Static & Dynamic Memory

http://www.cplusplus.com/doc/tutorial/dynamic/

Outline:

- Static memory & the stack
- Static keyword
- Dynamic memory & the heap

Stack Memory

Recap:

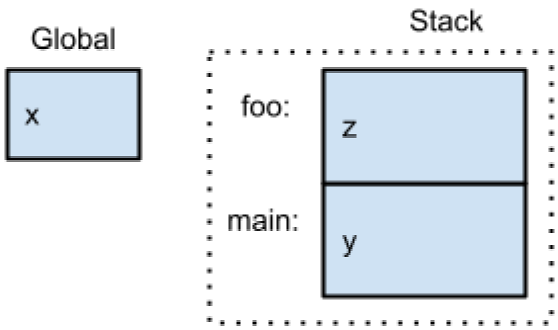
- When you declare a variable in C++, the appropriate memory space is allocated on the stack.
- With "normal" variable declarations, the memory is allocated for as long as the declaring function is active.
- Let's look at an example:

```
#include <iostream>
using namespace std;

int x=5;              // global variable, allocated for as long as program is active

void foo(){
       int z=3;              // local to foo, allocated only for duration of foo

       cout << x << endl;
       //cout << y;          <-- error
       cout << z << endl;
}

int main() {

       int y=4;              // local to main, also allocated for program duration
       foo();

       cout << x << endl;
       cout << y << endl;
       //cout << z << endl;  <-- error

       return 0;
}
```

See memory diagram below.

Memory diagram:

- The **stack** is the area where non-global variables is allocated.
- Each function's memory on the stack is called a **stack frame.**



Static memory

Although variables in one stack frame are not directly accessible from other frames, pointers can work around this. However, your results might not be what you expect.

- What happens to the local variable x when foo terminates?

```
#include <iostream>
using namespace std;

int* foo(){

       int x = 5;
       //static int x = 5;
       return &x;

}

int main() {

       int* y = foo();
       cout << *y;

       return 0;
}
```

★ It *might* do the same thing with or without the **static** keyword,

★ Without **static**, the local variable x is "freed" when foo terminates, so its memory could be overwritten.

★ The **static** keyword keeps the variable "alive," even after its declaring function has terminated.

Dynamic Memory

The primary **disadvantage of stack / static memory** is:

- Memory needs must be determined before program execution.
- This leaves no room for allocating memory in response to a user's needs.

For example:

- Say you're writing a program that read's information about students, line by line, until a sentinel keyword (like 'exit') is entered.
- You don't know how many lines of information your program will need to process.

A solution:

- Allocate an array that is as long as you will possibly need.
- The main drawback of this solution should be readily apparent.

```
// previous program code…

string student[5000];

// start getting lines of data…
```

Better solution:

- Allow the user to declare how many lines they will enter.
- Allocate memory on demand with the **new** keyword.
- Using **new** returns a pointer to the data on the heap (more on this below).

```
//previous code…

int n;

cout << "How many students?";
cin >> n;

string *student_data = new string[n];

// get data
for( int i=0; i<n; i++ ){
    cin >> student_data[i];
}

// PROCESS DATA

// when you're done, make sure you deallocate the memory
delete[] student_data;
```

Dynamic memory and the heap:

- When you use the **new** keyword, the memory is allocated on the **heap.**
- The heap is another memory space, specifically for dynamic memory.
- Memory allocated on the heap is not freed until the programmer frees it explicitly with the **delete** keyword.