Pointers

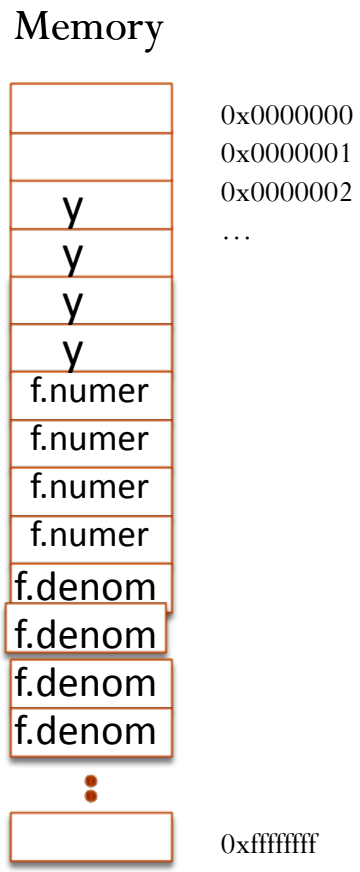
Memory

- Computer memory is used to store programs and variables in the programs
- The memory can be considered as a giant array
- Each byte in the memory can be indexed by the “address” -- in a 32-bit computer, the number of bits in each “address” is 32 bits.

Memory

- Each variable is stored somewhere in the memory – each variable is associated with an address that allows for access to the variable.
- Different types of variables may have different memory size
  - int – 4 bytes
  - char 1 byte
  - float – 4 bytes
  - double – 8 bytes.
  - you can use sizeof() function in C++ to check the size of each variable. See sizeof.cpp.

Memory



- The data members of a class are also stored in the memory.

```
main()
{
  int y;
  Fraction f;
    ……
}
```

- The memory is partitioned into different regions
  - different variables are stored in different regions
  - static memory – global variables
  - stack memory – local variables
  - heap memory – dynamically allocated memory.

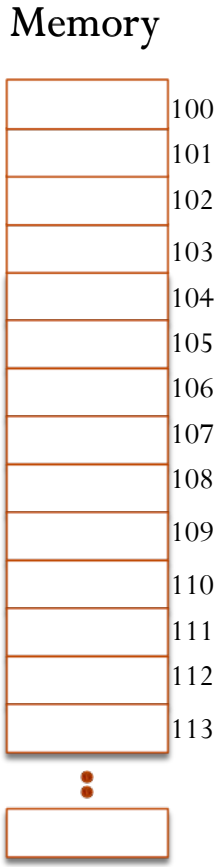
Pointers

- Every variable (object, data member, etc) is stored at a location in memory
- Every location in memory has a unique number assigned to it called it's address – index into the memory array.
- A pointer is a variable that holds a memory address
- A pointer can be used to store an object or variable's location in memory
- We can later "dereference" a pointer to have direct access to the object or variable the pointer points to.

Pointers

- Each memory address refers to the location of a single byte of data and consecutive memory addresses refer to contiguous bytes (i.e. bytes 100 and 101 are next to each other). Another way to think of memory is like a giant char array.

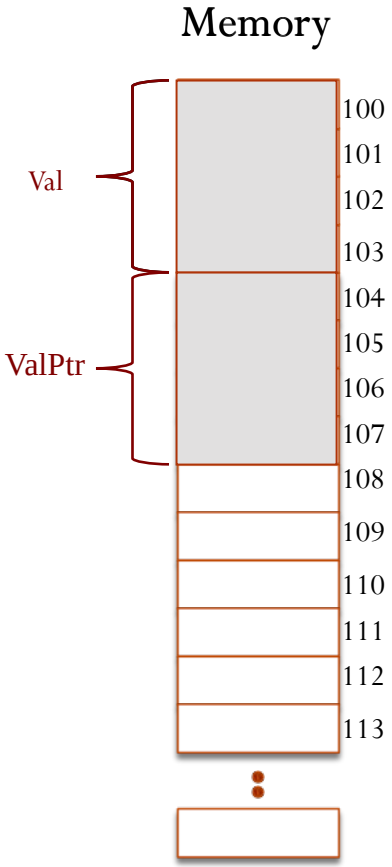
```
void foo() {




}//return
```



Pointers

- A pointer is allocated space in memory just like any other variable.
- Since a pointer holds an address, 32-bit systems use 32 bit addresses and therefore need 4 bytes to represent an address (32 bits * 1byte/8bits = 4 bytes). **Note that all pointers are allocated the same amount of memory independent of type (char*,int*,double*).

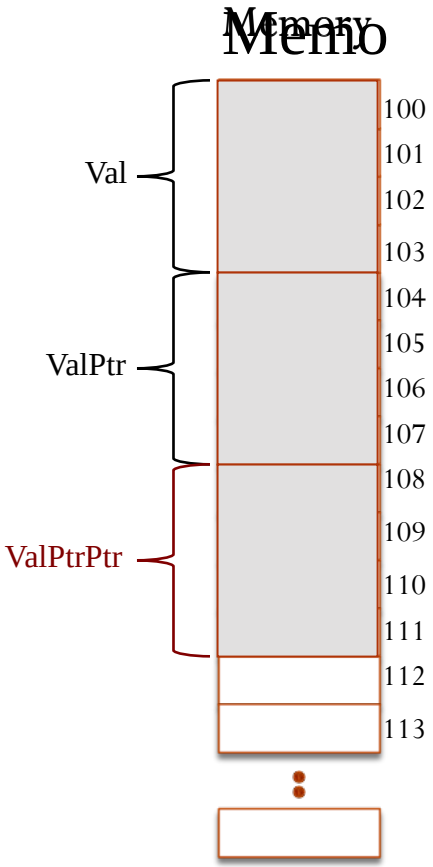
```
void foo() {
  int Val;
  int *ValPtr;
  int **ValPtrPtr;









} //return
```



Pointers

- A pointer pointer holds the address of a pointer of that same type and therefore is allocated the same space as a pointer. An int* holds the address of an int, and int** holds the address of an int*, an int*** holds the address of an int** etc.

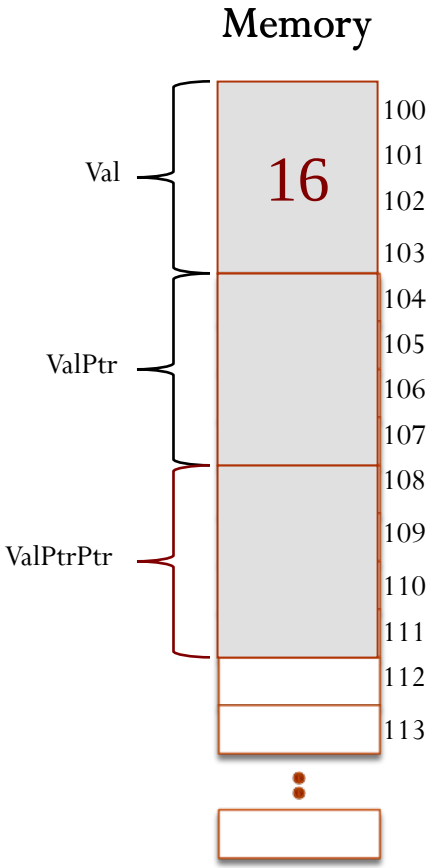
```
void foo() {
  int Val;
  int *ValPtr;
  int **ValPtrPtr;









}//return
```



Pointers

- When the compiler performs assignment, it goes to the address of a variable and updates the value. More generally, when a programmer writes the name of a variable "X" it can be interpreted as "the value in the memory reserved for X."

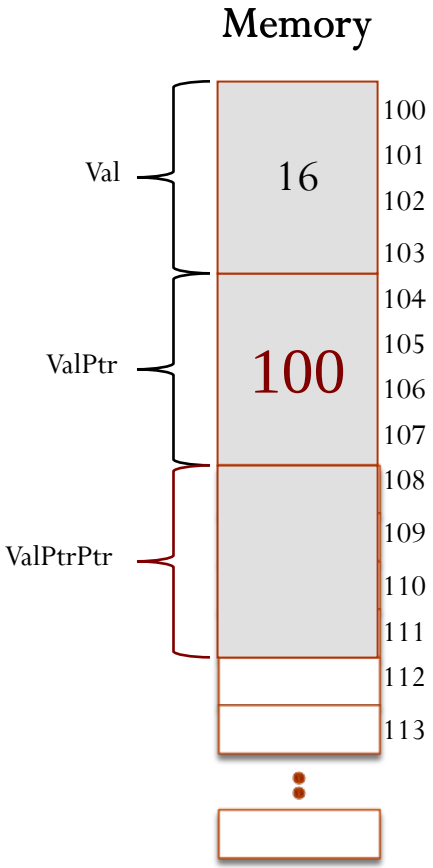
```
void foo() {
  int Val;
  int *ValPtr;
  int **ValPtrPtr;

  Val = 16;
  ValPtr = &Val;
  *ValPtr = 5;

  ValPtrPtr = &ValPtr;
  *ValPtrPtr = NULL;




}//return
```



Pointers

- By placing the '&' operator in front of a variable name, it is possible to refer to the address of that variable rather than its value. In other words, "&X" is interpreted as "the address of the memory reserved for X." The address of a variable is the address of the first byte it occupies in memory.

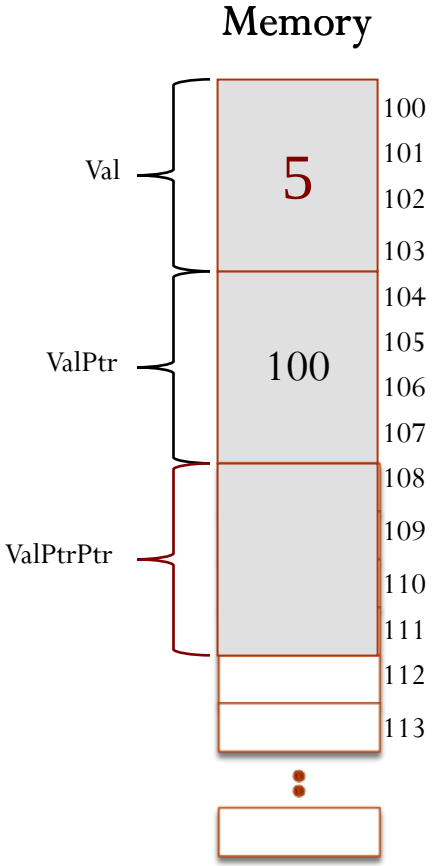
```
void foo() {
  int Val;
  int *ValPtr;
  int **ValPtrPtr;

  Val = 16;
  ValPtr = &Val;
  *ValPtr = 5;

  ValPtrPtr = &ValPtr;
  *ValPtrPtr = NULL;




}//return
```



Pointers

- For variables that are pointers, the '*' operator allows a programmer access to the value at the address stored in the pointer. This is called dereferencing a pointer and the process is known as indirection. Indirection is one reason pointers have types, although all pointers hold addresses, the compiler must know what type is stored at an address to access the value.

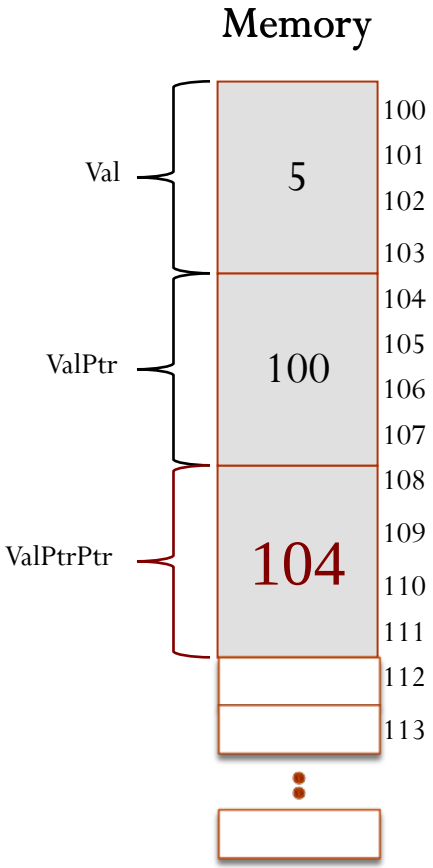
```
void foo() {
  int Val;
  int *ValPtr;
  int **ValPtrPtr;

  Val = 16;
  ValPtr = &Val;
  *ValPtr = 5;

  ValPtrPtr = &ValPtr;
  *ValPtrPtr = NULL;




} //return
```



Pointers

- Since a pointer pointer holds an address, it can be used almost exactly as a pointer. The extra "pointer" just says that the value at the address is an int* rather than an int.

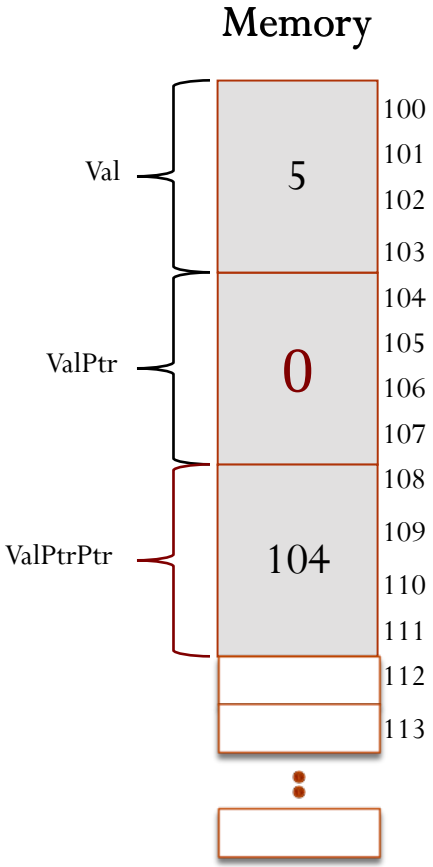
```
void foo() {
  int Val;
  int *ValPtr;
  int **ValPtrPtr;

  Val = 16;
  ValPtr = &Val;
  *ValPtr = 5;

  ValPtrPtr = &ValPtr;
  *ValPtrPtr = NULL;



} //return
```



Pointers

- Since a pointer pointer holds an address, it can be used almost exactly as a pointer. The extra "pointer" just says that the value at the address is an int* rather than an int.
  - A special memory NULL (0x00000000) is reserved for the not usable memory (invalid memory) – should be used to initialize any pointer.

```
void foo() {
  int Val;
  int *ValPtr;
  int **ValPtrPtr;

  Val = 16;
  ValPtr = &Val;
  *ValPtr = 5;

  ValPtrPtr = &ValPtr;
  *ValPtrPtr = NULL;



}//return
```



sample1.cpp

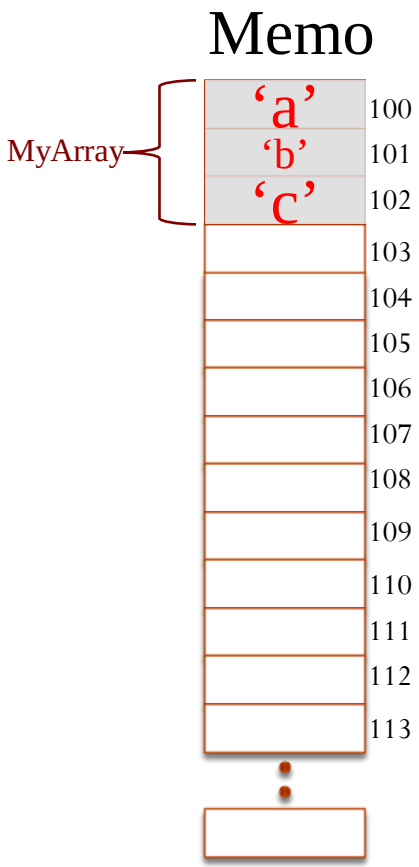
Life of a local variable

- Local variables are allocated on the stack.
- The life span of a local variable is within the subroutine
  - The variable is created when the routine is called.
  - When the routine is returned, all local variables do not exist anymore (the memory for these variables is released to the system).
  - Implication to the use of pointers?
    - Never return a pointer to a local variable.
  - Look at sample2.cpp and sample2_1.cpp.

Arrays and Pointer Arithmetic

- When an array is declared the compiler allocates enough space for all the elements of that array in memory. The array name now acts like a pointer to the first element in the array although this is just an abstraction because there is actually no address (pointer) stored in memory.

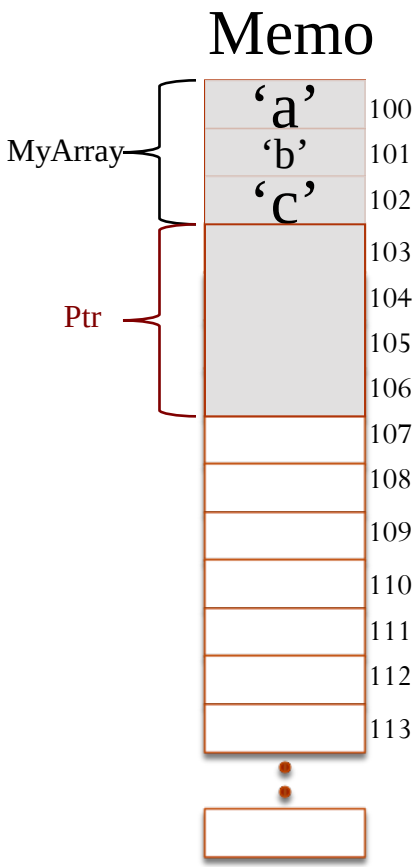
```
void foo() {
char MyArray[3] ={'a','b','c'};
char *Ptr;

Ptr = MyArray;









}
```



Arrays and Pointer Arithmetic

- An “actual” pointer will reserve memory to hold an address value.

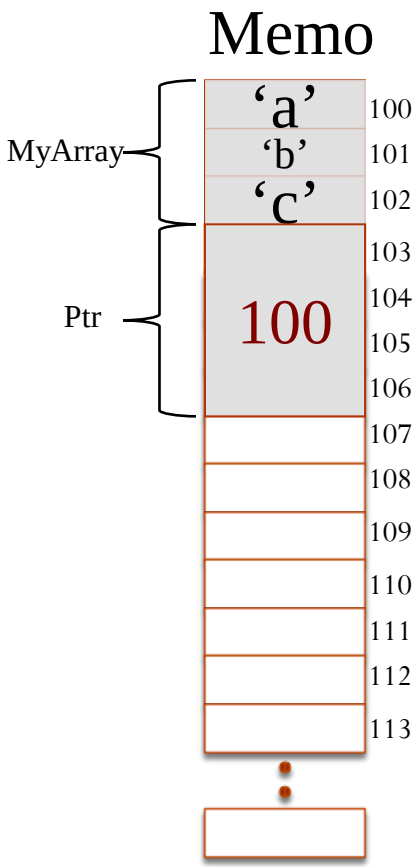
```
void foo() {
char MyArray[3] ={‘a’,’b’,’c’};
char *Ptr;

Ptr = MyArray;









}
```



Arrays and Pointer Arithmetic

- When an array is referenced in a program without an index [] the compiler "pretends" the name refers to a pointer that stores the address of the first element of the array. This allows us to assign a pointer to the value of MyArray.

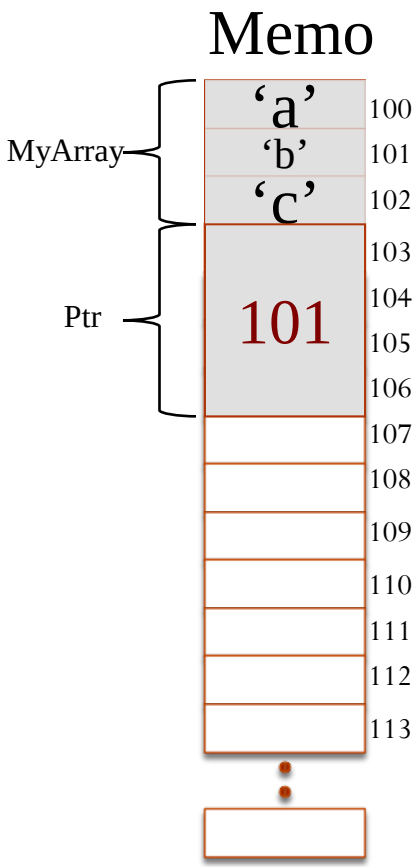
```
void foo() {
char MyArray[3] ={'a','b','c'};
char *Ptr;

Ptr = MyArray;









}
```



Arrays and Pointer Arithmetic

- In C++, pointers can be modified with +,-,++, and -- operators. A key difference between a pointer and an integer is that the statement Ptr = Ptr + Num, is translated to Ptr = Ptr + Num*(the size of the element the pointer points to). In this case, the pointer points to a char that is only allocated a single byte in memory.

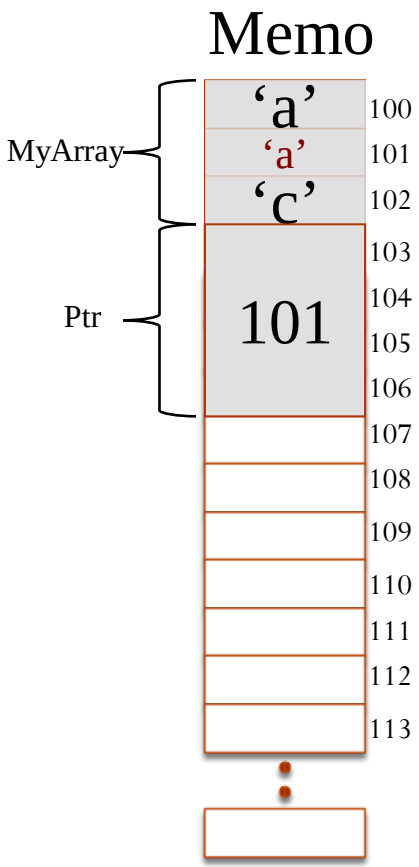
```
void foo() {
char MyArray[3] ={‘a’,’b’,’c’};
char *Ptr;

Ptr = MyArray;

Ptr = Ptr + 1;
*Ptr = *MyArray
*(Ptr+1) = ‘d’;
Ptr[1] = ‘a’;




}
```



Arrays and Pointer Arithmetic

- Changing the value of a pointer with arithmetic operations is known as pointer arithmetic. Among other things, it can be useful for traversing an array.

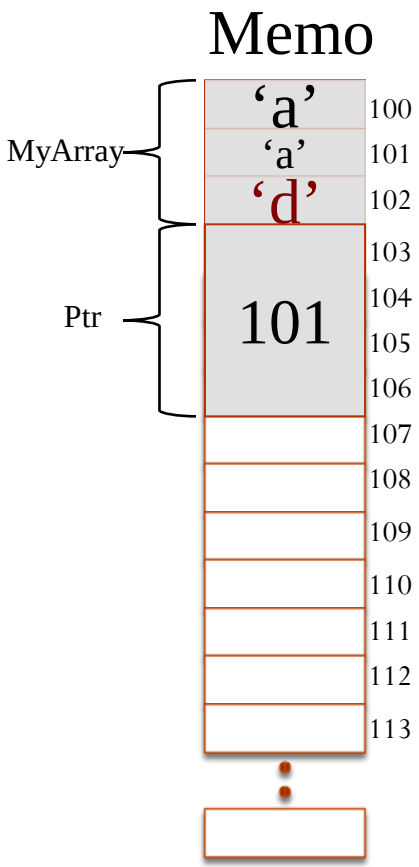
```
void foo() {
char MyArray[3] ={‘a’,’b’,’c’};
char *Ptr;

Ptr = MyArray;

Ptr = Ptr + 1;
*Ptr = *MyArray
*(Ptr+1) = ‘d’;
Ptr[1] = ‘a’;




}
```



Arrays and Pointer Arithmetic

- Pointer arithmetic and dereference can be combined into one statement to access offsets of an array. When the compiler encounters an indexed array, the statement is translated from array[index] into *(array+index).

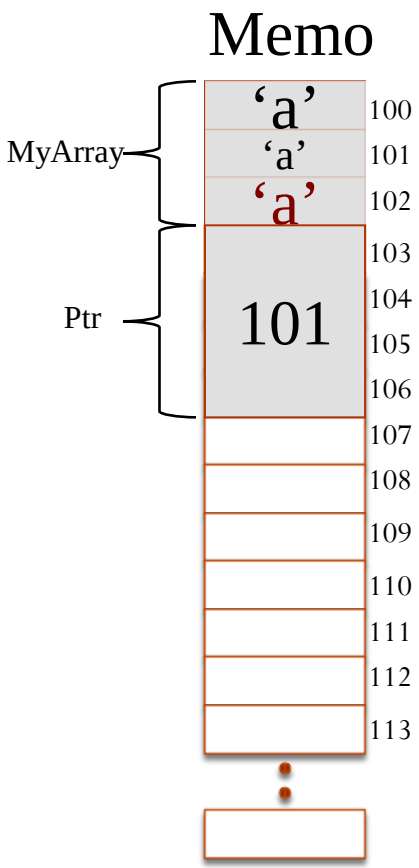
```
void foo() {
char MyArray[3] ={‘a’,’b’,’c’};
char *Ptr;

Ptr = MyArray;

Ptr = Ptr + 1;
*Ptr = *MyArray
*(Ptr+1) = ‘d’;
Ptr[1] = ‘a’;




}
```



Arrays and Pointer Arithmetic

- The left hand side of the last two statements are equivalent.

```
void foo() {
char MyArray[3] ={'a','b','c'};
char *Ptr;

Ptr = MyArray;

Ptr = Ptr + 1;
*Ptr = *MyArray
*(Ptr+1) = 'd';
Ptr[1] = 'a';




}
```



sample3.cpp
sample4.cpp

More about Pointers

- One pointer can be assigned to another if they are the same type or a typecast is used
- Any pointer type may be assigned to a void* type pointer
- Dereferencing an uninitialized pointer or a NULL (0) pointer will cause a segmentation fault

sample5.cpp

References

- A subclass of pointers specific to C++
- Basically a restricted pointer that only points to one object or variable
- Declared using ‘&’ operator instead of ‘*’ (int& IntRef)
- Only 1 dimensional (no int&&)
- Must be initialized (int& IntRef; will cause compiler error)
- Once initialized, a reference cannot be “reseated” to another object or variable
- No arithmetic operations allowed
- Automatically dereferenced so that only the value of the variable referenced is accessible

sample6.cpp

References

- Another way to think of a reference is giving a particular object or variable another name or alias (sample7.cpp)
- C++ automatically creates a reference for function calls that specify a reference in their parameter list
  - pass by reference (sample8.cpp)

- (sample9.cpp)

- Using the declaration, what is the size of p1, p2, and p3?

```
int *p1;
char *p2;
fraction *p3;
```