LECTURE 32: More Memory Management

Today

- Reading
  - Weiss Ch. 3, 4
- Objectives
  - Pointers in C++
  - Call-by-value vs. call-by-reference
  - (Back to classes: The Big Three)

Pointers in C++

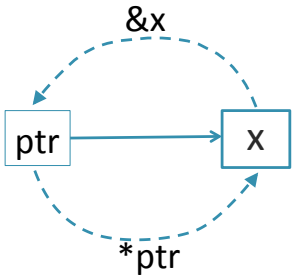
A pointer is a variable that stores the memory address of another entity of a given type

- To declare a pointer, use the * symbol

```
int *ptr; // Uninitialized!
```

Pointers in C++

- The address operator & returns the address of a variable
- The dereference operator * maps from the pointer to the data being pointed to



Pointers in C++

- *Be careful when altering memory via a dereferenced pointer!*
- Precedent rules are important
  - *ptr++;
  - (*vecPtr).push_back(5) // use -> instead
- Declaring multiple pointers on a line
  - int *ptr1, *ptr2; // each pointer needs a *

Pointers in C++

What are the values of the following expressions?

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

a. ptr
b. *ptr
c. ptr == a
d. ptr == &a
e. &ptr
f. *a
g. *&a
h. **&ptr
```

Dynamically allocated memory

- The `new` keyword allocates memory from the heap
- See ptr_exs.cpp

Dynamically allocated memory

- Don't use `new` when a stack allocated variable will suffice
- If you do use `new`, make sure you use `delete`
- If you have multiple pointers to the same piece of memory
  - Beware of stale pointers (point to already freed memory)
  - Beware of double deleting (deleting same memory twice)

Pointers in C++

```
int x, y;
x = 10;
int *p, *q;
p = new int(3);
*p = 47;
q = p;
*q = 23;
delete p;
*q = 17;
p = NULL;
p = &x;
cout << *p << endl;
```

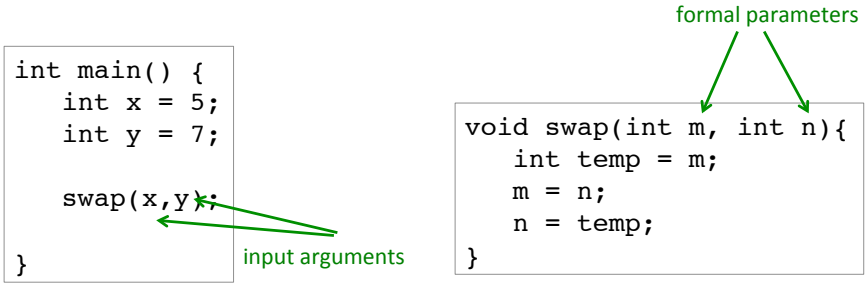
Pointers in C++

```
int x, y;            // declare two ints
x = 10;              // stack allocated int equals 10
int *p, *q;          // p, q pointers to ints
p = new int(3);      // p points to value 3
*p = 47;             // p now points to 47
q = p;               // q points to value 47
*q = 23;             // both p,q point to 23
delete p;            // memory is recycled
*q = 17;             // ERROR! memory was recycled
p = NULL;            // p points to nowhere
p = &x;             // p holds address of x - points to x
cout << *p << endl;  // prints value of x
```

Call-by-value

- Java and C++ use call-by-value when passing parameters
- Call-by-value: the input arguments are copied into the formal parameters

Call-by-value



x and y aren't swapped and lots of time spent copying actual objects

Call-by-value

- Change the formal parameters to now be pointers

```
int main() {
   int x = 5;
   int y = 7;

   swap(&x,&y);

}
```

```
void swap(int *m, int *n){
   int temp = *m;
   *m = *n;
   *n = temp;
}
```

x and y are now swapped but requires changing syntax!

References in C++

A reference is a constant pointer that is automatically dereferenced

- See ptr_exs.cpp
- To declare a reference, use & symbol

```
int x = 5;
int &int_ref = x;
```

Call-by-reference

- C++ also allows call-by-reference

```
int main() {
   int x = 5;
   int y = 7;

   swap(x,y);

}
```

```
void swap(int &m, int &n){
   int temp = m;
   m = n;
   n = temp;
}
```

The best solution!

Binary Search Example

```
int binarySearch(int val,vector<int> arr,int lo, int hi) {
      if( lo > hi) { return -1; }
      int mid = (lo+hi)/2;
      if(arr[mid] == val) {
            return mid;
      }
      else if(val < arr[mid]) {
            return binarySearch(val,arr,lo,mid-1);
      }
      else {
            return binarySearch(val, arr, mid, hi);

      }
}
```

This is inefficient. Why?

Binary Search Example

new function prototype

```
int binarySearch(int val, const vector<int>& arr,int lo, int hi);
```

- The `&` operator means no copying of input arguments
- `const` means this function will not change (mutate) this input parameter
- Only const methods can be called on `arr`

References in C++

- Benefits
  - Get the low-memory overhead of using a pointer
  - Without the need to use the dereference operator

The Big Three

- Destructor, copy constructor, `operator=`
- Default implementations of these methods are provided
- *Rule-of-thumb: If you need to overwrite one of these, overwrite them all*

Destructor

- Called when the object goes out of scope or when delete is called on an object
- Releases all resources
  - memory, files, streams

Copy Constructor

- Constructs a new object from an existing object
- The copy constructor is called when,

```
IntCell copy = original;
IntCell copy(original);
an input parameter to a call-by-value function
an object returned by value
```

- It would not be called in this instance:

```
IntCell copy;
copy = original;
```

operator=

- Assignment for two *already constructed* objects
- Example usage,

```
IntCell first(3);
IntCell scnd;
scnd = first;
```