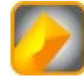


2007-02-01
Lecture #2
(Material on Test 1)

<u>Today's Outline</u>
C++ vs Java
Memory

Your Recommended Text Book

C++ for Java Programmers - (by Mark Allen Weiss)

- **Use it as a reference during your homework assignments**
- **You should cover the entire book on your own (if needed)**
- **Chapter 1, you should cover on your own**
- **Chapter 2, we cover some in class (functions, parameter passing)**
- **Chapter 3, we cover a lot in class (pointers, heap)**
- **Chapter 4, we cover in class (OOP)**
- **Chapter 5, we cover in class (Operator Overloading)**
- **Chapter 6, we cover in class (OOP Inheritance)**
- **Chapter 7, we cover some in class (Templates)**
- **Chapter 8, you should cover on your own (Exceptions)**
- **Chapter 9, we cover some in class (File I/O)**
- **Chapter 10, we cover a lot in class (STL)**
- **Chapter 11, we in class (Arrays, Strings, & Pointers)**
- **Chapter 12, you should cover on your own (C style)**
- **Chapter 13, we will not cover (JNI), but you can view on your own**

The Lifetime of Variables

Global Variables are the easy ones...

born? when the program starts executing

die? when the program stops executing

```
#include <iostream>
#include <cstdlib>
using namespace std ;
int age = 5 ;      // global variables
void print_age() ; // func prototype
int main(){
        cout << "age is " << age << endl ;
        print_age() ;
        return EXIT_SUCCESS ;
}
void print_age(){ // func definition
        cout << "age is " << age << endl ;
}
```

The Viewing of Variables

Global Variables...

Everybody below the declaration can view it.

Example:

```
#include <iostream>
#include <cstdlib>
using namespace std ;
int age = 5 ;      // global variables
void print_age() ; // func prototype
int main(){
        cout << "age is " << age << endl ;
        print_age() ;
        return EXIT_SUCCESS ;
}
void print_age(){ // func definition
        cout << "age is " << age << endl ;
}
```

The Lifetime of Variables

Local Variables are the easy too...

born? when the program enters their "scope"

die? when the program leaves their "scope"

```
#include <iostream>
#include <cstdlib>
using namespace std ;
void print_age() ; // func prototype
int main(){
        int age = 5 ;  // local variables
        cout << "age is " << age << endl ;
        print_age() ;
        return EXIT_SUCCESS ;
}
void print_age(){ // func definition
        // can't do anything here
}
```

The Viewing of Variables

Local Variables...

Typically, a variable's scope starts with the declaration and ends with the next curly bracket

```
#include <iostream>
#include <cstdlib>
using namespace std ;
void print_age() ; // func prototype
int main(){
        int age = 5 ;  // local variables
        cout << "age is " << age << endl ;
        print_age() ;
        return EXIT_SUCCESS ;
}
void print_age(){ // func definition
        // can't access age from here
}
```

The Viewing of Variables (2nd local example)

```
void print_age() ; // func prototype
void print_it(int x) ;
int main(){
        cout << “this is illegal “ << age ;
        int age = 5 ;  // local variables
        cout << “age is “ << age << endl ;
        if(age > 10){
                cout << age << “ is old “ ;
                print_age() ;
        }//if
        print_it(age) ;
        return EXIT_SUCCESS ;
}
void print_age(){ /* no age here */ }
void print_it(int x)
 { cout << x << “ is a party!” ; }
```

The Viewing of Variables

```
void print_age() ; // func prototype
int main(){
        int age = 1 ;
        for(int i = 0 ; i < 1 ; i++){
                int age = 2 ;
                cout << "age is " << age << endl ;
        }//for
        cout << "age is " << age << endl ;
        print_age() ;
        return EXIT_SUCCESS ;
}
void print_age(){
        int age = 3 ;
        cout << "age is " << age << endl ;
}
```

Note: 3 different variables, each "named" age

Memory Concepts: What's a Stack?

How is it possible to have 3 different variables, each “named” age, and C++ knows which one to use at any given time?

Let's introduce a new concept...

A Stack holds data...

It has 2 operations

1. **Push something on**
2. **Pop something off**

With a stack, the first thing you push on is the last thing you pop off

The Stack & Scope

In the previous example, an integer age is needed, so memory is allocated on the stack for it at the beginning of main.

The variable age now has a scope of main.

So, when the closing curly bracket of main is reached, the variable age can be destroyed.

Let's draw a diagram to explain what happens in the main function in the previous example

There is more than just the Stack

The stack only deals with local variables

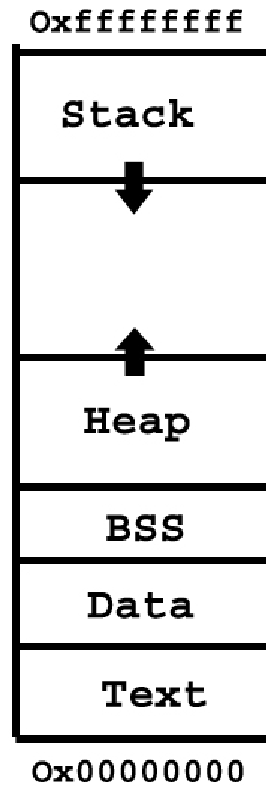
**Global variables, for example,
have their own section in memory**

Functions have their own section

There are also a few other sections...

Variables, Data Types, Storage

5 Parts of Memory for your Program in Linux



Variables, Data Types, Storage

5 Parts of Memory for you Program

Oxffffffff

- **Stack** (Local Variables) **– we know about**
- **Heap** (Dynamic Memory)
- **BSS** (Unitialized Global & Static Variables) ("Block Started by Symbol")
- **Data Segment** (Initialized Global & Static Variables)
- **Text Segment** (Where the actual compiled code is stored)

Ox00000000

Variables, Data Types, Storage

```
int s ;     // global uninit so in BSS
int p = 3 ; // global init so in data segment
int power(int number, int n) ; // text segment
int main(){  // text segment
        int x = power(10,2); // local variable, stack
        return EXIT_SUCCESS ;
}
int power(int number, int n){
        int value = 1 ; // local variable, stack
        if(n == 0) return value ;
        while(n > 0){
                value = value * number ;
                n = n – 1 ;
        }
        return value ;
}
```

Heap

Notice, that the Stack, BSS, and Data Segment are all basically handled “for us” ...

we just declare a variable, and C++
- grabs memory
- deletes it

The heap is something we'll learn about in the near future, where we have to
- grab memory for the variable
- delete the variable when done

Data Types and Sizes

The 4 common data types in C++ are...

1. **char** - 1 byte
2. **int** - size of integers on host
3. **double** - double precision float
4. **bool** - 1 byte

Data Types and Sizes

Typically...

character = 1 byte (8 bits)
short int = 2 bytes (16 bits)
int = 4 bytes (16 bits)
long int = 4 bytes (32 bits)
double = 8 bytes (64 bits)
bool = 1 byte (8 bits)

But not always...
Each compiler is actually free to choose the size of their data, thus you must not write programs that depend on certain sized data types

Pointers (on the Heap)

> Create a Pointer to the Memory

```
int * myintegerptr = (int*) NULL ;
```

> Grab Memory for the Pointer

```
myintegerptr = new int() ;    // (one integer, no value yet)
myintegerptr = new int(5) ;   // (one integer, 5 is the value)
myintegerptr = new int[3] ;   // (3 integer array, no values)
myintegerptr = {1,2,3} ;      // (3 integer array, 1,2,3 as values)
```

> Delete Memory

```
delete myintegerptr ; // deletes one integer
delete [] myintegerptr ;      // deletes entire array
```

Memory Viewer Program

> Sometimes it's tough to imagine the Stack, Heap, etc. in memory
> C++ is very cool though, in that it allows you to print the memory addresses of variables and functions!
> I've created a sample program to show you what I mean

Homework Extras

```
char MyInputString[501] ;

cin.getline(MyInputString, 500) ;



// Why 501?
// What about the \n?

// What's different with just regular cin?
// Where does the \n goes with regular cin?
```