Integers

Arquitectura de Computadores
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

Today: Integers

- Encoding integers
- Conversion, casting
- Expanding, truncating
- When should you use unsigned?
- Integer arithmetic

Introduction

- **Integers represent a growing and underestimated source of vulnerabilities in C and C++ programs**

- **Integer range checking has not been systematically applied in the development of most C and C++ software**
  - Security flaws involving integers exist
  - A portion of these are likely to be vulnerabilities

- **A software vulnerability may result when a program evaluates an integer to an unexpected value**
  - It is no longer acceptable to assume a program will operate normally given a range of expected inputs when an attacker is looking for input values that produce an abnormal effect

Encoding integers

- **C supports a variety of *integral* data types**
  - `char`, `short`, `int`, `long`, `long long`
  - Along with an indication of whether the represented numbers are all nonnegative (declared as `unsigned`), or possibly negative (the default)

- **The most common computer representation of signed numbers is known as two's-complement form**

- **In two's complement, most significant bit indicates sign**
  - 0 for non-negative
  - 1 for negative

Encoding integers

- **Unsigned**

$$B2U(X) \;=\; \sum_{i=0}^{w-1} x_i \cdot 2^i$$

- **Signed: Two's Complement**

$$B2T(X) \;=\; -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign bit

Two's complement: Simple example (*w*=5)

	-16	8	4	2	1	
10 =	0	1	0	1	0	8+2 = 10

	-16	8	4	2	1	
-10 =	1	0	1	1	0	-16+4+2 = -10

Encoding integers: another example

```
/* 00111011 01101101(2) */
 short x = 15213;


/* 11000100 10010011(2) */
 short y = -15213;
```

Weight	15213	-15213
1	1	1
2	0	1
4	1	0
8	1	0
16	0	1
32	1	0
64	1	0
128	0	1
256	1	0
512	1	0
1024	0	1
2048	1	0
4096	1	0
8192	1	0
16384	0	1
-32768	0	1
Sum	**15213**	**-15213**

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
  - -1 = 111…1

Example: values for *w* = 16 (short)

	Decimal	Hex	Binary
UMax	65 535	FF FF	11111111 11111111
TMax	32 767	7F FF	01111111 11111111
TMin	-32 768	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

- **Observations**
  - Asymmetric range
    - |TMin | = TMax + 1
  - UMax = 2 * TMax + 1

- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - INT_MIN
  - Values are platform specific

Today: Integers

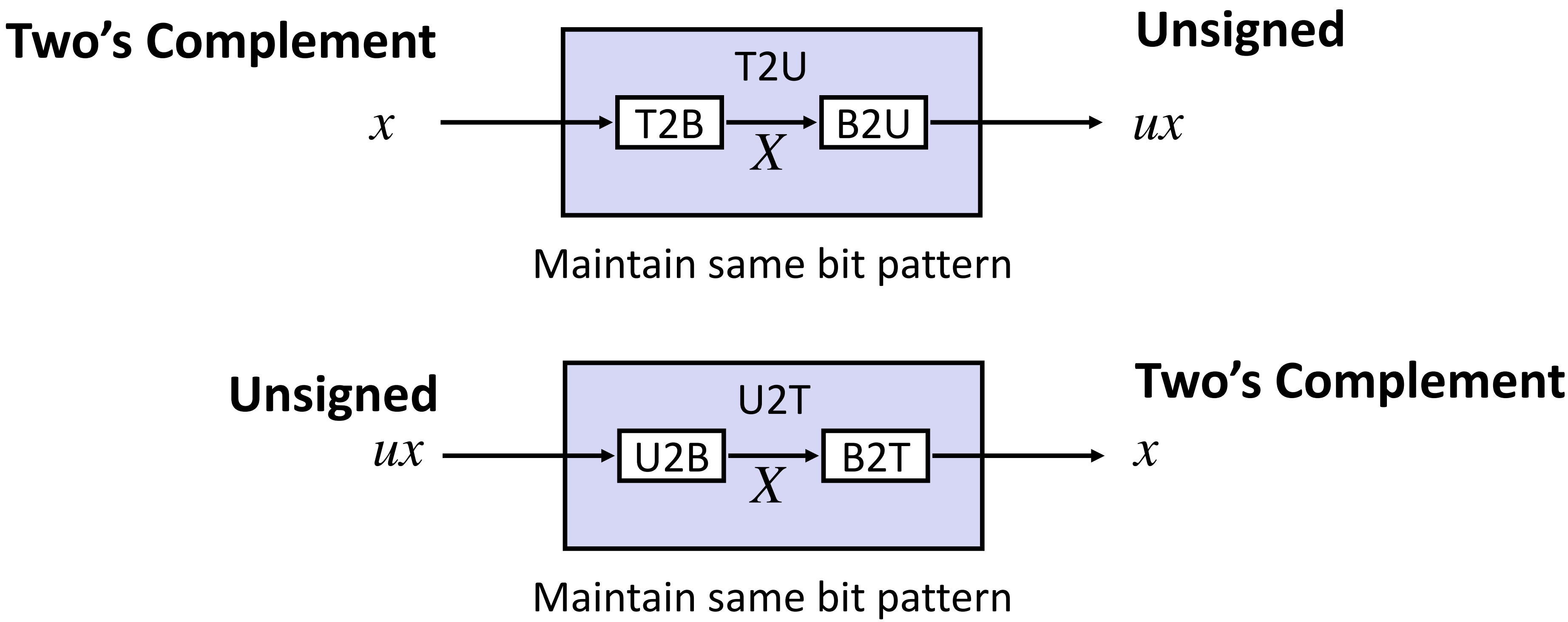
- Encoding integers
- **Conversion, casting**
- Expanding, truncating
- When should you use unsigned?
- Integer arithmetic

Unsigned & signed numeric values

X	U(*X*)	T(*X*)
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

- **Same encodings for non-negative values**
- **Every bit pattern represents an unique integer value**
- **Each representable integer has an unique bit encoding**

Mapping signed ↔ unsigned

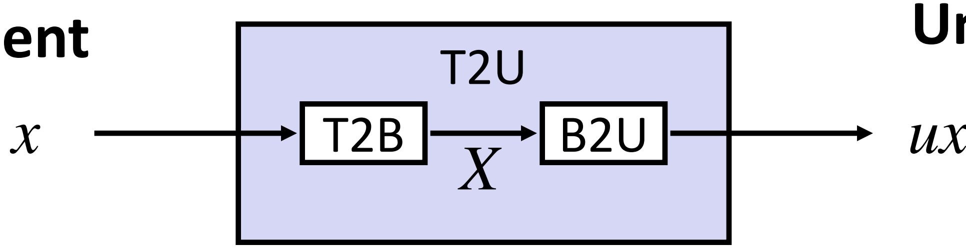


Mappings between unsigned and two's complement numbers:
Keep bit representations and reinterpret

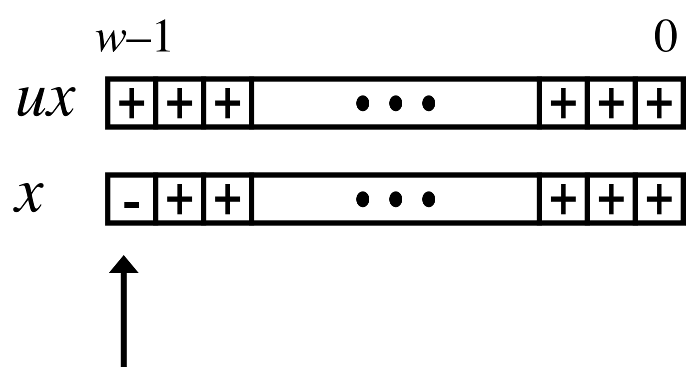
Relation between signed & unsigned

Two's Complement

Unsigned



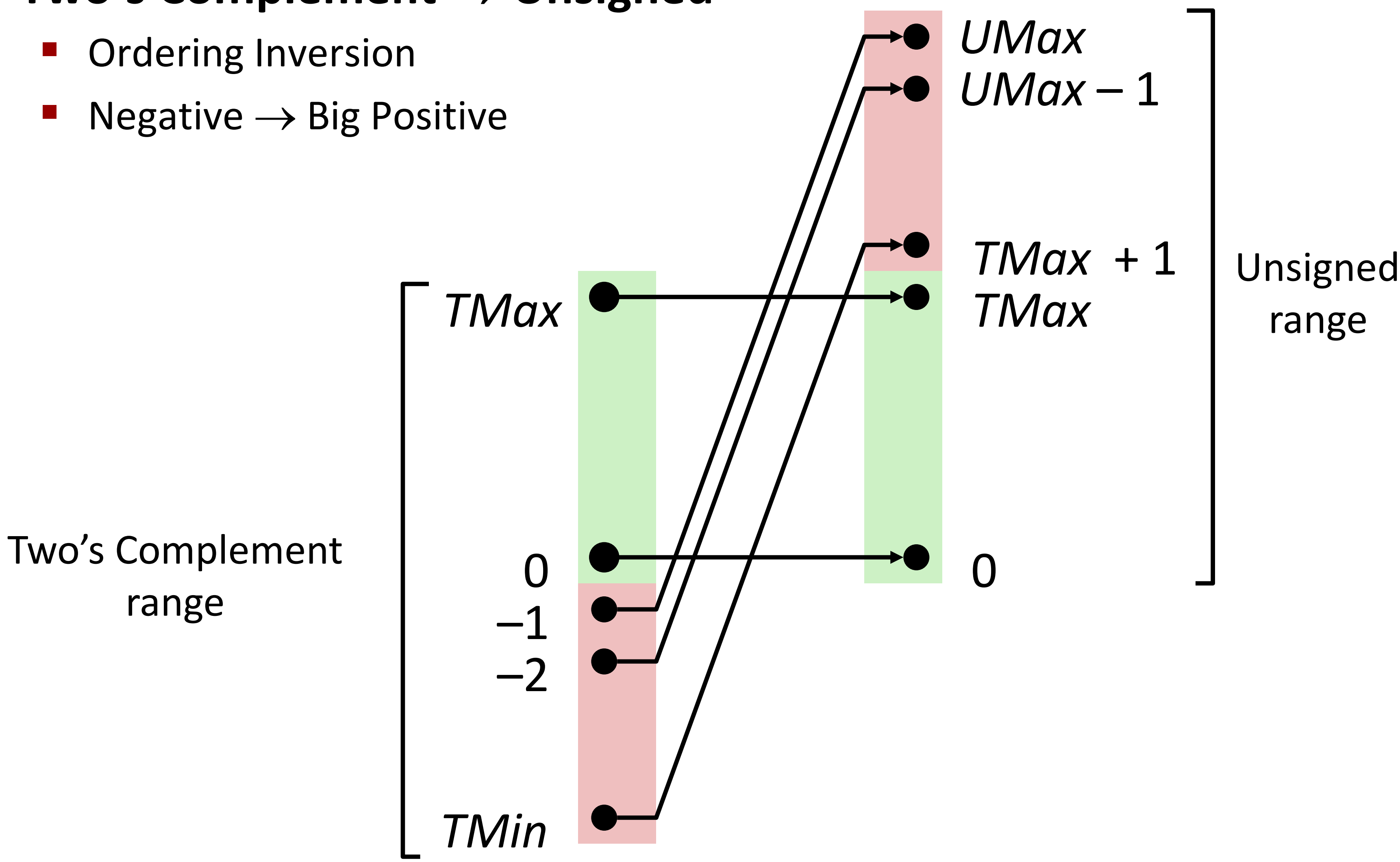
Maintain same bit pattern



Large negative weight
becomes
large positive weight

Conversion visualized

- **Two's Complement → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive



Casting: Signed vs. unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix
    - Examples: **0U, 4294967259U**

- **Explicit casting between *signed* and *unsigned***
  - Same as U2T and T2U

```
int tx, ty;
unsigned int ux, uy;

tx = (int)ux;
uy = (unsigned int)ty;
```

Casting: Signed vs. unsigned in C

- **Implicit casting**
  - Occurs via assignments and procedure calls

```
int tx, ty;
unsigned int ux, uy;

tx = ux;              /* Cast to signed */
uy = ty;              /* Cast to unsigned */
uy = fun(ty);         /* int fun(unsigned int x); */
```

- **Expression evaluation**
  - If there is a mix of unsigned and signed in a single expression, ***signed values implicitly cast to unsigned***
  - Including comparison operations **<, >, ==, <=, >=**

Casting surprises

TMin = -2147483648 TMax = 2147483647
UMax = 4294967295 UMin = 0

- Examples for *w* = 32

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483648	>	signed
2147483647U	-2147483648	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int)2147483648U	>	signed

Summary
Casting signed ↔ unsigned: Basic rules

- **Bit pattern is maintained but reinterpreted**
- **Can have unexpected effects: adding or subtracting 2^w**
- **Expression containing *signed* and *unsigned* values →**
  signed* cast to *unsigned

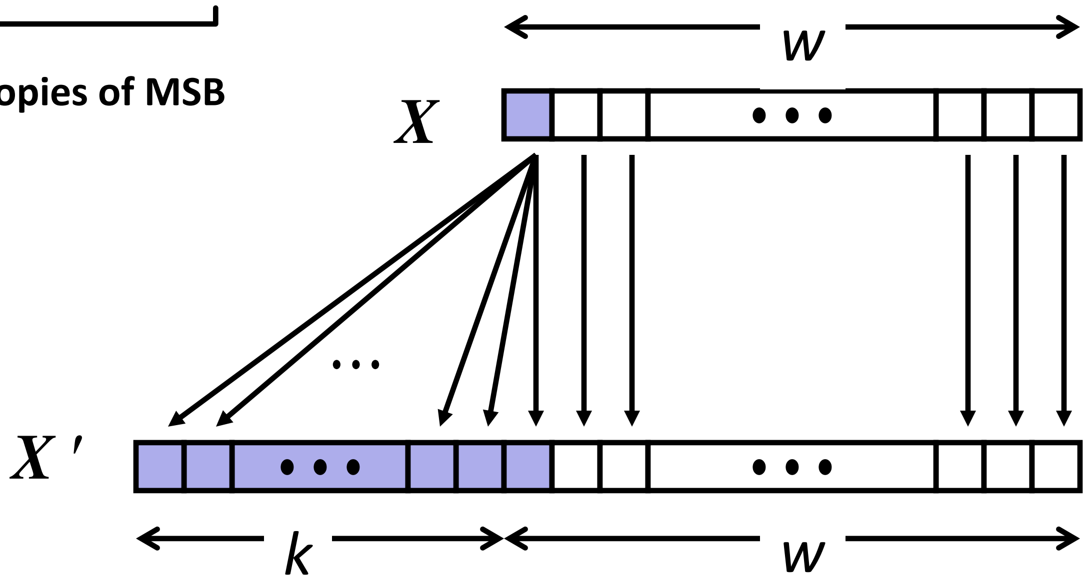
Today: Integers

- Representation: Signed and Unsigned
- Conversion, casting
- **Expanding, truncating**
- When should you use unsigned?
- Integer arithmetic

Expanding bit representation

- **Task:**
  - Given w-bit signed integer x
  - Convert it to $(w+k)$-bit integer **with same value**

- **Rule:**
  - Make k copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign extension: Simple example ($w = 5$)

Positive number

	-16	8	4	2	1
10 =	0	1	0	1	0

	-32	16	8	4	2	1
10 =	0	0	1	0	1	0

Negative number

	-16	8	4	2	1
-10 =	1	0	1	1	0

	-32	16	8	4	2	1
-10 =	1	1	0	1	1	0

Expanding bit representation: Example

```
short x =  15213;
int  ix = (int) x;  /* expand to 32-bit value */

short y = -15213;
int  iy = (int) y;  /* expand to 32-bit value */
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

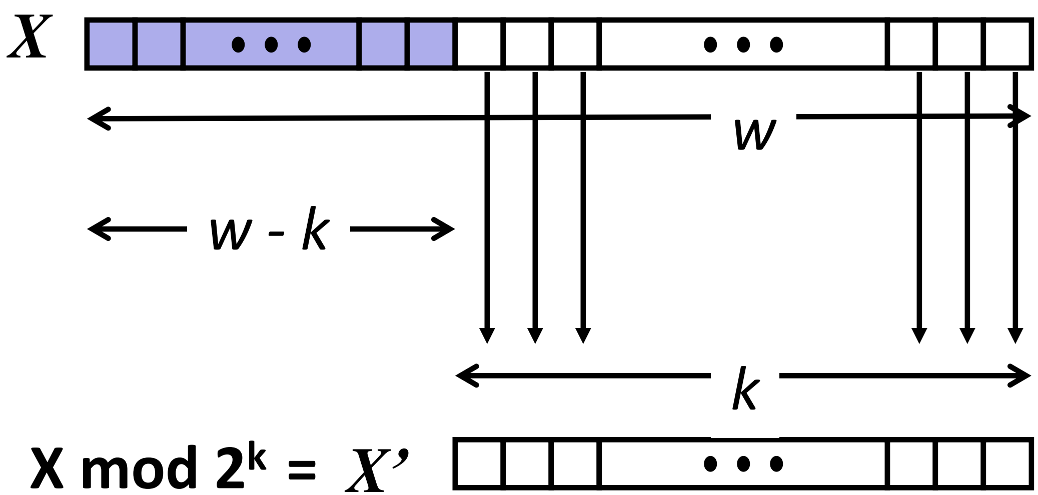
Truncating numbers

- **Task:**
  - Given w-bit signed/unsigned integer x
  - Convert it to k-bit integer ($k<w$), **possibly changing its value**

- **Rule:**
  - Drop the high-order $w-k$ bits:
  - $X' = \underbrace{x_{k-1}, x_{k-2}, \ldots, x_0}$

    ***w-k* low-order bits**

  - **Reinterpret number**



Truncation: Simple example

No sign change

	-16	8	4	2	1
2 =	0	0	0	1	0

	-8	4	2	1
2 =	0	0	1	0

	-16	8	4	2	1
-6 =	1	1	0	1	0

	-8	4	2	1
-6 =	1	0	1	0

Sign change

	-16	8	4	2	1
10 =	0	1	0	1	0

	-8	4	2	1
-6 =	1	0	1	0

	-16	8	4	2	1
-10 =	1	0	1	1	0

	-8	4	2	1
6 =	0	1	1	0

Truncating numbers: Example

```
int   x  = 53191;
short sx = (short)x;      /* truncate to 16-bit value */
int   y  = sx;            /* expand to 32-bit value */
```

	Decimal	Hex	Binary
x	53191	00 00 CF C7	00000000 00000000 11001111 11000111
sx	-12345	CF C7	11001111 11000111
y	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

Relative order of conversion

- **Consider the following code:**

```
short sx = -12345;              /* 0xcfc7 */
unsigned int uy = sx;           /* Mistery!!! */

printf("uy = %u :\t", uy);
show_bytes((unsigned char*) &uy, sizeof(unsigned int));
```

- **When run on a little-endian machine, we get:**

```
uy = 4294954951 : c7 cf ff ff
```

- **First changes the size and then from *signed* to *unsigned***
  - Otherwise, we would get `uy = 53191 : c7 cf 00 00`

Summary
Expanding, Truncating: Basic rules

- **Expanding (e.g., *short* to *int*)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- **Truncating (e.g., *unsigned int* to *unsigned short*)**
  - Unsigned/signed: bits are truncated (mod operation)
  - Result is reinterpreted
  - For small numbers yields expected behavior

Today: Integers

- Representation: Signed and Unsigned
- Conversion, casting
- Expanding, truncating
- **When should you use unsigned?**
- Integer arithmetic

When should you use unsigned?

- *Do not* use without understanding its implications

- Easy to make mistakes

```
unsigned int i;
int cnt;
. . .
for (i = cnt-2; i >= 0; i--)
 a[i] += a[i+1];
```

- Can be very subtle

```
#define DELTA sizeof(int)
int i, cnt;
. . .
for (i = cnt; i-DELTA >= 0; i-= DELTA)
   . . .
```

Counting down with unsigned

- **Proper way to use *unsigned* as loop index**

```
unsigned int i;
int cnt;

for (i = cnt-2; i < cnt; i--)
 a[i] += a[i+1];
```

- **Even better**

```
size_t i;

for (i = cnt-2; i < cnt; i--)
  a[i] += a[i+1];
```

  - Data type **`size_t`** defined as *unsigned* value with length equal to word size
  - Code will work even if **`cnt`** = *UMax*
  - What if **`cnt`** is signed and < 0?

When should you use unsigned? (cont.)

- ***Do* use when performing modular arithmetic**
  - Multi-precision arithmetic, in which numbers are represented by arrays of words

- ***Do* use when using bits to represent sets**
  - Logical right shift, no sign extension

- **Addresses are naturally unsigned, so system programmers find unsigned types to be helpful**

- **See Robert Seacord, *"Secure Coding in C and C++"***

Today: Integers

- Representation: Signed and Unsigned
- Conversion, casting
- Expanding, truncating
- When should you use unsigned?
- Integer arithmetic

Integer arithmetic

- **Many beginning programmers are surprised to find that…**
  - adding two positive numbers can yield a negative result
  - the comparison $x < y$ can yield a different result than the comparison $x - y < 0$
- **These properties are artifacts of the finite nature of computer arithmetic**
  - Integer arithmetic performed by computers is really a form of **modular arithmetic**
  - See "Integer arithmetic" document for details
- **Understanding the nuances of computer arithmetic can help programmers write more reliable code**

Arithmetic: Basic rules

- **Addition**
  - True value requires **w+1 bits** in the worst case
  - Unsigned/signed: Normal addition **followed by truncate and reinterpretation**
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^w
- **CPUs can use the same machine instruction to perform either unsigned or signed addition/subtraction**
  - Same operation on bit level

Unsigned vs Signed addition

unsigned char

```
     1110 1001        E9        223
+    1101 0101      + D5      + 213
---------------------------------
   1 1011 1110       1BE        446
---------------------------------
     1011 1110        BE        190
```

char

```
     1110 1001        E9        -23
+    1101 0101      + D5      + -43
---------------------------------
   1 1011 1110       1BE        -66
---------------------------------
     1011 1110        BE        -66
```

Arithmetic: Basic rules

- **Multiplication**
  - True value requires **2*w bits** in the worst case
  - Unsigned/signed: Normal multiplication **followed by truncate and reinterpretation**
  - Unsigned: multiplication mod 2^w
  - Signed: modified multiplication mod 2^w (result in proper range)

- **On most machines, integer multiplication and division are fairly slow**
  - Separate instructions are provided in IA32 for signed and unsigned multiplication/division
  - Replace them by constant factors with combinations of shift and addition/subtraction operations

Unsigned vs Signed multiplication

unsigned char

```
              1110 1001        E9         233
*             1101 0101     *  D5    *    213
  1100 0001 1101 1101        C1DD       49629
              1101 1101        DD         221
```

char

```
              1110 1001        E9         -23
*             1101 0101     *  D5    *    -43
  0000 0011 1101 1101        03DD         989
              1101 1101        DD         -35
```

Arithmetic: Basic rules

- **Left shift**
  - Unsigned/signed: multiplication by 2^k
  - **Always logical shift**

- **Right shift**
  - Unsigned: logical shift, div (division + round to zero) by 2^k
  - Signed: arithmetic shift
    - Positive numbers: div (division + round to zero) by 2^k
    - Negative numbers: div (division + round away from zero) by 2^k
      - Use biasing to fix

C Puzzle

- **For each of the following C expressions, either:**
  - Argue that is true for all argument values
  - Give example where not true

Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- `x < 0` ⇒ `((x*2) < 0)`
- `ux >= 0`
- `x & 7 == 7` ⇒ `(x<<30) < 0`
- `ux > -1`
- `x > y` ⇒ `-x < -y`
- `x * x >= 0`
- `x > 0 && y > 0` ⇒ `x + y > 0`
- `x >= 0` ⇒ `-x <= 0`
- `x <= 0` ⇒ `-x >= 0`

C Puzzle answers

TMin makes a good counterexample in many cases

- `x < 0` ⇒ `((x*2) < 0)` **False**: *TMin*
- `ux >= 0` **True**: 0 = *UMin*
- `x & 7 == 7` ⇒ `(x<<30) < 0` **True**: $x_1 = 1$
- `ux > -1` **False**: 0
- `x > y` ⇒ `-x < -y` **False**: `-1`, *TMin*
- `x * x >= 0` **False**: `65535` `(x*x=-131071)`
- `x > 0 && y > 0` ⇒ `x + y > 0` **False**: *TMax*, *TMax*
- `x >= 0` ⇒ `-x <= 0` **True**: $-TMax < 0$
- `x <= 0` ⇒ `-x >= 0` **False**: *TMin*

Summary

- **In C there are two different ways bits can be used to encode integers**
  - One that can only represent nonnegative numbers
  - One that can represent negative, zero, and positive numbers

- **Unsigned values can be very useful**
  - But the implicit conversion of signed to unsigned can lead to errors or vulnerabilities

- **It is possible to expand or shrink an encoded integer to fit a representation with a different length**

Summary

- **Integer arithmetic performed by computers is really a form of modular arithmetic**
  - These properties are artifacts of the finite nature of computer arithmetic