CS197c: Programming in C++

Lecture 4

Marc Cartright

http://ciir.cs.umass.edu/~irmarc/cs197c/index.html

Programming assignment Recap

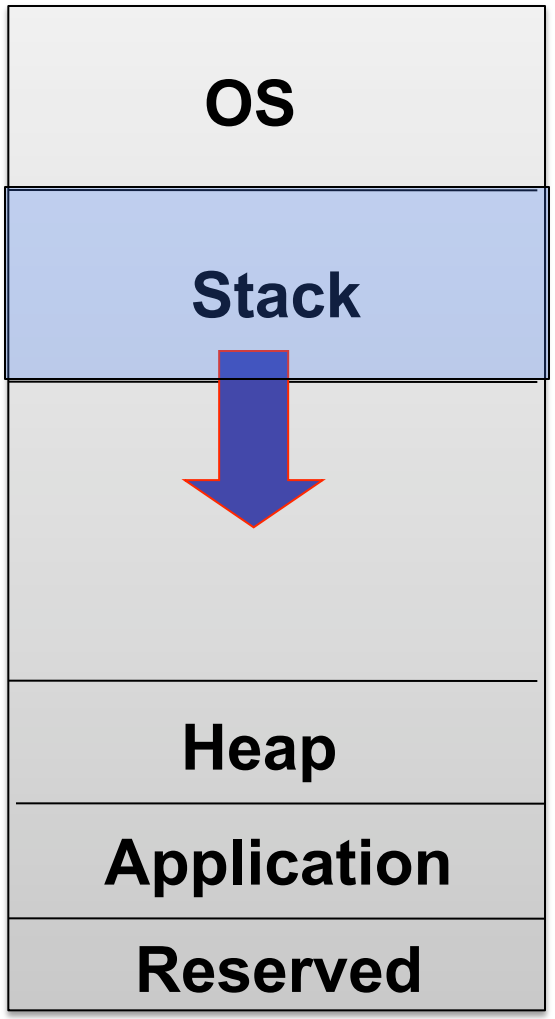
- PA2 due 9am *next* morning after class
  - Please try to attend
  - Ask questions earlier
  - Easier than PA1 (I think...)

- Things to not do:
  - Leave part of an assignment blank
  - Ignore the scaffolding given for the assignment
  - Not look at lecture notes or examples

Syllabus

- ~~Lecture 1 : C/C++ basics, tools, Makefiles, C data types, ADTs~~
- ~~Lecture 2 : C libraries~~
- ~~Lecture 3 : Classes in C++, C++ I/O~~
- Lecture 4 : Memory & Pointers
- Lecture 5 : More Pointers
- Lecture 6 : Templates and the STL
- Lecture 7 : Reflection, Exceptions, C++11
- Lecture 8 : Adv. Topics: Boost and OpenGL

The modern memory model



the stack

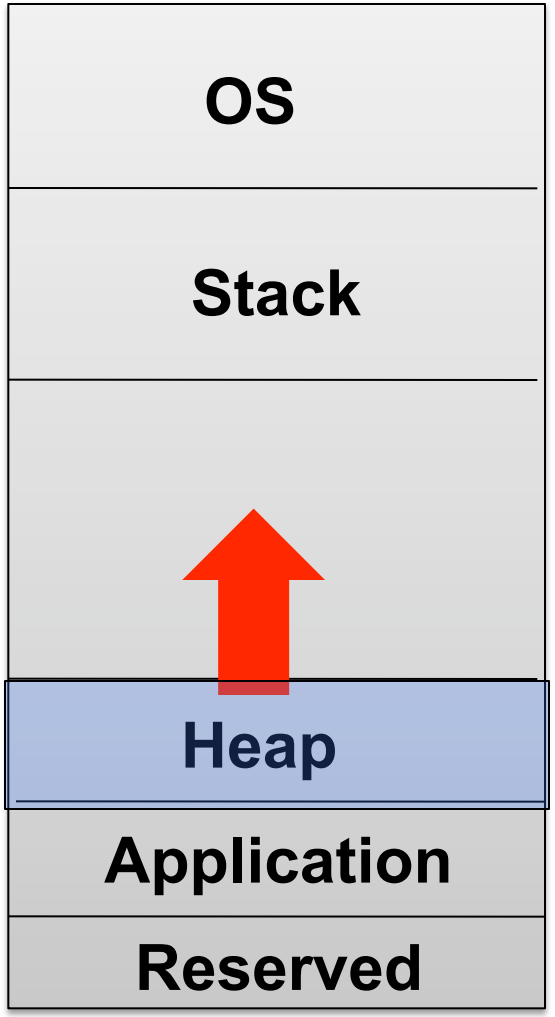
- Contents
  - local primitives
  - return addresses
  - function params

Growth : Ordered from function calls

Direction : high to low

Control : automatic

The modern memory model



the heap

Contents

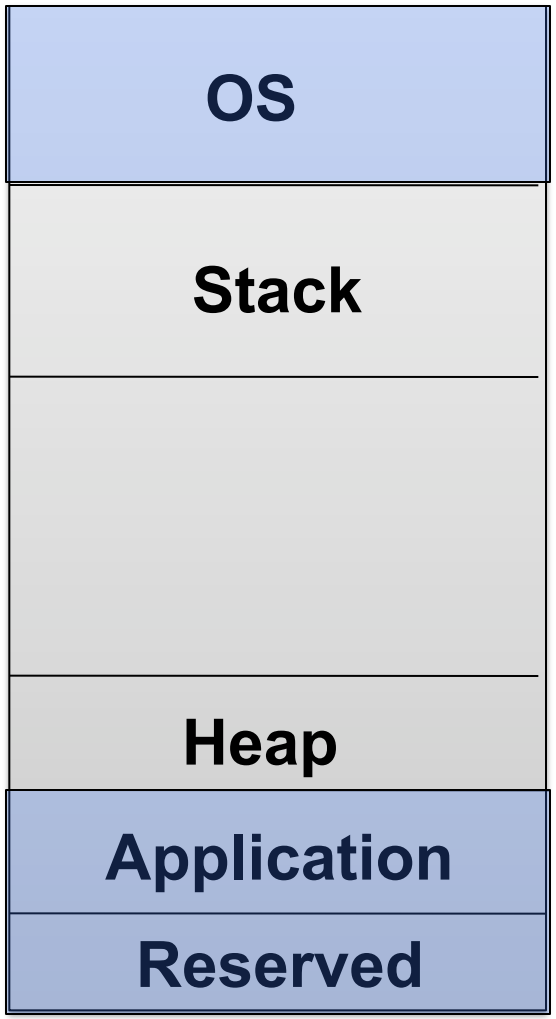
allocated objects

growth : chaotic, from **new**

direction : low to high

control : manual

The modern memory model



other stuff

- the operating system
- the application code
- etc.

How Java looks at things ?

- Primitives
  - allocated on the runtime stack
- Objects
  - allocated on the heap
  - accessed using references
  - created using **new()**
  - **destroyed using garbage collection**

How does C++ look at things ?

- Primitives
  - allocated on the runtime stack
- Objects
  - can be allocated on the stack or heap
  - accessed using **pointers**
  - created using **new()**
  - destroyed using **delete()**

What are pointers anyway ?

- variables which store locations of other vars
  - locations = addresses
- pointers usually point to things of only one kind
- since pointers hold locations, they hold machine words (they are ints)
- C/C++ has pointers
- Java has pointers (sort of)
  - Luckily it spares you from this

Common operators

- * operator
  - declaration: `int *p; int *q;`
  - dereference: `int a = (*p) + (*q);`
- & operator
  - address of: `int a = 12; int *p = &a;`
- -> operator
  - class/structure reference:

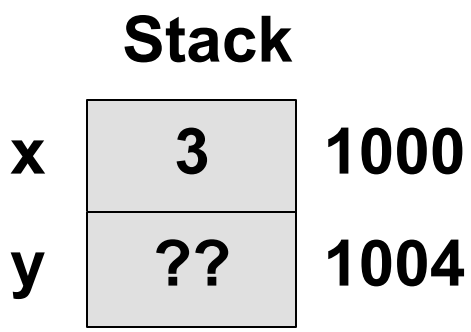
```
vector<int> *v = new vector<int>();
int len = v->length();
```

Pointer Sample code

```
int main()
{
        int x = 3;
        int *y;
        y = &x;
        x = 4;
       *y = 5;
}
```

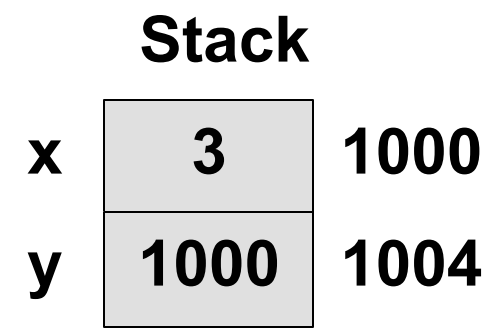
Pointer Sample code

```
int main()
{
        int x = 3;
        int *y;
        y = &x;
        x = 4;
       *y = 5;
}
```



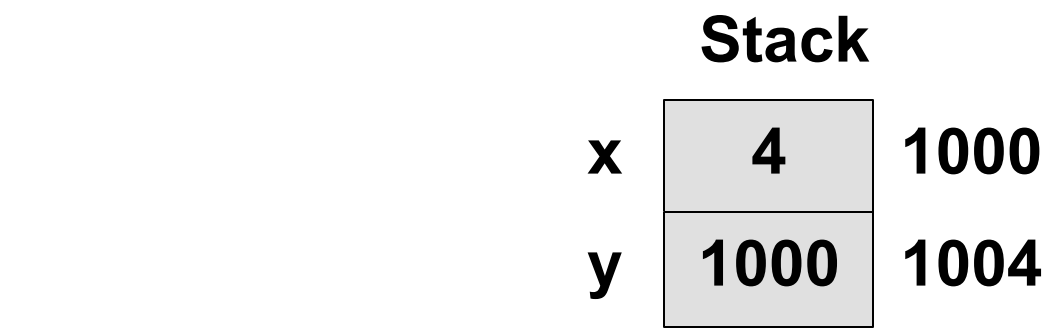
Pointer Sample code

```
int main()
{
        int x = 3;
        int *y;
        y = &x;
        x = 4;
       *y = 5;
}
```



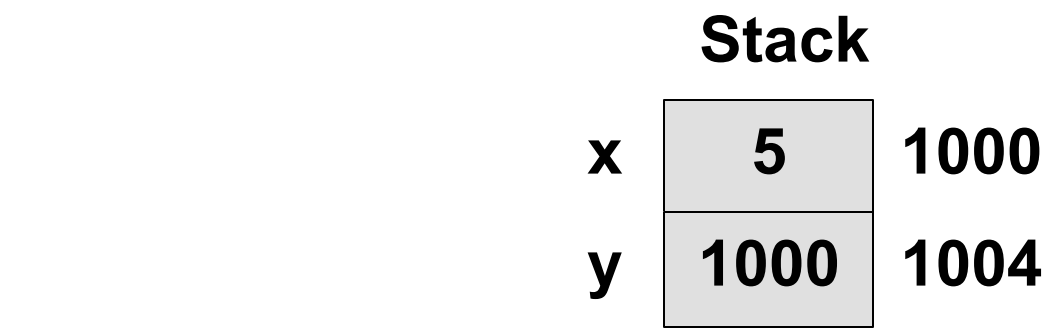
Pointer Sample code

```
int main()
{
        int x = 3;
        int *y;
        y = &x;
        x = 4;
       *y = 5;
}
```



Pointer Sample code

```
int main()
{
        int x = 3;
        int *y;
        y = &x;
        x = 4;
        *y = 5;
}
```



Other Pointers about pointers

- NULL
  - Shorthand for 0 pointer – any type
  - Points to nowhere
- pointer declaration
  - `int *ptr1, ptr; // error`
  - `int *ptr1, *ptr; // correct`
- be careful about precedence
  - `*ptr += 1;`
  - `*ptr++;`

Pointers vs. arrays

- Functionally equivalent, syntactically different
- `int *nums` is the same as `int nums[]`
- `*(nums + 24)` is also the same as `nums[24]`
- How does the pointer know how far to skip if you say `nums[24]` (or `nums + 24`)?

Pointers vs. arrays

- Functionally equivalent, syntactically different
- `int *nums` is the same as `int nums[]`
- `*(nums + 24)` is also the same as `nums[24]`
- How does the pointer know how far to skip if you say `nums[24]` (or `nums + 24`)?
- A single array increment moves `sizeof(type)` bytes over in the array.

new()

- `new()` is similar to Java
  - allocates objects from the heap
  - Calls constructor functions
- But not totally
  - You just cannot forget about the object
    - You have to garbage collect on your own
  - Arrays
    - We can dynamically allocate primitive arrays
    - `int *array = new int[10];`

delete()

- C++ has no garbage collection
  - you must `delete()` everything you `new()`
  - otherwise : memory leak
  - activates destructor
- `delete()` has no Java counterpart
  - Finalize does not count
- you can delete arrays too
  - `delete[] ()` operator

Example of delete() and new()

```
Obj o1;
Obj *op1 = &o1;
o1.foo();
op1 -> foo();
(*op1).foo();
Obj *op2 = new Obj();
op1 = op2;
delete op2;
op1->foo();
delete op1;
```

Example of delete() and new()

```
Obj o1;
Obj *op1 = &o1;
o1.foo();
op1 -> foo();
(*op1).foo();
Obj *op2 = new Obj();
op1 = op2;
delete op2;
op1->foo();
delete op1;
```

Example of delete() and new()

```
Obj o1;
Obj *op1 = &o1;
o1.foo();
op1 -> foo();
(*op1).foo();
Obj *op2 = new Obj();
op1 = op2;
delete op2;
op1->foo();
delete op1;
```

Example of delete() and new()

```
Obj o1;
Obj *op1 = &o1;
o1.foo();
op1 -> foo();
(*op1).foo();
Obj *op2 = new Obj();
op1 = op2;
delete op2;
op1->foo(); // can't do this; why?
delete op1; // can't do this either
```

Memory headaches

- stale pointers
- multiple deletes
- deleting something not newed
- dereferencing null pointers
- out-of-bound pointers
- and so, so much more...

Function pointers

- function name is actually a const pointer
  - address of the code that implements it
- Pointer to a function is a pointer whose address is the address of the function name
- Example

```
int f(int);
int (*pf) (int);
pf = &f;
```

References : a remedy

- Essentially an alias
  - `int &i = k;`
- Usable for primitives and objects
- Can never be repointed
- Can never be null
- No explicit dereferencing required
  - C++ does it for you
- All objects in Java do this under the hood

Other remedies

- Use C++ types where ever possible
  - strings Vs C-strings
- Use references where ever possible
- Use a debugger to detect memory errors
  - gdb
  - The ever-useful “printline debugging”
  - Or you can pray

Next Lecture

More Pointer Practice