Static and pointer variables, and other things you can do with functions.

CS 10A – FUNCTIONS PART 2

Static Variables

- A static variable is a variable that is essentially declared once and its place in memory is permanent for the remainder of the program’s execution.
- In other words, a static variable never reinitializes. A declaration statement doesn’t reset the value of a static variable. You’ll have to manually reset the variable value.
- Use of static variables is generally not recommended since it can be hard keep track of their value at any given time without the reinitialization.
- Functions can be static too, but that’s for a later class.

Using Static Variables

Program

```
int main()
{
        for(int i = 0; i < 5; i++)
        {
                static int x = 0;
                x += 5;
                cout << x << endl;
        }
        // Without the static keyword, the output
        // will always be 5 due to reinitialization.
        return 0;
}
```

Console

```
➢ ./a.exe
5
10
15
20
25
```

External Libraries – Member Functions

- Functions like substr() and length() from the string library are used by attaching it to the end of a variable with a period. These are known as member functions.
- Member functions use periods because they act on the variable they're attached to. The function is a member of a variable type it can act on. In a sense, the variable in question is the primary input parameter for the member function. They're almost never void types.
- Declaring our own member functions is outside the scope of this class. Just know what they're called now that you know how to use functions.

Recursion

- Recursion is a type of looping, applied to functions. A recursive function one that calls itself from within its own function. This is the most literal form of inception you can get in programming.
- Logic statements lets us control how and when recursive functions should occur. Without them, we'd have an infinite loop, despite the lack of while and for statements.
- Recursive functions are widely used in the world of algorithms, and useful for AI design or puzzle games.

Recursive Functions

Program

```
int recur(int x)
{
        if(x > 50)
                return x;
        else
                return recur(x + 5);
}

int main()
{
        cout << recur(1) << endl;
        return 0;
}
```

Console

```
➢ ./a.exe
51
```

Pointers

- A pointer is a type of variable that, instead of storing the value of the variable, it stores its memory address.
- The memory address is some hexadecimal value attached to a variable once it has been declared (use a * to declare a pointer).
- To access the address of any non-pointer variable, use & just before the variable name.
- To access the value stored at the address given by the pointer variable, use * just before the pointer name.
- Aside from simply allowing us to pass around arrays between functions, pointers also allow us to pass around local variables, allowing functions to write back to a specific variable instead of having to the an additional step of reassigning. This is useful in the event we want to store multiple outputs from a function.

Pointers and Non-Pointer Variables

Program

```
int main()
{
	int x = 5;
	int * p = &x;	// Address

	cout << p << endl;	// Address
	cout << *p << endl;	// Value

	return 0;
}
```

Console

```
➢ ./a.exe
0xffffcc14
5
```

Passing Local Variables via Pointers

Program

```
void pass(int * save_slot)
{
        *save_slot += 10;
}

int main()
{
        int x = 0;
        pass(&x);
        cout << x << endl;
        return 0;
}
```

Console

```
➢ ./a.exe
10
```

Empty Loops and Functions

```
bool isRNGOdd(int i, int * save)          // Checks whether if input number is odd, saves number
{
        *save = i;
        return (i % 2);                   // Remember that int and bool are interchangable
}

int main()
{
        int num;
        srand(time(NULL));                // RNG seeding
        while(!isRNGOdd(rand(), &num));           // For/while statement followed by ‘;’ is an empty loop
                                                  // But is still a functional finite loop due to function
        // Without an accompanying do, the while statement is a self-contained loop.
        return 0;
}
```

Overloading Functions

- You can create multiple versions of the same function by having different parameters for each one.
- The function name and output type can be completely identical between two or more functions. However, if there is a difference between the SET of input parameters, the compiler considers the functions to be unique.
- This way, you can create a function with optional inputs. You can also create a function to accept multiple parameter types in the same slot. If you need to change how each version of a function behaves, you can do that too.
- You can now adjust your function behavior depending on what info the user provides the program.

Overloaded Function Example 1

Program

```
int sum(int a, int b)
{
		return a+b;
}
double sum(double a, double b)
{
		return a+b;
}
int main()
{
		cout << sum(3, 5) << endl;
		cout << sum(3.5, 5.5) << endl;
		return 0;
}
```

Console

```
➢ ./a.exe
8
9
```

Overloaded Function Example 2

Program

```
int sum(int a, int b)
{
		return a+b;
}
int sum(int a, int b, int c)
{
		return a+b+c;
}
int main()
{
		cout << sum(1, 6) << endl;
		cout << sum(1, 6, 4) << endl;
		return 0;
}
```

Console

```
➢ ./a.exe
7
11
```