Lab-2

(Pointers and Dynamic Memory allocation)

Data Structures

Khalid Mengal

Contents

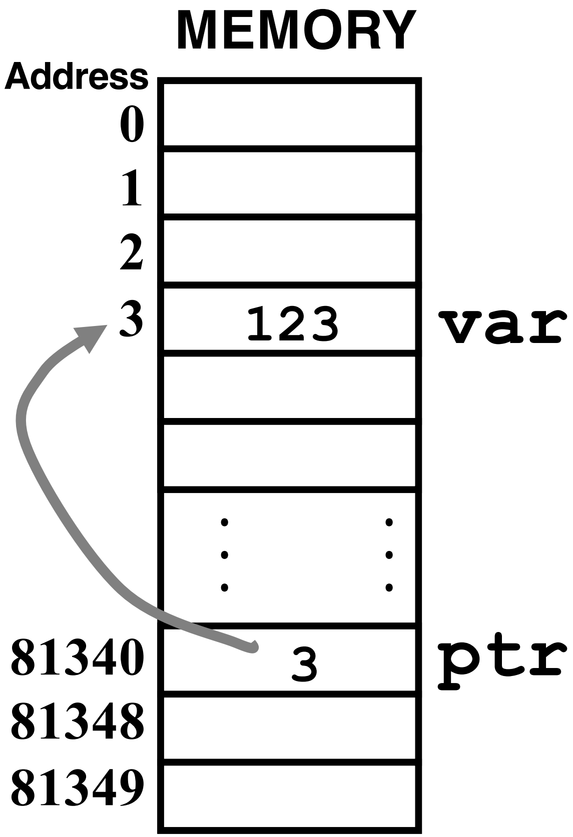
- Pointers
- Dynamic variables
- new and delete operator
- Pointers and functions
- Pass by Reference
- Exercise

Pointers

- A pointer is a variable that holds the ***address*** of something else.

```
int var;
int *ptr;

var = 123;
ptr = &var;
```



MEMORY

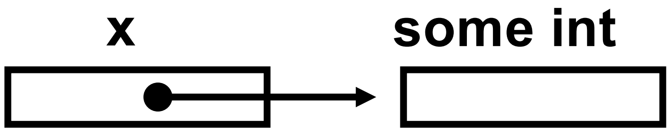
Address		
0		
1		
2		
3	123	var
	: :	
81340	3	ptr
81348		
81349		

Advantages of using Pointers

- Provide direct access to memory
- Make the program simple and efficient
- Allocate / deallocate memory during the execution of the program
- Pass arrays and c-strings to functions
- Return more than one value from a function

Pointers to anything

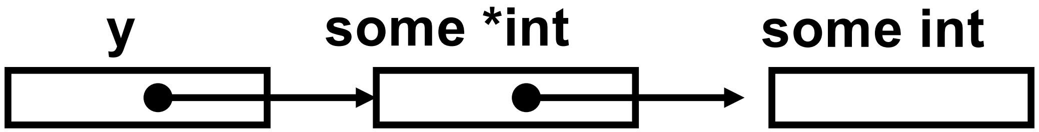
int *x;



x

some int

int **y;

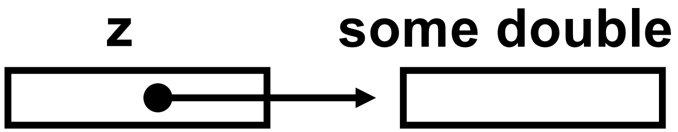


y

some *int

some int

double *z;



z

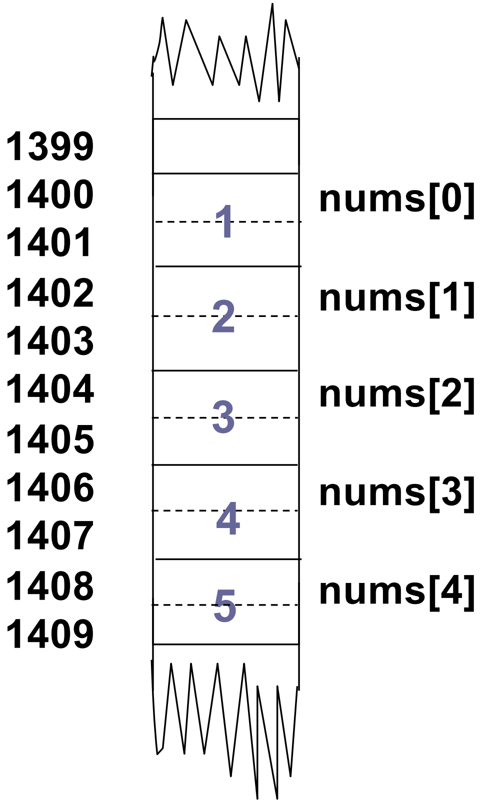
some double

Array Notation

There is a close association between pointers and arrays.
An array name is basically a ***const*** pointer.

```
short nums[ ]= { 1, 2, 3, 4, 5 };

for(int index=0; index<5; index++)
    cout<< nums[index];
```

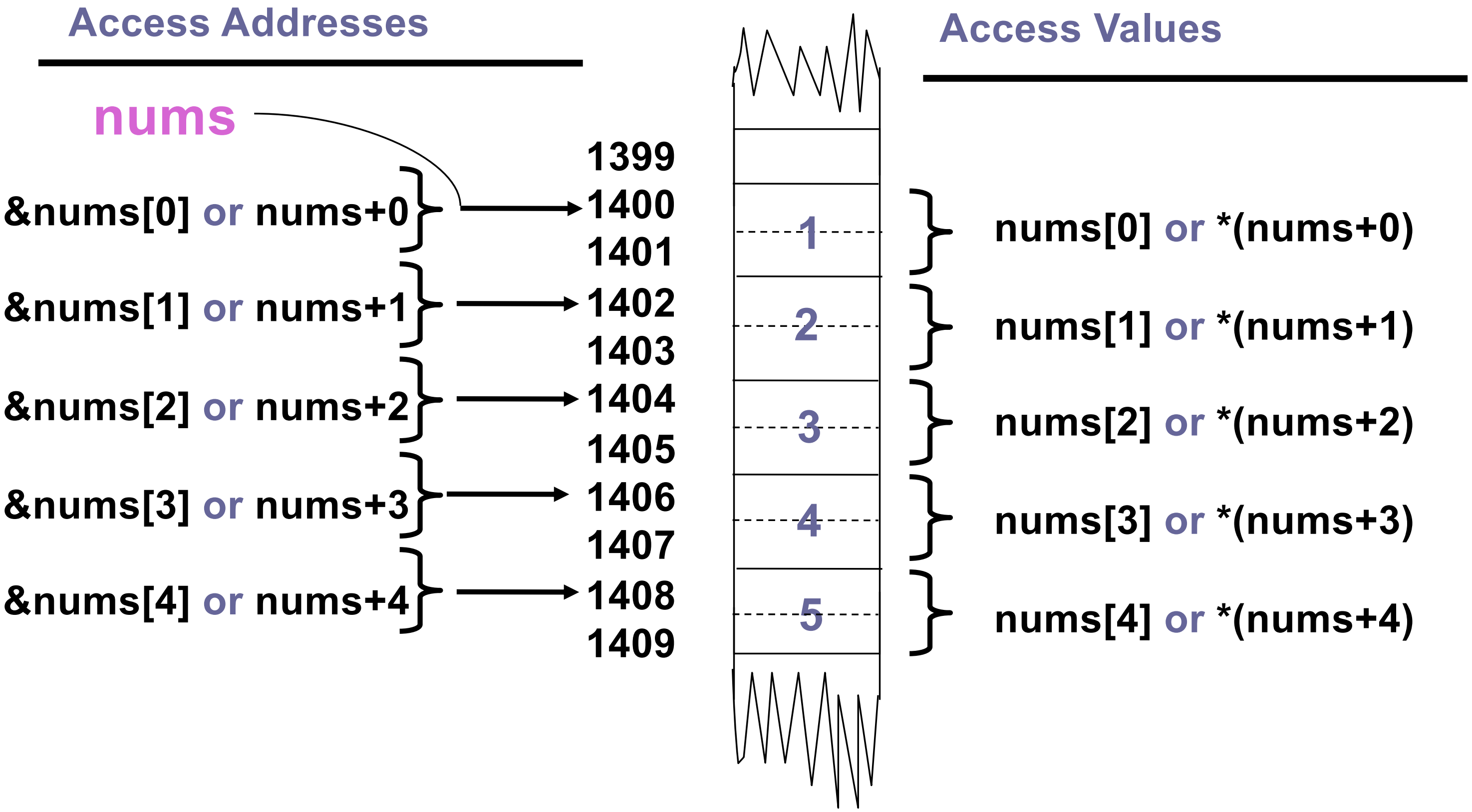


1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409

1
2
3
4
5

nums[0]
nums[1]
nums[2]
nums[3]
nums[4]

Pointer Notation



Access Addresses

nums

&nums[0] or nums+0

&nums[1] or nums+1

&nums[2] or nums+2

&nums[3] or nums+3

&nums[4] or nums+4

1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409

1
2
3
4
5

Access Values

nums[0] or *(nums+0)

nums[1] or *(nums+1)

nums[2] or *(nums+2)

nums[3] or *(nums+3)

nums[4] or *(nums+4)

Array Notation vs Pointer Notation

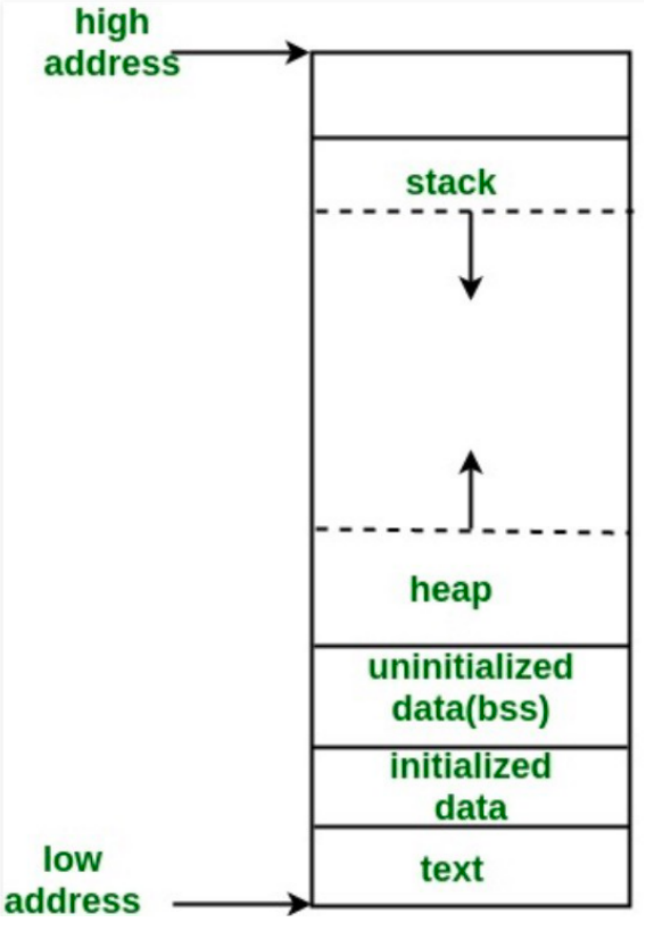
Array Notation

```
short nums[ ]= { 92, 81, 70, 69, 58 };
for(int index=0; index<5; index++)
    cout<< nums[index];
```

Pointer Notation

```
short nums[ ]= { 92, 81, 70, 69, 58 };
for(int index=0; index<5; index++)
    cout<< *(nums+index);
```

Program Address Space



- A ***program's address space*** is the range of ***logical*** addresses a program can operate on.
  - Note that the logical address space is not the same as physical memory addresses.
- A program address space is divided into three main areas:
  1) **Text/Code area** - near start of space
  2) **Initialized Data -** global and static variables
  3) **Un-initialized Data -** global and static variables
  4) **Heap** - middle of address space
  5) **Stack** - near top of address space

  stack grows, but direction of stack growth is OS dependent

Heap vs Stack Memory

- Stack
  - Fast Access
  - Contiguous
  - Automatic allocation/deallocation
  - Variables can not be resized
  - Limited Size (e.g. 8.192 MB)
    - ulimit –s (command to check stack size)

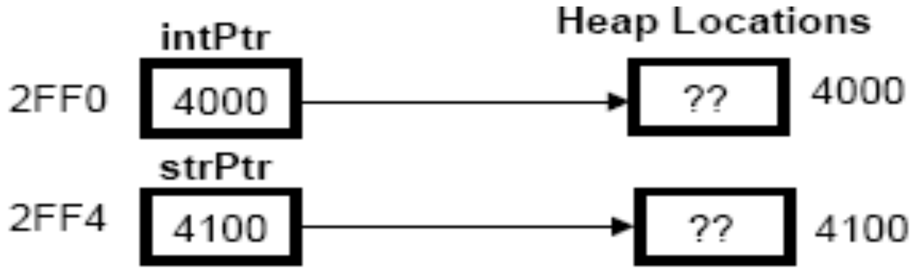
- Heap
  - Slow Access
  - Fragmented
  - Manual allocation/deallocation
  - Variables can be resized
  - Unlimited Size (determined by Physical RAM)

Allocating Variables Using new

- **new** operator can be used to allocate dynamic memory in the heap
  - ***variableType *variableName* = new *variableType*();**
- For example:

```
int *intPtr = new int();      // On same line

string *strPtr;               // Two lines
strPtr = new string();
```



The delete Operator

- Unlike Java, you as a programmer in C++ are responsible for deleting any dynamic memory objects you create.
- Deletion is accomplished by using the **delete** operator and passing the pointer to the object to be deleted:

```
delete ptrName;              // Single-object version
delete[ ] arrayPtrName;      // Array version
```

- For example:

```
int *intArray = new int[35];
double *dPtr = new double;   // () are optional  for base types
delete[] intArray;           // Delete array
delete dPtr;                   // Delete double
```

Pointers & Functions: **swap()**

```
void swap(int *p, int *q)
  {
     int tmp;
     tmp = *p;
     *p = *q;
     *q = tmp;
  }
  .
  .
  .
  swap(&a, &b)         //Call swap function
```

Pass By Reference

```
void swap(int &p, int &q)
  {
      int tmp;
      tmp = p;
      p = q;
      q = tmp;
   }
   .
   .
   .
   swap(a, b)              //Call swap function
```

Pass by Reference vs Pass by Pointers

- Pointers holds the memory address of the variable, whereas the reference has the same address as the item it references
- Pointer can be re-assigned a difference address, whereas the reference can not be
- Pointer can be assigned NULL whereas reference cant be
- Pointer has be de-referenced to get the variable it is pointing to, whereas reference is directly pointing to the same variable

Random Number in C++

- The **rand**() function computes a sequence of pseudo-random integers in the range of **0** to RAND_MAX

```
int number = rand();    //number will be assigned a value between 0-RAND_MAX
int number = rand() % 101;  //number will get a value between 0 and 100
```

- **srand**(arg) function sets its argument as the seed for a new sequence of pseudo-random numbers returned by **rand**().

```
srand(time(NULL));
```