6.096 Introduction to C++
January (IAP) 2009

char vs. char *:

Just to clarify, a char variable stores a single character value – the letter T, a colon, a newline (\n), etc. To the computer this is no more than a small integer number that represents a particular character – 97 for the letter 'A', for instance. This is what was meant by the range of number values for a char variable. Normally, though, a char should be used primarily as a character, not as a number. Each constant char value is expressed by including the character in single quotes – e.g. '\n'.

A char *, or a string, stores a series of 0 or more characters, each of which can be any one of the values a char variable could have. A constant char * value is indicated with "double quotes".

Functions

Functions as modules:

A function is a block of code with a name.

To motivate the idea of functions, imagine trying to program a robot to launch someone from Timubktu to Boston via catapult. Dumping all the code into main would be absurdly long and difficult to keep track of, and nobody who read a single line would have a clue where that line fit in. We'd lose track of our programming goals.

It'd be much more intuitive to break it up into:

- Build the catapult
- Set up the catapult
- Fire the catapult

Before we go any further, this is a good way to code – write "pseudocode" first, using plain English to describe what's supposed to happen, then keep expanding each sentence until it's sufficiently detailed that you can express it as if-statements, loops, etc. Often some of the initial English descriptions will describe good ways to divide up the code into functions.

To turn the catapult pseudocode into code:

```
int main() {
        buildCatapult();
        setUpCatapult();
        fireCatapult();
}
```

This style is often a good design for main – main a few calls to some functions that do all the real work.

Each of the functions buildCatapult, setUpCatapult, and fireCatapult is said to be "invoked" or "called" via a "function call" from "calling function" or "caller" (in this case, main). To call a function, type the name of the function, followed by parentheses.

Now that we've packaged up each part of our big procedure and given it a name, we have a bunch of smaller units, each of which we can independently

- Write
- Debug
- Test
- Reuse in later programs

We can also now split development of the program among multiple people, with each developing a different set of functions.

This sort of independence of program components is called *modularity*, and is critical to good software engineering. Splitting code up into functions increases the modularity of a program.

Function Definitions:

Example:

```
void printHello() {
	cout << "Hello world!";
}
```

This definition specifies that we want to name the sequence of commands within the curly braces ({…}) printHello, so that we can then call it from another function, such as main, with the syntax printHello();.

Example:

```
bool isMultiple(int a, int b) {
	if( a % b == 0 )
		return true;
	else
		return false;
}
```

Here, bool is the *return type*, isMultiple is the function name, a and b are argument names, and return true; and return false; are return statements. Each concept is discussed in detail below.

Returning values:

The printHello function was just a shorthand way of telling the computer to run a sequence of commands – it issued instructions. A function like isMultiple can be thought of as asking the computer a question, which will require some commands to be run to figure out, but which ultimately gives back an answer.

In this example, the question we are asking is "Is a a multiple of b?" The name of the function must be preceded by a keyword specifying the data type of the answer we are expecting – the *return type*. In this case, since we are asking a yes or no question, we want our answer – our *return value* – to be a bool. At various points in the program, we specify *return statements* (here, return true; and return false;), which indicate what the answer should be if the conditions leading to that statement are fulfilled.

The *void return type* specifies that there is no return value, which generally means that this function is for issuing instructions, not asking a question. Returning a value from a void function is a syntax error. Not returning a value from a non-void function is not a syntax error, but will usually cause runtime errors.

Parameters/Arguments:

In the catapult example, we might want to make our setUpCatapult function reusable for more than this one trip. We'd at least need to specify the weight of the projectile, direction, and distance each time we used it. If we add *parameters* or *arguments* to our function call to specify the weight as 150 pounds, the compass direction as 71.5 degrees, and the distance as 5584 miles, the function call might look like: setUpCatapult(150, 71.5, 5584);. The order of arguments is specified by the function definition – it must specify that the first argument is weight, the second distance, and so on.

In the isMultiple function definition, a and b are arguments – variables that are declared between the parentheses of the function definition and can then be used throughout the function body. The function call isMultiple(25, 5) specifies that when the body of the isMultiple function is executed for this call, the variables a and b in the function should store the numbers 25 and 5, respectively.

Functions can have arbitrarily large numbers of arguments – arbitrarily long *argument lists*.

Any expression can be passed as an argument into a function – for instance, the following is a legal function call for a maximum function defined to take 3 arguments:
maximum(a, b, a + b, maximum(c + d, e, f)).

C++ functions in can be compared to mathematical functions: just like f(x, y, z) means some mathematical expression named *f* evaluated for certain values of x, y, and z, isMultiple(a, b) means some set of instructions named isMultiple evaluated for certain values of a and b.

Argument Promotion/Demotion:

Normally, the arguments must be of the types specified in the function definition. If the compiler has a predefined way of converting the arguments you pass into the types specified by the function definition, it will do this automatically. This can result in promotion or demotion of the argument value – conversion into a higher or lower-precision data type, respectively.

Example: A function defined as void myFunc(int a) {…} can be called as myFunc(24.3), since the compiler can convert 24.3 to an int by truncating the decimal.

Early exit:

No more code from a function is executed after one of that function's return statements; a return statement ends the function's execution.

Usually you want the logical path to return statements to be straightforward, but sometimes it's useful to return early from a function. You often want to do this in response to an error or an invalid argument.

Example:

```
bool isPrime(int number) {
        if(number <= 1)
                return false;
        …
}
```

No number less than 2 can be tested for primality, so if the argument the function received violated this constraint, it simply exits without even bothering to run the tests in the rest of the function.

Function Prototypes:

If you try to call a function before it is defined in your source file, you will get a syntax error. The solution is to know the compiler know ahead of time what's coming with a *function prototype*. This looks exactly like the first line of a function declaration: return type, function name, parentheses, and argument list. Instead of braces, though, you just put a semicolon. You also don't need to include the names of the variables in the argument list.

Function prototypes are usually put either at the beginning of source files, or in separate *header files* (usually with .h extensions) which are then included with the #include preprocessor directive. Usually only large groups of prototypes should be put in header files.

Example: If you define isPrime after main in your source file but want to call it from main, put the line bool isPrime(int); or the line bool isPrime(int number); before main.

Standard Library Functions:

There are lots of prepackaged functions to use in standard C++ libraries. To use them, you must include the appropriate header files.

These functions are very efficient and well-implemented. They should be used in preference to hand-crafted functions whenever possible.

Example: The sqrt function (in header file cmath) takes square roots. The rand function (in cstdlib) generates random integers.

Scope:

Variables exist within *scopes* – blocks of code within which identifiers are valid. An identifier can be referenced anywhere within its scope, as long as the reference comes after its declaration.

Global variables – variables declared outside of any function – have *file scope*, meaning they can be referred to from anywhere in the file. Global variables should generally be avoided, except for global named constants.

Every set of braces is its own scope, and can contain *local* (i.e. non-global) *variables*. The moment the set of braces in which a variable was declared ends, the variable *goes out of scope*, i.e. it can no longer be referenced as an identifier. The program usually erases variables that have gone out of scope from memory. The scope of arguments to a function is the entire function body.

Example: A variable declared in the first line of a function can be referred to anywhere in the function, but nowhere outside of it. The moment the function exits, the variable ceases to exist in memory.

No two variables may share the same name within a scope. The exception is when you have opened another scope with a curly brace ({), at which point you may declare new variables of the same name. If another variable is declared within a smaller scope, the variable name refers to the variable in the narrowest enclosing scope.

Example: In the following code, there are two variables named i. When i is referenced from within the loop, it is interpreted as referring to the one declared in the loop. Every time the loop executes again, i is created anew.

```
void sampleFunc(int i) {
        for( int j = 0; j < 10; j++ ) {
                int i = 0;
                cout << i;
        }
```

By-Value vs. By-Reference:

The arguments we've seen so far have been passed *by value* – the computer makes a copy of the values passed as an argument and sets new variable (e.g. a and b in isPrime) to those values.

Sometimes we want to allow a function to have access to variables that exist in the scope of the caller. For certain variable types (which we'll get to later), copying the value may be very computation-intensive, and may take up too much memory. To give the callee access to the caller's data, we pass the arguments *by reference* – the variable in the function actually refers to the same value in memory as in the caller.

The syntax for declaring arguments to be passed by value is simply an ampersand after the data type of the variable name.

Example: `void squareByReference(int &value) { value = value * value; }`

If we have a variable number and pass it into squareByReference, the value of number will be updated when squareByReference exits. Since it updates the original value in memory, the function does not need to return a value.

Allowing the callee to modify the caller's data is very useful, but sometimes very dangerous.

const Arguments:

The "*principle of least privilege*" dictates that no module of a program should have the right to change more data than it needs to. When a function is passed an argument by reference, very often the caller doesn't want it changing the value of the argument. Even if an argument is passed by value, it still may be that the function doesn't need to change the argument's value anywhere, and trying to do so would likely be a logic error.

To prevent unwanted changes to arguments, the arguments can be declared const. For the duration of the function, an argument declared const cannot be changed.

Example:

int square(const int number) { return x * x;}

Trying to change the value of number in the body of square would be a syntax error.

Default Arguments:

Default arguments allow you to specify default values for the last few arguments of a function, so that you need not pass any value for these arguments unless you want the arguments to be different from the default.

Example:

```
void print(const char *str = "\n") { std::cout << str; }
```

This specifies to the compiler that the function call print() should be treated as equivalent to print("\n") – that is, the default thing to print is a newline, and we only need to explicitly pass a string if we want to print something else. If you call a function relying on default values, you cannot pass any values for arguments after the first argument for which you want to use the default. (Of course, this means that all the arguments after that must have default values.)

Example: If we were going to use our catapult mostly to launch items from Timbuktu to Boston, but sometimes between other places, we might use the function prototype:

void setUpCatapult(double weight, double compassDirection = 71.5, double distance = 5584) …