Programming in RobotC

Bits and Bytes

`1` or `0`

Every bit of information stored in a computer is made out of bits, which are found in two states: `1` or `0`, true or false, on or off. Thus, it is able to store two states. Essentially, it can store a yes/no response to a closed question. For example, the question: *Do you live in New Zealand* is a question with a yes or no answer, which can be represented by 1 bit, as a `1` state.

8 bits- a byte

Things get interesting when you have more than one bit. In modern computers, 8 bits are called a byte (or octet). Using 8 bits, you can have 256, or 2^8 states, instead of the two states (2^1) you can have with just one byte. These 256 states can be represented as numbers.

`0000 0000`, a byte made of all zeros represents the number 0.

`0000 0001` represents the number 1.

And so on, until you reach `1111 1111`: 255. This is not 256 because we have already used one of the possible 256 states for the number zero.

The table

But what would `1010 1001` mean in decimal (base 10, the way normal people count)? You could count up by one each time like we did with the numbers 0 and 1, but that would take way too long. Instead, we can use a table with 8 rows.

The very right hand column of the table takes a value of 2^0, which equals 1 (n^0 always equals 1). The one to the right of it takes the value of 2^1, or 2, and so on.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
`1`	`0`	`1`	`0`	`1`	`0`	`0`	`1`

Enter each bit into the table, and you get the above. Here, each `1` value corresponds to the number above it, and the resulting number is the sum of all `1`'s. Because the first bit is a `1`, you count the 128 above it. The second bit is a `0`, so you don't add 64. The third is `1`, so you add 32 to 128. You do this until you reach the end:

$$128 + 32 + 8 + 1 = 169$$

And now you have the result: 169.

Sign or unsign

Currently, we only have 0 and positive numbers- this is called *unsigned* as there is no positive or negative sign on the number. When a number is *signed*, the range of 256 states is split between negative values, 0, and positive values. In the case of 1 byte, the range changes from 0 to 255 into -127 to 127. This is why the motors and joysticks receive an output and input ranging from the latter. Read this Wikipedia article to learn more.

Note: Usually, the range would be between -128 and 127, but RobotC seems to stop 128 from being achieved for symmetrical performance forwards and backwards.

Types

Char

A *char* has a maximum and minimum value as written above: -128 to 127 (or similar), and can also display a single character, such as `a`, `!` or `;`. If you specify the type as `unsigned char` instead of just `char`, then the number range changes to from 0 to 255. If it is a character (not a number), it must be between `'` - single quotes.

String

a `string` is an *array* of `char`'s, and it can hold what is known as a string. A string is essentially a number of `char` values grouped together. For example:

```
//In RobotC:
string myString = "Hello World";

//Ignore these
//In Normal C:
char *myArray = "Hello World";
//is the same as:
char myArray[] = "Hello World";
//Which is the same as:
char myArray[11] = "Hello World";
```

See Arrays for more details.

Integer

Integers take up either 16 or 32 bits, and are specified as either a `int` or `unsigned int`. They are much the same as a `char`, except that it must be an integer and has a much greater range- up to 4.2 billion.

The Vex Cortex uses 32 bit integers.

Long and long long

Same as `int`, but `long` is 32 or 64 bits, and `long long` is 64 bits.

Short

`short` or `short int`. 16 bits. A range between -32768 to 32767 for `signed short`.

Floats

You have learnt about signed and unsigned numbers- more specifically, signed and unsigned *integers*. Integers are any whole numbers, from 0, -12 to 1234567891 (that's a prime). But what if you want a decimal number like 3.14, or 0.1? Here, we need a new type of number. These are called *floats* or *doubles*.

Floats are stored in 32 bits, or 4 bytes, and use a slightly modified form of *scientific notation*.

The format is:

```
1    0001 1010    1010 1011 1011 0001 0000 011
sign  exponent              fraction
```

1. The first section, the one bit indicates the sign- `1` is negative, `0` is positive
2. The second section is made out of 8 bits is the exponent. Its range is somewhere around -126 to 128
3. The last is made of 23 bits, and is a fraction. It is the decimal section of it

Doubles, surprisingly, use double the storage- 8 bytes.

The numbers can be very large and very small, but can also be inaccurate. For example, in C:

```
float number = 0.1+0.2;
printf("0.1 + 0.2 equals %.15f", number);
> 0.1 + 0.2 equals 0.300000011920929
```

This happens because, like how:

$$\frac{1}{3} = 0.33333333...$$

The number cannot be represented as a decimal- it is recurring and would require an infinite amount of space to write in full. Such the same thing can happen with binary. Therefore, you should use integers when possible.

Comments

When writing code, other people, including your future self may not understand what code does. This is why you use comments, explaining what a piece of code does. This is why we have comments, which are sections of code ignored by the computer.

You start a line with `//` for a single-line comment, and start with `/*` and end with `*/` for a multi-line comment:

```
//This is a comment
This is not a comment
/*This is a multi-line comment.
This is still a comment
*/
```

Note: There are also Pointers, which will be covered later

Variables

Introduction

You know about some data types, but how can you store them? You use variables:

```
int age = 10;
type variableName = variableValue;
```

You define a variable by having the type of it, in this case, `int`, then the name of it: `age` in this case- this is what you will refer to to access it-. You should use a descriptive name for this so you know what kind of information is stored such as the age of someone- having it named `pineapple` isn't too helpful. Next is the `=` sign: you want age to equal a number, which is the 4th part, the `10`. Finally, you need to end it with a semicolon `;` to tell the computer to finish that instruction.

Note: you cannot have a space in the name of the variable, start it with a digit, and there are other character restrictions.

You can change the value of the variables after you define them:

```
age = 11;
age = age + 10;
```

In the second example, the value of `age` defined as being the current of `age` (11) plus 10, giving it a final value of 21. DO NOT DO THIS IN MATHS. IT WILL BE MARKED WRONG. Note the you only needed to write down the type of the variable, `int` in this case, when you first defined it.

Constants/literals

There are sometimes when you do not want the value of a variable to be changed. In this case, use a constant `const` during the declaration of a variable:

```
const unsigned int MAXDRIVESPEED = 80;
//maxDriveSpeed cannot be changed
```

Constants are, by convention, written in `ALLUPPERCASE`, not `lowerCamelCase`.

Type Casting/Conversion

This works as expected:

```
int x = 10;
x = x / 2;
printf("x equals %d", x);

> x equals 5
```

10 divided 2 equals 5. Simple. However, what would happen if you do this:

```
int x = 10;
x = x / 3;
printf("x equals %d",x);

> x equals 3
```

The result of this is not a whole number- not an integer. This is why The value of 10 / 3 becomes rounded to the nearest whole number. You cannot change the type of a variable. However, what you can do is create a new variable of type `float` or `double`:

```
int x = 10;
float y = x/3;
printf("y equals %.3f", y);
//The %.3f prints out a float to 3 decimal places
> y equals 3.000
```

Why does y equal 3.000 and not 3.333? That is because both numbers in the division, `10` and `3` are integers. To solve, this, at least one of them must be a `float`. We can do this using two ways. Either add a decimal place in the `3` to make it `3.0`:

```
float y = x/3.0
```

This implicitly converts 3, an integer into a float, 3.0.

By doing this, the variable with least accuracy, `x` will change type to a `float` in this context, giving the final result as a `float` and not as a `int`.

The other way this can be done is by prefixing a `(float)` in front of one of the numbers, doing a **type casting** in the form, `(type) expression`:

```
float y = (float)x/3;
//OR
float y = x/(float)3;
printf("%.3f", y);
> 3.333

printf("&d", (int)y);
> 3
```

The top two converts one of the numbers into a float, which implicitly converts the other into a type with the same accuracy. Note that the `(type)` is first in the order of operations compared to `/` /

The third one does something similar, except that it loses accuracy, going from an `int` to a `float`. Here, the number is rounded from 3.333... to 3.

Default Values

If you simply declare a variable but do not give it any value, it will be set to the default value of that type.

```
int test;
printf("%d", test);
> 0
```

Type	Default Value
`int`	`0`
`char`	`'\0'`
`float`	`0`
`double`	`0`
pointer	`NULL`

Booleans and operators

Booleans, or `bool` is a true/false, 1/0 value as would happen with a single bit. These are often created using *comparison operators*, usually a *binary operator*. A binary operator gets two values, and outputs a single value.

Greater than

For example, a binary operator could be `>`, the greater than sign. Something you could do with it is:

```
int age = 18;
bool over18 = (age > 18);
//False
```

Here, the variable, `over18` is a boolean that can take one of two states: `true` or `false`. It checks if the variable `age` is greater than 18. It is not- although equal to, it is not *bigger than* 18. Therfore, `over18` is equal to `false`, which is identical to being `0`. A value of true is the same as having a non-zero value, often `1`.

There is also `>=`, which means *is greater than or equal to*. If the variable `over18` equaled `age >= 18`, it would have a value of `true` as it is equal to 18.

There are also the `<` and `<=` operators, which mean *smaller than* and *smaller than or equal to* respectively.

Equals

`==` is another operator. This checks if something *equals* another thing. This is different from a single `=` sign, as that assigns a value to a variable, such as `int age = 14`. For example:

```
int yearLevel = 9;
bool isYear10 = (yearLevel == 10);
//False
```

Like how `<` has its inverse, `>=`, `==` also has its inverse: `!=`. This checks if one value is *not equal to* the other, returning `true` if they are different and `false` if they are the same.

Logical Operators

Logical operators are similar to comparison operators, except that they take booleans as their two arguments, or values instead of two numbers/strings/whatever.

AND

The AND operator, or gate has the symbol `&&`. This returns `true` if and only if both of its arguments are `true`:

Value 1	Value 2	Output
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

OR

The OR operator has the symbol `||`, made of two pipe symbols. It should be next to the backslash symbol on the keyboard. This returns a `true` value if at least one of its arguments are `true`.

Value 1	Value 2	Output
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

NOT

The NOT operator is slightly different- it is a unary operator, meaning it only takes one input. Its symbol is `!`, and like the `!=` operator, it flips the value of a boolean: `true` becomes `false` and `false` becomes `true`. For example, `!(1)` returns a value of `false`, as `1` is the equivalent of `true`.

Arithmetic Operators

Maths operators such a `+` are used in programming.

Symbol	Description	Example
`+`	Adds two things together	`int x = 5 + 8; //13`
`-`	Subracts two things together	`x = 4 - 2; //2`
`*`	Multiplies two things together	`x = 5 * 2; //10`
`/`	Divides two things together	`x = 16 / 2; //8`
`%`	Modulus- Gives the remainder of a division	`x = 11 % 3; //2`
`++`	Increments by 1: adds one. Shorthand for `vari = vari + 1`	`x++; //13`
`--`	Decrements by 1. Shorthand for `vari = vari - 1`	`x==; //12`
`+=`	Adds a number to a variable. Shorthand for `vari = vari + x`	`x+=4; //16`
`-=`	Subtracts a number to a variable. Shorthand for `vari = vari - x`	`x-=10; //6`
`*=`	Multiplies a number to a variable. Shorthand for `vari = vari * x`	`x*=2; //12`
`/=`	Divides a number to a variable. Shorthand for `vari = vari / x`	`x/=3; //4`
`%=`	Divides a number to a variable and gives remainder.	`x%=3; //1`

Bitwise Operators

Bitwise operators are operators that work directly on the bits and bytes. Let us have two values, `A`, which equals `0000 1010`, or 10 and `B`, which equals `0001 1111`, or `31`:

Operator	Explanation	Example	Result
`&` `AND`	Binary. Returns `1` if `A` AND `B` are `1`	`A & B`	`0010 1010`, `10`
`|` `OR`	Binary. Returns `1` if `A` OR `B` are `1`	`A | B`	`0001 1111`, `63`
`^` `XOR`	Binary. Returns `1` if one of `A` and `B` are `1`	`A ^ B`	`0001 0000`, `16`
`~` `NOT`	Unary. Flips bits: `1` ⟶ `0`, `0` ⟶ `1`	`~A`	`1111 0101`, `3`
`<<` `LEFT SHIFT`	Moves bits by `x` bits to the left	`A << 2`	`0010 1000`, `40`
`>>` `RIGHT SHIFT`	Moves bits by `x` bits to the right	`A >> 1`	`0000 0101`, `5`

Notice anything about the last two, `<<` and `>>`? $10 \times 2^2 = 40$, and $10 \div 2^1 = 5$.

Conditional Statements

`if` and `else`

Conditional Statements are statements that occur only if something- a boolean is true. A common type of this is the `if` statement. In this conditional, some action happens only *if* some condition is true. They follow the form:

```
if (boolean) {
  //Do something
}
//Is the same as
if (boolean == true) {
  //Do something
}
```

You can also have something happen if that condition is *not* met. For example:

```
int year = 2017;
if (2000 < year && year <= 2100) {
  //If the year is between 2000 and 2100, do this
  printf("Currently the 21st Century");
}
else {
  printf("Probably not alive");
}
```

This is useful as it can react to certain conditions. In Vex, this might be:

```
if (vexRT[Btn8D]) {
  motor[driveLeft] = 127;
}
else {
  motor[driveLeft] = -127;
}
```

Here, it checks if a button, `Btn8D` has been pressed, giving it a value of 1. Remember that `1` is equal to `true` as a boolean, so you do not need to write `if(vexRT[Btn8D] == 1)`. If it is pressed, it sets the left driving motor to run at full speed, and if the button is not pressed, it will run backwards.

`else if`

Else if is a combination of a latter, and allows for more than two outcomes. For example:

```
int year = 2017;
if (year < 2000) {
  //1999 or earlier
  printf("It\'s before the 21st Century");
}
else if (year < 2100) {
  //Between 2000 and 2099. If the above if statement was not there, it would be 2099 or earlier.
  printf("It\'s the 21st Century");
}
else {
  //Later than 2099
  printf("The 21st Century has ended");
}
```

The ternary operator

The ternary operator takes *three* inputs instead of the usual two, and is in the form:

```
int foo = (boolean)? valueIfTrue: valueIfFalse;
```

It is a compressed form of:

```
int foo;
if (boolean) {
  foo = valueIfTrue;
}
else {
  foo = valueIfFalse;
}
```

It can be a bit harder to read for beginners, but on the other hand, being more compressed can make overall readability better.

Loops

Loops are things that allow you to do an action multiple times

The first of these will be the `while loop`

`while`

The `while` loop is a loop that runs as long as some condition is true

For example:

```
while (batteryLevel > 50) {
  printf("A bustard is a large bird");
}
```

You can also have what are called **infinite loops**. Infinite loops, as you might imagine, run infinitely due to the condition- boolean always being true. It only stops only when it is forcefully done so, such as when the battery is turned of or program forcefully quit. Usually, this is a bad thing, but is done extensively in RobotC in order to update commands such as controller values and motor values:

```
while (true) {
  motor[driveLeft] = vexRT[Ch3];
  motor[driveRight] = vexRT[Ch2];
}
```

Above, this loop will run multiple times per second, updating the motor speed as the controller values change.

`do while` loops

`do while` loops are similar to `while` loops, except that the `do` part- what is done is at the start instead of the end. This means that a `do while` loop always runs **at least once**.

```
do {
  printf("Hello");
} while (year == 2016);
//Runs once as year is not 2016
> Hello
```

`for` loops

`for` loops are more complex than `while` loops. They take the form:

```
for (init; condition; increment) {
  //Do stuff
}
//e.g.
for (int i = 0; i < 100; i++) {
  printf("%d", i);
}
//Prints numbers from 0 to 99
```

The first part *initializes* a variable. That is, it creates a variable, which is, by convention, often called `i`. Here, `i` is set to a value of 0.

The second part gives the loop a *condition*- a boolean. It will keep on running until that boolean is `false`. In this case, when `i` is not smaller than 100.

The final part tells you what to do after each time the loop runs. In this case, the value of `i` is *incremented*, and so will increase by 1 each time it runs.

Each of the 3 parts are optional. You could have a loop like `for(;;)`, which is the equivalent of an infinite loop.

`break`, `continue`

These two things go inside the above loops, and allow you to modify the behavior of the loop.

`break` breaks you out of the loop. That is, the loop finishes before the condition/boolean is false. For example,

```
for (int i = 0; i < 100; i++) {
  if (i == 10) {
    break; //exits loop when i equals 10
  }
}
```

`continue` does something slightly different. It allows you to skip over a single iteration of a loop.

```
for (int i = 0; i < 10; i++) {
  if ( i == 5) {
    continue;
  }
  printf("%d ", i);
  //Prints the numbers EXCEPT for 5
}

> 0 1 2 3 4 6 7 8 9
```

Functions

Functions are reusable pieces of code which allow a specific task to be repeated:

```
void driveForwards() {
  motor[driveLeft] = 127;
  motor[driveRight] = 127;
}

driveForwards(); //Causes robot to move fowards
```

You must *define* the function, giving it a type and name like a variable, and then what it should do in curly braces `{}` like loops.

Notice that in this case, the type of the function is `void`. Every function must *return* a single value (of the type the function is defined as), but if the type is `void`, it does not have to.

The name of the function, `driveForwards()` is descriptive of what it does, and ends with parentheses `()`. Inside the function are the *arguments*, which will be discussed later.

Inside the curly braces is what the function should do, similar to a loop, except that it only runs once.

Finally, the function must be *called*, which essentially means that the stuff inside the curly braces are run. Note that when it is called, it uses the function name, plus the parentheses at the end.

Return value

The return value is a single value of the same type as the function was when it was defined. Every function, except one with type `void` must return a value. For example:

```
int currentYear() {
  return 2017;
}
printf("%i", currentYear());
//%d and %i is a placeholder for an integer

> 2017
```

Here, the return value of the function, `currentYear` is outputted into the print statement.

Arguments

Arguments are the stuff that go into the parentheses after the name of the function. This allows for a change in behavior, instead of the function doing the exact same thing every time. The arguments must be of a specific type, which can be different from the type of the argument. If there is more than one argument, they are separated by commas `,`.

```
void setMotorSpeed(int rightSpeed, int leftSpeed) {
  motor[driveRight] = rightSpeed;
  motor[driveLeft] = leftSpeed;
  //Type of function, void is different from type of arguments, int.
}

setMotorSpeed(65, -12);
//Right motor at speed of 65, left motor at speed of -12
```

In the above function, `setMotorSpeed` sets the speed of the two motors, and are *given* specific values when the function is *called*. In this case, it is given the values 65 and -12 as its arguments, which it uses to set the speed of the motors.

Scope

Take this example:

```
int hi = 10; //A variable defined outside a function

int main() {
  printf("%d\n", hi); // > 10
  int hi = 15;
  printf("%d\n", hi); // > 15

  other();
}

int other() {
  printf("%d\n", hi); // > 10
}
```

You define the variable `hi` outside of any function. In `main()`, you print out the value of `hi` - 10. Then, you define a variable called `hi` with the value of 15. Now, `hi` equals 15. Here, everything seems to be all right.

But in another function, `other()`, you print the value of `hi`, which remains as 10. Why? Because of **scope**.

When you first defined `hi`, its scope was *global*: anything could access it. But then, in `main()`, you defined another variable with the same name with a different value. This created another variable with a *local* scope, meaning only that function could access it. And when there are two variables, one local and one global, the local one is used.

This is why, when `hi` was defined for the second time, it did not change the original value of `hi`, but created another one of the same name. And is why when `other()` tried to access it, it accessed the global `hi`.

This will occur in loops as well, but not in conditional statements:

```
int hi = 10;

int main() {
  printf("%d\n", hi); // > 10
  hi = 15;
  printf("%d\n", hi); // > 15
  other(); // > 15

  for (int i = 0; i < 3; i++) {
    int hi = 100;
    printf("%d\n", hi); // > 100
  }

  printf("%d\n", hi); // > 15
  return 0;
}

int other() {
  printf("%d\n", hi); // > 10
}
```

Pointers

Last, but not least, we have pointers. Every variable is stored in *memory*, or physical RAM. You can use the *address*, which is the value a pointer takes to access the value of the variable. For example, lets have a variable and a pointer:

```
int myVariable = 10;
int *myPointer = &myVariable;
//Pointer is same type as the thing it points to
```

You use the `*` before the name of the pointer to signify that you want it to be a pointer, and the use an ampersand `&` to get the address of the variable that you want to point to. The address will probably be something like `0x006FF9D8`, where the `0x` signifies that it is a `hexadecimal` number, covered below.

The useful thing is, that if you have the address, you can also manipulate its value:

```
int myVariable = 10;
int *myPointer = &myVariable;

*myPointer += 10; //Add 10 to myVariable
printf("%d\n", myVariable);//Prints value of myVariable
> 20
```

This is useful because functions can only return one value. However, if you use the address of a variable as a argument, you can modify more than one:

```
void main() {
  int age = 14;
  int yearLevel = 9;
  point(&age, &yearLevel);
  printf("Age: %d. Year level: %d\n", age, yearLevel);

  > Age: 15. Year level: 10
}

void point(int *theirAge, int *theirYearLevel) {
  *theirAge += 1;
  *theirYearLevel += 1;
}
```

Hexadecimal Numbers

Decimal, or base 10 has 10 characters you can use: 1, 2, 3, 4, 5, 6, 7, 8, and 9. Binary, or base 2 has 2: 0 and 1. Hexadecimal is a base that is often used, and has *16 characters*: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.

Hexadecimal is useful as it conveys more information than either decimal or binary while still being human readable. One of the useful properties of hexadecimal is that you can convey 1 byte, or 8 bits using 2 hexadecimal characters, as $\mathbf{16 = 2^4}$, meaning one character can hold 4 bits. With 2, you can hold 8 bits, ora byte.

Tasks

Tasks are slightly different from functions, and exists in RobotC but not in normal C. There is a task called `task main`, the equivalent of `int main` in C. This is the starting point of the program- the processor needs to have an entry point into the program- somewhere to start, and that is `task main`. A basic `task main` could look like:

```
void setDriveSpeed(int left, int right) {
  motor[driveLeft] = left;
  motor[driveRight] = right;
}
task main() {
  while (true) {
    setDriveSpeed(vexRT[Ch3], vexRT[Ch2]);
  }
}
```

It is important to note that tasks CANNOT have any arguments.

You can have up to 19 other tasks running simultaneously in the Cortex in RobotC. To do this, use:

```
task other() {
  //do something
}
task main() {
  StartTask(other); //starts the task 'other'
  EndTask(other); //ends the task 'other'
}
```

Structures

A `struct` is a block of code containing undefined/uninitialized variables that can be defined later. Note that you MUST have a semicolon `;` at the end of the `struct` definition.

For example, this is a basic `struct`:

```
struct myStruct {
  //Creates undefined variables
  int x;
  char* y;
  unsigned z;
};

int main() {
  struct myStruct gainAccess; //The variable, gainAccess now has access to myStruc
  gainAccess.x = 8; //gainAccess.x now has a value, although the original myStruct does not
  printf("gainAccess.x = %d", gainAccess.x);

  > gainAccess.x = 8
  return 0;
}
```

This essentially creates a blank template which you can make copies of and fill in later.

Another way of making a `struct` is by using `typedef`. Use that keyword, at at the end of it, after the closing brace `}`, add another name for it. To create the new copy of it use the second name as the type of it instead of `struct myStruct`. The benefit of doing this is that once the `struct` has been defined or created, you do not need the `struct` keyword to create a copy.

```
typedef struct myStruct{
  int x;
  char* y;
}my_struct;

int main(){
  my_struct gainAccess; //gainAccess now has access to myStruct. Note that you did not need to write `struct`.
  //struct myStruct is equivalent to my_struct
  gainAccess.x = 8;
  printf("gainAccess.x = %d", gainAccess.x);
  return 0;
}
```

Gaining access to a `struct` using a pointer then deleting it to make space:

```
#include <stdlib.h>
#include <stdio.h>
//You need these two libraries or modules to use `malloc` and other functions

typedef struct myStruct myStruct;
//forward declaration: declaring, or creating before actually using it. Required if you reference it before it is defined

struct myStruct {
  int x;
  char* y;
  myStruct* xy; //Pointer to itself. This is why the forwards declaration is required, as myStruct would not have otherwise been created or defined yet, and so making it of type myStruct would fail.
};

int main() {
  myStruct *gainAccess = malloc(sizeof(myStruct)); //GainAccess is a pointer that allocates the memory required for a `myStruct` to the pointer, which points to the first byte in that sequence
  gainAccess->x = 8; //Gives value to x. Equvialent to:
  //(*gainAccess).x = 8; where you get the `struct` in that location, then get the x value in that struct
  gainAccess->xy = malloc(sizeof(myStruct)); //allow xy to get enough memory to fit in memory required for a `myStruft` struct, like above.
  gainAccess->xy->x = 4; //Value of the pointer points to a pointer and the value of that pointer points to a struct from where we access x. Essentally a `struct` in a `struct`
  gainAccess->xy->xy = malloc(sizeof(myStruct)); //Struct in a struct in a struct
  gainAccess->xy->xy->x = 11;
  printf("gainAccess.x = %d\n", gainAccess->x);
  printf("gainAccess.xy.x = %d", gainAccess->xy->x);
  printf("gainAccess->xy->xy->x = %d\n", gainAccess->xy->xy->x);
  free(gainAccess); //deletes the data allocated to the pointer to free space
}
```

Unions

Unions are similar to `struct` in its structure, but allows you to store **one** variable of any data type defined in the `union`:

```
union myUnion {
  int myInt;
  float myFloat;
  char myChar;
}; //This semicolon is important

union myUnion newUnion; //Creates copy of union
newUnion.myInt = 11; //Access members like you would members of a struct
newUnion.myFloat = 9.6;
newUnion.myChar = 'A'; //Make sure you use '' not ""
printf("%d\n", newUnion.myInt);
printf("%f\n", newUnion.myFloat);
printf("%c\n", newUnion.myChar);

> 1092196705
> 9.599946
> a
```

As you can see, the first two, `myInt` and `myFloat` have been corrupted as it can only store enough information for the biggest variable, `float`. It is overwritten by the value of `myChar`, `A`.

Recursion

Recursion is a concept in programming in which a function is *recursively* called multiple times by itself, layering itself like a `struct` inside a `struct`. An example of this is the Fibonacci sequence, a sequence of numbers going `1, 1, 2, 3, 5, 8, 13, 21` and so on, where the `nth` term is the sum of the previous two terms.

```
long long fibonacci(long long);
//Long long type allows for bigger numbers. Forwards declaration here.

long long fibonacci(long long n) {
  //Fibonacci sequence: 1, 1, 2, 3, 5, 8...
  //Sum of previous 2 in sequence, starting with one
  if (n <= 2) {
    return 1;
    //First two terms are 1
  }
  else {
    return fibonacci(n-1) + fibonacci(n-2);
    //Otherwise, get the last two terms and add together
  }
}
```

Of course, there are alternatives to recursion:

```
long long fibonacciNonRecursion(long long n) {
  long long first = 1;
  long long second = 1;
  for (long long i = 2; i < n; i++) {
    long long temp = second;
    second += first;
    first = temp;
  }
  return second;
}
```

`malloc` and `free`

This is something that allows for **dynamic memory allocation**. This is important as it allows you to allocate and then free up memory. An example of use of `malloc` might be when making an array. This is because when making an array, you usually need to specify exactly how many elements are in it, and so how much memory is required.

The syntax for `malloc` usually uses a pointer, which is the starting point of the allocation. The pointer needs the type of data it will contain, and you need to tell it how much memory (in bytes) you want to allocate. This is usually done with `sizeof()`, which tells you the size of a certain data type, such as `sizeof(int)` which might give you 4 or 8. If this is an array, it is usually multiplied by the number of elements inside it.

```
type *name = (type*) malloc(numberOfBytesToAllocate);
//e.g.
int *myArray = (int*) malloc(sizeof(int) * 50); //Array of type int with 50 elements. Enough data for 50 times (sizeof(int)) bytes.
```

```
#include <stdlib.h>
#include <stdio.h>
//fibonacci() already defined

int main() {
  long long sizeOfArr = 20; //Define size of array so you can change it easily
  long long *myArray = (long long*) malloc(sizeOfArr * sizeof(long long));
  if (!myArray) {
    //NULL, or 0 means that it was not allocated (probably because there was not enough
avaliable. 0 also means false. So, if false:
    printf("Error: Memory not allocated");
    return 0;
  }

  //pointer with enough space to store sizeOfArr long longs
  for (long long i = 0; i < sizeOfArr; i++) {
    //a for loop going between 0 and 49 (<, not <=)
    *(myArray + i * sizeof(long long)) = fibonacci(i + 1);
    //get the integer stored in the (address of first array plus the size of an int times i).
  }
  for (long long i = 0; i < sizeOfArr; i++) {
    printf("%lld\n", *(myArray + i * sizeof(long long)));
    //now print them out. %lld for signed long long, %llu for unsigned long long.
  }
  free(myArray); //free the memory allocated to the pointer
}
```

Arrays

Arrays are a type of structure (in a general sense, not a `struct`) that allows you to store *multiple* elements, or pieces of data in of a single data type in a single variable. This is already done with strings in normal C (robotC has its own implementation). Strings are actually an array of `char`'s:

```
char *myArray1 = "Hello World";
//is the same as:
char myArray2[] = "Hello World";
//Which is the same as:
char myArray3[11] = "Hello World";
```

In the first example, you are setting `myArray` as a pointer to the first element of the array, `"Hello World"`. You can access the `"H"` by writing `*(myArray)`, and any other element by writing `*(myArray + n * sizeof(int))` where n is the `nth` element in the array, **starting at 0**. The first element is 0, the second is 1, and so on. You can only print each character, not the whole string.

The second example does something similar to the third, except that **the number of elements are implicitly declared due to the length of the string** , which is 11 characters long and so up to an index of 10. As **you cannot change the length of the array after you set it**, you can only do this for the first time you set a value to it. You can print it using `printf("%s", myArray2);` or a single element using `printf("%c", myArray2[2]);`.

The third sets it to an array with 11 elements explicitly, and so it can be up to and including 11 characters. You can set any character in the array using `myArray3[n] = 'y'`. **You must use a single quote (') for char**.

There is also another way:

```
#include <stdio.h>
#include <string.h>
//You need <string.h> for `strcpy`
int main() {
  char str[20];
  strcpy(str, "Hello World");
  printf("%s", str);

  > Hello World
}
```

You can also use arrays to store other data types:

```
int myAges[] = {14, 15, 14, 17, 13, 14, 16, 15, 13, 14};
```

RobotC commands

- `vexRT[ChX]` where X is a number between 1 and 4. Returns a value between -127 and 127
- `vexRT[BtnYZ]` where Y is a number between 5 and 8, and Z is either U, D, L or R. Returns 1 for pressed, 0 for not pressed
- `vexRT[AccelDIM]` where DIM is either X, Y or Z. Returns a value between -127 and 127
- `motor[motorName] = val` where motorName is the motor name you gave it at the top of the program (in motor and sensor setup), and val is a integer between -127 or 127. If it is greater than or smaller than the range, it defaults to the maximum value (i.e. -127 or 127).
- `SensorValue[sensorName]` where sensorName is the name you defined at the top of the program (in motor and sensor setup). Could be a boolean: 0 or 1, but could also be a large range such as 0 to 4095.
- `StartTask(taskName, priority)` starts a task. priority is an optional argument telling the Cortex what kind of priority/importance the task has, ranging from 0 to 255. The default is 7.
- `EndTask(taskName)` ends a task
- `wait1MSec(numMilli)` where numMilli is an integer telling you how many milliseconds to pause for. There is also `wait10MSec()` which is the same except that is is in 10 milliseconds instead of 1.
- `time1[X]` where X is either T1, T2, T3, or T4. Vex Cortex has 4 timers. It gives you the current value of the timer. There is also `time10[X]` and `time100[X]`. Setting `time1[X] = 0` resets the timer.
- clearTimer[X] where X is one of the 4 timers. Clears the timer.

Note: You can have two controllers. In this case, append the name of the channel or button with `Xmtr2` e.g. `vexRT[Ch3]` becomes `vexRT[Ch3Xmtr2]`.

Sensors

Sensor Name	Type	Range	Notes
Light Sensor	Analog	0 to 4095	0 is brightest
Potentiometer	Analog	0 to 4095	
Line Follower	Analog	0 to 4095	0 is brightest
Gyro	Analog	-3600 to 3600	1 = 1/10 of degree. Range is 3600, depends on direction of rotation.
Accelerometer	Analog	0 to 4095	Req. 3 ports
Touch	Digital	0 or 1	1 is closed
Sonar	Digital	-1 to	-1 means no reflection. Otherwise in cm.
Quadrature Encoder	Digital	0 to	360 means 1 rotation. One wire between int3 and int6.
Digital In	Digital		
Digital Out	Digital	0 or 1	1 sends digital signal. Use for solenoids.

RobotC 4.0 Math functions

Name	Input	Output	Description
`abs`	`int/float/double/long`	`float`	Returns absolute value. -ve ⟶ +ve, +ve ⟶ +ve
`acos`	`float`	`float`	Inverse cos/sin/tan. Returns angle in radians
`atof`	`string`	`float`	Converts `string` to `float`
`atoi`	`string`	`long`	Converts `string` to `long`
`ceil`	`float`	`float`	Rounds up to nearest integer
`cos`	`float`	`float`	cos/sin/tan with input in radians
`cosDegrees`	`float`	`float`	cos/sin/tan with input in degrees
`degreesToRadians`	`int`	`float`	Converts degrees to radians
`exp`	`float`	`float`	e^x. Returns e to the power of argument
`floor`	`float`	`float`	Rounds down to nearest integer
`log` , `log10`	`float`	`float`	ln, log of a number
`PI`	Not a function	`float`	π is a constant
`pow`	`float` , `float`	`float`	Raises base to the power of exponent
`radiansToDegrees`	`float`	`short`	Converts radians to degrees, returning as a integer.
`sqrt`	`float`	`float`	Square root function. Equivalent of `pow(base,0.5);`

`printF` function

The `printf` function allows you to print text to the console. You can print variables, and those require you to specify what type of variable it is. To do this, you enter a `%` before the variable, and there are many options for formatting.

The string must be between double quotes `""`, and after that, you enter the variables as arguments for the function.

The `\n` at the end makes it go to a new line so that the output text isreadableandwillhavespacesbetweentwo `printf` statements.

Specifier	Type	Example
`d` / `i`	signed integer	`-15`
`u`	unsigned integer	`18`
`o`	unsigned octal	`77`
`x` , `X`	unsigned hexadecimal integer. lower/upper	`7fa` , `7FA`
`f` , `F`	Decimal floating point. lower/upper	`392.65`
`e` , `E`	Scientific Notation. lower/upper	`3.2e+2` , `3.2E+2`
`g` , `G`	%e or %f. The shortest. lower/upper	`3200`
`a` , `A`	Hexadecimal floating point. lower/upper	`0xc.90fep` , `oXC.90FE`
`c`	Character	`a`
`s`	String	`Hello World`
`p`	Pointer	`0x7ffec12b1edc`

```
float printThis = 123.456;
int printThat = 1234567;
printf("%.1f\n", printThis);
printf("%.10d\n", printThat);

>123.5 //Rounded to 1 d.p.
>0001234567 //10 digits printed.
```

You can round floating point numbers by having a `%.dpf` where `dp` is the number of decimal places you want. For decimals, you can make sure at least x amount of characters will be printed with the same notation.