Variables in C

What is a variable?
Each variable is just a block of memory
Block of memory that equates to a certain value
Actual value is determined by the programmer
- Integer, Byte, A few bits, etc.

Example:
- ASCII character 'A'
  numeric value 65
  In hex = 0x41
  Depending on the debugger, it may appear as 'A', 65, or 0x41
- Array Example
  The string "CprE281x" is represented in memory as
  'C'   'P'   'R'   'E'   '2'   '8'   '1'   'x'   '\0'
  Memory contains
  0x43 0x50 0x52 0x45 0x32 0x38 0x31 0x58 0x00

Variables in C

Variable declaration: compiler knows two things – the name and type of the variable
- int k – reserves 32 bits of memory to hold integer value
  of k
- k is called an object or "a named region of storage"
- In variable assignment the compiler places the
  value in the memory location of the object
  - k = 2 -> places 2 (a 32 bit value) at memory location k
  - The two values associated with k are
    - rvalue – the right value, 2
    - lvalue – the left value, address of k or the object `k'
  - The statement assign on the right side of an
    equation, i.e. 2=k is not acceptable

Variables in C

- myBaseAddr is the name of the pointer
  and it contains an address (lvalue), thus
  - It generally does not make sense to say
    myBaseAddr = 50 or myBaseAddr = myInt
  - This works but does assigning an rvalue to an lvalue
    really make sense? The second case usually gives a
    warning
  - We should write myBaseAddr = 0x0000 0005
    or myBaseAddr = &myInt
  - The statements assign an address to myBaseAddr
  - The & symbol is used to access the lvalue of the
    variable myInt – the andress of myInt
  - Now myBaseAddr points to the address of myInt

Variables in C

- The notation of a variable is just a way of representing a
  specific quantity. The programmer interprets how the
  information is represented and decide how the value is used
  - Passed in the value 50 in an 8 bit quantity
    - Binary   = 0011 0010
    - Hex      = 0x32
    - Decimal  = 50
  - Could be the actual number 50
  - Could be various bits of information
    - If bit 6 is set, do something
    - Could be a combination
      - If bit 6 is set, x = x + lower nibble of the value

Variables in C

- So how do we access the value stored at the
  address pointed to by myBaseAddr?
  - Use the " symbol again - called the dereferencing
  - *myBaseAddr = 50 assigns 50 the address pointed to
    by myBaseAddr
  - How many bytes of memory used to store this value?
  - What should be the value of myInt now?
  - What is the lvalue of myBaseAddr?
  - The rvalue?
  - What happens if we increment myBaseAddr by 1 as in
    myBaseAddr++?
  - What does the statement (*myBaseAddr++) do?
Arrays

- What is an array?
  - Sequence of a specific variable type stored in memory
  - Not a specific type
  - Pointer to a block of memory
- Define an array as
  - `type variableName[arraySize];`
  - Declares "arraySize" elements of type "type" denoted by "variableName"
- Zero-indexed (starts at zero rather than one)
- Last element is found at arraySize-1

Variables in C

- **Strings**
  - What is a string?
    - Special array of type `char` that is ended by the NULL (\0) character
  - Remember to create an array of `N+1` characters to allow space for the NULL character
  - 20 character string `char szString[21]; /* 20 + 1 */`
  - Why is there a NULL character?
  - Otherwise, how can you identify actual chars in a string?

Variables in C

- Be careful of boundaries in C
  - No guard to prevent you from accessing beyond array edge
  - Write beyond array = Potential for disaster

What exactly is an array?
- Not a specific type
- Pointer to a block of memory
- No built-in mechanism for copying arrays

Accessing Arrays – pointers

- As a pointer point to the address of another variable the same is true for arrays, for example
  - `myBaseAddr = &myIntArray[0]` sets the pointer `myBaseAddr` to the address of `myIntArray[0]`
  - What do `*myBaseAddr` and `myIntArray[0]` have in common
  - What does `*(myBaseAddr + 1)` represent in array?
  - What about `*(++myBaseAddr)`?
  - Difference between the previous two statements?
- Can also write `myBaseAddr = myIntArray`
  - I.E. the name of the array is actually the address of the first element of the array

Arrays and Pointers

- Be careful when using pointers and arrays interchangeably – what is wrong with the following code?
  - `char myCharArray[20] = “this is my string”;
  - int *myArrayBaseAddr;
    - `myArrayBaseAddr = myCharArray;`
  - `While(*myArrayBaseAddr != 0)`
    - `printf(“%c”, *myArrayBaseAddr);
      myArrayBaseAddr++;`
Pointers

Points to a spot in memory
- Pointer size is dependent upon addressability of system, not
  type of variable that is being pointed to
  Most microprocessors and 32-bit memory addressable
- char * 32-bit memory address
- long * 32-bit memory address
- float * 32-bit memory address

sizeof function
- Returns the size in bytes of a variable
- Figuring out sizes of a variable on a system (e.g., int)
- Calculating the size of a block of memory

Examples

```
sizeof(char)  = 1
sizeof(char *) = 4
sizeof(long *) = 4
```

Pointers Example

```
int nVal;
int *pnVal;
pnVal = &nVal; /* let address be 0x20000000 */
nVal = 10;
*pnVal = 5;
pnVal is 0x20000000
*pnVal is 5
nVal is 5
```

Pointers

Draw a memory diagram

Three key steps when using pointers:
1. Declare the pointer
   type * pName;
   char * pChar;
   long * pHistory;
2. Initialize the pointer
   In order to use the pointer, we need to point it somewhere.
   pChar = (char *) 0x00001800;
   pHistory = &lValue;
   The (char *) tells the compiler this is a 32-bit memory address, not a 32-bit value.
3. Access the pointer (Read/Write)
   In order to get the value, we must use a * in front of the name.
   n = *pChar | 0x80;
   if(*pHistory + 25 > TOL_HISTORY)
   *pHistory = TOL_MINIMUM;

Embedded Programming Example

Given:
- Temperature 0x2500 float
- AC 0x2520 byte
If temp>80 then turn on AC by setting bit 0 to true

```
float * pfTemp;
char * pAC;
pfTemp = (float *) 0x2500;
pAC = (char *) 0x2520;
if (*pfTemp > 80 )
  *pAC = *pAC | 0x01;
```

Memory Diagram Example

- Assume the following C code
  int myInt;
  char myArray[10] = "CPRE281x";
  char *myCharAddress;
  int *myIntAddress;
  myCharAddress = myArray;
  myIntAddress = &myInt;
  myInt = 200;
- What is myIntAddress?
- What is *myCharAddress?
- What is *myCharAddress++?
- What is myCharAddress now?
- What is *(myCharAddress++)?
- What is *(myIntAddress++)?
Operations in C
Arithmetic operators: + - * / % ()
Shift: the shift operation may be done via an arithmetic shift or by a logical shift

Arithmetic - MSB stays the same on a right shift
Logical - Always shift in a zero

0x0F >> 2 = 0x03;
0x0F << 2 = 0x3C;
0x0F >> 1 =

Operator Precedence
~ ! (unary) ++ --
* / % arithmetic
+ -
<< >> bit shift
< <= >= relational
== != bitwise logical
& ^ | Boolean

Functions
• Goal - Calculate some value or do some task
• Subroutines - May/may not return a value
• Syntax
ReturnType FunctionName
(Type Parameter1Name, Type Parameter2Name, ...)
{ return (expression of ReturnType);
}
• main function is the startup point for all C programs
main ()
{
}

Functions
• Return Types
    void No Return Value
    May return any variable type but an array
    Note: Don't return a pointer to a local variable (more later)
• Examples
    return (0);
    return (nVal);
    return 1;
    return; /* void function */
    return keyword immediately exits the function

Functions
• Parameters
    May have zero or more parameters
    Typically, standard practice is to keep the number of parameters below 5 to 8
    Any type, even an array
def PassArray (char szString[])
    For an array, may or may not declare size
    If the size is not declared, make sure to either know the size ahead of time or to pass the size in as a parameter
    Arrays are passed in as pointers
    All parameters are local variables, i.e. altering the local variable does not affect the actual caller unless the variable is a pointer

Functions
• Prototyping
    How does C look up a function?
    C -> top-down compilation
    Compiler only knows about what it has seen so far i.e at line 20, knows contents of lines 1-20
    Problem: Write the function definition at the bottom, call it at the top
    Solution 1: Move the function definition earlier
    Solution 2: Write a prototype
    Prototype - Tells the compiler the function is defined somewhere in the code
    If the function is prototyped but not defined, linker error
Functions

Prototype
Declaration or header line of function, up to first curly brace, plus semicolon
- No semicolon = compiler expects function body (i.e., code)
- Semicolon = prototype

- Declaration
  void WritePrototype (char szString[], short nStringLen) {}
- Prototype
  void WritePrototype (char szString[], short nStringLen);
- Call
  Syntax: FunctionName (parameter1, parameter2, etc.):
  if(x > 5)
    WritePrototype(szName,20);

Passing Variables
Can pass via one of two ways:
1. Pass to be read only (Write - No effect)
2. Pass allowing changes (Write - Changes actual variables)

Pass by value ("call by value"), i.e. no changes
void DoValue (int, float, char);
...
DoValue (5, 2.5, 'A');
DoValue (nTest, fPressure, szName);

Value - A local variable on the stack

Pass by pointer ("call by reference")
i.e., allow changes
void DoChanges(int *, float *, char[]);
DoChanges(5, 2.5,"test"); /* Can't do this, need a variable to use */
DoChanges(&nTest,&fPressure,szName);

In order to allow changes to the variable, must pass as a pointer
- Memory Address - Access to actual variable itself

How does this happen?
Parameters are set up as local variables
- Created on the stack
- Visible only to the function
- Enter the function: Space is created
- Exit the function: Space is destroyed
  - Not really destroyed, just changed to garbage status

Why is returning a pointer to a local variable bad?
Return a value - OK - actual value and mechanisms are set up for that
Return an address - Address to memory that may/may not be garbage

Global vs. Local
Global variable
- Declared outside of all functions
- May be initialized upon program startup
- Visible and usable everywhere from .c file

What happens when local/global have the same name?
- Local takes precedence

Summary
- Local - declared inside of a function, visible only to function
- Global - declared outside all functions, visible to all functions

What happens when you want a local variable to stick around but do not want to use a global variable?
Create a static variable
Syntax:
  static Type Name;

Static variables are initialized once
Think of static variables as a "local" global
Sticks around (has persistence) but only the function can access it
Control-Flow in C

Flow Control - Making the program behave in a particular manner depending on the input given to the program.

Why do we need flow control?
Not all program parts are executed all of the time, i.e., we want the program to intelligently choose what to do.

Statements for Boolean flow control
if, else if, else
The evaluation for Boolean flow control is done on a TRUE / FALSE basis. TRUE / FALSE in the context of a computer is defined as non-zero (TRUE) or zero (FALSE).
-1, 5, 15, 225, 325.33 TRUE
0 FALSE

Flow Control/Control Flow in C

if, else if, else
Must always have "if"; may/may not have "else if" or "else"
Syntax
if { (Condition1) 
  { ...
  } else if (Condition2)
  { ...
  } else if (Condition3)
  { ...
  } else
  { ...

Example

if ( nVal > 10)
{ 
  nVal += 5;
}
else if (nVal > 5) /* If we reach this point, */
{ /* nVal must be <= 10 */
  nVal -= 3;
}
else /* If we reach this point, */
{ /* nVal must be <= 10 and */
  nVal = 0; /* nVal must be <= 5 */
}

Comparisons (Relational Operators) - Numeric
> (greater), >= (geq), < (less), <= (leq), == (Equal), != (Net Equal)
Comparison gives a result of zero (FALSE) or !zero (TRUE).
A TRUE result may not necessarily be a 1
Equality
Double equals sign ==
- Assigns a value
- Tests for equality, returns non-zero or zero
if (nVal == 5) versus if (nVal = 5)
The second expression always evaluates to TRUE.
Why?

Multiple condition tie together using Boolean (logical) operators, & (AND), | (OR), ! (NOT)
if ( nVal > 0 ) && (nArea < 10)
if ( (nVal < 3) || (nArea > 50) )
if ( !(nVal <= 10) )

Conditions are evaluated using lazy evaluation.
Lazy evaluation - Once a condition is found that completes the condition, stop
OR any condition is found to be TRUE
OR anything = 1
AND any condition is found to be FALSE
AND anything = 0

Why is lazy evaluation important?
Makes code run faster - skips unnecessary code
Know condition will/will not evaluate, why evaluate other terms
Can use lazy evaluation to guard against unwanted conditions
Checking for a NULL pointer before using the pointer