Constant Variables

- The NXT has a limited amount of space available for variables
- It’s best to intelligently use variables to preserve as much of this space as possible
- Constant variables and #define statements can help:
  - Adding “const” to any variable will make it a compile-time constant and not use any variable space.
  - A #define statement will create an alias to a value or statement in a single variable name.

Constant Variables

- Examples:

```
#define HalfSpeed 50
#define FullSpeed 10*10
#define DefaultWait wait1Msec(1000)

task main()
{
  motor[motorB] = FullSpeed;
  motor[motorC] = HalfSpeed;
  DefaultWait;
}
```

```
task main()
{
 const int threshold = 50;

 while(true)
 {
   if(SensorValue[light2] < threshold)
   {
     motor[motorB] = 50;
     motor[motorC] = -50;
   }
   else
   {
     motor[motorB] = -50;
     motor[motorC] = 50;
   }
}
```

Variable Locations

- It is important where you declare your variables:
  - Variables declare outside of any structure (function or task) are considered to be “global”
  - Variables declared inside of any structure (function to task) are considered to be “localized” to that structure.
  - Variables declared in a loop/conditional statement are localized to that statement block
- It is always best to declare your variables in the correct location (usually at the top of your structure)

Variable Locations

- In this example, “test2” is not known outside of the “if” statement block.

```
1   task main()
2   {
3     int n = 0;
4
5     if(n == 0)
6     {
7       int test2 = 50;
8     }
9     nxtDisplayString(2, "%d", test2);
10  }
```

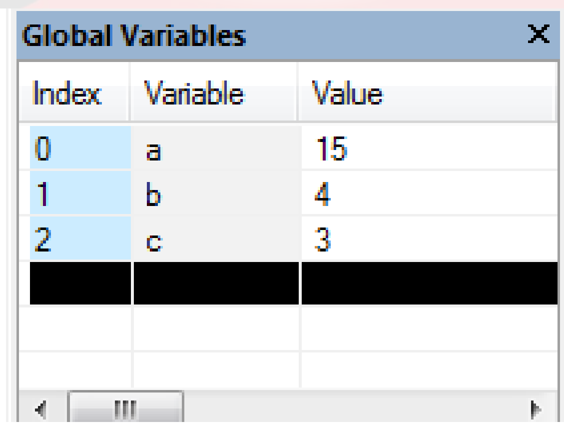
```
  File "Test Variables.c" compiled on Jul 14 2011 21:56:45
7 **Info***:'test2' is written but has no read references
9 **Error**:Undefined variable 'test2'. 'short' assumed.
```

Type Casting Variables

- Sometimes in order to prevent the loss of precision, you will have to type cast a variable
  - Precision: When a variable loses data
  - i.e. 15/4 should be 3.75 (float), but will actually be truncated to a value of 3 (int)

```
task main()
{
  int a = 15;
  int b = 4;
  float c = 0.0;

  c = a/b;
}
```



Robomatter INCORPORATED

Carnegie Mellon Robotics Academy

Type Casting Variables

- Even though "c" is a float, the math that occurred is integer math
  - Int / Int = Int
  - Float / Int = Float
  - Float / Float = Float
- ROBOTC is trying to preserve by using the least amount of memory as possible
  - This can have unintended consequences

```
task main()
{
  int a = 15;
  int b = 4;
  float c = 0.0;

  c = a/b;
}
```

Global Variables

Index	Variable	Value
0	a	15
1	b	4
2	c	3

Type Casting Variables

- You can force a variable to be a different “type” of variable by type casting with an explicit conversion:

```
task main()
{
  int a = 15;
  int b = 4.0;
  float c = 0.0;

  c = a/(float)b;
}
```

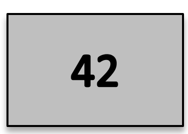
Global Variables

Index	Variable	Value
0	a	15
1	b	4
2	c	3.75
[redacted]		

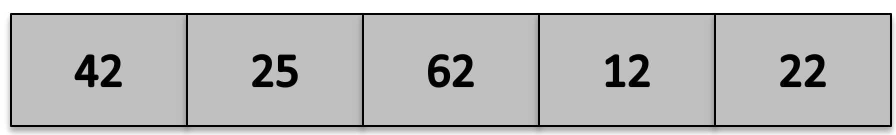
- By putting a (data type) in front of a variable, you can cast that variable into a new type
  - This example casted the integer variable “b” as a float to allow the calculation to perform as expected.

Array Variables

- An “array” variable type is a data type that is meant to describe a collection of elements (values or variables)
- The idea is to store a common set of data into a common variable name with an index.
  - Think of an index as a mailbox number and the variable name as the street name.



Variable ”MyVar”



Array Variable “MyVar[5]”

Array Variables

- An array must be declared before it can be used – like every other variable
  - type name[# of elements];
  - int myVar[5];
- You can have arrays of any data type – including strings
- To set the initial value of the array, you will have to use the following structure to define the data set:
  - int myVar[5] = {42, 25, 62, 12, 22};

42	**25**	**62**	**12**	**22**

Array Variable “MyVar[5]”

Array Variables

- To access the value of an element of the array, use its array index:
  - while(myVar[2] == 62)

[0]	[1]	[2]	[3]	[4]
42	**25**	**62**	**12**	**22**

Array Variable “MyVar[5]”

- Note that the index begins at zero, not one!

Array Variables

- For loops are very handy for "iterating" through an array to perform an operation
- The iterating value "i" can be used to grab a specific element of our array through each pass of the loop.

```
task main()
{
  int myVar[5] = {42, 25, 62, 12, 22};
  int sumValue = 0;

  for(int i = 0; i < 5; i++)
  {
    sumValue = sumValue + myVar[i];
  }
}
```

[0]	[1]	[2]	[3]	[4]
42	**25**	**62**	**12**	**22**

Array Variable "MyVar[5]"

163

Variable "sumValue"

Arithmetic Operators

- ROBOTC accepts a number of arithmetic operators:

- Assignment: **a = b**
- Addition: **a + b**
- Subtraction: **a – b**
- Multiplication: **a * b**
- Division: **a / b**
- Modulo (remainder): **a % b**
- Increment: **++x, x++**
- Decrement: **––x, x––**
- Unary Positive: **+x**
- Unary Negative (inverse): **–x**

Arithmetic Operators

- Differences between ++i and i++
  - ++i will increment the value of i, and then return the incremented value
  - i++ will increment the value of i, but return the pre-incremented value.

```
int i = 1;
int j = i++;
//i should be 2, j should be 1

int k = 1;
int l = ++k;
//k should be 2, l should be 2
```

Global Variables

Index	Variable	Value
0	i	2
1	j	1
2	k	2
3	l	2

Comparison Operators

- ROBOTC supports 6 different comparison/relational operators between two values:
  - Equal to: **a == b**
  - Not Equal to: **a != b**
  - Greater than: **a > b**
  - Less than: **a < b**
  - Greater than or equal to: **a >= b**
  - Less than or equal to: **a <= b**

Logical Operators

- ROBOTC has 3 logical operators to assist when making complex decisions:
  - Logical Negation (NOT): **!a**
  - Logical AND: **a && b**
  - Logical OR: **a || b**

Logical Truth Tables

NOT (!)

p	¬p
True	False
False	True

AND (&&)

INPUT		OUTPUT
A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

OR (||)

INPUT		OUTPUT
A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Compound Assignment Operators

- There are 5 different "compound assignment" operators to provide some shortcuts when working with variables:
  - Addition Assignment: **a += b**
  - Subtraction Assignment: **a -= b**
  - Multiplication Assignment: **a *= b**
  - Division Assignment: **a /= b**
  - Modulo Assignment: **a %= b**

Unary, Binary and Ternary

- Commands that work on a single value or variables are called “unary” operations.
  - Examples: x++, i--, -x, !x
- Commands that work on two values or variables are called “binary” operations.
  - Examples: a > b, a == b, a += b
- Commands that work with three values or variables are called “ternary” operations.
  - Examples?

Ternary Operations

- A ternary operation is a way to rewrite a basic “if/else” statement involving the assignment of values:
  - Normal Example →
  - Ternary Example:

```
task main()
{
  int a = 5;
  int b = 10;
  int c = 0;

  c = a > b ? 25 : 50;
}
```

```
task main()
{
  int a = 5;
  int b = 10;
  int c = 0;

  if(a > b)
  {
    c = 25;
  }
  else
  {
    c = 50;
  }
}
```

Ternary Operation

- A Ternary operation is a quick way to assign a variable a different value depending upon a condition
- The structure is as follows:
  - *condition* **?** *value if true* **:** *value if false*

```
task main()
{
  int a = 5;
  int b = 10;
  int c = 0;

  c = a > b ? 25 : 50;
}
```

Structure Statements

- A structure statement (struct) is a custom type of variable that allows the user to create a fix set of labeled objects of different types
  - Example: If I wanted to create a single variable to store a coordinate – I would create a "struct" that contained two integer variables

```
task main()
{
  typedef struct
  {
    int x;
    int y;
  } XYcoordinates;

  XYcoordinates StartingPoint;
  XYcoordinates EndingPoint;

  StartingPoint.x = 5;
  StartingPoint.y = 10;

  EndingPoint.x = 2;
  EndingPoint.y = 3;
}
```

Struct Definition

1. The keyword "typedef struct" is required to define a struct – This stands for "Type Definition of a Structure"
2. Inside of the definition of the structure, you can have as many variables (members) of any type as you would like. You can even mix and match types
   - i.e. string, int, float in the same struct
3. At the end of the definition, you give your structure a unique name.

```
typedef struct
{
  int x;
  int y;
} XYcoordinates;
```

1. Keyword to define struct
2. Members of the struct
3. Name of the struct

Using Structs and Member Variables

- Struct Utilization
  - Once you have a created a struct, you can now create a struct variable by declaring it
    - Example:
      XYCoordinates WaypointOne;
  - Once declared, you can read and write t your struct and its members like a variable.
    - Example:
      WaypointOne.x = 5;
      WaypointOne.y = 3;

```
task main()
{
  typedef struct
  {
    int x;
    int y;
  } XYcoordinates;

  XYcoordinates waypointOne;

  waypointOne.x = 3;
  waypointOne.y = 2;
}
```

Arrays of Structs

- Just like normal array variables, you can create arrays of your structs:
  - Example: XYCoordinates myWaypoints[4];
- Accessing the struct variables is similar with an array as well
  - Example: myWaypoints[2].x = 3;
- This can be useful for iterating through a struct with a for loop!

```
task main()
{
  typedef struct
  {
    int x;
    int y;
  } XYcoordinates;

  XYcoordinates myWaypoints[5];

  myWaypoints[2].x = 3;
  myWaypoints[2].y = 2;
}
```

Robomatter INCORPORATED

Carnegie Mellon Robotics Academy