Pointers!

Explicit Memory
Access & Management

By Jason Agron

Variables

- Variables are heavily used constructs in any programming language.
- Variables are often “nicknames” of storage locations:
  - Registers or memory locations.
- Who cares where variables are stored?
  - Usually, not the programmer.
    - It can be handled solely by the compiler.

Example: Variable Referencing

- Original Source:

  int x = 0;

  int y = 1;

  x = x + 3;

  y = x + y;

- Assume…
  - Compiler chooses variable locations.
  - Address of x = 0x0A.
  - Address of y = 0xFF.

- Program and Associated Actions:
  - int x = 0;
    - Mem[0x0A] = 0x00000000
  - int y = 1;
    - Mem[0xFF] = 0x00000001
  - x = x + 3;
    - R12 = Mem[0x0A]
    - R12 = R12 + 3
    - Mem[0x0A] = R12
  - y = x + y;
    - R13 = Mem[0xFF]
    - R13 = R12 + R13
    - Mem[0xFF] = R13

"Fixed" Variables

- Variables in a user's program can usually be put *anywhere*.
- But what about variables (data) that corresponds to fixed devices?
  - i.e. hardware devices.
- These variables have fixed addresses.
  - *How does one read variables at a fixed address?*

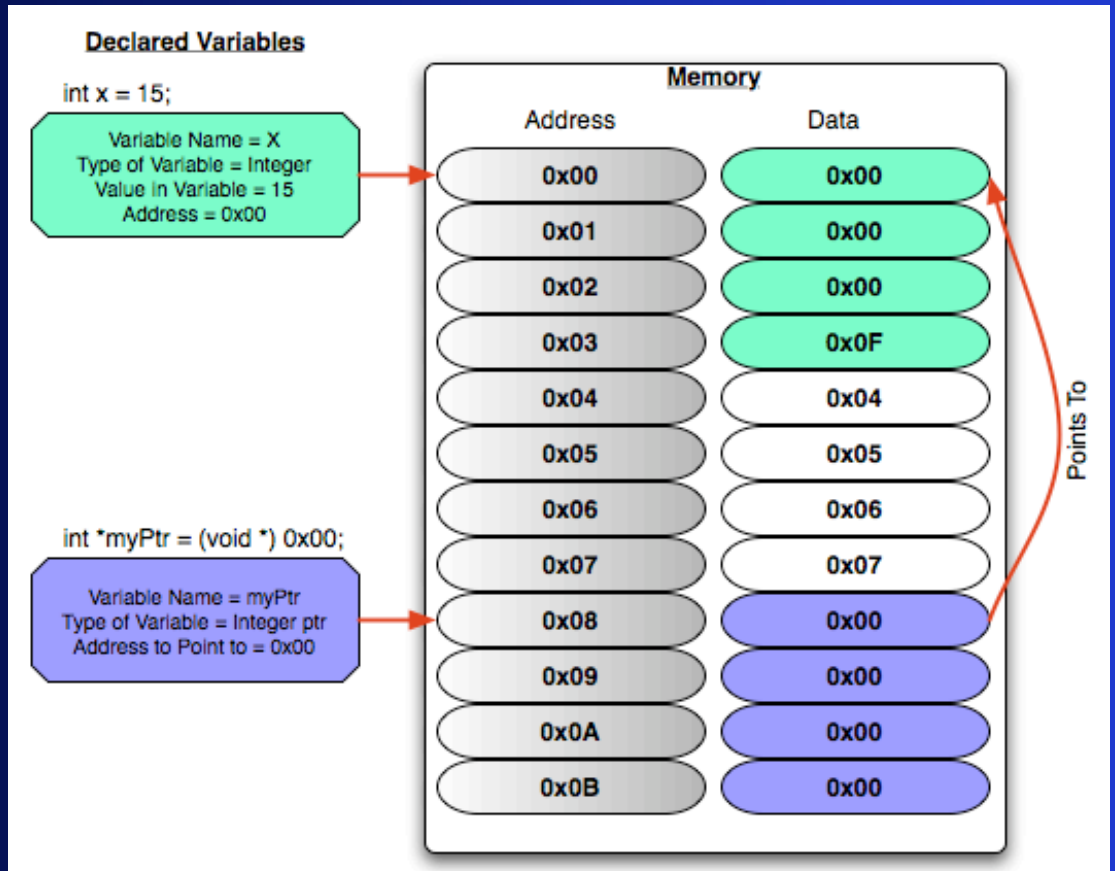
Pointers!

- Pointers are programming constructs:
  - They provide the idea of “indirection”.
  - They are merely variables which are used to access other places.
- All pointers have the following:
  - The “address” that the pointer “points to”.
  - The “data” that is being pointed at.

Pointers Vs. Variables

- In the previous example…
  - Variable names were merely “nicknames” for addresses.
    - Memory location 0x0A was named ‘x’.
    - Memory location 0xFF was named ‘y’.
  - The value stored at the memory location is data.
- Pointers work a bit differently.
  - They too are nicknames for a memory location.
  - The value stored in that memory location is an address, not data!

Example - Pointer Vs. Variable



In this case, the address of variable 'x' is known to be 0x00. Therefore it is possible to set the pointer 'myPtr' to point to 'x' by setting it's value to 0x00.

If the address of 'x' is not known to the programmer, then it can be looked up instead by using '&' which is the "address-of" operator:

int *myPtr = (void *) (&x);

Features of Pointers (1)

- The “value” of a pointer is the address that it points to.
  - int *myPtr = <ADDRESS_TO_POINT_TO>;
- The “type” in front specifies what type of data you are pointing at.
  - Important!!
  - Integers are 32-bit (4 bytes), Chars are 8-bit (1 byte)…
- To get the address pointed to…
  - <addressPointedTo> = myPtr;
- To set the address pointed to…
  - myPtr = <newAddressPointedTo>;

Features of Pointers (2)

- To get the “data” that is pointed to, the pointer must be dereferenced.
  - This is done using the ‘*’ operator.
- To read the data being pointed to…
  - <dataPointedTo> = *myPtr;
- To write the data being pointed to…
  - *myPtr = <newData>;

Address-Of Operator

- The ‘&’ operator is used to calculate the address of a variable.
- In our first example…
  - &x would return 0x0A.
  - &y would return 0xFF.
- This is how pass-by-reference works!
  - You don’t pass the value.
  - Instead you pass a “reference” (address) to where the value is located.

Example: Use Case

- There is a hardware device with three 32-bit registers attached to the system bus.
  - Base address of the device is 0x40000000.
  - Reg0 = base + 0x0.
  - Reg1 = base + 0x4.
  - Reg2 = base + 0x8.
  - Registers are readable and writable.
- *Why are the register addresses each separated by 0x4??*

Accessing the Registers (1)

- First declare pointers to access the registers:
  - volatile int *reg0 = (int *)(0x40000000);
  - volatile int *reg1 = (int *)(0x40000004);
  - volatile int *reg2 = (int *)(0x40000008);
- The ‘volatile’ keyword tells the compiler:
  - Variable cannot be stored in on-CPU registers.
  - Because it can be modified by “external” processes.
- Forces the compiler to produce code that always does bus transactions to write/read such variables.
  - EXCEPT if caching is enabled!!!!!

Accessing the Registers (2)

- Now let's write/read the variables…
  - Put values in reg0, reg1, and reg2…
    - *reg0 = 15678;
    - *reg1 = 1 + 3;
    - *reg2 = -99;
  - Read values from registers…
    - *reg2 = (*reg1 + 10) / (*reg2);
- Now you are interacting with the data stored in the device's registers!!