National University of Computer and Emerging Sciences

OBJECT-ORIENTED PRORAMMING

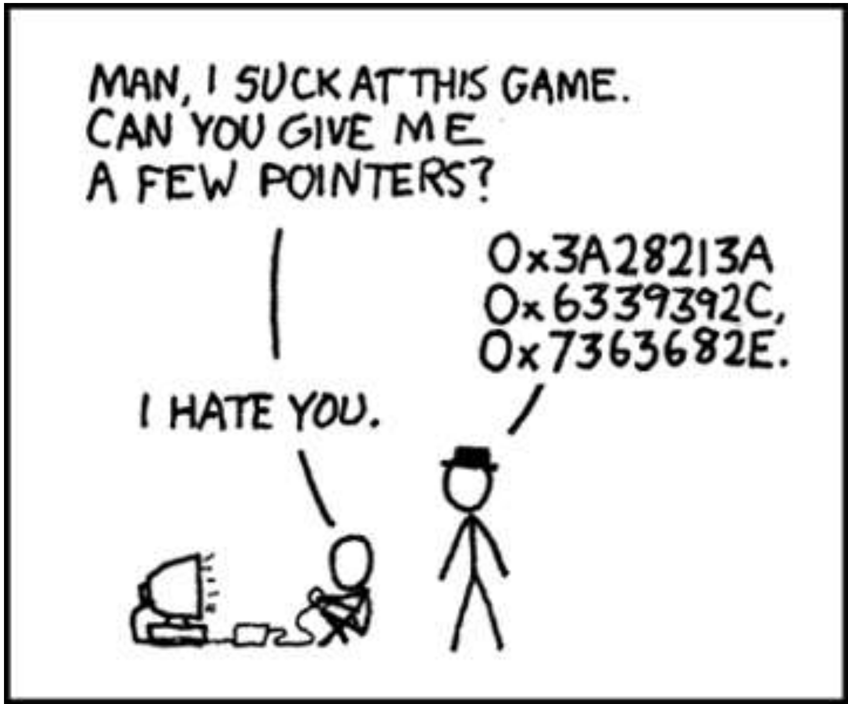
Summer 2023

Pir Sami Ullah Shah

Lecture # 2 Pointers

Pointers

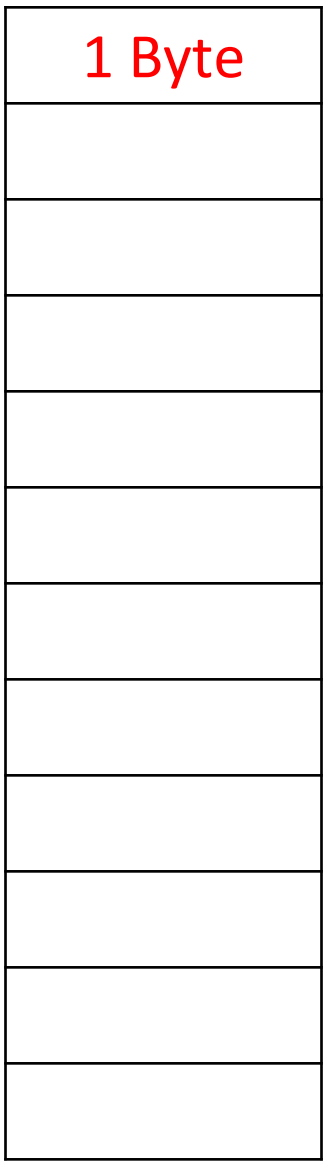
- Introduction
- Address of and Dereference operator
- Memory Leaks and dangling pointers
- Void pointer
- Casting pointers



Memory (RAM)

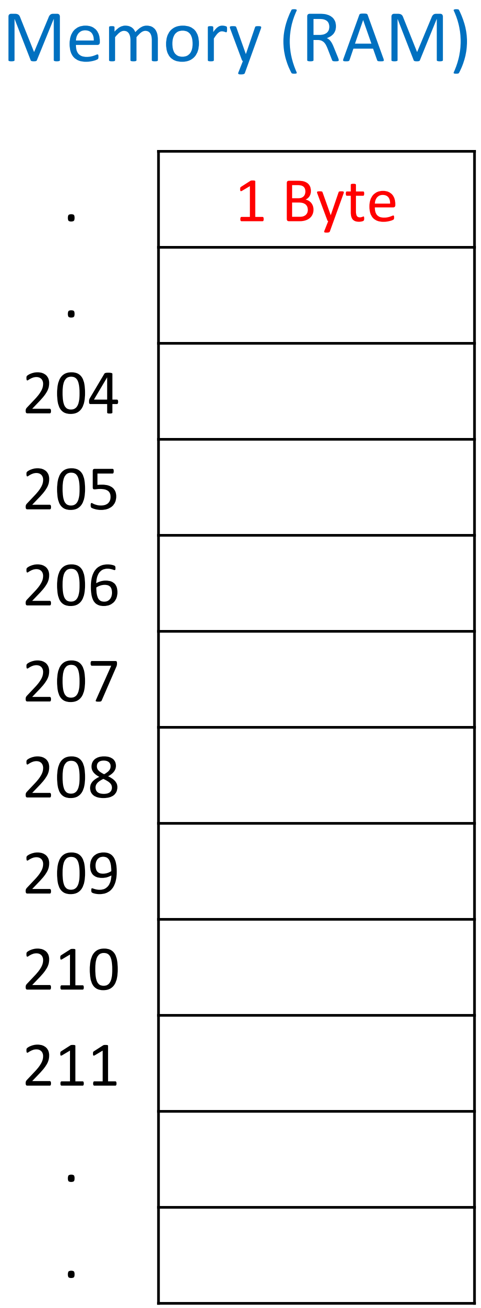
Program Memory

- Each block in memory represents 1 byte



Program Memory

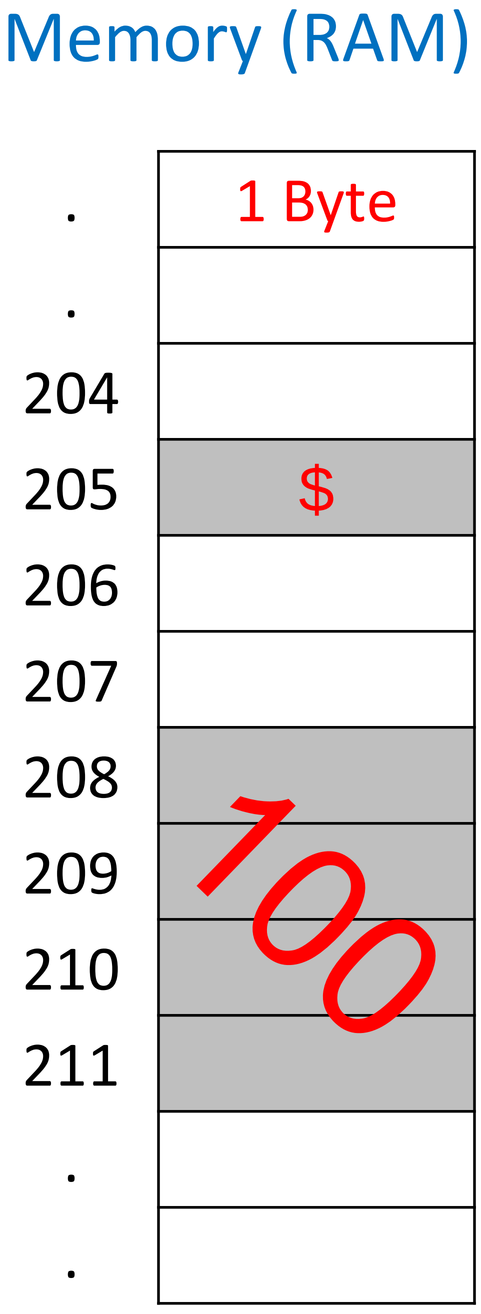
- Each block in memory represents 1 byte
- Each byte in memory has an address
- Whenever a variable is declared, the computer reserves some memory for it.



Program Memory

Integers are stored in 4 bytes
Characters stored in 1 byte

```
int num = 100;
char letter = ‘$’;
```



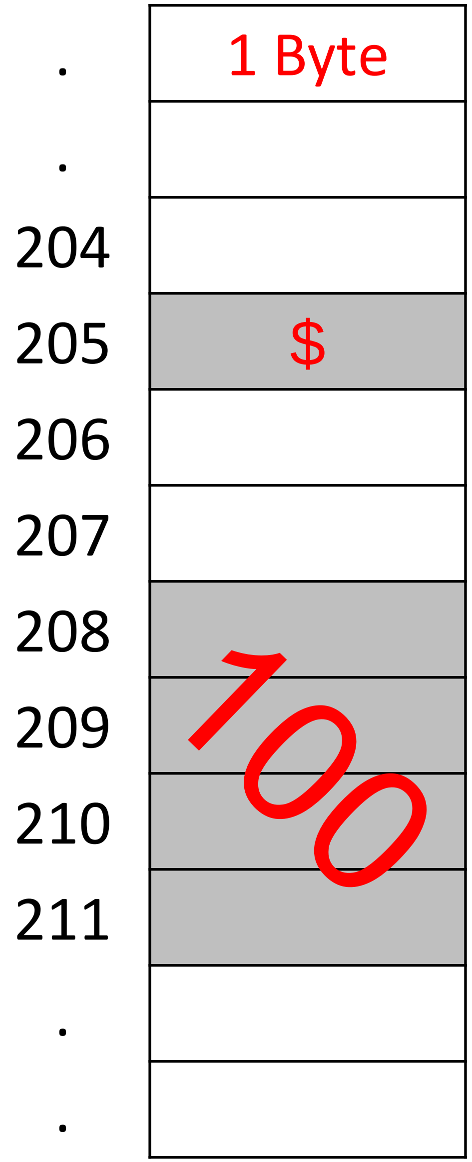
Program Memory

Integers are stored in 4 bytes
Characters stored in 1 byte

```
int num = 100;
char letter = '$';
```

Can we find the memory address of these variables??

Memory (RAM)



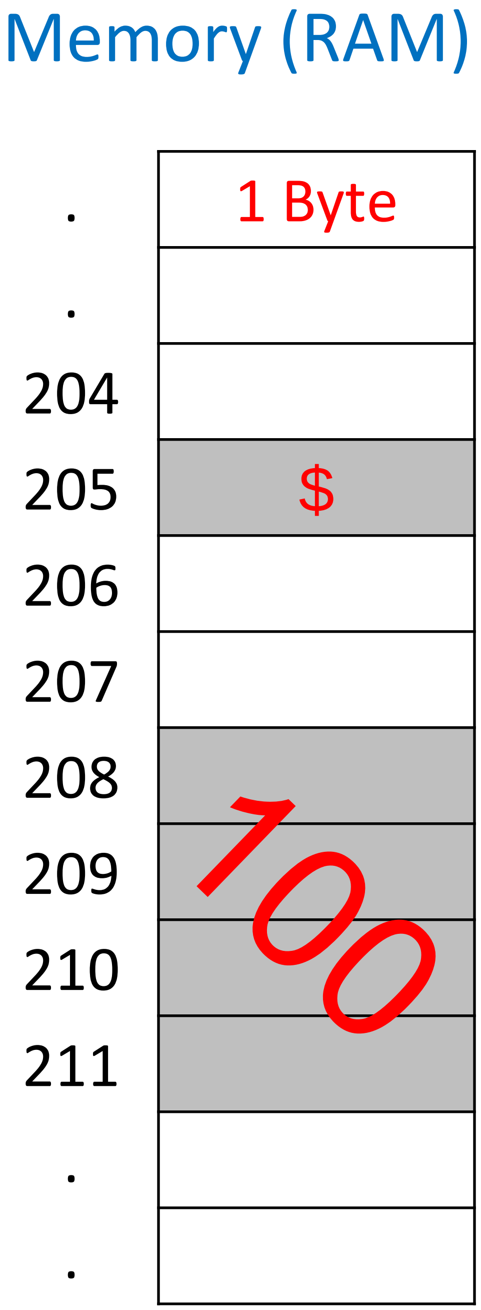
Program Memory

Integers are stored in 4 bytes

```
int num = 100;
char letter = '$';
```

Can we find the memory address of these variables??

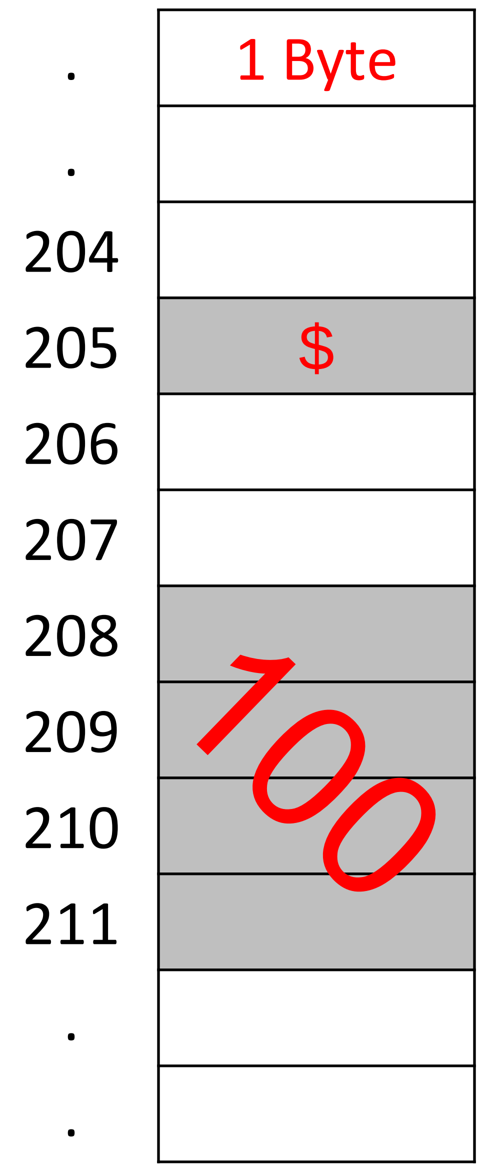
```
cout<< &num;
```



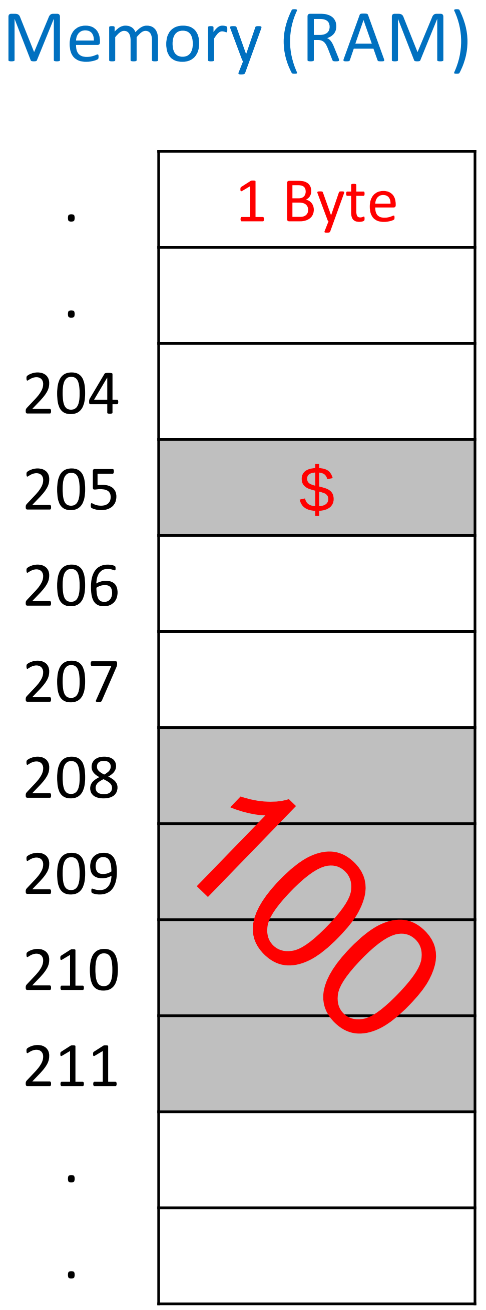
Memory (RAM)

Program Memory

```
int num = 100;
char letter = ‘$’;

cout<< &num;
```



Variable Name	Data Type	Starting Address
num	int	208
letter	char	205



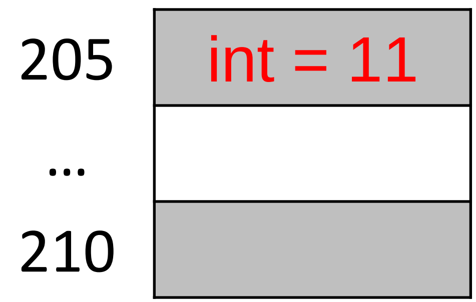
Program Memory

```
int num = 100;
char letter = '$';

cout<< &num; //prints 208
```

Variable Name	Data Type	Starting Address
num	int	208
letter	char	205

Pointer Variables

- Pointer variable (pointer): a variable that holds an address




```
int x = 11;
cout<< &x;
```

dataType * variable Name;

```
int *ptr;
ptr = &x;
```

Pointer Variables

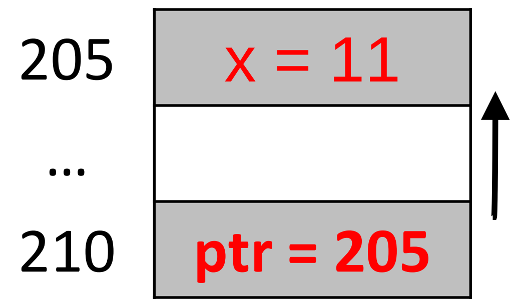
- Pointer variable (pointer): a variable that holds an address

205	x = 11
…	
210	**ptr = 205**

```
int x = 11;
cout<< &x;

int *ptr;
ptr = &x;
```

Pointer Variables

- Because a pointer variable holds the address of another piece of data, it "points" to the data




```
int x = 11;
cout<< &x;

int *ptr;
ptr = &x;
```

The dereference Operator

- The dereference operator (*) dereferences a pointer.

- It allows you to access the value that the pointer points to.

```
int x = 25;
int *intptr = &x;
cout << *intptr << endl;
```

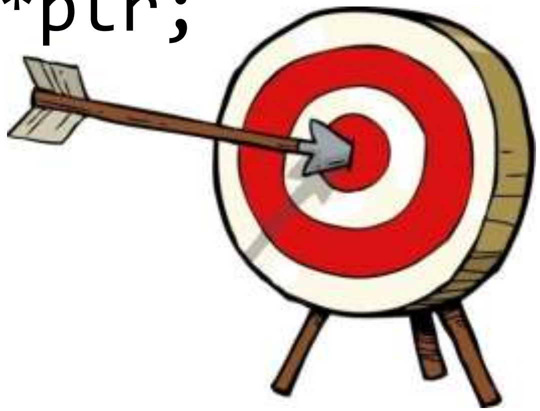
This prints 25.

The dereference Operator

- Note that the * sign can be confusing here, as it does two different things:

- When used in declaration e.g. `string* ptr`, it creates a pointer variable.

- When not used in declaration, it act as a dereference operator e.g `cout << *ptr;`



The dereference Operator

- Another Example:

```
int v1 = 99;
int* p = &v1;
cout<<" P points to the value: "<< *p;
```

Dereferencing Pointer Example

```
int v1 = 0;
int* p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

v1 and *p1 now refer to
the same variable

Output:

```
42
42
```

Pointer Assignment and Dereferencing

- Assignment operator (**=**) can be used to assign value of one pointer to another

- Pointer stores addresses so **p1 = p2** copies an address value into another pointer

```
int v1 = 55;
int* p1 = &v1;
int* p2;
p2=p1;
cout << *p1 << endl;
cout << *p2 << endl;
```

```
Output:

        55
        55
```

Dynamic Memory Allocation

- Used when space requirements are unknown at compile time

- Most of the time the amount of space required is unknown at compile time

- **Dynamic Memory Allocation** (DMA):-
  - We can allocate/deallocate memory at runtime or execution time.

Differences between Static and Dynamic Memory Allocation

- Dynamically allocated memory is kept on the memory heap (also known as the free store)

- Dynamically allocated memory cannot have a name, it must be referred to

- *Declarations* are used to statically allocate memory,
  - the ***new*** operator is used to dynamically allocate memory

Dynamic Memory Allocation

- Heap management in C++ is explicit:

```
ptr = new datatype;
//allocate memory for one element

ptr = new datatype[size];
//allocate memory for fixed number of elements
```

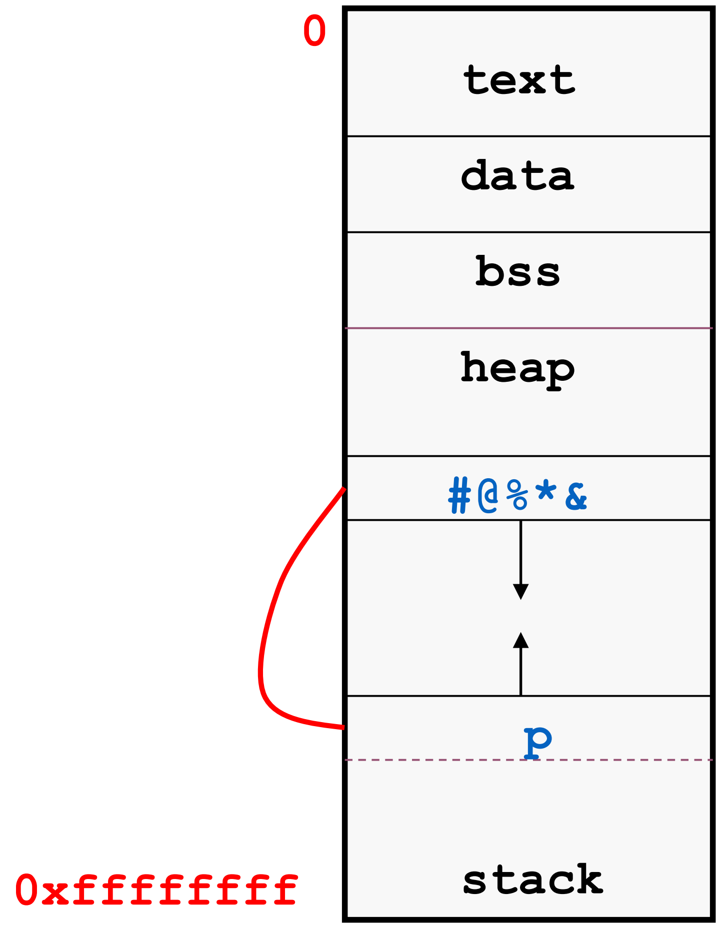
Dynamic Memory Allocation

- Heap management in C++ is explicit:

```
delete ptr;
//deallocate memory for one element

delete [] ptr;
//deallocate memory for array
```

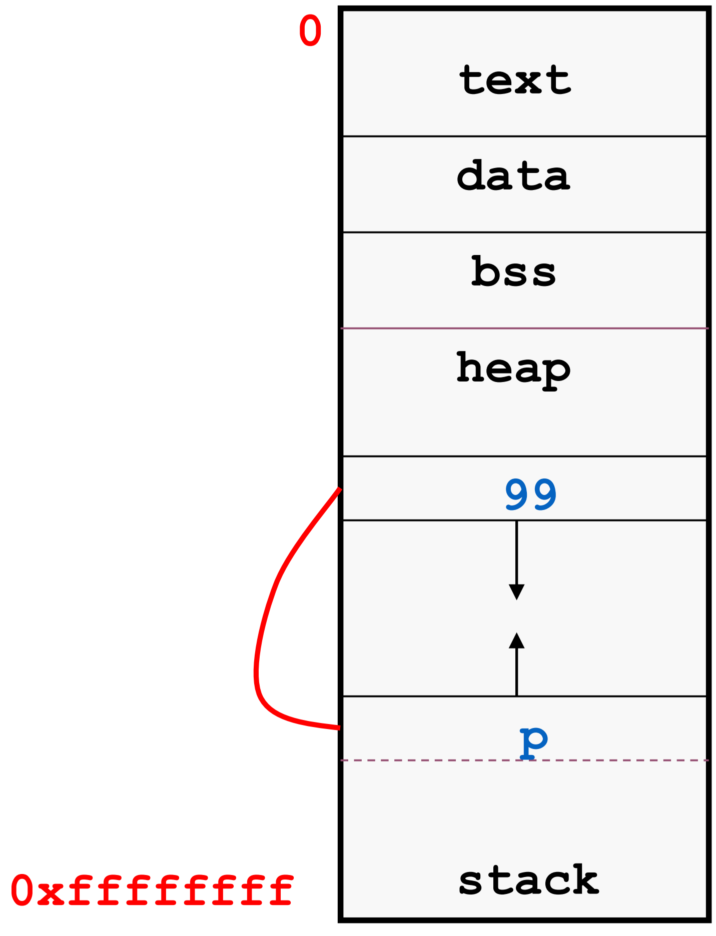
Example

```
int main()
{
  int *p;

  p = new int;

 *p = 99;

  return 0;
}
```



Example

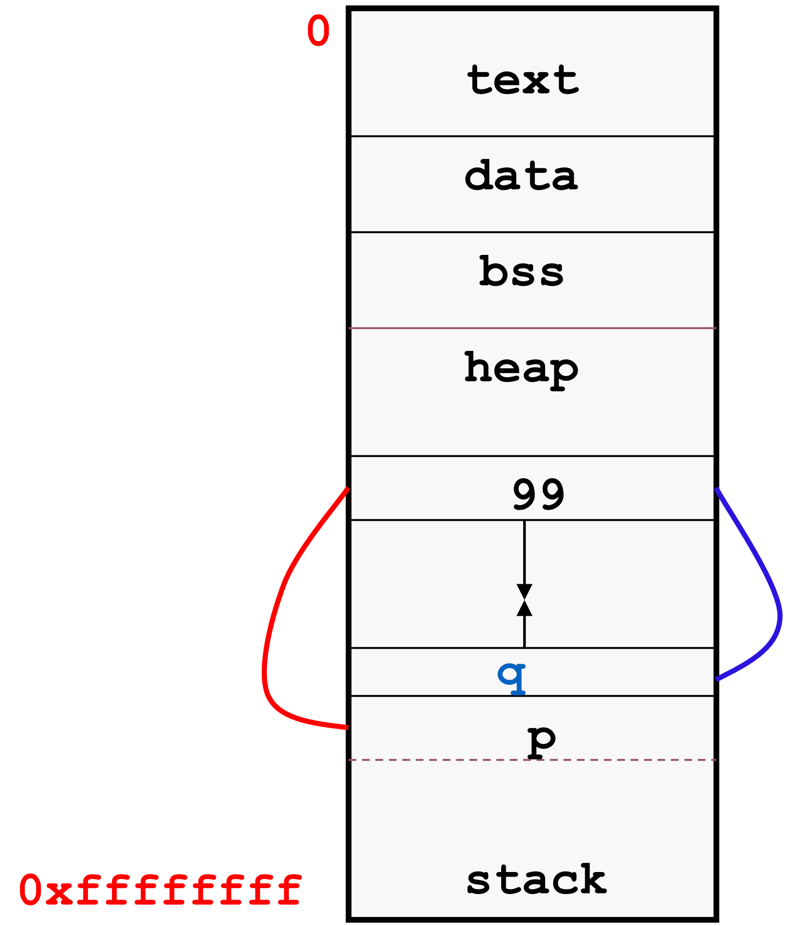
```
int main()
{
  int *p;

  p = new int;

 *p = 99;

  return 0;
}
```



Aliasing

If two pointers have the same value, they are aliases of each other.

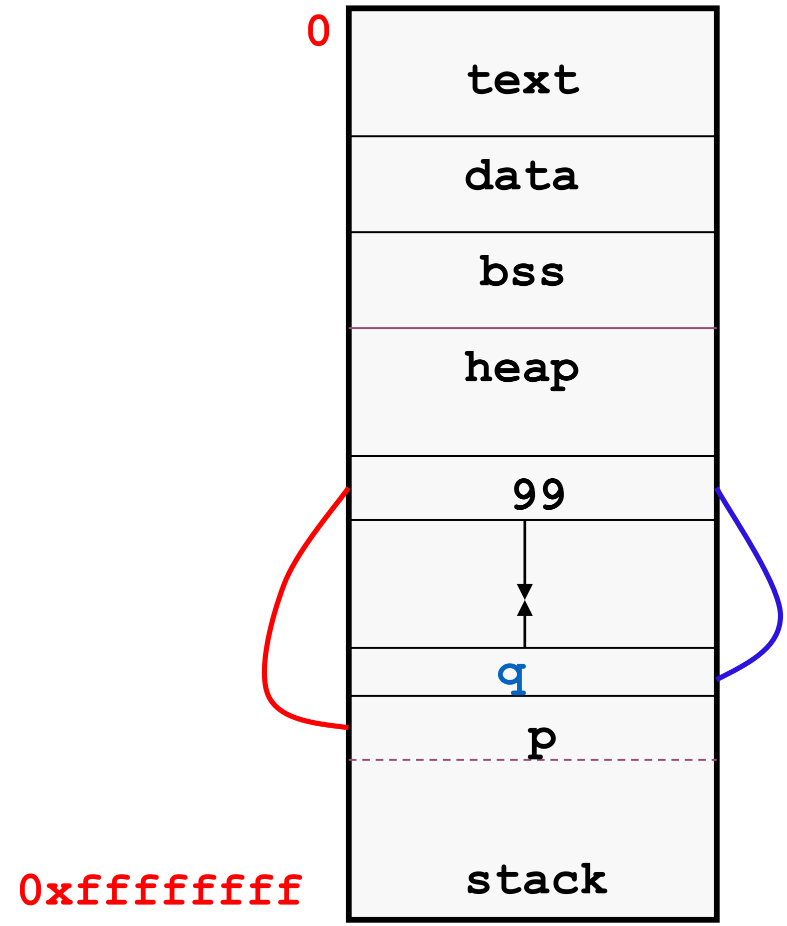
```
int main()
{
   int *p, *q;
   p = new int;
  *p = 99;
   q = p;

   return 0;
}
```



Aliasing

If two pointers have the same value, they are aliases of each other.

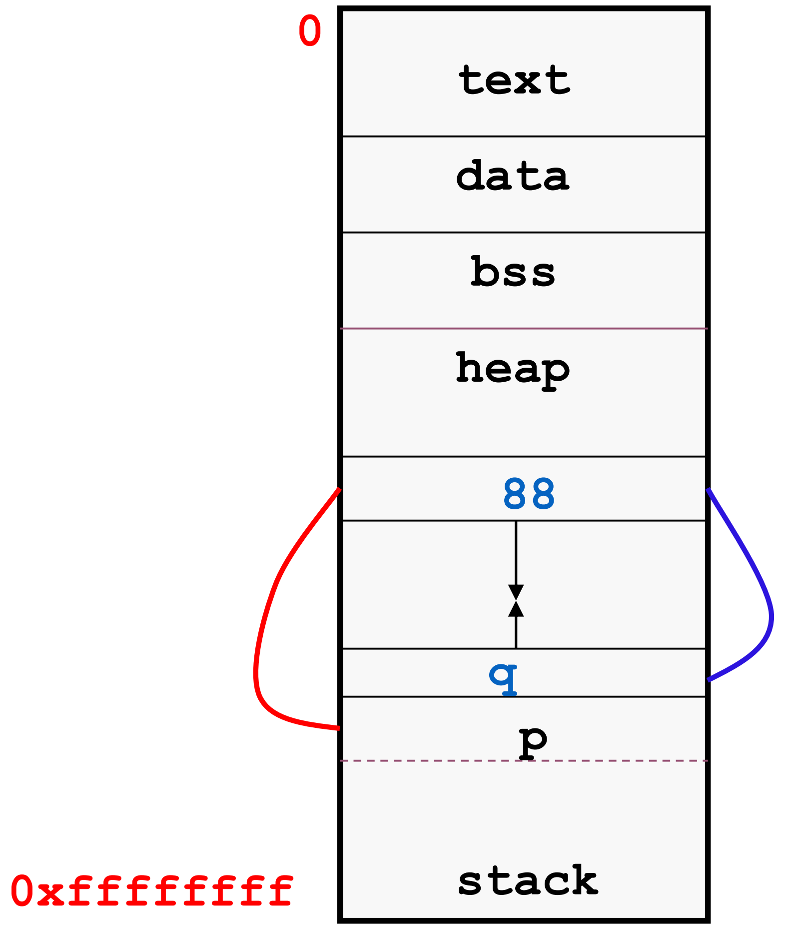
```
int main()
{
  int *p, *q;
  p = new int;
 *p = 99;
  p = q;
 *q = 88;
  return 0;
}
```



Aliasing

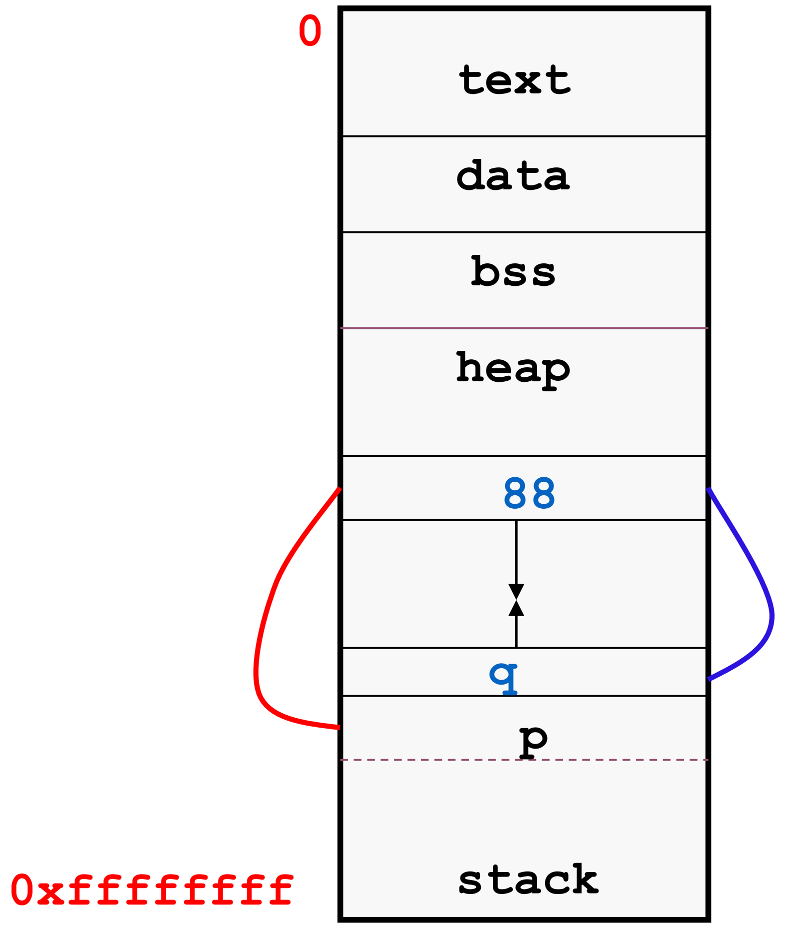
If two pointers have the same value, they are aliases of each other.

```
int main()
{
  int *p, *q;
  p = new int;
 *p = 99;
  q = p;
 *q = 88;
  return 0;
}
```



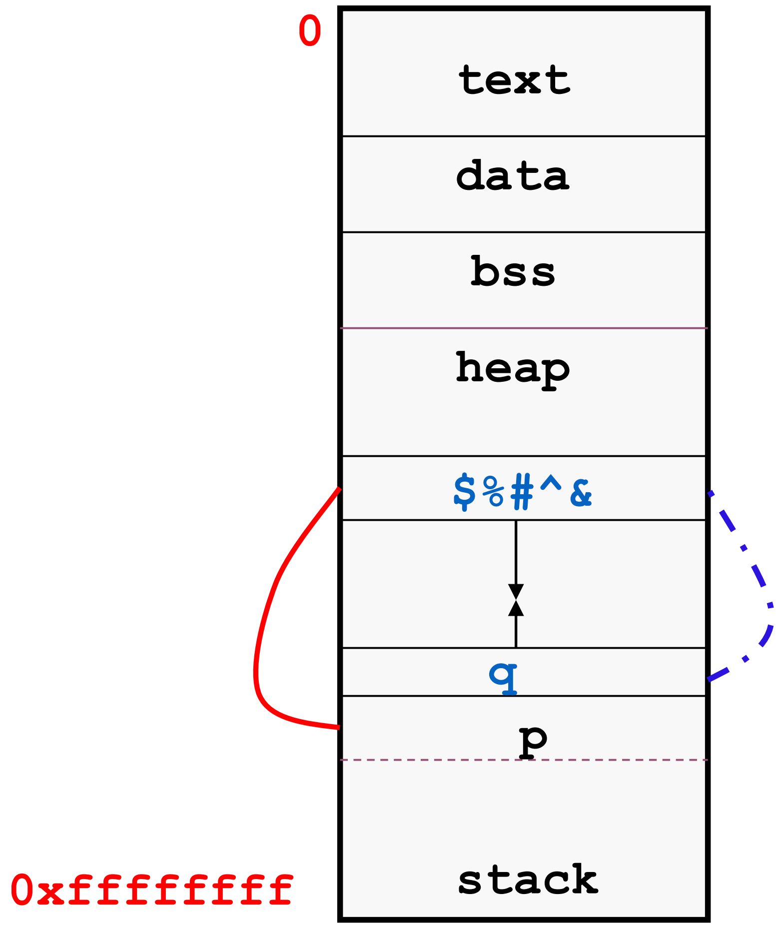
Dangling Pointers

```
int main()
{
  int *p, *q;
  p = new int;
 *p = 99;
  p = q;
 *q = 88;
  delete q;
  return 0;
}
```



Dangling Pointers

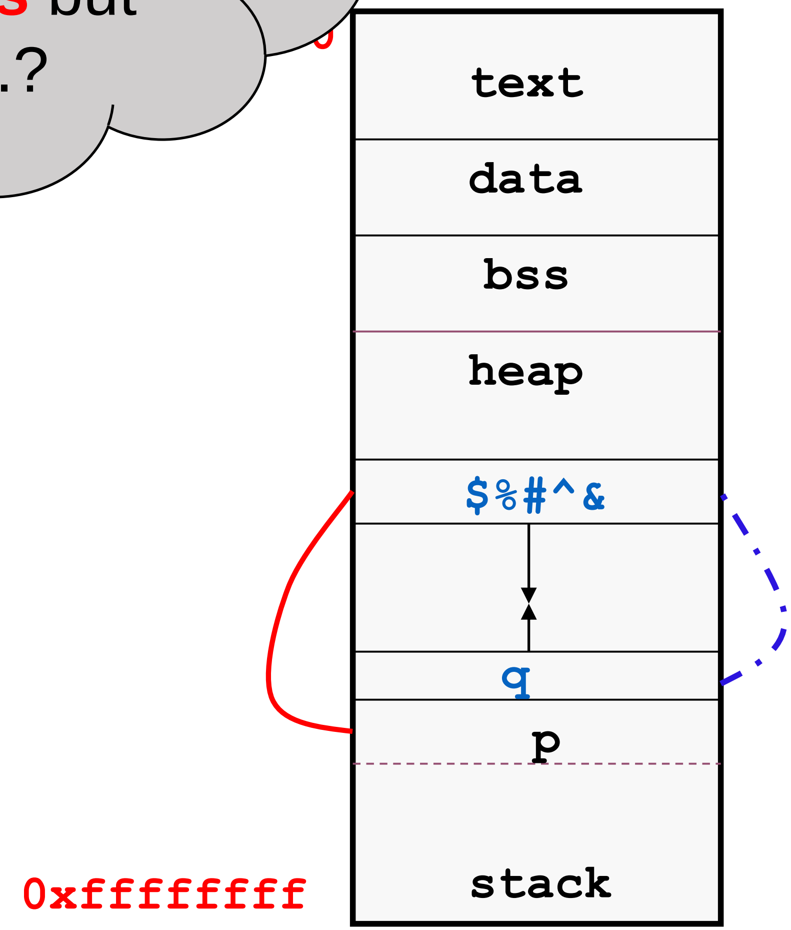
```
int main()
{
  int *p, *q;
  p = new int;
 *p = 99;
  p = q;
 *q = 88;
  delete q;
  return 0;
}
```



Dangling Pointers

p and q are now **dangling pointers** but why..?

```
int main()
{
  int *p, *q;
  p = new int;
 *p = 99;
  p = q;
 *q = 88;
  delete q;
 *p = 71;
  return 0;
}
```



Dangling Pointers

- The delete operator does not delete the pointer, it takes the memory being pointed to and returns it to the heap

- It does not even change the contents of the pointer

- Since the memory being pointed to is no longer available (and may even be given to another application), such a pointer is said to be dangling

Errors due to Dangling Pointers

If the program writes to memory referenced by a dangling pointer, a silent corruption of unrelated data may result, leading to subtle bugs that can be extremely difficult to find

Avoiding a Dangling Pointer

- For Variables:

```
delete v1;
v1 = NULL;
```

- For Arrays:

```
delete[ ] arr;
arr = NULL;
```

Returning Memory to the Heap

- Most applications request memory from the heap when they are running;
- It is possible to run out of memory (you may even have gotten a message like "*Running Low On Virtual Memory*")
- So, it is important to return memory to the heap when you no longer need it

Returning Memory to the Heap

- Always return memory to th … fore *undangling* the pointer

Memory Leak!!!

- What happens if you d…

```
ptr = NULL;
delete ptr;
```

Memory Leaking

```
int main()
{
   int *p;
   p = new int;
// memory allocated above is unreachable now
   p = new int;
   return 0;
}
```

Memory Leaking

```
void f()
{
    int *p;
    p = new int;
    return;
}

int main()
{
    f();
    return 0;
}
```

p is a local variable, destroyed after function returns

Memory Leaks

- Memory leaks when it is allocated from the heap using the new operator but not returned to the heap using the delete operator

Memory Leaking and Dangling Pointers

- Dangling pointers and memory leaking are <u>evil sources of bugs</u>:

- **<u>Hard to debug</u>**
  - may appear after a long time of run
  - may far from the bug point

- **<u>Difficult to prevent</u>**

Null Address

- Like a local variable, a pointer is assigned a random value (i.e., address) if not initialized

- 0 is a pointer constant that represents the empty or NULL address

- Should be used to avoid dangling pointers

```
int *ptr = 0; OR int *ptr = NULL;
```

Null Address

- Cannot Dereference a NULL Pointer

```
int *ptr = 0;

cout << *ptr << endl;

// ERROR: ptr does not point to a valid
//address
```

Pointers Data-Type

Why does a pointer need a data type?

Aren’t **all memory addresses of the same length**?

Pointers Type

- In a given OS, all memory addresses are of the same length

- However, with operations like dereference or increment/decrement the compiler needs to know the data type of the pointer variable (**to get data or jump to the next memory location**)

- Examples:
  - If “p” is a **character-pointer** then “p++” will increment “p” by **one byte** **(next location)**
  - if “p” is an **integer-pointer** its value on “p++” would be incremented by **4 bytes** **(next loc.)**

Void Pointer

- `void*` is a **pointer** with
  - ***void * can point at***

```
int iVar=5;
float fVar=4.3;
char cVar='Z';
int* p1;
void* vp2;
p1 = &iVar; // Allowed
p1 = &fvar; // Not Allowed
P1 = &cVar; // Not Allowed
vp2 = &fvar; // Allowed
vp2 = &cVar; // Allowed
vp2 = &iVar; // Allowed
```

This is a great advantage…
So, What are the limitations?

Void Pointer

- `void*` is a **pointer** with
    - ***void * can point at***

This is a great advantage…
So, What are the limitations?

- <u>Cannot be dereferenced</u>

- <u>Cannot perform any arithmetic operations</u>, not even increment/decrement

Casting pointers

- Pointers have types, so you cannot just do this

```
int *pi;
double *pd;
pd = pi;
```

- Even if they are both the same size, C++ will still give an error on implicit typecasting

Casting pointers

- C++ will let you change the type of a pointer with an explicit cast

```
int *pi; char *pd;
pd = (char*) pi;
```