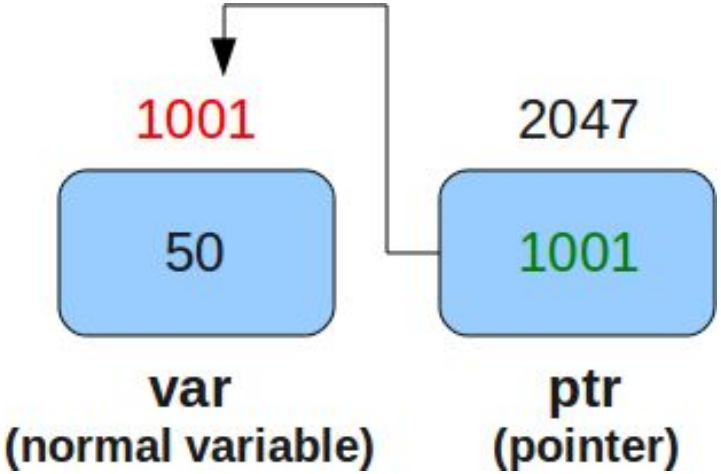
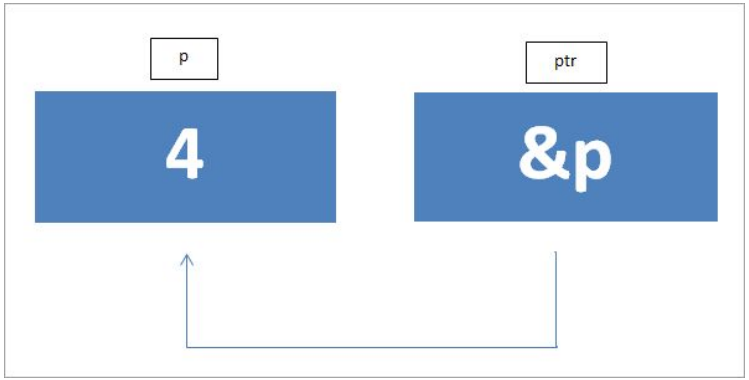
C++ Pointers and References





Uses of pointers and references

Uses of &

1. Obtain the address of a variable

```
int a = 5;
int* p = &a;
```

2. Declaring a reference

```
int a = 5;
int& b = a;
```

Examples of passing by reference and by address

```
1.  void swapByReference (int &a, int &b){
2.       int temp = a;
3.       a = b;
4.       b = temp;
5.  }
```

```
1.  void swapByAddress (int *a, int *b){
2.       int temp = *a;
3.       *a = *b;
4.       *b = temp;
5.  }
```

Uses of pointers and references

An example of passing in po

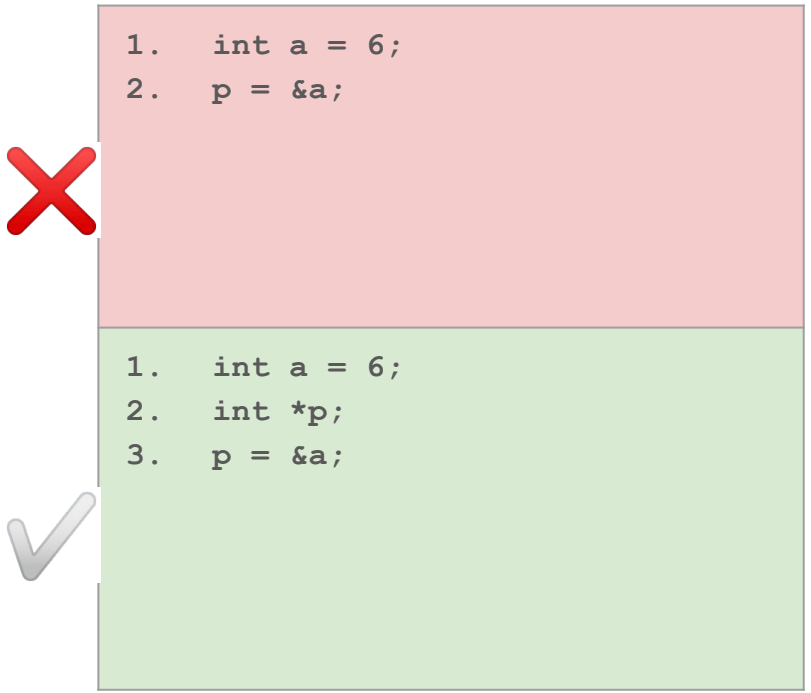
```
1.   void pass_by_value(int* p) {
2.       //Allocate memory for int and store the address in p
3.       p = new int;
4.   }
5.
6.   void pass_by_reference(int*& p) {
7.        p = new int;
8.   }
9.
10.  int main() {
11.      int* p1 = NULL;
12.      int* p2 = NULL;
13.      pass_by_value(p1); //p1 will still be NULL after this call
14.      pass_by_reference(p2); //p2 's value is changed to point to the newly allocate memory
15.      return 0;
16.  }
```

Code example above is from: https://stackoverflow.com/a/5789842/9706459

Dos and Don’ts for pointers

Always make sure your pointers are initialized

You need to initialize your pointers before using it!



```
1.   int a = 6;
2.   p = &a;
```

```
1.   int a = 6;
2.   int *p;
3.   p = &a;
```

Dos and Don’ts for pointers

Always check for NULL value when working with pointers.

Especially when you want dereferencing a point, check for NULL before dereference.

```
1.  void getIntVal (int* p) {
2.       cout << *p << endl;
3.  }
```

```
1.  void getIntVal (int* p) {
2.       if (p != NULL){
3.           cout << *p << endl;
4.       }
5.  }
```

Dos and Don’ts for pointers

Always delete the pointers after you finish using them.

Never lose track of your pointers

Before you want to reuse a pointer, make sure this pointer is not pointing anything, or there is another pointer pointing to the same object, so you don’t get memory leak.

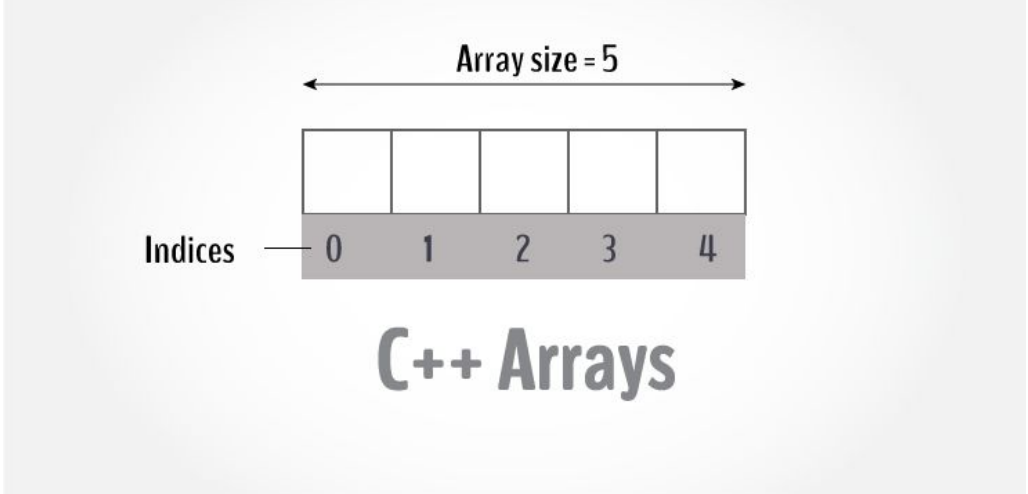
```
struct Point {
    double x;
    double y;
};
```

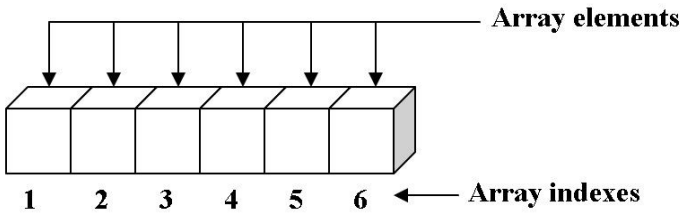
```
1.  Node* p = new Point(5,5);
2.  p = new Point(7,2);
```

```
1.  Node* p = new Point(5,5);
2.  delete p;
3.  p = new Point(7,2);
```

```
1.  Node* p1 = new Point(5,5);
2.  Point* p2 = p1;
3.  p1 = new Point(7,2);
```

Pointer and Array





One-dimensional array with six elements

Different ways of declaring an array:

```
int  arr1[5];
int  arr2[5] = {}; // all elements are 0s;
int  arr3[] = {1,2,3,4,5};
int  arr4[5] = {1,2,3,4,5};
```

How we pass an array as a parameter to a function:

```
void printArray(int arr[], int len);
```

Which is the same as:

```
void printArray(int* arr, int len);
```

How we access element at certain index of an array (arr2 as declared on the left):

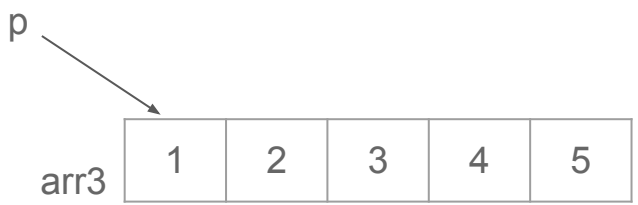
```
arr3[1]; // accessing the second
              //  element of the array
```

Which is the same as:

```
*(arr3+1);
```

Which is the same as:

```
int p = arr3;
*(p+1);
```



Code example: https://github.com/NathanWoo/cs16-final-review

Common mistakes in arrays

Copying all the elements in an array into another array

```
1.   int arr1[5];
2.   int arr3[] = {1,2,3,4,5};
3.   arr1 = arr3;
```

```
1.   int arr1[5];
2.   int arr3[] = {1,2,3,4,5};
3.   for (int i = 0; i < 5; i++){
4.        arr1[i] = arr3[i];
5.   }
```

Code example: https://github.com/NathanWoo/cs16-final-review

Common mistakes in arrays

Invalid return type when returning an array.

When you want to return an array, you should use pointer representation for the return value in the function declaration.

```
1.  int[] doubleVal(int arr[], int len){
2.        int* resultArr = new int[len];
3.        for (int i = 0; i < len; i++){
4.             resultArr[i] = 2 * arr[i];
5.        }
6.        return resultArr;
7.  }
```

```
1.  int* doubleVal(int arr[], int len){
2.        int* resultArr = new int[len];
3.        for (int i = 0; i < len; i++){
4.             resultArr[i] = 2 * arr[i];
5.        }
6.        return resultArr;
7.  }
```

P.S. Error code in this example would give you compilations errors

Code example: https://github.com/NathanWoo/cs16-final-review

Common mistakes in arrays

Creating an array on the stack.

If you want to return the array you create within a function, you should create that array on the heap.

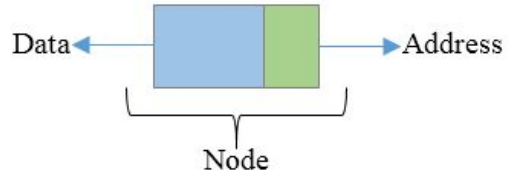
```
1.  int* doubleVal(int arr[], int len){
2.        int resultArr[len];
3.        for (int i = 0; i < len; i++){
4.             resultArr[i] = 2 * arr[i];
5.        }
6.        return resultArr;
7.  }
```

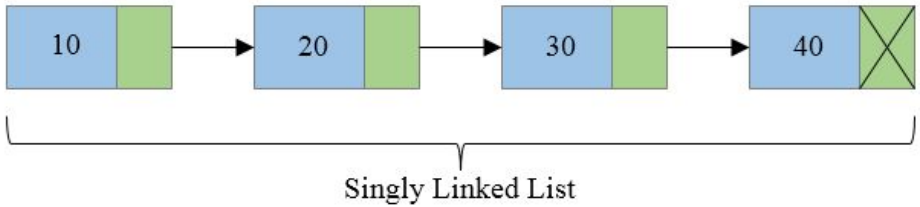
```
1.  int* doubleVal(int arr[], int len){
2.        int* resultArr = new int[len];
3.        for (int i = 0; i < len; i++){
4.             resultArr[i] = 2 * arr[i];
5.        }
6.        return resultArr;
7.  }
```

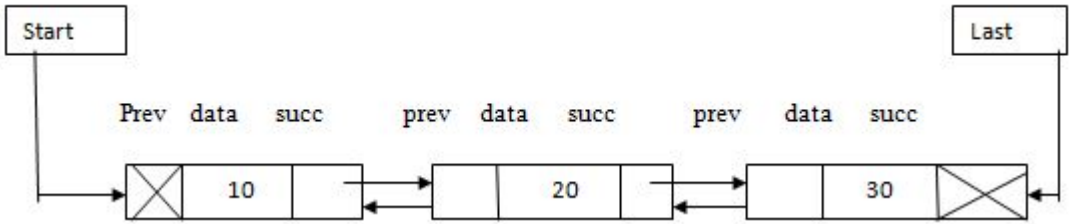
P.S. Error code in this example would give you a warning during compilation, but would still compile. The array returned contains junk values.

Code example: https://github.com/NathanWoo/cs16-final-review

Linked Lists







Two structs used to construct a singly linked list model:

Node:

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

LinkedList

```
struct LinkedList{
  Node* head;
  Node* tail;
};
```

Two structs used to construct a doubly linked list model:

Node:

```
struct Node{
  Node* prev;
  int data;
  Node* next;
};
```

LinkedList

```
struct LinkedList{
  Node* head;
  Node* tail;
};
```

Code example: https://github.com/NathanWoo/cs16-final-review

Common mistakes in LinkedList

Not checking whether the input point is valid.

```
 1.   int countEven(LinkedList *list){
 2.       // assert (list != NULL) would
 3.       // also be invalid
 4.       Node* temp = list->head;
 5.       int count = 0;
 6.       while (temp != NULL){
 7.           if (temp->data % 2 == 0)
 8.                count++;
 9.           temp = temp->next;
10.       }
11.       return count;
12.   }
```

```
 1.   int countEven(LinkedList *list){
 2.       if (list == NULL)
 3.           return 0;
 4.       Node* temp = list->head;
 5.       int count = 0;
 6.       while (temp != NULL){
 7.           if (temp->data % 2 == 0)
 8.                count++;
 9.           temp = temp->next;
10.       }
11.       return count;
12.   }
```

Common mistakes in LinkedList

Using a wrong operator to access the child element of a struct pointer

```
1.   int countEven(LinkedList *list){
2.       Node* temp = list.head;
3.       int count = 0;
4.       while (temp != NULL){
5.           if (temp.data % 2 == 0)
6.                count++;
7.           temp = temp.next;
8.       }
9.       return count;
10.  }
```

```
1.   int countEven(LinkedList *list){
2.       Node* temp = list->head;
3.       int count = 0;
4.       while (temp != NULL){
5.           if (temp->data % 2 == 0)
6.                count++;
7.           temp = temp->next;
8.       }
9.       return count;
10.  }
```

Common mistakes in LinkedList

LindkedList is not the same as Node

E.g. when you want to get the value of the head pointer of a linkedlist

```
1.   int countEven(LinkedList *list){
2.       // type 1
3.       LinkedList* head = list;
4.       // type 2
5.       LinkedList* head = list->head
6.       // type 3
7.       Node* head = list;
8.       . . . (some code) . . .
9.   }
```

```
1.   int countEven(LinkedList *list){
2.       Node* head = list->head;
3.       . . . (some code) . . .
4.   }
```

Common mistakes in LinkedList

Miss spell NULL

```
1.   Node* p1 = null;
2.   Node* p2 = Null;
```

```
1.   Node* p1 = NULL;
2.   Node* p2 = nullptr;
```

Return: check your return!

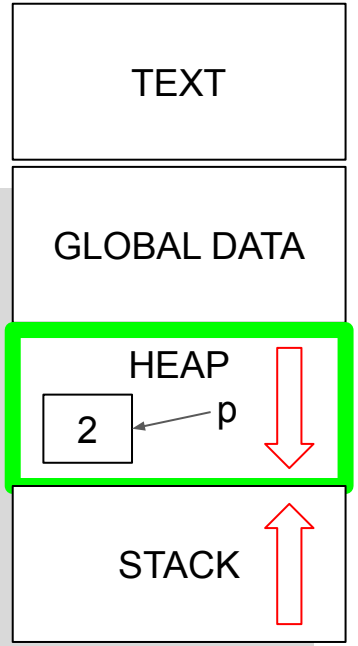
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return
return return return return return return return return return return return return return return return

Heap vs Stack

Heap

- Managed by programmer
  - Explicitly delete data from heap, otherwise data in heap stays there forever
- `new` operator used to allocate memory on heap, `delete` used to free memory on heap
- Memory leaks
  - Heap memory not deallocated before program ends
  - Heap memory is no longer accessible

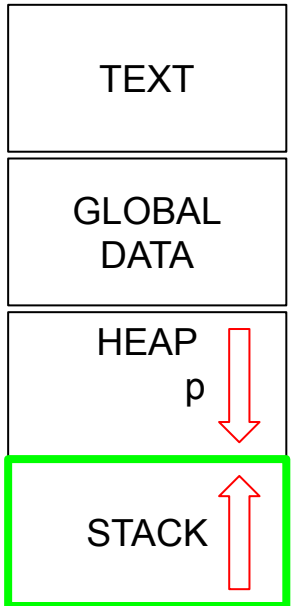
```
1.   int* p;
2.   P = new int;
3.   *p = 2

OR

1.   int* p;
2.   p = new int(2);

OR

1.   int* p = new int(2);
```



Heap vs Stack

Stack

- LIFO (Last In, First Out)
  - Think of a stack of papers or a stack of books
- Managed automatically




```
1.  void foo() {
2.      int x = 10;
3.      bar();
4.  }
```

Heap vs Stack

```
1.   int* p = new int;
2.   Student* n = new Student;
3.   delete p;
4.   delete n;
5.   delete n;
```

`n` has already been freed, so line 5 will result in an error