CSCI 2270: Data Structures

Recitation #3 (Section 101)

Office Hours

- Name: Himanshu Gupta
  Email: himanshu.gupta@colorado.edu

- **Office Hours - 10am to 2pm on Mondays in ECAE 128**
  - In case that doesn't work for you, shoot me an email. We will figure something out that works for both of us.
  - Also, you can attend any TA's office hours. Timings are available on moodle in calendar.

- Does this work for you guys or should I reschedule? (Survey)

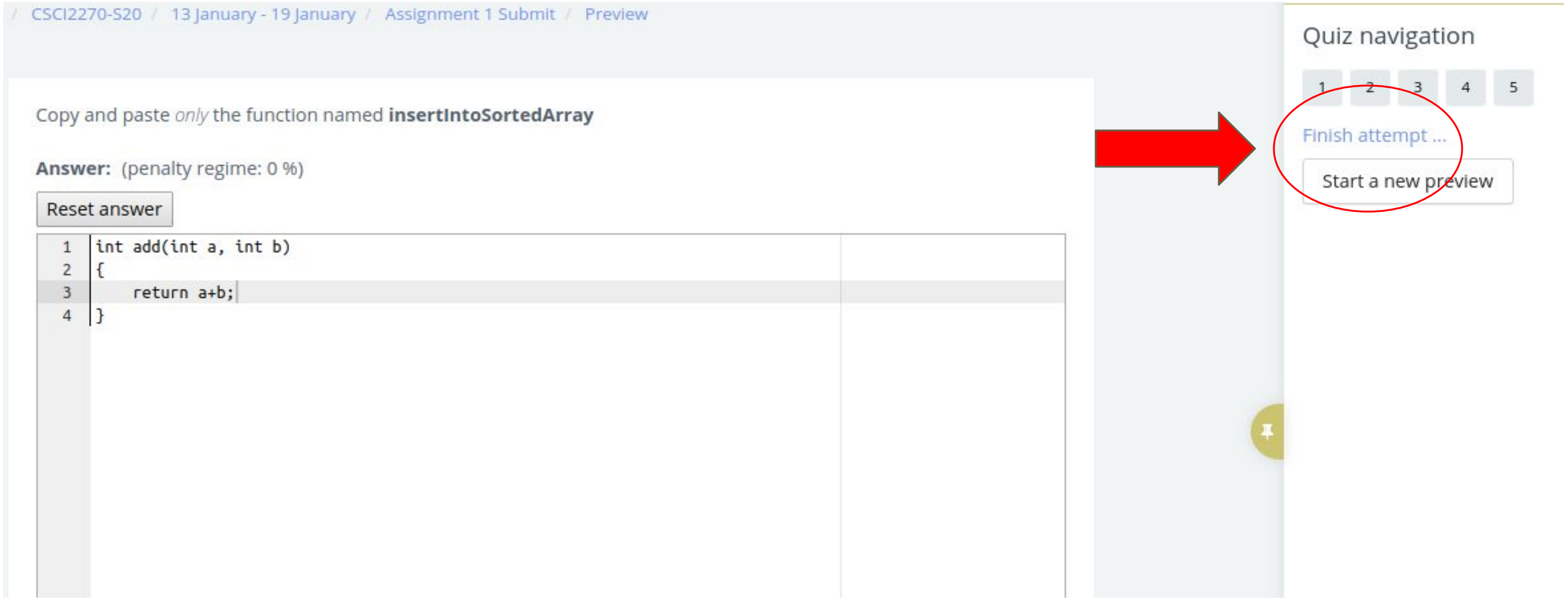
Logistics

- From this recitation onwards, you get points only after you have finished (or at least tried sincerely) the exercise assigned for that recitation.
  - Finish your code. If stuck, call us and ask for help.
  - Show your code to the TA or CA and ensure there is a tick against your name on the attendance sheet.

- Please ensure that your work has been noticed and you get points for it. Emails like “ I forgot to get my code checked” will not be entertained again and again.

- **Assignment 2 is due on Sunday, 2 February 2020, 11:59 PM.**
  GOOD LUCK!

Please click on “Finish Attempt” after you are done!


/ CSCI2270-S20 / 13 January - 19 January / Assignment 1 Submit / Preview
Quiz navigation
1
2
3
4
5
Finish attempt ...
Start a new preview
Reset answer

Copy and paste *only* the function named **insertIntoSortedArray**

Answer: (penalty regime: 0 %)

```
int add(int a, int b)
{
    return a+b;
}
```

Today’s Agenda

- Quick review on pointers and pass by value, reference, pointers.

- Review (20 ~ 30 mins)
    - Memory Allocation
    - Dynamic Memory
    - Freeing Memory
    - Linked Lists (will be covered either on Friday or next week)
    - Basic Sorting algorithm

- Exercise
    - Array doubling with dynamic memory allocation.

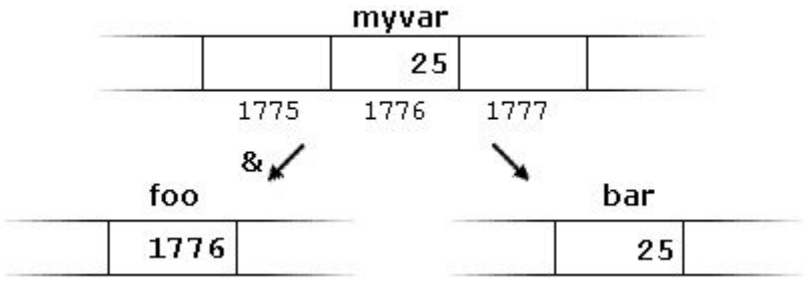
Quick Review: Pointers

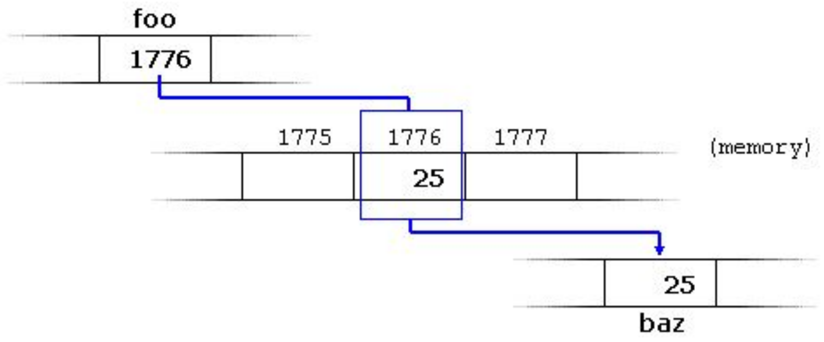
- Pointer is just a variable that stores a memory address.
- int* p (**or** int *p **or** int * p)
  - Initialize a pointer to an integer type
- &a
  - Address-of operator (of a variable a)
  - For example: **int* p = &a;**
- p
  - Address of where the pointer is pointing
- *p
  - Dereferencing operator i.e., refers to the value of the variable at the address stored in p

Pointers

- int myvar = 25;
  - cout<<myvar<<endl; (Output is 25)

- int * foo = &myvar;
  - cout<<foo<<endl; (Output is 1776)
  - cout<<*foo<<endl; (Output is 25)





Any Questions?

Discuss this with people on your table

```
#include <iostream>
using namespace std;
int main()
{
	int num[3];
	int* p;
	p = num;
	*p = 10;
	p++;
	*p = 20;
	p = &num[2];
	*p = 30;
	for (int i = 0; i < 3; i++)
		cout << num[i] << ", ";
	return 0;
}
```

What's the output?

a. 10, 20, 30,
b. 10, 20, 30
c. compile error
d. runtime error

Discuss this with people on your table

```
#include <iostream>
using namespace std;
int main()
{
	int arr[] = { 4, 9, 11, 17 };
	int* p = (arr + 1);
	cout << *arr + 2<<”,”;
	cout <<*p;
	return 0;
}
```

What’s the output?

a. 6, 5
b. 6, 9
c. 11,5
d. 11,9

Discuss this with people on your table

```
#include <iostream>
using namespace std;
int main()
{
        int a = 10,
        int* pa;
        pa = &a;
        cout << "a” << “   “ <<pa;
        return 0;
}
```

What’s the output?

a) 10, some hexadecimal value
b) 10, 10
c) a, some hexadecimal value
d) a,10

Discuss this with people on your table

Giving the following definition statements shown below, which are not valid among the options a, b, c, d, e and f?

```
int x;
float f;
int *pi;
float *pf;
```

a) x=10;   b) f=10;   c) pi=&x;   d) pi=*x;   e) pf=&x;   f) pf=&pi;

Quick Review: Pass-by-Value

- Creates a local copy of variables

```cpp
void add2 (int num)
{
    num = num + 2;
}

int main ()
{
    int a = 10;
    add2(a);
    cout << a;
}
```

Address	Stack
0x7ffeeacf8e2c	int a = 10
	...
SOME ADDRESS	int num = 12

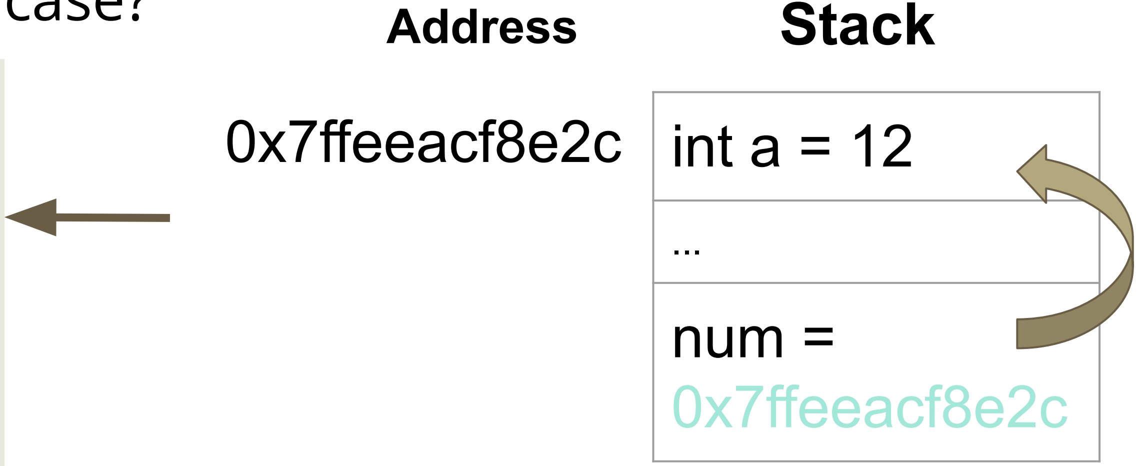
Terminal

```
10
```

Quick Review: Pass-by-Pointers

- What happens in this case?

```
void add2 (int * num)
{
    *num = *num + 2;
}

int main ()
{
    int a = 10;
    add2( &a );
    cout << a;
}
```



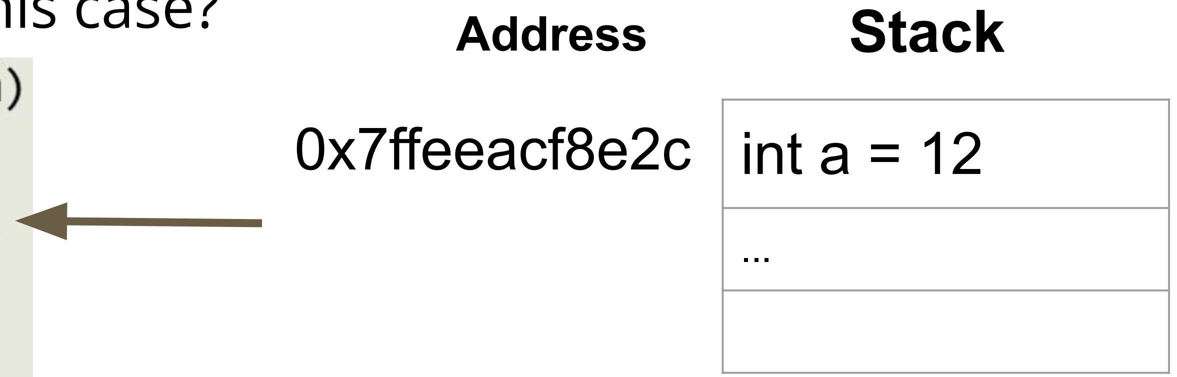
Terminal

```
12
```

Quick Review: Pass-by-Reference

- What happens in this case?

```cpp
void add2 (int &num)
{
    num = num + 2 ;
}

int main ()
{
    int a = 10 ;
    add2( a );
    cout << a;
}
```



Terminal

```
12
```

Any Questions?

Memory Allocation

- Variables represent storage space in the computer's memory.

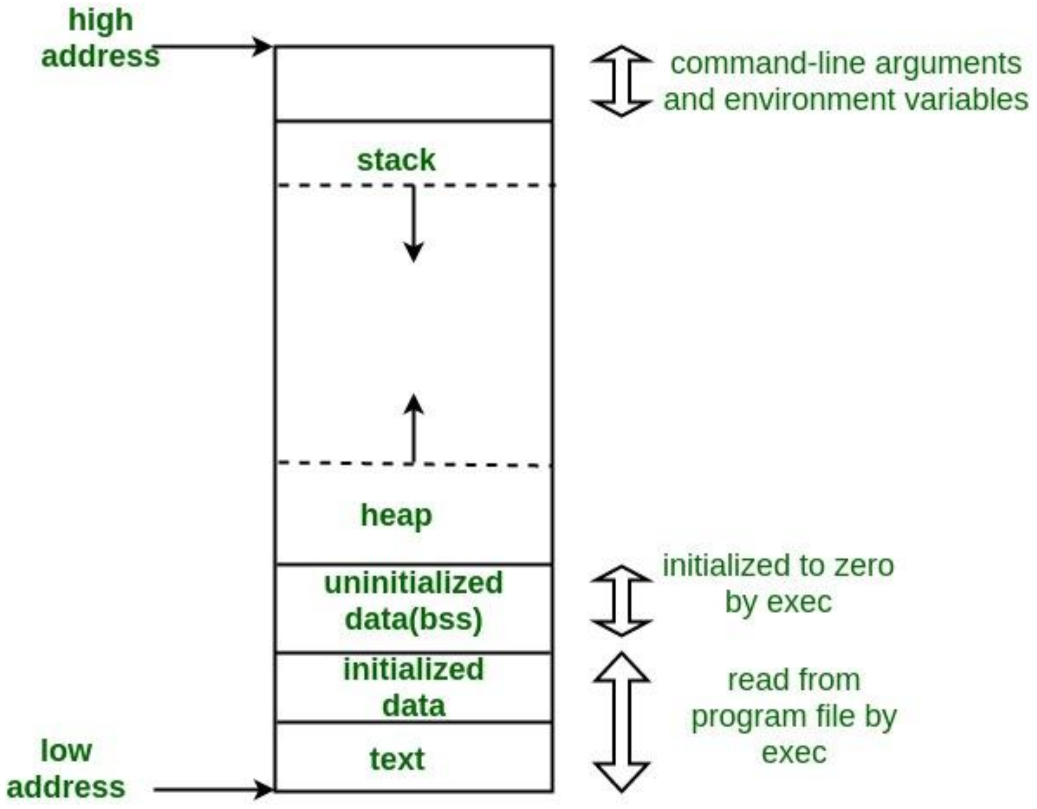
- Each variable has a convenient name like “a” or “num” in the code that we write. However, behind the scenes at runtime, each variable uses an area of the computer's memory to store its value.

- In C++ we interact with memory in a more low-level way. Not handling the memory properly can create a lot of problems, especially : **“SEGMENTATION FAULT (CORE DUMPED)”**. These errors are rather annoying, and can cause you a lot of trouble. They are often indicators that you’re trying to use/access memory you shouldn’t use/access.

Memory Allocation

- Static Memory Allocation
  - Memory for named variables is allocated by the compiler at compile time.
  - Exact size and type of storage must be known at compile time.
  - For standard array declarations, this is why the size has to be constant.
  - Performed using Stack Memory.
- Dynamic Memory Allocation
  - Memory is allocated to variables “on the fly” during run time.
  - Performed using Heap Memory.

Memory Layout of a C++ program


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

- Memory is actually sub-divided different areas, with different properties. We are only concerned about two types.
- Type 1: Stack Memory
- Type 2: Heap Memory

Static Memory Allocation (using Stacks)

- Stack memory keeps track of all the active functions (those that have been called but have not yet terminated) from the start of the program to the current point of execution, and handles allocation of all function parameters and local variables.

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
         z = add(x,y);
        return 0;
}
```

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
         z = add(x,y);
        return 0;
}
```

STACK (empty)

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
         z = add(x,y);
        return 0;
}
```

STACK
main()

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
        z = add(x,y);
        return 0;
}
```

STACK
int z
int y=15
int x=5
main()

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
        z = add(x,y);
        return 0;
}
```

STACK
add()
int y=15
int x=5
int z
main()

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
        z = add(x,y);
        return 0;
}
```

STACK
int b
int a
add()
int y=15
int x=5
int z
main()

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
        z = add(x,y);
        return 0;
}
```

STACK
int b
int a
add()
int y=15
int x=5
int z
main()

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
        z = add(x,y);
        return 0;
}
```

STACK
add()
int y=15
int x=5
int z
main()

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
        z = add(x,y);
        return 0;
}
```

STACK
add()
int y=15
int x=5
int z
main()

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
        z = add(x,y);
        return 0;
}
```

STACK
int y=15
int x=5
int z
main()

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
        z = add(x,y);
➡       return 0;
}
```

STACK
int y=15
int x=5
int z
main()

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
        z = add(x,y);
        return 0;
}
```

STACK
main()

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
        z = add(x,y);
        return 0;
}
```

STACK
main()

Static Memory Allocation (using Stacks)

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
        z = add(x,y);
        return 0;
}
```

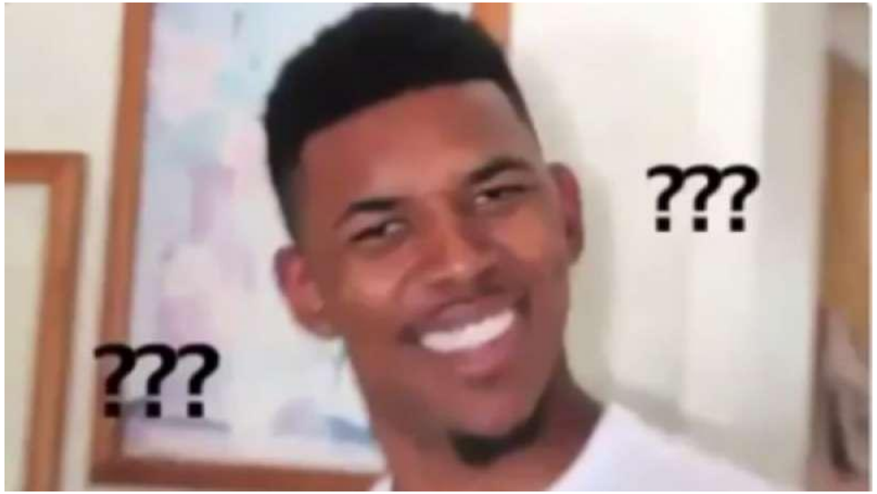
STACK (empty)

Any Questions?

Dynamic Memory Allocation

Dynamic Memory Allocation

Question: Why do I need it when Static Memory allocation technique already does the job of assigning memory to my variables?



Dynamic Memory Allocation

Question: Why do I need it when Static Memory allocation technique already does the job?

Answer:

- Well, let’s say I want you to read numbers from a file and store in an array.
    - What’s the first question that comes to your mind?

Dynamic Memory Allocation

Question: Why do I need it when Static Memory allocation technique already does the job?

Answer:

- Well, let’s say I want you to read numbers from a file and store in an array.
    - What’s the first question that comes to your mind ?
        - How many elements do I need to store in my array? (you will use that information to define your array, right?)

Dynamic Memory Allocation

Question: Why do I need it when Static Memory allocation technique already does the job?

Answer:

- Well, let’s say I want you to read numbers from a file and store in an array.
  - What’s the first question that comes to your mind ?
    - How many elements do I need to store in my array? (you will use that information to define your array, right?)
    - If I am not willing to share that information. You have now reached a dead end. What do you do?

Dynamic Memory Allocation

Question: Why do I need it when Static Memory allocation technique already does the job?

Answer:

- Well, let’s say I want you to read numbers from a file and store in an array.
  - What’s the first question that comes to your mind ?
    - How many elements do I need to store in my array? (you will use that information to define your array, right?)
    - If I am not willing to share that information. You have now reached a dead end. What do you do?
    - One possible solution - Define a really big array. However, you are probably wasting a lot of memory here. THIS IS UNDESIRABLE!!

Dynamic Memory Allocation

Question: Why do I need it when Static Memory allocation technique already does the job?

Answer:

- Well, let's say I want you to read numbers from a file and store in an array.
  - What's the first question that comes to your mind ?
    - How many elements do I need to store in my array? (you will use that information to define your array, right?)
    - If I am not willing to share that information. You have now reached a dead end. What do you do?
    - One possible solution - Define a really big array. However, you are probably wasting a lot of memory here. THIS IS UNDESIRABLE!!
    - Dynamic Memory Allocation to the rescue.

Dynamic Memory Allocation (Using Heap)

- In order to use the heap memory in C++ we use the **“new”** and **“delete”** keywords.
    - “new” for allocating memory on the heap
    - “delete” for deallocating/freeing memory from the heap

- Creating variables on the heap is very different from creating variables on the stack. On stack memory, variables get actual names and addresses. On heap memory, they don’t get names, JUST ADDRESSES.

- Instead, we use a pointer to allocate the memory for the variable.

Dynamic Memory Allocation

- “new” operator

  I just mentioned that **“On heap memory, variables don’t get names, JUST ADDRESSES “**

  - How do you access those variables then?
  - **ANSWER: Using pointers, right? (because pointers are used to store addresses and can be perfectly used here)**

Dynamic Memory Allocation

- “new” operator

  I just mentioned that **“On heap memory, variables don’t get names, JUST ADDRESSES “**
  - How do you access/use those variables then?

Dynamic Memory Allocation

- "new" operator

int* p1; → Stored on the stack

STACK	Address
int* p1	0x7ffd9181ddec

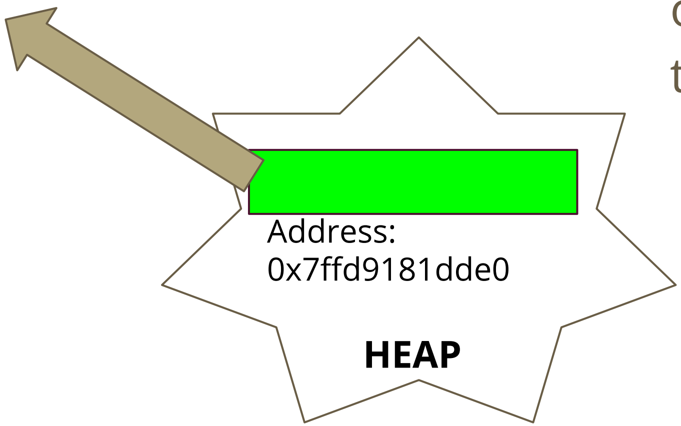
Dynamic Memory Allocation

- “new” operator

int* p1; → Stored on the stack

STACK	Address
int* p1	0x7ffd9181ddec

p1 = new int; → Allocates a memory block on Heap of size int and stores the address of that block in pointer p1



```
cout<<p1;
```

Output: 0x7ffd9181dde0

Dynamic Memory Allocation

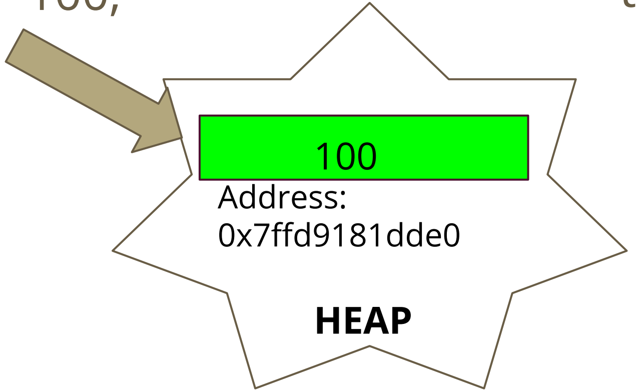
- “new” operator

int* p1; → Stored on the stack

STACK	Address
int* p1	0x7ffd9181ddec

p1 = new int; → Allocates a memory block on Heap of size int and stores the address of that block in pointer p1

*p1 = 100;



cout<<p1<< “   “ << *p1;

Output: 0x7ffd9181dde0   100

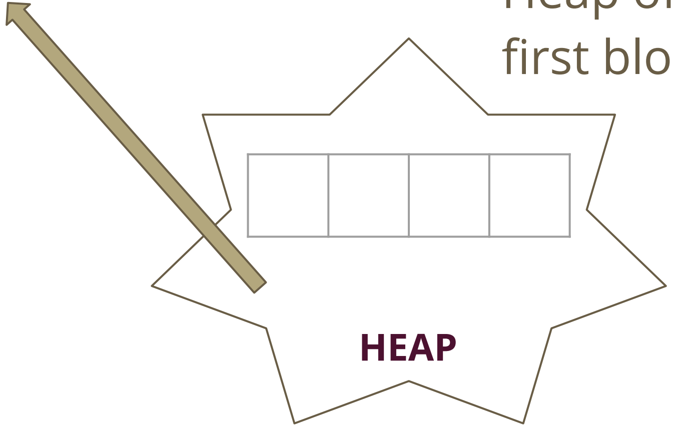
Dynamic Memory Allocation

- You can declare arrays dynamically as well. They are allocated memory on the heap. HOW DO YOU DO THAT?

STACK	Address
int* p1	0x7ffd9181ddec

int* p1; $\Rightarrow$ Stored on the stack

p1 = new int[4]; $\Rightarrow$ Allocates a contiguous memory segment on Heap of size int * 4 and stores the address of the first block in that segment in pointer p1



```
cout<<p1;

Output:  0x7ffd9181dde0
```

Any Questions?

Dynamic Memory Deallocation

- In Static Memory Allocation technique using Stacks, we observed that a variable is removed from the stack as soon as the function in which it was declared terminates (or when its scope ends)

  Since they are automatically deallocated from the stack, they are also called automatic variables.

- Unlike automatic variables, dynamically allocated variables don't get deallocated automatically. We must do this manually.

- We do this using the **"delete"** operator.

Dynamic Memory Deallocation

- **“delete”** operator

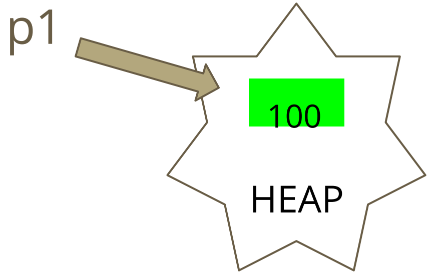
```
int* p1;   // pointer p1 stored on the STACK

p1 = new int;  // pointer p1 stores the address of a memory block on HEAP

*p1 = 100; // modifies the value pointed by p1 to 100

delete p1; // deallocates/frees the memory back to HEAP
           // It doesn’t delete the pointer variable p1
           // p1 is still available on the STACK.
```

Dynamic Memory Deallocation

- **“delete”** operator (Visualization)

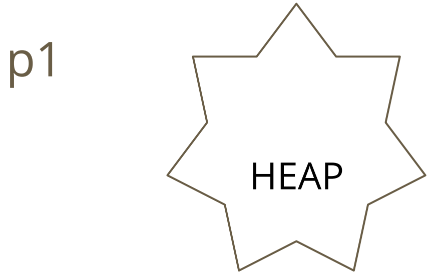
  int* p1; // pointer p1 stored on the STACK
  p1 = new int; // pointer p1 stores the address of a memory block on HEAP
  *p1 = 100; // modifies the value pointed by p1 to 100

STACK	ADDRESS
int* p1	0x7ffd9181ddec



delete p1;

STACK	ADDRESS
int* p1	0x7ffd9181ddec



Dynamic Memory Deallocation

- **“delete”** operator

  How to delete a dynamically allocated array?

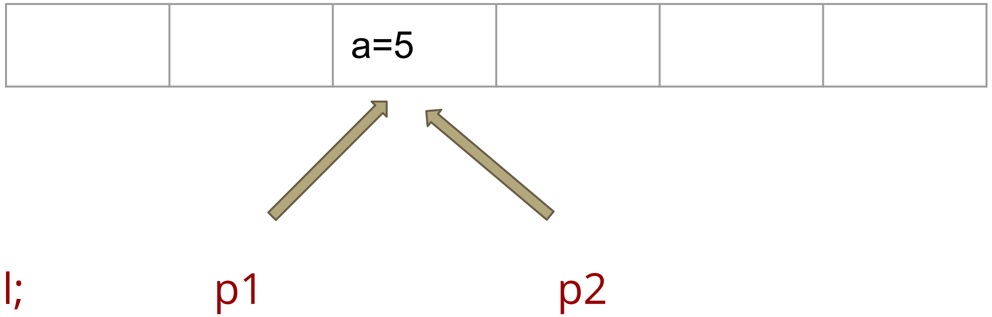
```
int* p1;
p1 = new int[100];
int i=0;
for(i=0;i<100;i++){
        cout<<p1+i;
}
delete [] p1;
```

What happens if you don’t deallocate?

CASE1 - Dangling Pointer (a pointer which still exists, even though the object it pointed to no longer exists)

```
#include<iostream>
using namespace std;
int main(){
        int *p1 = new int;
        *p1 = 5;
        int *p2 = p1;
        cout<<p1<<”   “<<*p1<<endl;
        delete p1;
        p1 = nullptr;
        cout<<p2<<”   “<<*p2<<endl;
        return 0; }
```

HEAP MEMORY (



What happens if you don’t deallocate?

- CASE 2 - Memory Leak - A memory leak occurs in C++ when you allocate memory dynamically and deallocate it.
- If a program has memory leaks, then its memory usage is satirically increasing since all systems have limited amount of memory and memory is costly. Hence it will create problems.

```
// function with memory leak
void func_to_show_mem_leak() {
        int* ptr = new int(5);
        return;
}
int main() {
        // Call the function to get the memory leak
        func_to_show_mem_leak();
        return 0;
}
```

Takeaway from new and delete operator

- “new” operator gives you the power to allocate memory dynamically at run time.

- However, always ensure that you use the “delete” operator to deallocate memory to avoid errors.

Today’s exercise

- Array Doubling

- Download Recitation3 Writeup from moodle.
    - Look at file exercise.cpp
        - It takes a filename as a command line argument and stores elements from that file into an array.
        - This array is resized dynamically.
        - You just have to implement the array doubling function called “resize”.

Let’s look at the code

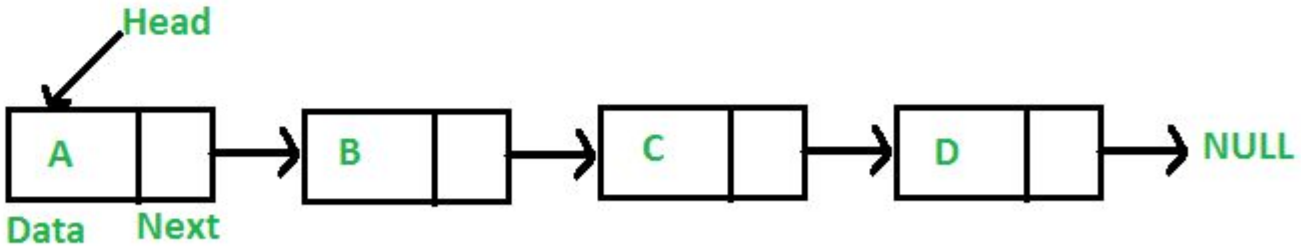
BubbleSort?

Refer to your Recitation3 Write-up

Linked Lists (Introduction)

- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.
    - Every element in a linked list is called a **“node”**
    - Most basic Linked list has a node with two objects in it.
        - int data; (value stored in that node)
        - *node next; (pointer to the next node)

- The elements in a linked list are linked using pointers as shown in the below image:



Nodes in Linked Lists

Using Structs

```
struct node
{
   int data;
   node *next;
};
```

Using Class

```
class Node
{
Public:
      int data;
      Node* next;
};
```

I have arrays to store data. Why do I need linked lists?

- Well, arrays always allocate memory in contiguous blocks. This leads to inefficient/ sub-optimal CPU memory usage. LLs solve this problem.
  - For example:

a[0]	a[1]	a[2]	c	d	b[0]	b[1]	b[2]	b[3]	b[4]	e

  - Now, let’s say you delete c, d and e

a[0]	a[1]	a[2]			b[0]	b[1]	b[2]	b[3]	b[4]	

  - These intermediate empty memory blocks can never be used and this is wastage of your CPU memory which is undesirable.

- Also, arrays have predefined sizes. LLs have no predefined size or restriction (well, can’t be more than the memory size of course)

Office Hours

- Name: Himanshu Gupta
  Email: himanshu.gupta@colorado.edu

- **Office Hours - 10am to 2pm on Mondays in ECAE 128**
  - In case that doesn't work for you, shoot me an email. We will figure something out that works for both of us.
  - Also, you can attend any TA's office hours. Timings are available on moodle in calendar.