

# For Wednesday

- Read Savitch, chapter 12
- C++ Practice 4 due

# Friend Functions

- What?
- Why?
- How?

# Pointers

- Value is a memory address
- Similar to references
- Can manipulate the value of the address directly
- Must explicitly dereference the pointer to access the thing being referred to (or pointed at)

# Declaring Pointers

```
int *p, *q;
```

```
char *str;
```

```
Student *stuPtr;
```

# Notes on Pointers

- In a multi-variable declaration, the \* is required for each individual variable
- The \* is not part of the name of the variable; it is part of the type
- Each pointer has an associated type, called the target type
- The target type does not affect the actual value of the variable, but it does affect C++'s manipulation of the variable

# Values of Pointers

- Pointers are not automatically initialized to anything
- You must place an address in them
- Size of pointers varies based on the capabilities of the machine
- Often pointers are the same size as longs

# The Address Operator

- Allows us to determine the address of an item in memory.

# The Address Operator

```
int v, *ptr;
```

```
double x, *y;
```

```
ptr = &v;
```

```
y = &x;
```

# Dereferencing Pointers

- Use the asterisk before the pointer variable

```
v = *ptr;
```

# Comments

- Do not dereference a pointer unless the address points to an appropriate value
- NULL is usually used as the value to indicate a pointer that is pointing to nothing
- NULL is actually the 0 value pointer

# Practice with Pointers

- Write a declaration-initialization to establish a pointer, `charPtr`, to a location that stores a character and place the letter 'B' in that location. Declare any other variable necessary.

# Arrays and Pointers

- What **is** an array name?
- If we declare

```
int    numArray[10];
```

what is the value of numArray?

# Arrays and Pointers

- The name of an array is a constant pointer
- That is, `numArray` is the address of `numArray[0]`

# Using Pointers As Arrays

```
int scoreArray[10];  
int *scorePtr = scoreArray;  
*scorePtr = 10;  
scoreArray[1] = 12;  
scorePtr[2] = 11;
```

# Addresses of Array Elements

- Address of array elements are computed by adding the address of the first element to the product of the size of the elements and the element index
- Thus in an integer array `ar` starting at address `100`, the address of `ar[2]` would be `100 + 2 * sizeof(int)` or `108` if we assume 4 byte integers

# You Try It

- We have the following declarations:  
`char carr[10];`  
`long larr[20];`
- Assume chars are 1 byte and longs are 4 bytes.
- The address of carr is 240 and the address of larr is 320
- What is the address of carr[6]?
- What is the address of larr[4]?

# Pointer Arithmetic

- We can actually do arithmetic (addition and subtraction) with pointers
- This works very much like the computation to find an element in an array
- You take the current value of the pointer and add the number being added times the size of the target data type

# Pointer Arithmetic Practice

- Given:

```
int *ptr1;  
char *ptr2;  
float *ptr3;
```

- What is:

```
ptr1 + 5  
ptr2 + 11  
ptr3 + 3
```

# Final Note

- For any array **a**:

**a[i] == \*(a+i)**

# Pointers and Classes

- Can declare pointers to a class type (just like to an int or double)
- Have two choices for dereferencing.
- Can use  
`(*objPtr).Method()`  
or  
`objPtr->Method()`

# this

- Special variable to refer to the calling object.
- Most C++ programmers always call methods of the calling object using the *this* pointer.

# Dynamic Allocation

- Pointer values created using the address operator are of limited utility
- Dynamic allocation creates variables in memory that only have a pointer (no direct variable name)
- Note that this allows us to create an array of arbitrary size at run-time

# How Do We Do It?

- Memory allocation is done using `new`
- Examples:

```
int *intPtr = new int;
```

```
int *arrPtr;
```

```
arrPtr = new int[50];
```

```
Student *myStuPtr1, *myStuPtr2;
```

```
myStuPtr1 = new Student;
```

```
myStuPtr2 = new Student("Mary Smith", 3.45);
```

# Freeing the Memory

- When we finish using statically allocated variables, the memory used by them is freed up (actually when the function ends).
- Dynamically declared variables must be freed by the programmer.
- Done using delete:

```
delete intPtr;
```

```
delete [] arrPtr;
```

# Memory Leaks

```
int *intPtr = new int;
```

```
*intPtr = 5;
```

```
intPtr = new int;
```

```
*intPtr = 10;
```

```
// What happened to the first one?
```

# Notes on Dynamic Arrays

- Need three pieces of information associated with an array
  - Where's the data?
  - What's the capacity?
  - What's the size (current number of elements)?

# Dynamic Classes

- Classes that use dynamic memory to store some/all of their data members
- Constructor typically allocates/initializes the dynamic portions of the class

# Destructors

- Used to free the dynamically allocated portions of an object when the object is destroyed (either by delete or by going out of scope if statically allocated).
- Called ~classname
- Like constructor, has no return type
- Also always has no parameters

# Assignment and Initialization

- What's the difference?

# Assignment

- Default assignment operator
  - does a byte-by-byte copy of the object
  - Why is this an issue?
- Solution:
  - Create your own assignment operator
  - Either copy the pointer or the data as appropriate for your class
  - Must be a member function
  - Always returns `*this`

# Initialization

- Copy Constructor