Data Types and Type Conversions
ENEE 140

Prof. Tudor Dumitraș
Assistant Professor, ECE
University of Maryland, College Park

http://ter.ps/enee140

Today’s Lecture

• Where we’ve been
  – Scalar data types (int, long, float, double, char)
  – Basic control flow (while and if)
  – Functions

• Where we’re going today
  – Data types and type conversion
  – Bitwise operations
  – Branching
  – Global variables
  – Random number generation
  – Testing
  – Project 1

• Where we’re going next
  – Vector data types (arrays and strings)
Limits for Integers

- We’ve seen:
  - UINT_MIN = 0
  - UINT_MAX = $2^w - 1$ ($w = 32$ on the GRACE machines)

- Binary representation:
  - UINT_MIN: $(000...0)\text{ w bits}$
  - UINT_MAX: $(111...1)\text{ w bits}$

Machine Representation of Integers

- Math deals with an infinite set of integers

- On a computer you can only represent a finite set of numbers
  - The limits of the int numbers you can use in your C programs are architecture-dependent
  - Example, on the GRACE machines:
    - unsigned a; 4 bytes (32 bits)
    - unsigned long a; 8 bytes (64 bits)

- How many values can you represent using 32 bits?
  - $2^{32}$
  - That’s why UINT_MAX is $2^{32}-1$
    - Between 0 and UINT_MAX there are $2^{32}$ numbers.
The sizeof Operator

• Yields the **number of bytes** required to store a variable of the type of its operand
  
  – Can provide a variable or a type name
  
  – For example, on the GRACE machines:

```c
int a;
sizeof(a) 4
sizeof(char) 1
sizeof(int) 4
sizeof(unsigned) 4
sizeof(long) 8
sizeof(unsigned long) 8
sizeof(float) 4
sizeof(double) 8
```

x 8 = **number of bits**

Binary Representation of Numbers

• We commonly use numbers in **base 10**
  
  – 10 possible digits:
  
  ```
  0 .. 9
  ```
  
  – Carry to the next order of magnitude:
  
  ```
  9 + 1 = 10
  ```
  
  – Value of 4-digit number $d_3 \, d_2 \, d_1 \, d_0$:
  
  ```
  D = \sum_{i=0}^3 d_i \cdot 10^i
  ```
  
  – Example:
  
  ```
  15 = 1*10^1 + 5*10^0
  ```

• Computers use numbers in **base 2**
  
  – 2 possible digits:
  
  ```
  0, 1
  ```
  
  – Carry to the next order of magnitude:
  
  ```
  1_2 + 1_2 = 10_2
  ```
  
  – Value of 32-bit binary number $B=b_31 \, b_30 \, \ldots \, b_1 \, b_0$:
  
  ```
  B = \sum_{i=0}^{31} b_i \cdot 2^i
  ```
  
  – Example: $0101_2 = 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 5_{10}$
Binary Representation of Numbers – cont’d

• Value of 32-bit binary number $B=b_{31} \ b_{30} \ldots \ b_1 \ b_0$:  
  
  $$B = \sum_{i=0}^{31} b_i \cdot 2^i$$

• This is the representation of unsigned variables
  – Signed integers and floating point variables use more complex representations (more on this in ENEE 350)

• Signed integers use one bit to store the sign
  – Using 32-bit int’s you can represent as many values as with 32-bit unsigned
  – However, only about half of these values are positive

Bitwise Operations

• Operators for manipulating bits:
  – & bitwise AND
  – | bitwise OR
  – ^ bitwise XOR (exclusive OR)
  – << left shift
  – >> right shift
  – ~ flip all bits (unary)

• Common usage: bit masks
  – $a = a \ & \ 1$; set all but lowest order bit to 0
  – $a = a \ \mid \ 1$; set lowest order bit to 1;
  – $b = (a \gg 3) \ & \ 1$; find value of bit $b_3$ from $b_{31} \ldots b_3 \ b_2 \ b_1 \ b_0$
Integer Overflow Revisited

- We’ve seen:
  - `UINT_MAX + 1 = 0`

- Why?
  - Say \( w = 4 \)
  - We can represent \( 2^w = 16 \) numbers
    - Unsigned range: \( 0 .. 15 \)
    - `UINT_MAX = 2^w - 1 = 15_{10} = 1111_2`
    - `UINT_MAX + 1 = 1111_2 + 1 = 10000_2`

Review: Integer Limits and Overflow

- We’ve seen
  - `sizeof(unsigned) == 32` (on GRACE machines)
  - Maximum unsigned value `UINT_MAX` is \( 2^{32} - 1 \approx 4.3 \) billion
  - Unsigned arithmetic operations are done modulo \( 2^{32} \)

```c
unsigned a = 1;
1  a = 2 * a;  a is 2
2  a = 2 * a;  a is 4
3  a = 2 * a;  a is \( 2^3 = 8 \)
...
31 a = 2 * a;  a is \( 2^{31} \)
32 a = 2 * a;  a is 0 (overflow!)
33 a = 2 * a;  a is 0
```
Implicit and Explicit Type Casts

• We’ve seen

```c
float b = 1 / 2;  // value of b is 0
float b = 1.0 / 2;  // value of b is 0.5
```

– In the first example, 0 (the result of integer division) is converted to
  `float` and assigned to `b`

– In the second example, 2 is converted to `float` to perform the operation
  using the rules of floating-point arithmetic

– These are **implicit type casts**

• You can also specify the type conversion using **explicit casts**

```c
float b = (float)1 / 2;  // value of b is 0.5
```

Rules for Type Conversions in C

• In expressions with floating point and integer variables:
  • **Integers** are cast to **floating point**

• In expressions with unsigned and int:
  • **Signed** values are cast to **unsigned**

• In expressions with variables of different storage sizes:
  – The smaller-size numbers are converted to the larger size
    (e.g. `int` is converted to `long int`)
  – This does not incur overflow or loss of precision

• In assignments
  – The value on the **right side** of an assignment is cast to the type of the **left side**
  – This happens after the operation is performed

• The complete rules are in K&R Chapter 2.7
Random Number Generation

- Many computer applications require random numbers
  - Example: coin toss results in heads or tails, each with probability $p = \frac{1}{2}$

- Computers produce pseudo-random numbers
  - Sequence of numbers that appears random
  - The numbers in the sequence follow certain mathematical properties, e.g. uniform distribution
    - Uniform distribution: all values have equal probabilities
    - More about probability distributions in ENEE 324

- Random number generators (RNGs) typically require the programmer to provide a seed before generating the sequence
  - Same seed provided => same sequence generated
  - Seed must be a unique number

Generating Random Numbers in C

- The C standard library provides a basic RNG
  - Must include stdlib.h

- Seed the random number generator (RNG) only once
  ```c
  #include <stdlib.h>
  #include <time.h>
  srand( time(NULL) );
  ```
  seed RNG with current time

- Generate multiple (pseudo) random numbers
  ```c
  int x = rand(), y = rand(), z = rand();
  ```
  - rand() returns a pseudo-random integer in the range $[0, \text{RAND\_MAX}]$
  - RAND\_MAX is also defined in stdlib.h
How Does a Random Number Generator Work?

• A common method: **linear congruential (LC) generator**
  
  – Generates sequence $X_0, X_1, X_2, ...$
  – $X_0$ is initialized with the seed
  – $X_{i+1}$ is computed based on $X_i$ using the following formula:

$$X_{i+1} = (A \times X_i + B) \mod M$$

– Three parameters:
  
  - **$A$**: the multiplier
  - **$B$**: the increment
  - **$M$**: the modulus

Some Properties of LC Generators

• $X_{i+1}$ is computed based on $X_i$ using the following formula:

$$X_{i+1} = (A \times X_i + B) \mod M$$

– The largest number that can be generated is $M - 1$
– When $M = 2^{32}$ and operations done on 32-bit integers, modulus operation can be omitted
– Sequence $X_i$ is a cycle of numbers that are repeated periodically (**orbit**)
– With good choices for $A$, $B$ and $M$, the orbit is a complete permutation: every 32-bit integer is generated exactly once
– **Example**: $A = 214013$, $B = 2531011$, $M = 2^{32}$
Global Variables

• We’ve seen: variables declared inside a function
  ```c
  void fun()
  {
      int a;     // variable a declared inside function fun()
  ...
  }
  ```
  – Only visible inside that function

• **Global variables**: variables declared outside any function
  ```c
  int b;     // global variable b
  int main()
  {
      ...
  }
  ```
  – Global variables are visible in any function of the program
    (more on variable scope later)

Testing

• Complex programs are more likely to have bugs

• It is important to test these programs thoroughly, with a broad range of inputs
  – Create several sets of input values (**test cases**)
  – Think about **corner cases** (e.g. limit > RAND_MAX)

• Good programming practice: write test cases **before** writing the program
  – This helps you clarify what the program should do

• **Debugging is not enough** for writing correct programs
  – You must also create **rigorous tests**
Review of Lecture

- What did we learn?
  - Binary representation of unsigned integers
  - Bitwise operations and bit masks
  - Type conversions
  - Global variables
  - Random numbers
  - The linear-congruential random number generator
  - Testing

- Next lecture
  - Arrays and strings

- Assignments for this week
  - Read K&R Chapters 1.6, 1.9, 2.3, 2.4, 4.1, 4.2, B3
  - Weekly challenge: strncpy.c
  - Homework: lab06.pdf (on http://ter.ps/enee140), due on Friday at 11:59 pm
  - Quiz 5, due on Monday at 11:59 pm
  - Project 1: enee140_s16_p1.pdf (on http://ter.ps/enee140), due on March 21 at 11:59 pm