Last Time

- C++ Introduction
  - C++ is a federation of related languages
  - Object Orientation is only achieved by the programmer
  - Allows a mix of procedural and oo elements

Last Time

- Variables, references, and pointers
  - The stack vs. the heap
- Structure
  - Class definition file (.h), class implementation
  - Namespaces
  - Include Guard

Pointers

- Pointers have two pieces of information

•A memory address

•The type (or void) of what is at that address

```
int a = 5;
int * b = &a;
*b = 10;
cout << a << endl;
```

Reference Type

- Holds the address of a variable (or object), but behaves syntactically like a variable (or object)

```
int a = 5;
int& c = a;
c = 10; //does not need dereference
cout << a << endl;
```

Reference Type vs. Pointers

- When defined:
  - Refers directly to what it references
    Cannot be null
    Cannot be left uninitialized
  - Cannot be reassigned to reference other variables (or objects)

*****	translates to "pointer" **`(int*) b -`** 'b' holds an int pointer **`int (*b)`** - pointer 'b' holds an int
&	translates to "reference" **`&a`** - gets the address of the variable 'a' b = &a; pointer can only hold that address if the types match *type*& **a** - defines 'a' as a reference type

Functions

```
returnType name (parameters)
{ statements; }
```

- Function calls allocate a new frame on the stack
  - Automatically allocates space for local variables
- This space goes away after function exits

Functions

- Parameter Passing
  - By value, pointer and reference
- Return values
  - By value, pointer and reference

Pass by Value

- Value of the actual parameter is copied into the new stack frame of the function

Function Definition

```
void addTen( int num )
{
     num = num + 10
}
```

Function Call

```
int x;
addTen(x);
//value of x unchanged
```

Pass by Pointer

- Value of the actual parameter is copied into the new stack frame of the function
  - But, that value is a pointer value

Function Definition

```
void addTen( int* p_num )
{
    *p_num = (*p_num) + 10
}
```

Function Call

```
int x;
addTen(&x);
//value of x changes
```

Pass by Reference

- Written like pass by value
  - From both the function definition and call perspectives
  - (except for the & in the formal parameter
- Behaves like pass by pointer

Function Definition

```
void addTen( int& num )
{
    num = num + 10
}
```

Function Call

```
int x;
addTen(x);
//value of x changes
```

Passing Objects

- Pass by value
  - Copy of object (its member data) placed on the call stack
  - Changes lost
- Pass by pointer, and by reference
  - Copy of pointer placed on call stack
  - Changes persist

Parameter Passing Example

Return Types

- By Value
  - Returns a value (or object) of the specified type
- By Pointer
  - Returns a pointer to the value (or object) of the specified type

Return by Reference Example

Function Definition

```
int& max(int &x, int &y)
{
    if ( x > y )
    {
       return x;
    }
    else
    {
    return y;
    }
}
```

Function Call

```
int a = 3, b = 5;
max(a,b) = 10;
cout << b << endl;
```

Return by Reference Example

Safe Pointers

- From stack to heap
- From heap to stack
- From stack to stack in earlier frames
  - Like pass by pointer/reference
- Will point to what you expect

Not Safe

- From stack to stack in later frames
  - Possible when returning pointers/references
- From heap to stack
- Not guaranteed to point to what you expect

Objects are safe to live on the Stack

- What does this mean?
  - Objects CAN be local variables
    - When the Object is only used within that function
    - Pointers to the object are passed to other functions as parameters
    - No pointer to the object is given to heap objects
    - No pointer to the objects is returned

Should be on the heap...

- When we want to point to the object from other objects on the heap
- When we want to pass the object back as a return value