SOUTHERN ADVENTIST UNIVERSITY School of Computing

More on Functions

Chapter 10

For Next Time

- Read Chapter 10

Global Variables

- Local variables: declared within functions
  - "Local" to that function
  - Scope is limited to function
  - Names do not conflict with any other variables
- Global variables: declared outside any function
  - "Global" to the file and/or program
  - Scope is entire file and/or program
  - Declared **`static`**: scope limited to file

Prefer Locals

- Space for local variables occupied only when the function is executing
- Local variables in one function are completely independent of local variables in another function (they cannot interact or influence each other across function boundaries)
- Local variables "start fresh" each time the function is called
- A function that uses no global variables can be used as is in other programs

Locals Hide Globals

- If a local variable has the same name as a global variable, the local variable is the one used within the function
- To access the like-named global, use the scope resolution operator, **`::`**
  - **`::x`**
- The compiler resolves names as follows
  - If it is declared locally, it's local
  - If it is not declared within the function, it checks for global declaration
  - If it is not local and it is not global, it is undefined

Locals vs. Globals

- Locals are uninitialized
- Globals are automatically initialized to "zero"
  - Numbers are 0
  - Booleans are false
  - Other types (to be seen) are "zero-ish"

Static Locals

- If a local variable is qualified as **`static`**, it lives between function invocations
- Its value is retained between function calls
- Space for **`static`** locals is allocated when the program begins executing and is deallocated when the program terminates

Static Globals

- If a global variable is qualified as **`static`**, its scope is limited to the file in which it is declared
- The **`static`** qualifier renders a variable "file local"
- Functions can also be declared **`static`**
  - Such functions cannot be used outside of the file in which they are declared

Function Signature

- A function's *signature* consists of its name and types of parameters
- A function's signature can be determined from its prototype and/or its definition
  - `void f(int x, double y) { … }`
  - `void f(int, double);`
- A function's return type is not included in a function's signature

Function Overloading

- C++ allows definition of multiple functions with the same name
- Two functions within the same program that have the same name are said to be *overloaded*
- Examples:
  - `void f(int x, double y) { … }`
  - `void f(int) { … }`
- Overloaded functions **must** have different signatures

Factorial

- One definition of the mathematical factorial function:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 2 \cdot 1$$

- Another, recursive, definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise.} \end{cases}$$

C++ Recursive Factorial

```
int factorial(int n)
{
    if ( n == 0 )
        return 1;
    else
        return n * factorial(n - 1);
}
```

How it Works

```
factorial(6) = ?
```


How it Works

```
factorial(6) = 6 * factorial(5)
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
```

How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
```

How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
             = 6 * 5 * 24
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
             = 6 * 5 * 24
             = 6 * 120
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
             = 6 * 5 * 24
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
             = 6 * 120
```

How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
             = 6 * 120
```


How it Works

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
             = 6 * 120
             = 720
```


Correct Recursion

A correct recursive function must

- Call itself within its definition
  - This is the recursive case
- Provide a way that it does not call itself
  - This is the base case
- Provide some sort of conditional construct to select between the recursive and base cases
  - **`if/else`** or **`switch`** statement
- Each recursive call must move the execution closer to the base case
  - Otherwise infinite recursion (stack overflow) will result

Making Functions Reusable

- Commercial C++ programs often consist of hundreds of separate C++ files that are compiled separately and linked together to make the executable file
- A function defined in one file may be used by code in many other files
- A function must have exactly one definition, in one file
- Since each file has to compiled separately, how can we use the same function in multiple files?

Interface vs. Implementation

- Put the function's prototype in a **`.h`** header file
- **`#include`** the header file in all files that use the function
- Define the function in one **`.cpp`** file to be linked in with all the other compiled files
- The **`.h`** header file specifies the function's *interface*
- The **`.cpp`** source file provides the function's *implementation*

Variables

All variables have a

- name
- type
- value
- location in memory

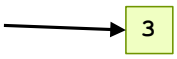
```
int x = 3;
```

3

To this point we have not been concerned about the variable's memory location

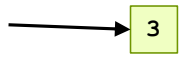
Address of a Variable

- Memory locations are addressed with whole numbers beginning at address 0
- `&` is the "address of" operator
- If `x` is a variable, `&x` is the memory location of `x`
- `&x` is therefore a number, but it is more convenient to think of it as a pointer to a place



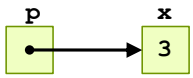
Pointer Variable

- The address of a variable can be assigned to a special variable called a pointer
- While an address is really a number, and so a number is assigned to a pointer, C++ is very strict and does not allow the free interchange of pointers and numeric values
  - To attempt to do so is usually an error



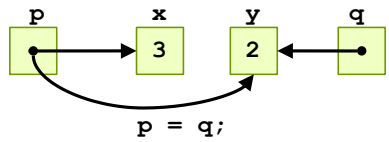
Pointer Variable

```
int x = 3;
int *p = &x;
```



Reassigning the pointer

```
int x = 3, y = 2;
int *p = &x, *q = &y;
```



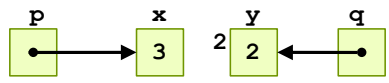
Dereferencing the Pointer

```
int x = 3, y = 2;
int *p = &x, *q = &y;
```


```
*p = *q;
```


Dereferencing the Pointer

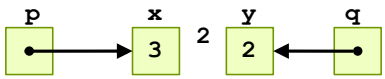
```
int x = 3, y = 2;
int *p = &x, *q = &y;
```




```
*p = *q;
```

Dereferencing the Pointer

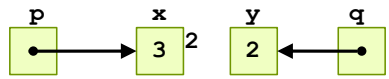
```
int x = 3, y = 2;
int *p = &x, *q = &y;
```




```
*p = *q;
```


Dereferencing the Pointer

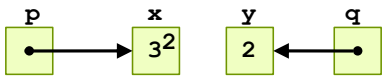
```
int x = 3, y = 2;
int *p = &x, *q = &y;
```




```
*p = *q;
```


Dereferencing the Pointer

```
int x = 3, y = 2;
int *p = &x, *q = &y;
```




```
*p = *q;
```


Dereferencing the Pointer

```
int x = 3, y = 2;
int *p = &x, *q = &y;
```


```
*p = *q;
```


Assignment with Pointers

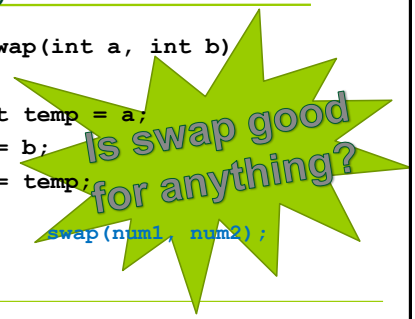
```
int x = 3, y = 2;
int *p = &x, *q = &y;

p = ...;   Makes p point somewhere else
*p = ...;  Changes the value that p points to
```


Call by Value

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

swap(num1, num2);
```



Call by Reference with Pointers

```
void swap(int *a, int *b)
{
     int temp = *a;
     *a = *b;
     *b = temp;
}
          swap(&num1, &num2);
```


Reference Variables

- An alternative to pointer variables

```
int x = 3;
int& r = x;
r = 5;    //  Reassigns x!
```


Call by Reference via References

```
void swap(int& a, int& b)
{
     int temp = a;
     a = b;
     b = temp;
}
               swap(num1, num2);
```


Pointers in C and C++

- Pointer variables are supported in both C and C++
- Reference variables are available only in C++
- Since C++ programs often use C library functions, knowledge of pointers is essential
- Call-by-reference via
  - Pointers: absolutely no question—call by reference is happening
  - References: looks just like call by value—have to look at the function prototype to be sure
- References were introduced to allow C++ to do some interesting things like operator overloading for objects

Next . . .

Vectors