

by Xander Soldaat (with assistance from Timothy Friez and John Watson)

Pointers and Data Structures in ROBOTC

New capabilities enhance a leading programming environment

Editor's note: This article is for those with an interest in learning programming. To see the code involved, please scan the barcode or type in the URL.

Tens of thousands of kids are getting their first programming experience through robotics. From an educational software developer's perspective it is critical that the software has a low entry point, but a very high ceiling. With the release of ROBOTC 3.5, the ROBOTC development team included many new features designed to teach advanced programming concepts like variable pointers and recursive functions allowing students to learn the higher level programming concepts used by professional programmers today. Programmers are now able to create efficient and effective code while developing complex algorithms for applications such as autonomous path planning and advanced sensor processing using complex data structures. This article is a tutorial on how pointers work in the ROBOTC programming environment specifically with the LEGO MINDSTORMS NXT robotics controller. The tutorial is written with both new and experienced programmers in mind. New programmers will be introduced to the concepts of variables and pointers using diagrams and examples while experienced programmers will be able to see applications of these advanced concepts being used with a robotics-based application.

The release of ROBOTC 3.5 brought a myriad of new features, including the long-awaited implementation of pointers to variables. Having this functionality opens a whole range of new possibilities, such as the implementation of complex data structures.

WHAT ARE VARIABLES?

Prior to version 3.5, ROBOTC only supported normal variables; in other words, a variable, be it an int, float or anything else, only had a value, like 712, 0.383 or "howdy". For example:

```
long foo = 76278;
float baz = 2.9121;
string greet= "hello";
```

In effect, when you declare a variable, the compiler puts aside an appropriately sized amount of memory and gives it a user-defined symbolic name, like "I" or "foo" or "baz". An integer (int) is 4 bytes large, a float point number (float) is also 4 bytes but a string of character (string - in the case of ROBOTC) is usually 20 bytes large. Take a look at the drawing (right—you see the memory blocks, the labels assigned to them, their contents and the memory address for that block (the hex numbers under the blocks).



DISSECTING A VARIABLE

A variable has two parts, an r-value and an l-value. The term r-value refers to the right side of the assignment statement and l-value refers to the left side. Now consider this snippet of code:

```
// ptr tutorial example 1
task main()
{
    int i, j;
    i = 10;
    j = 51;
    // This should print out - i: 10, j: 51
    writeDebugStreamLine("i: %d, j: %d", i, j);

    // Assign i's r-value to j
    j = i;
    // This should print out - i: 10, j: 10
    writeDebugStreamLine("i: %d, j: %d", i, j);

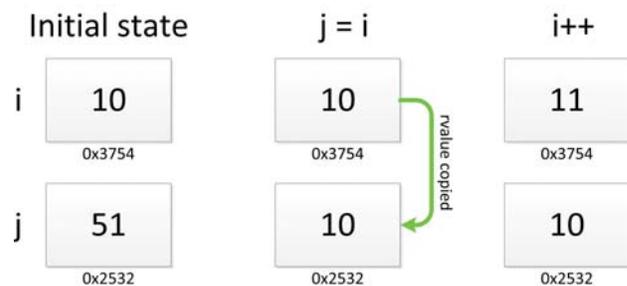
    i++;
    // This should print out - i: 11, j: 10
    writeDebugStreamLine("i: %d, j: %d", i, j);
}
```

When this program is executed, the output to ROBOTC's "Debug Stream" should look like this:

```
i: 10, j: 51
i: 10, j: 10
i: 11, j: 10
```

(Note: To Open the ROBOTC "Debug Stream", make sure your ROBOTC interface is in "Expert" or "Super User" mode and open the Debug Stream from the normal debugger windows menu.)

When looking at the memory locations and their contents after each operation, it would look something like this:



On the first assignment at the beginning of the program, variable i's r-value is given the value 10. Right below this assignment, there's a second assignment where variable j's r-value is given the value 51. This is considered setting an initial value.

Later in the program, what will happen with the line "j = i"? Simple, the r-value of i is copied and assigned to j's r-value. Now both j and i's r-values are 10. What happens when we increment i? Does it change j's r-value? In short, no, the two r-values are completely separated.

EDUBOTS

rate entities. When the compiler assigned the value of variable i to variable j, it will “copy” that value from one variable to the other.

GETTING TO THE POINT(ER)

As we mentioned earlier, when a variable is declared, the compiler assigns an appropriately sized chunk of memory to it. This chunk has an address, a number, much like a house. Should you want to access this address, you can do so with the reference (&) operator. This isn't very useful in itself, though. Fear not, the creators of C didn't add this ability to get the address without being able to doing something practical with it as well. This is where the pointer (*) operator enters the picture.

Previously, we saw that the r-value of a variable contains the actual value that we assigned. With a pointer, the r-value is in fact the address of the variable we're pointing at. But what if we want to see the value of this variable we're pointing at? We can do that with the dereference operator, also a “*”. Combining all this new-found knowledge, we get the following:

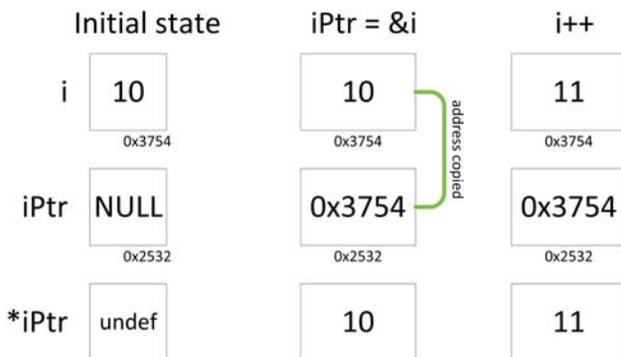
```
// ptr tutorial example 2
task main()
{
    int i = 10;
    int *iPtr;
    iPtr = &i; // iPtr now points at the address of i
    // This should print out
    // i: 10, iPtr: 656F6968, *iPtr: 10
    // (value for iPtr may vary)
    writeDebugStreamLine("i: %d, iPtr: %p, *iPtr:
%d", i, iPtr, *iPtr);

    i++;
    // This should print out
    // i: 11, iPtr: 656F6968, *iPtr: 11
    // (value for iPtr may vary)
    writeDebugStreamLine("i: %d, iPtr: %p, *iPtr:
%d", i, iPtr, *iPtr);
}
```

The output in the ROBOTC Debug Stream should look like this:

```
i: 10, iPtr: 656F6968, *iPtr: 10
i: 11, iPtr: 656F6968, *iPtr: 11
```

If you look at it from a memory perspective, our program will should look like this:



POINTER ARITHMETIC

So now that you know how what pointers are, let's have some fun with them. Consider the program below:

```
// ptr tutorial example 3
```

```
task main()
{
    ubyte arr[3] = {1, 2, 3};
    ubyte *arrPtr;

    arrPtr = &arr[0]; // arrPtr now points to the
address of arr[0]

    // This should print out
    // arr[0]: 1, arrPtr: 656F6968, *arrPtr: 1
    // (value for arrPtr may vary)
    writeDebugStreamLine("arr[0]: %d, arrPtr: %p,
*arrPtr: %d", arr[0], arrPtr, *arrPtr);

    arrPtr++; // we're now pointing at arr[1]

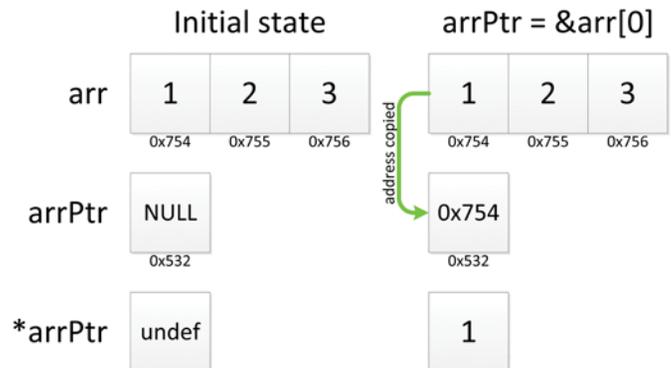
    // This should print out
    // arr[1]: 2, arrPtr: 656F6969, *arrPtr: 2
    // (value for arrPtr may vary)
    writeDebugStreamLine("arr[1]: %d, arrPtr: %p,
*arrPtr: %d", arr[1], arrPtr, *arrPtr);
}
```

That should print out something like this in the Debug Stream:

```
arr[0]: 1, arrPtr: 656F6968, *arrPtr: 1
arr[1]: 2, arrPtr: 656F6969, *arrPtr: 2
```

Please note that the address arrPtr may be different in your case. It is important that on the second line, the address arrPtr points to is one higher than in the first one.

When you look at what it looks like in memory, you can get a good idea of what's going on:



INTERESTING FACT ABOUT POINTERS

When you use sizeof() on a normal variable, like a ubyte, int or long, you get the number of bytes this type takes up. In the case of a ubyte, that's one byte, int takes up two and a long uses four. Doing a “sizeof” of a pointer is, well, pointless. You won't be getting the size of the item you're pointing at, it's always 4 in the case of ROBOTC. That's because the address stored in the pointer is a 4 byte number, a long. Keep that in mind when you want to use sizeof() to get the number of bytes you want to wipe with a memset(), for example.

PROJECT: DATALOGGING

If you've ever built a robot, may have found that it doesn't always do what you think it ought to do. Reality has a nasty habit of messing with your perfectly programmed robot. To deal with this, we have two choices: either change reality (not an easy task) or find out what's going on and change the behaviour of our robot. Sounds

easy, but how will you know what change if you don't know what's going on? This is where datalogging comes in. Datalogging is the practice of constantly taking snapshots of the current state and saving that information in safe place so you can look at it later. This is useful because as your robot disappears under the sofa, it's a little hard to keep an eye on the screen!



PUTTING IT INTO PRACTICE

To get started with a simple example of datalogging, we'll need the following parts:

- An NXT running ROBOTC 3.5 firmware or higher
- A LEGO Colour Sensor in Sensor Port 1 (S1)
- A LEGO Touch Sensor in Sensor Port 2 (S2)
- A standard LEGO NXT Motor in Motor Port A
- 3 NXT cables to connect the sensors and motor

What we want to do is the following:

- Log, for a short period of time:
- The LEGO Colour Sensor
- The encoder count of the motor
- Whether or not the LEGO Touch Sensor is pressed
- A timestamp for the above measurements

At the end of the run, it should be possible to review the data. So just how do we intend to use pointers for this? Well, they are great for passing variables around in a program. In this example, we'll use pointers to allow various functions in the program access to the data. Suppose we wish to log sensor and motor encoder data at specific intervals and inspect them later. There are two ways to go about it:

- Pass around all of the data
- Pass around only a data entry struct

If you don't have pointers, the first option may be your only way to go about this. However, ROBOTC is not impeded by this, so we'll use the second way. To achieve this, we'll use a "struct" to hold the data. A "struct" (short for structure) is a way to package multiple variables of different types into a single object. You can then use this object to pass a lot of data around your program in a simple efficient way. The struct that we'll use to store the data looks as follows:

```
struct
{
  TColors colourNum;
  bool touchSensorPressed;
  long motorEncoder;
  long timeStamp;
} tDataEntry;
```

Since this would only hold one data point, we must create an array of them:

```
tDataEntry dataEntries [MAX_DATAPOINTS];
```

If you're familiar with structs, then you'll know that you can access the individual members through the "." operator. For example, to access the motorEncoder member, you would type something like this:

```
tDataEntry entry;
entry.timestamp = 42;
```

To get back to what we were working on, we need a loop to go through the array and read the data:

```
// Pointer to tDataEntry struct
tDataEntry *dataPtr;
for (index = 0; index < MAX_DATAPOINTS; index++)
{
  // Point dataPtr to a fresh new data entry struct
  dataPtr = &dataEntries [index];

  // Get the data from the sensors and motor
  readData (dataPtr);

  // Wait a little bit
  wait1Msec (100);
}
```

Our function to read the data from the sensors and put it into a tDataEntry struct looks as follows:

```
void readData (tDataEntry *data)
{
  data->colourNum = (TColors) Sensor-value [COLOUR];
  data->touchSensorPressed =
  (bool) SensorBoolean [TOUCH];
  data->motorEncoder = nMotorEncoder [MOTOR_A];
  data->timeStamp = nPgmTime;
}
```

As you can see, the individual members of the struct are accessed through the "->" operator. Why not the "."? Take a look at the function "readData" parameters. We're not passing "readData" a structure, but rather a pointer to a structure. This allows us to keep a single set of data, but pass it around so other functions can process the actual data rather than dealing with copies of it. When working with members of a pointer to a structure, you'll have to use the "->" to access or modify the member variables of the structure. To recap:

- Use "->" to access members of a pointer to a struct: structPtr->member
- Use "." To access member of a regular struct variable: struct.member

As you can see, ROBOTC has made great strides in recent updates to bring advanced functionality to the end user. The inclusion of pointers opens up a whole new level of flexibility and functionality that creative and advanced users can tap into to achieve more efficient code. Pairing pointer and structures together allow the programs to manage data much more efficiently and effectively. In addition to pointer and data structure support, ROBOTC 3.5 also includes new recursive and re-entrant functions. Now that programmers have access to these all of these industry standard tools, they will truly be able to test the limits of your robot's capabilities by implementing complex algorithms and other computer science concepts.

To see the complete program, including a function to print all the data after the initial logging, please go to <http://botbench.com/botmag>. ©

Links
 Carnegie Mellon Robotics
 Academy,
www.education.rec.ri.cmu.edu,
 (412) 681-7160



See More Online!

Scan bar code or type in botmag.com/051304

For more information, please see our source guide on page 80.