable
0x1004	
0x1000	arr = 0x7224
The Stack	



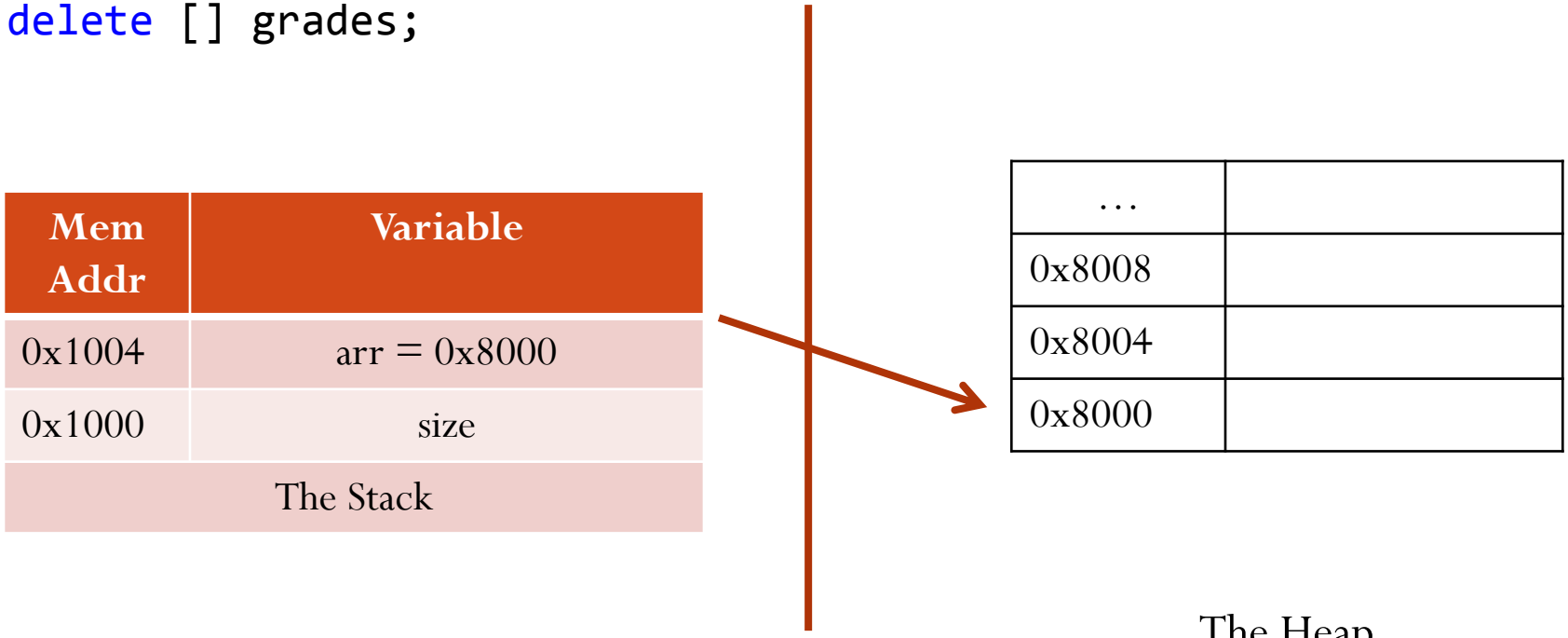
The Heap

Dynamic Array with Variable Size

```
int size;
double *grades;

cin >> size;
grades = new double[size];
...
delete [] grades;
```

- Here we have the user select the size of the array
- Allocation is based on the variable size



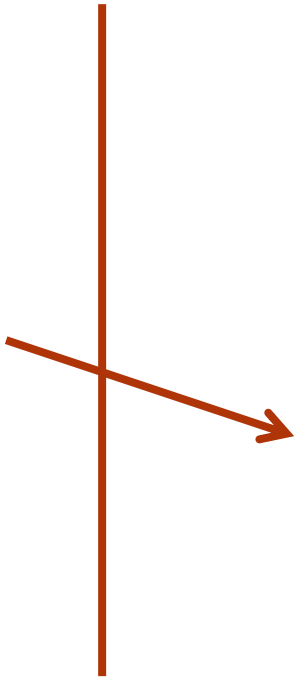
Dynamic Array with Variable Size

```
int size;
double *grades;

cin >> size;
grades = new double[size];
...
delete [] grades;
```

- Notice the allocation can occur at any time
- This time it must at least be after size is initialized to a value

Mem Addr	Variable
0x1004	arr = 0x8000
0x1000	size
The Stack	



…	
0x8008	
0x8004	
0x8000	

The Heap

Example 15.1

- Create a program which calculates the mean and variance of a set of numbers
- Mean and Variance are given by:

$$Mean(x) = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad Var(x) = \frac{1}{n}\sum_{i=1}^{n} (x_i - \mu)^2$$

- Prompt the user for how many numbers they wish to enter
  - Create an array that will accommodate the requested numbers
- Fill the array through user input
- Print the mean, and variance

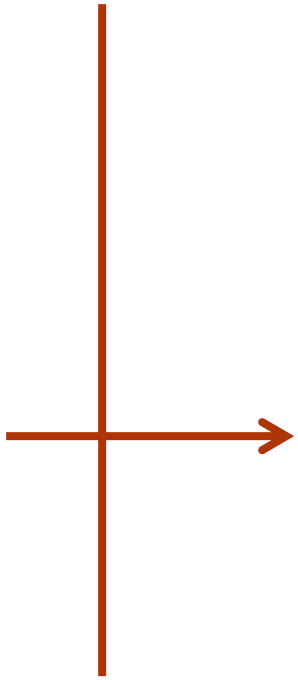
Reusing Pointers

```
int *arr;
arr = new int[4];
arr[0] = 5;
…
delete [] arr;

arr = new int[5];
arr[4] = 3;
…
delete [] arr;
```

- Allocate an array and place number 5 into it

Mem Addr	Variable
0x1004	
0x1000	arr = 0x8000
The Stack	



0x800C	
0x8008	
0x8004	
0x8000	5

The Heap

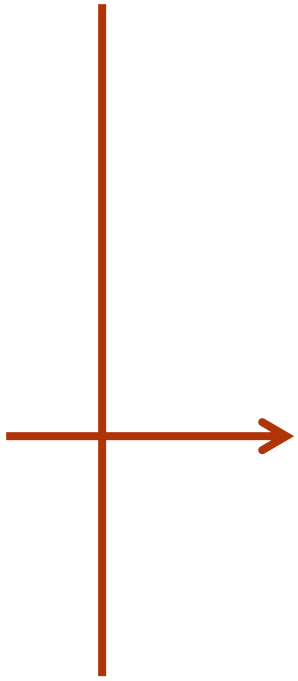
Reusing Pointers

```
int *arr;
arr = new int[4];
arr[0] = 5;
…
delete [] arr;

arr = new int[5];
arr[4] = 3;
…
delete [] arr;
```

- Deallocate the array after we are finished

Mem Addr	Variable
0x1004	
0x1000	arr = 0x8000
The Stack	



The Heap

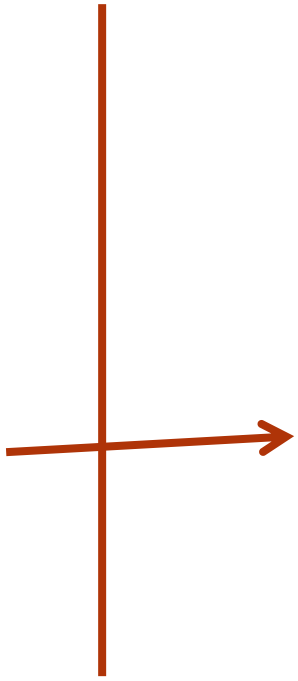
Reusing Pointers

```
int *arr;
arr = new int[4];
arr[0] = 5;
…
delete [] arr;

arr = new int[5];
arr[4] = 3;
…
delete [] arr;
```

- Pointer *arr* is reused to create another array of a different size
- Notice *arr* points to a new location now

Mem Addr	Variable
0x1004	
0x1000	arr = 0x6544
The Stack	



0x6554	3
0x6550	
0x654C	
0x6548	
0x6544	

The Heap

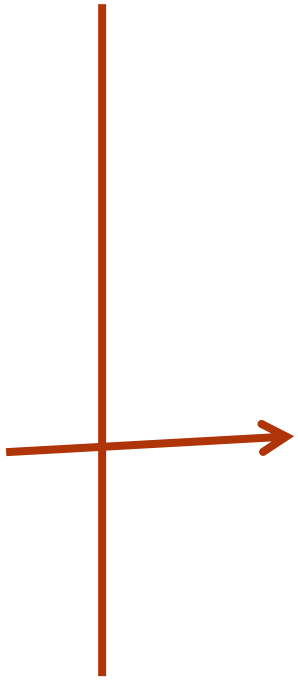
Reusing Pointers

```
int *arr;
arr = new int[4];
arr[0] = 5;
…
delete [] arr;

arr = new int[5];
arr[4] = 3;
…
delete [] arr;
```

- Pointer *arr* is reused to create another array of a different size
- Notice *arr* points to a new location now

Mem Addr	Variable
0x1004	
0x1000	arr = 0x6544
The Stack	



0x6554	3
0x6550	
0x654C	
0x6548	
0x6544	

The Heap

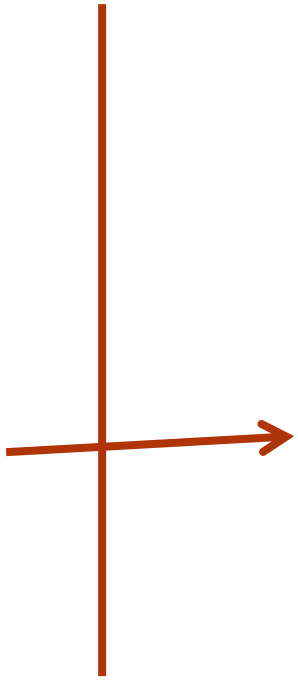
Reusing Pointers

```
int *arr;
arr = new int[4];
arr[0] = 5;
…
delete [] arr;

arr = new int[5];
arr[4] = 3;
…
delete [] arr;
```

- Finally we deallocate the array

Mem Addr	Variable
0x1004	
0x1000	arr = 0x6544
The Stack	



The Heap

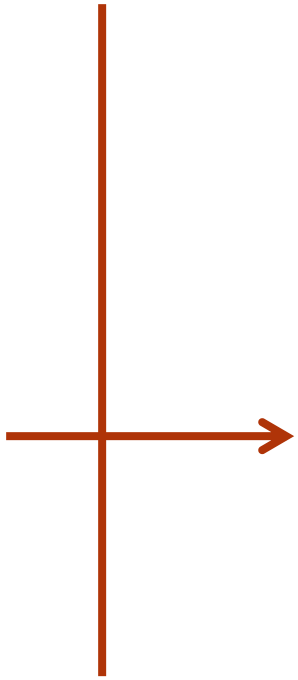
Memory Leak

```
int *arr;
arr = new int[4];
arr[0] = 5;
…

arr = new int[5];
arr[4] = 3;
…
delete [] arr;
```

- Here we modify the code to forget a deallocaton

Mem Addr	Variable
0x1004	
0x1000	arr = 0x8000
The Stack	



0x800C	
0x8008	
0x8004	
0x8000	5

The Heap

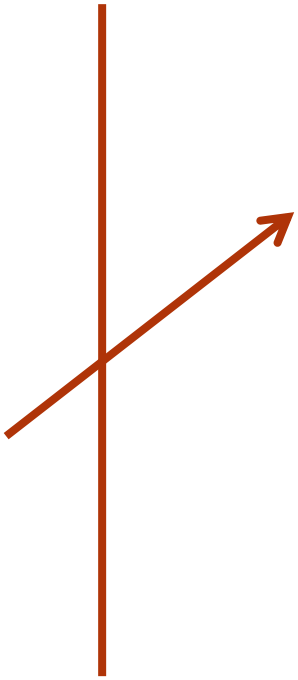
Memory Leak

```
int *arr;
arr = new int[4];
arr[0] = 5;
…

arr = new int[5];
arr[4] = 3;
…
delete [] arr;
```

- Here we perform another allocation but our previous allocation still exists

Mem Addr	Variable
0x1004	
0x1000	arr = 0x6554
The Stack	



0x6554	3
0x6550	
0x654C	
0x6548	
0x6544	

0x800C	
0x8008	
0x8004	
0x8000	5

The Heap

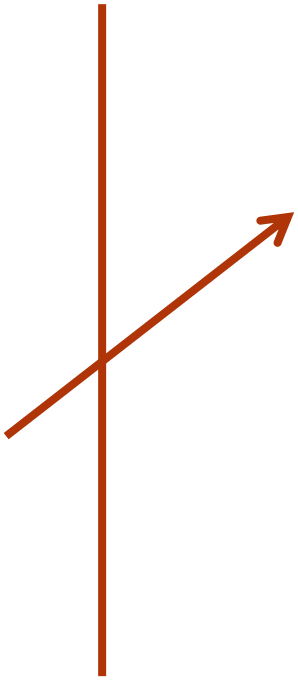
Memory Leak

```
int *arr;
arr = new int[4];
arr[0] = 5;
…

arr = new int[5];
arr[4] = 3;
…
delete [] arr;
```

- It is possible to delete the newest allocation but we have lost the pointer to our older one
- For the rest of the process's life the memory at 0x8000 will be reserved and unusable

Mem Addr	Variable
0x1004	
0x1000	arr = 0x6554
The Stack	



0x800C	
0x8008	
0x8004	
0x8000	5

The Heap

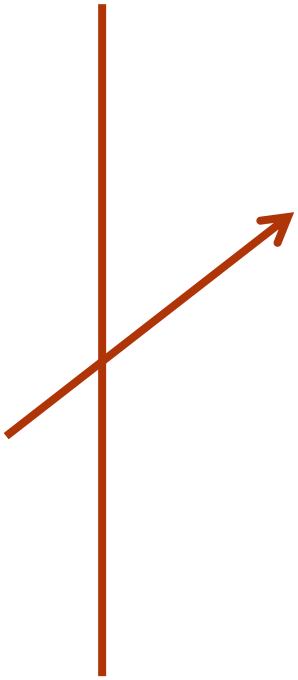
Memory Leak

```
int *arr;
arr = new int[4];
arr[0] = 5;
…

arr = new int[5];
arr[4] = 3;
…
delete [] arr;
```

- This is an example of a memory leak
- Rule of Allocations:
  - For every allocation there should be a deallocation

Mem Addr	Variable
0x1004	
0x1000	arr = 0x6554
The Stack	



0x800C	
0x8008	
0x8004	
0x8000	5

The Heap

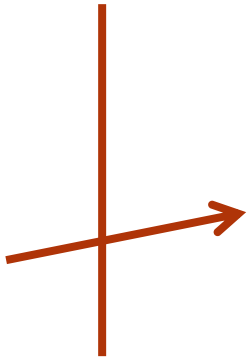
How bad can a memory leak be?

- Consider the following:

```
int *arr, i;

for(i = 0; i < 10000; i++)
    arr = new int[1000];
delete [] arr;
```

Mem Addr	Variable
0x1004	
0x1000	arr = …
The Stack	



…	
0x####	

…	
0x####	

…	
0x####	

…

…	
0x####	

The Heap

How bad can a memory leak be?

- Consider the following:

```
int *arr, i;

for(i = 0; i < 10000; i++)
   arr = new int[1000];
delete [] arr;
```

- How much memory is lost?
  - int is 32 bytes, each array is then 32 * 1000, 32000 bytes
  - 10000 – 1 arrays were allocated without a deallocation
  - Therefore 32000 * 99999 = 319968000 bytes
    - ≈ 320 MB

Memory Leaks

- While the example given may seem far fetched, often in practice memory allocation is performed in loops
- Such loops maybe executed millions of times
- Single instances of a memory leak are typically not an issue, however many repetitions of a single memory leak can spell disaster for your program
- Your program may run out of memory or worse effect the stability of your operating system

Memory Leaks

- Memory leaks are very hard to detect manually for the following reasons
  - They do not effect the functionality of your program
  - Only large leaks can be found by monitoring the memory footprint
  - Even if you know a leak is exists it is very difficult to determine which code is the culprit

- This is such an issue that many languages such as java will not allow the programmer to control memory allocation and deallocation

Memory Leak Strategy

- Besides being a perfect programmer there are tools to help solve this memory leak issue

- For Visual Studio there is a memory leak detection tool
  - Requires an additional define
  - Use through the debugger tool
  - A howTo for leak detection is uploaded to this lecture set
  - The tool is very limited compared to Valgrind

- For g++ based compilers (linux, mac)
  - Valgrind