CSE 31
Computer Organization

Lecture 2 – A Quick Start with C Programming, C Pointers

Announcements

- Lab
  - Lab 1 out this week
    - » Due at 11:59pm on the same day of your lab during week after next (with 7 days grace period after due date)
    - » You must demo your submission to your TA within 21 days
    - » Demo is REQUIRED to receive full credit
- Reading assignment
  - Chapter 4-6 of K&R (C book) to review C/C++ programming

History Lesson on C

- C developed by Dennis Ritchie at AT&T Bell Labs in the 1970s.
  - Used to maintain UNIX systems
  - C was derived from the B language
  - B was derived from the BCPL (Basic Combined Programming Language)
  - Many commercial applications are still written in C
- Current standard updates
  - C11: improved Unicode support, cross-platform multi-threading API
  - C99 or C9x remains the common standard

History Lesson on C

- References
  - `http://en.wikipedia.org/wiki/C99`
- Highlights of C99
  - Declarations in for-loops, like Java
  - Java-like `//` comments (to end of line)
  - Variable-length non-global arrays
  - `<inttypes.h>`: explicit integer types (intN_t, unintN_t)
  - `<stdbool.h>` for boolean logic def's
- Current version is C17 (or C18)
- Soon to be replaced by C2x
  - Expected to be voted on in 2023

Disclaimer

- Important: You will not learn how to fully code in C in these lectures! You'll still need your C reference for this course:
  - K&R is a must-have reference
  - Check online for more sources

Compilation: Overview

C compilers take C and convert it into an architecture specific machine code (string of 1s and 0s).

- Unlike Java which converts to architecture independent bytecode.
- Unlike most functional programing languages (e.g. Scheme) which interpret the code.
- These differ mainly in when your program is converted to machine instructions.
- For C, generally a 2-part process of compiling .c files to .o files, then linking the .o files into executables. Assembling is also done (but is hidden, i.e., done automatically, by default)
  - » We will learn these in later lectures.

Compilation: Advantages

- Great run-time performance: generally, much faster than Java (interpreted and compiled) or Python (interpreted) for comparable code (because it optimizes for a given architecture)
- OK compilation time: enhancements in compilation procedure (`Makefiles`) allow only modified files to be recompiled

Compilation: Disadvantages

- All compiled files (including the executable) are architecture specific, depending on both the CPU type and the operating system
- Executable must be rebuilt on each new system.
  - Called “porting your code” to a new architecture.
- The “change → compile → run [repeat]” iteration cycle is slow

C vs. Java™ Overview (1/2)

Java	C
Object-oriented (OOP)	No built-in object abstraction. Data separate from methods.
“Methods”	“Functions”
Class libraries of data structures	C libraries are lower-level
Automatic memory management	Manual memory management

C vs. Java™ Overview (1/2)

Java	C
High memory overhead from class libraries	Low memory overhead
Relatively Slow	Relatively Fast
Arrays initialize to zero	Arrays initialize to garbage

Syntax:
`/* comment */`
`// comment`
`System.out.print`
	Syntax:
	`/* comment */`
	`// comment`
	`printf`

You need newer C compilers to allow Java style comments, or just use C99

C Syntax: main

- To get the **main** function to accept arguments, use this:

```
int main (int argc, char *argv[])
```

- What does this mean?
  - **argc** will contain the number of strings on the command line (the executable counts as one, plus one for each argument). Here argc is 2:

```
   ./sort myFile
```

  - **argv** is a pointer to an array containing the arguments as strings (more on pointers later).
  - Always return a value according to ANSI (American National Standard Institute)

C Syntax: Variable Declarations

- Very similar to Java, but with a few minor but important differences
- All variable declarations must go before they are used (at the beginning of the block)*
- A variable may be initialized in its declaration; if not, it can hold garbage!
- Examples of declarations:
  - correct: `int a = 0, b = 10;`

    ```
    ...
    ```
  - Incorrect:* `for (int i = 0; i < 10; i++)`

*C99 overcomes these limitations

C Syntax: True or False?

- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (pointer: more on this later)
  - no such thing as a Boolean*
- What evaluates to TRUE in C?
  - everything else…

Boolean types provided by C99's `stdbool.h`

C syntax : flow control

- Within a function, remarkably close to constructs in several other languages such as Java, C++, Perl, PHP, Javascript, Go, etc. (showing their legacy) in terms of flow control
  - `if-else`
  - `switch`
  - `while` and `for`
  - `do-while`

Common C Error

`a = b` VS `a == b`

- There is a difference between assignment and equality

  `a = b` is assignment

  `a == b` is an equality test

- This is one of the most common errors for beginning programmers!
  - One solution (when comparing with constant) is to put the variable on the right!
    If you happen to use =, it won't compile.

    ```
    if (3 == a) { ... }
    ```

All objects have a size

- The size of their representation
- The size of static objects is given by sizeof operator (**in Bytes**)

```
#include <stdio.h>
int main() {
      char c = 'a';
      int  x = 34;
      int  y[4];
      printf("sizeof(c)=%d\n",   sizeof(c) );
      printf("sizeof(char)=%d\n",sizeof(char));
      printf("sizeof(x)=%d\n",   sizeof(x) );
      printf("sizeof(int)=%d\n", sizeof(int) );
       printf("sizeof(y)=%d\n",   sizeof(y) );
       printf("sizeof(7)=%d\n",   sizeof(7) );
      return 0;
}
```

Output:

```
sizeof(c)=1
sizeof(char)=1
sizeof(x)=4
sizeof(int)=4
sizeof(y)=16
sizeof(7)=4
```

Quiz:

```
void main(); {
  int *p, x=5, y; // init
  y = *(p = &x) + 1;
  int z;
  flip-sign(p);
  printf("x=%d, y=%d, *p=%d\n", x, y, p);
}
flip-sign(int *n){*n = -(*n)}
```

How many syntax+logic errors in this C99 code?

```
#Errors
  a)1
  b)2
  c)3
  d)4
  e)5
```

Quiz: Answer

```
void main(); {
   int *p, x=5, y; // init
   y = *(p = &x) + 1;
   int z;
   flip-sign(p);
   printf("x=%d, y=%d, *p=%d\n", x, y, *p);
}
flip-sign(int *n){*n = -(*n);}
```

How many syntax+logic errors in this C99 code?

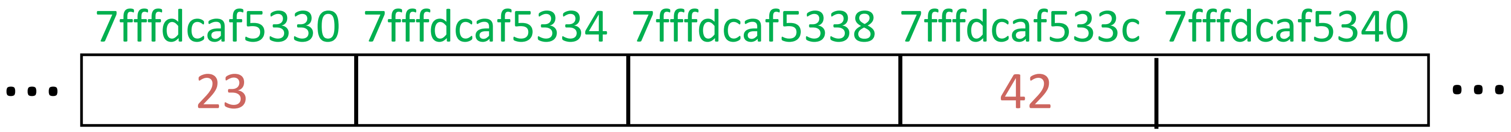
5…

(signed ptr print is logical err)

```
#Errors
  a) 1
  b) 2
  c) 3
  d) 4
  e) 5
```

Address vs. Value

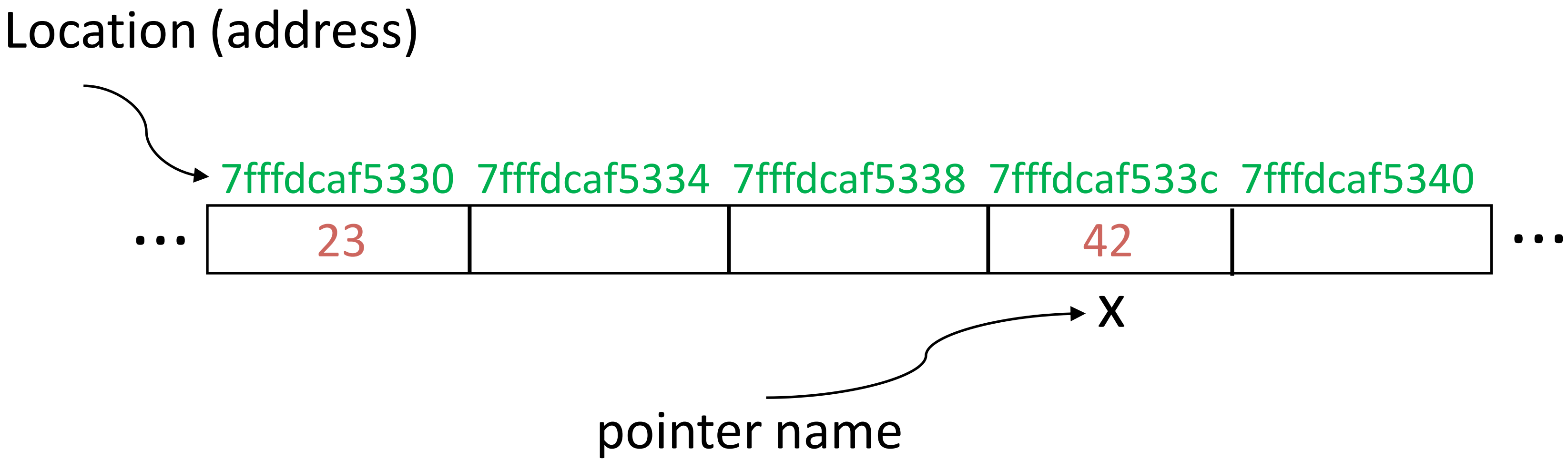
- Consider memory to be a single huge array:
  - Each cell of the array has an address associated with it.
  - Each cell also stores some value.
  - Do you think addresses use signed or unsigned numbers?
    - » Negative address?!
- Don’t confuse the address referring to a memory location with the value stored in that location.



	7fffdcaf5330	7fffdcaf5334	7fffdcaf5338	7fffdcaf533c	7fffdcaf5340	
…	23			42		…

Pointers

- An address refers to a particular memory location. In other words, it points to a memory location.
- Pointer: A variable that contains the address of a variable.



Location (address)

…	7fffdcaf5330	7fffdcaf5334	7fffdcaf5338	7fffdcaf533c	7fffdcaf5340	…
	23			42		

x

pointer name

Pointers

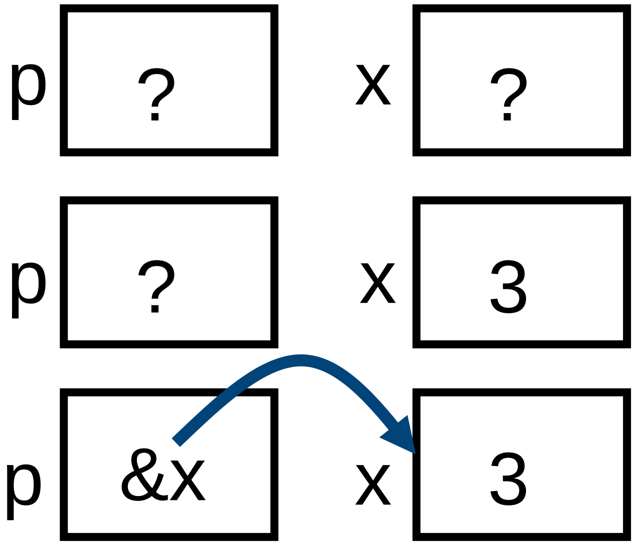
- How to create a pointer:

  & operator: get address of a variable

```
int *p, x;

x = 3;

p = &x;
```



Note the "*" gets used 2 different ways in this example. In the declaration to indicate that p is going to be a pointer, and in the printf to get the value pointed to by p.

- How to get a value pointed to?

  * "dereference operator": get value pointed to

```
printf("p points to %d\n",*p);
```