Pointers and Memory

Quick refresh!

The address-of operator

- The ‘address-of’ operator is ‘&’ (ampersand)
- You can use it to get the address of a variable
- Declare a variable : int x = 1000 ;
- Get the memory address : cout << &x ;
- Returns memory address, not value
- This is a ‘reference’ to the data

Creating a pointer

- A pointer can be more or less any type
- You can create a pointer with the ‘*’ operator
- int * myPointer = 0;
- A pointer points to the address of some data

Using a pointer

- cout << myPointer; // prints the memory address of the pointer
- myPointer = &x // now the pointer points to x
- cout << myPointer; // prints the memory address of x

The dereference operator

- You can also get the data stored at a memory location with a pointer
- You do this with the dereference operator ‘*’.
- This gets the data ‘referenced’ by the address held by the pointer.
- cout << * myPointer; // prints the value of x, which is 1000

Allocating memory

- You can allocate a bunch of memory and make reference to it with a pointer
- If you have to do this, you need to make sure you free or delete it when you no longer need it, and at the very least, when you quit your program.
- You can allocate and free memory in two different ways

malloc and free

- to allocate some memory to store 100 ints:
- int * myInts;
- myInts = (int*) malloc(100 * sizeof(int))
- you can now access each memory location in ‘myInts’ like you would an array - e.g. myInts[9];
- if you use malloc, you need to call free() before your program quits to make sure the memory gets freed up.
- free(myInts);

New and Delete

- Instead of using malloc and free, you can use new and delete

- int *myInts = new int[100];

- you can now access each memory location in ‘myInts’ like you would an array - e.g. myInts[9];

- If you do anything like this, before you quit your program, you need to delete the memory

- delete [] myInts; // don’t forget the square brackets!

STACK vs HEAP

- Using malloc / free or new / delete creates memory on the HEAP.
- If you don’t use these approaches, chances are your data is on the STACK.
- The stack is well defined and structured.
- The heap is dynamic, and useful when you want to create lots of data, process it, transform it and move it around.
- e.g. you can resize data on the heap, but not on the stack.

REMEMBER

- Don’t try to write lots of data to a pointer before allocating enough memory.
- Don’t forget to get rid of the pointer and clean up the memory when your program quits. You can use the class destructor to do this.
- e.g. myClass::~myClass() { free(myInts) };
- Don’t try to use a pointer after you’ve deleted it.
- Don’t delete if you malloc’d. Don’t free if you new’d.

Accessing methods for objects declared by pointers

- Any object can be declared with a pointer
- This places the object in a particular block of memory.
- BUT when you do this, the object’s methods / members can’t be accessed in the usual way
- myObject.myFunction() // pointers can’t do this.
- myObject->myFunction() // pointers can do this.

‘Smart’ pointers

- If you really want to use ‘smart’ pointers be my guest
- But remember that there’s lots of debate over how these should best be used, even after all this time (almost a decade)
- e.g. https://stackoverflow.com/questions/106508/what-is-a-smart-pointer-and-when-should-i-use-one