

## Data Structures and Algorithms II

### Lecture 3: Revision of Pointers and Dynamic Data

- Memory
- Pointers
- Dynamic Data
- Dynamic 2d arrays

1

### Pointers and Variables

Cells can be named, using variable names. If variable x refers to cell 12 then:

`x = 5`

results in

12 13 14 15...

-----  
| 5 | 7 | 1 | | | |  
-----

or as most datatypes take more than one byte..

12 13 14 15...

-----  
| 5 | | 1 | | | | | | |  
-----

3

### Pointers

We can think of computer memory as just a contiguous block of cells:

-----  
| | | | | | | |  
-----

each cell has a value (or contents) and an address, e.g.

12 13 14 15...

-----  
| 3 | 7 | 1 | | | | |  
-----

So the cell with address 12 contains value 3.

2

### Declaring Variables

In languages like C++ we have to allocate memory for variables:

`int x;`

will allocate a 2 byte space for x, and associate the address of the first byte with the variable name.

4

## Pointers

A pointer is basically like an address. In the above example, if ptr is a pointer variable, ptr might contain the value 14.

Pointers are declared as follows:

```
int* ptr;
```

declares that ptr is a pointer to an integer (int \*ptr; is also OK). It allocates space for a pointer but does NOT allocate space for the thing pointed to.

Generally the syntax is:

```
<type>* <identifier>
```

5

## Accessing and Modifying Pointer Values

We can modify what a pointer points to in various ways.

```
ptr = &x;
```

modifies ptr so it points to the same cell that the ordinary integer variable x points to.

In our example above this was memory cell 12, so the result would be that the value pointed to by ptr (\*ptr) would now be 5.

If two things point to the same cell weird things can happen. If we now say:

```
x = 2
```

then \*ptr will = 2 too. But sometimes that's what we want.

7

## Accessing and Modifying Pointer Values

We can find out what a pointer ptr points to (ie, the value in the cell with the address ptr) using the notation:

```
*ptr
```

and set this value using e.g.,:

```
*ptr = 6
```

From our last diagram, with ptr=14 this would result in:

```
12 13 14 15...
```

```
-----  
| 5 | 6 | | | | |  
-----
```

Almost never want to change address locations, so DON'T use e.g.:

```
ptr = 16;
```

6

## Pointer Diagrams and Arithmetic

Pointers are often represented using diagrams like the following, which avoid explicitly referring to a particular memory cell address:

```
---      ---  
|ptr|---> | 5 |  
---      ---
```

pointer ptr points to a cell containing 5

Pointers should always point to something and 'know' what sort of thing they point to (hence are more than just an address..).

If we increment a pointer, this results in it pointing to the NEXT object in memory, not just the next memory address.

```
int *ptr;  
ptr++;
```

will result in ptr pointing to next integer in memory.

8

## Pointers and Arrays

In C++ pointers and arrays are very closely related.

```
int vec[5];
```

makes the variable `vec` a pointer to the first cell of a block of memory big enough to store 5 integers.

Space for those 5 integers is reserved, so it doesn't get clobbered:

```
12    14    16    17    18
-----
| 5 | 6 |   |   |   |
-----
----- RESERVED -----
```

In the above example `vec=12` (an address) (but we wouldn't usually think about this).

9

## Pointers and Arrays

Don't need to reserve space if pointer points to something that already exists (e.g., string constant).

e.g.,

```
char* str;
str = "alison";
```

pointer `str` points to the same address as the (previously allocated) string constant "alison".

However, will be problems if you, say, read in someone's name into `str`, without allocating space.

The following causes the program to crash:

```
char* str;
cin >> str;
```

11

## Pointers and Arrays

We could have declared `vec` in the almost equivalent way:

```
int* vec;
vec = new int[5];
```

We STILL can set and access the 'elements' in the 'array' in the same manner:

```
vec[1] = 2;
```

etc. But we have to reserve space explicitly.

This is source both of flexibility (can reserve "just enough" space at runtime) and errors (forget to allocate space).

10

## Copying Pointer Variables

What's the difference between

```
int* a;
int* b;
...
a=b;
```

and

```
int* a;
int* b;
...
*a = *b;
```

What do you think the effect of the following is:

```
int vec[5];
int* vptr;
...
vptr = vec;
```

12

### Pointers, Functions and Call by Reference

We can pass a pointer to a function:

```
void myfn(int* ptr)
{
    *ptr = 1;
}
```

The result is that "the int ptr points to" is set to 1. If we call it with:

```
int* p1;
.....
myfn(p1);
```

\*p1 will now have the value 1.

When myfn is called, ptr is a copy of p1, pointing to the same place as p1. So when \*ptr is set, that sets the memory cell also pointed to by p1.

Similar to how call by reference args handled.

### Arrays and Functions

If we want a function to return an array, the pointer version is essential. The following illustrates this:

```
int* test()
{
    int* a;
    a = new int[2];
    a[0]=1; a[1]=2;
    return a;
}

void main()
{
    int* b;
    b = test();
    cout << b[0] << b[1];
}
```

Note that space for the array (being created) is explicitly allocated within function using new; it is still there after function exits.

### Arrays and Functions

As array name = pointer to first element, arrays are passed to functions as pointers. So when array elements modified in the function, also modified in the calling procedure:

```
void test(int c[])
{
    c[0] =1;
}

void main()
{
    int a[5];
    test(a);
    cout << a[0];
}
```

results in '1' being written out.

The test function may be written equivalently as:

```
void test(int* c)
{
    c[0] =1;
}
```

### Dynamic Data

When variables are declared in normal fashion, space in memory is assigned at compile time for those variables.

If an array is declared, enough space for the whole array is assigned, so:

```
int a[1000]
```

would use 2000 bytes (for 2 byte ints).

Wasteful if only a few of them are used in a particular run of the program.

And problematic if more than 1000 elements are needed.

So there's a case for being able to dynamically assign memory as a program runs, based on the particular needs for this run.

### Dynamic Data

This is possible using the 'new' operator that explicitly assigns memory:

```
int* a;  
....  
cout << "How many entries?";  
cin >> n;  
a = new int[n];
```

results in space for n integers being dynamically assigned as the program runs. Just enough, and not too much, for that run.

We could declare the variable at the same time:

```
int* a = new int[n];
```

We can also dynamically de-allocate memory to free it up when we're done. To delete a whole array use:

```
delete [] a;
```

de-allocates all that space.

17

### Variable Sized Arrays

We can create 2-d arrays of variable size based on pointers, using dynamic data. To create n x n array x:

```
int** x;  
  
x = new int* [n];  
for(i = 0; i<n; i++)  
    x[i] = new int[n];
```

- Allocates memory for n pointers to integers. x now points to start of this block (array) of pointers.
- For each of these pointers, allocate memory for n integers. The array elements x[i] now point to these blocks.

This is useful for our adjacency matrix graph representation.

19

### Dynamic Data

This can be useful for strings:

```
char* str = new char[n];
```

could be used to create space for a string of length n, allowing us to deal with variable length strings efficiently. [Strictly, allows string of length n-1, as need end of string char at end].

18

### Memory: Heap and Stack

- Dynamic data is put in a bit of memory called the *heap*. data. Items have to be explicitly *deleted* from this.
- The actual parameters and local variables of functions are put on the *run-time stack*. Items are automatically removed from this when a function finishes.

20

### **Exercise**

Write a constructor function for the graph class that takes an argument giving the size of the graph, and allocates space for an array of the correct size.

Assume that the graph class has private data:

```
private:  
    logical** g;
```