The Hardware/Software Interface

CSE351 Winter 2011

Module 3: Integers
(and more about C pointers)

Today's Topics

- **Representation of integers: unsigned and signed**
- **Arithmetic and shifting**
- **C Pointer arithmetic**
  - **And more about C pointers**

Encoding Integers

- **The hardware (and C) supports two flavors of integers:**
  - unsigned – only the non-negatives
  - signed – both negatives and non-negatives
- **There are only 2^W distinct bit patters of W bits, so...**
  - Can't represent all the integers
  - Unsigned values are 0 ... 2^W-1
  - Signed values are -2^{W-1}... 2^{W-1}-1

Unsigned Integers

- Unsigned values are just what you expect
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_7 2^7 + b_6 2^6 + b_5 2^5 + \ldots + b_1 2^1 + b_0 2^0$
    - Interesting aside: $1+2+4+8+\ldots+2^{N-1} = 2^N-1$

```
 00111111
+00000001
 01000000
```

```
 63
+ 1
 64
```

- You add/subtract them using the normal "carry/borrow" rules, just in binary
- An important use of unsigned integers in C is pointers
  - There are no negative memory addresses

2's Complement (Signed) Integers

- **Let's do the natural thing for the positives**
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): 0x00 = 0, 0x01 = 1, …, 0x7F = 127
- **But, we need to let about half of them be negative**
  - Use the high order bit to indicate 'negative'
  - Call it “the sign bit”
  - Examples (8 bits):
    - 0x00 = 00000000_2 is non-negative, because the sign bit is 0
    - 0x7F = 01111111_2 is non-negative
    - 0x80 = 10000000_2 is negative

2's Complement Negatives

- **How should we represent -1 in binary?**
  - Possibility 1: 10000001_2
    Use the MSB for “+ or -”, and the other bits to give magnitude
    (Unfortunate side effect: there are two representations of 0!)
  - Possibility 2: It would be handy if we could use the same hardware adder to add signed integers as unsigned
    - We add unsigned using the simple carry rule, so…
    - What should the 8-bit representation of -1 be?

```
 00000001
+????????      (want whichever bit string that gives right result)
 00000000

 00000010      00000011
+????????     +????????
 00000000      00000000
```

2's Complement Negatives

- **What should the value of -1 be, in binary?**
  - Possibility 1: 10000001_2
    Use the MSB for "+ or -", and the other bits to give magnitude
  - Possibility 2: It would be handy if we could use the same hardware adder to add signed integers as unsigned
    - We add unsigned using the simple carry rule, so…
    - What should the 8-bit representation of -1 be?

```
 00000001
+11111111      (want whichever bit string that gives right result)
 00000000

 00000010      00000011
+11111110     +11111101
 00000000      00000000
```


Unsigned & Signed Numeric Values

X	Unsigned	Signed
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- Both signed and unsigned integers have limits
  - If you compute a number that is too big, you wrap
  - If you compute a number that is too small, you wrap
- The CPU may be capable of "throwing an exception" for overflow on signed values
  - It won't for unsigned
- But C and Java just cruise along silently when overflow occurs…

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1
- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- **Other Values**
  - Minus 1
    111…1 0xFFFFFFFF (32 bits)

Values for *W* = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Values for Different Word Sizes

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

Arithmetic and Shift Operations

- **Left shift: `x << y`**
  - Multiplies x by 2**y
    - 00000010_2 << 2 == 00001000_2 *$2 * 2^2 = 8$*

      - *Undefined behavior when y < 0 or y ≥ word_size*
      - *You CAN get the wrong result (overflow)*

- **(Arithmetic) Right shift: `x >> y`**
  - Divide x by 2**y
    - 00000010_2 >> 1 == 00000001_2 *$2 / 2^1 == 1$*
    - 11111110_2 >> 1 == 11111111_2 *$-2 / 2^1 == -1$*
    - *Note: correct truncation (towards 0) requires some care with signed ints*

Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix
    - `0U, 4294967259U`
- **Casting**
  - `int tx, ty;`
  - `unsigned ux, uy;`
  - Explicit casting between signed & unsigned same as U2T and T2U
    - `tx = (int) ux;`
    - `uy = (unsigned) ty;`
  - Implicit casting also occurs via assignments and procedure calls
    - `tx = ux;`
    - `uy = ty;`
-

Pointer Arithmetic

- Pointer values are unsigned
  - Unfortunately, the software (e.g., the OS) sets things up in a way that you're unlikely ever to deal with values where signed vs. unsigned matters...

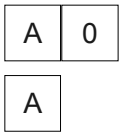
- int* pInt; // p can hold an address that points to an integer
  int * pInt2; // same thing
  int *pInt3; // also the same thing
  int* pInt4, pInt5; // pInt4 is a pointer, but pInt5 is an int
  int *pInt6, *pInt7; // both pInt6 and pInt7 are pointers

- pInt++; // is legal
  - It means "increment pInt to the next element of the type it points at"
  - The value in pInt is always incremented in units of bytes
    - So, in this case, pInt is increased by 4 (#bytes in an int)

- char* pChar = "This is a test"; // this is legal too; 'pChar' names 4 bytes; '*pChar' names one byte
  char c = *(pChar+8);
  - c == 'a'
  - *(pChar+8) and pChar[8] have exactly the same meaning in C

String functions in C

- **Strings are a <u>convention</u> in C**
  - A "null terminated array of char"
    - The "null" is a byte with value 0, written '\0' as a character
  - What's the difference between "A" and 'A'?
    - "A" occupies two bytes
    - 'A' occupies one byte



- **Operations on strings are done using library methods that understand the convention**
  - `strlen(), strcpy(), strcat(), strcmp(), ...`

Example `strlen()` implementation

```
int strlen(char* str) {
    char* p;
    for ( p=str; *p; p++ ) ;
    return (int)(p - str);
}
```

- **p is initialized to point to the 1st character of the string**
- **each loop iteration increments p – points to next char in string**
- **we're done when the character p points at is false (0)**
- **successive string bytes occupy higher memory addresses**
  - So the result is p – str
    - strlen doesn't count the '\0' as part of the length of the string

Error scenarios

```
int strlen(char* str) {
    char* p;
    for ( p=str; *p; p++ ) ;
    return (int)(p - str);
}
```

- **What if the caller forgot to null terminate the string?**
- **What if the caller passes in a pointer that doesn't point to a string?**
  - E.g., what does `strlen(NULL)` return?
- **Can we add code to `strlen()` to detect these errors?**

Badness Example 1

```
int strlen(char* str) {
    char* p;
    for ( p=str; *p; p++ ) ;
    return (int)(p - str);
}
```

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    char* p = "This is a test";
    printf( "%d\n", strlen(p) );
    printf( "%d\n", strlen(&p[3]) );
    printf( "%d\n", strlen(p[3]) );
    return 0;
}
```

What is printed?

(What does char* argv[] *mean?)*

Badness Example 1 Answers

```
$ gcc -Wall strtest.c
strtest.c: In function 'main':
strtest.c:15: warning: passing argument 1 of 'strlen' makes pointer from
integer without a cast
/usr/include/string.h:399: note: expected 'const char *' but argument is
of type 'char'


$ ./a.out
14
11
Segmentation fault
```

Note: I compiled the code for main() *from the previous slide, but with*
#include <string.h>
added at the top, and without the code for strlen() *in my file.*

Badness Example 2

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int intArray1[] = {1, 2, 3, 0};
    int intArray2[] = {-1, -2, -3};
    printf( "%d\n", strlen(intArray1) );
    printf( "%d\n", strlen(intArray2) );
    return 0;
}
```

What is printed?

Badness Example 2 Answers

```
int intArray1[] = {1, 2, 3, 0};
int intArray2[] = {-1, -2, -3};
```

```
$ gcc -Wall strtest.c
strtest.c: In function 'main':
strtest.c:18: warning: passing argument 1 of 'strlen' from incompatible
pointer type
/usr/include/string.h:399: note: expected 'const char *' but argument is
of type 'int *'
strtest.c:19: warning: passing argument 1 of 'strlen' from incompatible
pointer type
/usr/include/string.h:399: note: expected 'const char *' but argument is
of type 'int *'


$ ./a.out
1
16
```

Even Worse Example 3

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int intArray1[] = {1, 2, 3, 0};
    int intArray2[] = {-1, -2, -3};
    printf( "%d\n", strlen((char*)intArray1) );
    printf( "%d\n", strlen((char*)intArray2) );
    return 0;
}
```

```
$ gcc -Wall strtest.c
$ ./a.out
1
16
```