General Syntax

Statements are the basic building block of any C program. They can assign a value to a variable, or make a comparison, or make a function call. They must be terminated by a semicolon. Every statement has a value: the simplest statement, a number, has the value of the number. An arithmetic expression has the value of the expression; a conditional like 2 > 5 has the value false.

Comments are blocks of text ignored by the computer and are useful for reminding yourself of what some code does. Comments are any text between two flags: /* and */.

Operators

Arithmetic

+ - * / % — plus, minus, multiply, divide, modulus

Comparison

< <= > >= == != — less than, less than or equal, greater than, greater than or equal, equal, not equal

Assignment

= is the basic assignment operator – the variable on the left gets the value of the expression on the right. The assignment operator can also be combined with any of the arithmetic operators to update a variable's value. So, for example, `a += 2` is the same as `a = a + 2`.

Boolean

&& || ! – AND, OR, NOT. Note that AND and OR are short-circuit operators. That is, if the left side of the AND is false, the right side is never evaluated. Similarly, if the left side of the OR is true, the right side is never evaluated.

Variables

Variables are things that hold values. They must first be declared at the top of your function:

Types

All variables have types. They can be integers (int) or have decimals (float) or represent strings (char). You can change variables from one type to another by using a cast; this only makes sense for some types, like int to float or float to int. Casting a float to an int truncates the decimal part.

Syntax

```
int i;
float j = 1.35;
int k = (int) j;
```

The above code declares three variables, two integers, i and k, and a float, j. After this bit of code i is uninitialized and has some unknown value; j is equal to 1.35 and k is equal to 1, since it is an integer cast of j.

Scope

Variables are only accessible within a certain scope. Variables' scope is defined by braces: { }. A variable is accessible within the brace in which it is declared and all braces underneath that brace level. A variable declared outside a brace (i.e. just in a file) is accessible everywhere; it is called a global variable. Variables declared within braces are local variables.

```
int j;
{
     int k;
     /* point a */
     {
          int l;
          /* point b */
     }
}

{
     int m;
     /* point c */
}
```

In this example, variable j is accessible at points a, b, and c, but k is accessible only at points a and b, l is accessible at point b, and m is accessible at point c.

Local variables are useful because they allow you to reuse variable names throughout a program.

Arrays

Arrays are sequences of variables. To declare an array, you may use one of two syntaxes: `int myArray[5];` declares a array of 5 integers with arbitrary values; `int myArray = {0,1,2,3,4};` declares an array of 5 integers with values 0, 1, 2, 3, and 4.

To get at a value in an array, use square brackets. So, if we have:
`int myArray = {5,-1,6,90,4};`
`myArray[0]` is 5, `myArray[1]` is –1, and so on. You can also assign to an array by using the square brackets, so `myArray[0] = 9;` updates the first value in the array to 9.

Be careful to keep the value within the square brackets within the size of the array. Interactive C will allow you to access a value past the end of the array, which will be some undefined value and may cause your program to crash.

Conditionals

A conditional is the building block of programming: it allows a program to do different things based on some data. The basic conditional is the if statement:

```
if (expression) {
     /* executed if expression is true */
}
else {
     /* executed if expression is false */
}
```

The else part of the if statement is optional. Any code following the if statement (i.e. outside the braces) is executed regardless of the value of the expression.

The expression can be any valid C statement. Generally it will be a boolean expression with some comparisons, but it can also be an arithmetic expression. If it is an arithmetic expression, 0 is false and all other values are true.

Be careful not to confuse the comparison operator ==, with the assignment operator =. Suppose you have:

```
int myInt = 0;
```

`myInt == 5` evaluates to false, but `myInt = 5` evaluates to 5 and is therefore true.

Loops

Loops are the building block of complex programs; they allow you to do things lots of times depending on a condition. There are three basic loops: the for loop, while loop, and do-while loop:

```
for (initializer; conditional; increment) {
     /* your code here (body of loop) */
}
```

The for loop has three parts that control its behavior separated by semicolons. The initializer is executed before entry into the for loop. The conditional is evaluated before each entry into the for loop (including the first one) – if it is true, then the for loop is entered, if not, it is skipped. The increment expression is evaluated after each evaluation of the body of the for loop. Generally you will want to change something that is checked in the conditional in the increment or in the body of the loop; otherwise you'll get an infinite loop.

```
while (conditional) {
     /* your code here */
}
```

The while loop evaluates the conditional and executes the body as long as it is true.

```
do {
     /* your code here */
} while (conditional);
```

The do-while loop is a lot like the while loop, except that the body is guaranteed to be executed at least once since the conditional is checked only after the body has been executed.

Functions

Functions allow you to split up your program into easily manageable subunits. They are a collection of statements that can be run at any time, can take arguments, and can return values. One special function is main(), which is automatically called when the robot starts running.

Arguments

A function can take 0 or more arguments; these behave the same as local variables. The function's arguments can be changed without affecting the values that were given in the calling function.

Return Values

A function can return one expression of some type to whomever called it; the caller can then use the return value of the function in an expression or assign it to a variable. A function can also return type *void*, which means it returns no value.

Syntax

```
<return type> <function name>(<arg type> <arg name>, ...)

So, for example:
int myFunction(int a, int b) {
     /* code here */
}
```

defines a function called myFunction that takes two integers as arguments and returns an integer. myFunction can then be called from another function and have its return value assigned to a variable:

```
int a = myFunction(1, 2);
```

or you can use the function as an expression:

```
if (myFunction(1, 2)) {
     /* code here */
}
```

in which case the body of the conditional is executed if myFunction(1, 2) returns a non-zero value.

An important function is `main`, which is called by the Handyboard's operating system when it first starts; i.e. when you first turn the Handyboard on. You must write a main function if you want your Handyboard to do anything. `main` has the specification:

```
void main() { … }
```

That is, it takes no arguments and returns nothing.

Interactive C Library Calls

The library calls allow you to gather data from your sensors (your robot's inputs) and move your motors and servos (your robot's outputs). Basically, the idea is to use library calls to gather data from your sensors, execute some conditionals on that data, and respond to the environment with library calls to change the robot's relationship to the world.

Read the course notes and Interactive C manual for a complete reference. The essential calls are given below:

`void motor(int m, int speed)`
Sets motor port `m` to speed `speed`. Use this to activate the motors on your robot. `speed` can be positive or negative for different directions of motion.

`int analog(int port)`
Returns an analog reading from 0-255 representing the sensed voltage on analog sensor port `port`.

`void printf(char* format, ...)`
Prints a string on the LCD with any number of variables. See the Interactive C manual (or type `man printf` on an server machine).

`void enable_servos()`
`void servo(int port, int period)`
Use enable_servos to turn on power to your servos. `servo` sets the servo on port `port` to a position proportional to `period`; `period` is a value between 0-4000. This is a sweep of 180 degrees from 0 to 4000.

Other cool stuff you should look up in the Interactive C manual:
Multiple processes
Shaft encoders
Button presses
Pointers
Other variable types (unsigned int, double, short) – don't use long!