Bits, Bytes, and Integers
August 27, 2009

**Topics**
- Representing information as bits
- Bit-level manipulations
- Boolean algebra
- Expressing in C
- Representations of Integers
- Basic properties and operations
- Implications for C

**Binary Representations**

**Base 2 Number Representation**
- Represent 15213\(_{10}\) as 11101101101101\(_2\)
- Represent 1.2010 as 1.0011001100110011\[_{2}\]...
- Represent 1.5213 \(\times 10^4\) as 1.110110110101\(_2\) \(\times 2^{13}\)

**Electronic Implementation**
- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires

**Encoding Byte Values**

Byte = 8 bits
- Binary 00000000\(_2\) to 11111111\(_2\)
- Decimal: 0\(_{10}\) to 255\(_{10}\)
- Hexadecimal 00\(_{16}\) to FF\(_{16}\)
- Base 16 number representation
- Use characters '0' to '9' and 'A' to 'F'
- Write FA1D37B\(_{16}\) in C as 0xFA1D37B
- Or 0xfa1d37b

**General Boolean Algebras**

Operate on Bit Vectors
- Operations applied bitwise
  01010101 & 01010101 = 01010101
  01010101 | 01010101 = 01010101
  01010101 ^ 01010101 = 01010101

**Back to bits: Boolean Algebra**

Developed by George Boole in 19th Century
- Algebraic representation of logic
- Encode "True" as 1 and "False" as 0
- Developed by George Boole in 19th Century
- Algebraic representation of logic
- Encode "True" as 1 and "False" as 0
- And
- Or
- Not
- Exclusive-Or (Xor)

**Bit-Level Operations in C**

Operations &, |, ~, ^ Available in C
- Apply to any "integral" data type
- long, int, short, char, unsigned
- View arguments as bit vectors
- Arguments applied bit-wise

**Examples (Char data type)**
- ~0x41 --> 0xBE
- ~0x00 --> 0xFF
- 0x69 & 0x55 --> 0x41
- 0x69 | 0x55 --> 0x7D
- 0x11010101 | 0x10101010 --> 0x11111011

All of the Properties of Boolean Algebra Apply
Contrast: Logic Operations in C

Contrast to Logical Operators

- `&&`, `||`, `!`
- View 0 as "False"
- Anything nonzero as "True"
- Always return 0 or 1
- Early termination

Examples (char data type)

- `!0x41  -->  0x00`
- `!0x00  -->  0x01`
- `!!0x41  -->  0x01`
- `0x69 && 0x55  -->  0x01`
- `0x69 || 0x55  -->  0x01`
- `p && *p  (avoids null pointer access)`

Watch out for `&&` vs. `&` (and `||` vs. `|`)… one of the more common oopsies in C programming

Shift Operations

Left Shift: `x << y`
- Shift bit-vector `x` left `y` positions
- Throw away extra bits on left
- Fill with 0's on right

Right Shift: `x >> y`
- Shift bit-vector `x` right `y` positions
- Throw away extra bits on right
- Logical shift: Fill with 0's on left
- Arithmetic shift: Replicate most significant bit on right

Memory organization

Programs refer to data by address
- address space viewed as a large array of bytes
- an address is like an index into that array

Any given computer has a "Word Size"
- nominal size of integer-valued data
- and, usually, of addresses
- 32 bits is still most common
- though 64 bits is emerging

Data Representations

Sizes of C Objects (in Bytes)

- C Data Type Typical 32-bit Intel IA32 x86-64
  - unsigned [int] 4 4 4
  - int 4 4 4
  - long int 4 4 4
  - char 1 1 1
  - short 2 2 2
  - float 4 4 4
  - double 8 8 8
  - long double 10/12 10/12
  - char * 4 4 8

- Or any other pointer

Encoding Integers

Unsigned

<table>
<thead>
<tr>
<th>X</th>
<th>R2I(X) = (\sum_{j=0}^{w-1} x_j 2^j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(-15213)</td>
</tr>
<tr>
<td>1</td>
<td>(-15213)</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td>64</td>
<td>7</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
</tr>
<tr>
<td>256</td>
<td>9</td>
</tr>
<tr>
<td>512</td>
<td>10</td>
</tr>
<tr>
<td>1024</td>
<td>11</td>
</tr>
<tr>
<td>2048</td>
<td>12</td>
</tr>
<tr>
<td>4096</td>
<td>13</td>
</tr>
<tr>
<td>8192</td>
<td>14</td>
</tr>
<tr>
<td>16384</td>
<td>15</td>
</tr>
<tr>
<td>(-32768)</td>
<td>16</td>
</tr>
</tbody>
</table>

Binary

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>00000001</td>
<td>00000001</td>
</tr>
<tr>
<td>5</td>
<td>00000101</td>
<td>00000101</td>
</tr>
<tr>
<td>255</td>
<td>01111111</td>
<td>01111111</td>
</tr>
<tr>
<td>511</td>
<td>11111111</td>
<td>11111111</td>
</tr>
<tr>
<td>1023</td>
<td>11111111</td>
<td>11111111</td>
</tr>
<tr>
<td>2047</td>
<td>11111111</td>
<td>11111111</td>
</tr>
<tr>
<td>4095</td>
<td>11111111</td>
<td>11111111</td>
</tr>
<tr>
<td>8191</td>
<td>11111111</td>
<td>11111111</td>
</tr>
<tr>
<td>16383</td>
<td>11111111</td>
<td>11111111</td>
</tr>
<tr>
<td>(-32768)</td>
<td>11111111</td>
<td>11111111</td>
</tr>
</tbody>
</table>

Sum

| \(-15213\) | \(-15213\) |

Encoding Example (Cont.)
### Numeric Ranges

- **Unsigned Values**
  - $U_{\text{Min}} = 0$
  - $U_{\text{Max}} = 2^w - 1$
- **Two's Complement Values**
  - $T_{\text{Min}} = -2^{w-1}$
  - $T_{\text{Max}} = 2^{w-1} - 1$

#### Values for $W = 16$

<table>
<thead>
<tr>
<th>Value</th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{Max}}$</td>
<td>65535</td>
<td>FF FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Max}}$</td>
<td>32767</td>
<td>FF FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Min}}$</td>
<td>-32768</td>
<td>80 00 00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>$U_{\text{Min}}$</td>
<td>0</td>
<td>00 00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

### Signed vs. unsigned ints in C

- **Constants**
  - By default, considered to be signed integers
  - Unsigned if have “U” as suffix
- **Casting**
  - Can explicitly cast between signed & unsigned
    - `int tx, ty;`
    - `unsigned ux, uy;`
    - `tx = (int) ux;`
    - `uy = (unsigned) ty;`
  - Implicit casting also occurs via assignments (and function calls)
    - `tx = ux;`
    - `uy = ty;`

### Casting Surprises

#### Expression Evaluation
- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations `<`, `>`, `<=`, `>=`

### Visual of casting

- **2's Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive

### Sign Extension

- **Task:**
  - Given $w$-bit signed integer $x$
  - Convert it to $w+k$-bit integer with same value
- **Rule:**
  - Make $k$ copies of sign bit:
    - $X' = x_{w-1}, x_0, x_{w-1}, x_{w-2}, ..., x_0$
  - $k$ copies of MSB

### Sign Extension Example

- Converting from smaller to larger integer data type
- C automatically performs sign extension
Unsigned Addition

Operands: w bits
True Sum: w+1 bits
Discard Carry: w bits

Standard Addition Function
- Ignores carry output
- Implements Modular Arithmetic

\[ s = \text{UAdd}(u, v) = u + v \mod 2^w \]

Visualizing what should happen

Integer Addition
- As in, actual math, not computer math
- 4-bit integers \( u, v \)
- Compute true sum \( \text{Add}(u, v) \)
- Values increase linearly with \( u \) and \( v \)
- Forms planar surface

Two's Complement Addition

TAdd and UAdd have Identical Bit-Level Behavior
- Signed vs. unsigned addition in C:
  \[
  \text{int } s, t, u, v;
  s = (\text{int}) ((\text{unsigned}) u + (\text{unsigned}) v);
  t = u + v
  \]
  Will give \( s == t \)

Characterizing TAdd

Functionality
- True sum requires \( w+1 \) bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer

\[ \text{TAdd}(u, v) = \begin{cases} 
  s + 2^{-w} & u + v < \text{TMIn}_w \quad (\text{NegOver}) \\
  s & \text{TMIn}_w \leq u + v \leq \text{TMax}_w \\
  s - 2^{w+1} & u + v < -2^{w-1} \quad (\text{PosOver}) 
\end{cases} \]
**Unsigned Multiplication in C**

Operands: w bits
- True Product: 2^w bits
- Discard w bits: w bits

Standard Multiplication Function
- Ignores high order w bits
- Implements Modular Arithmetic

True Product: 2^w bits

---

**Signed Multiplication in C**

Operands: w bits
- True Product: 2^w bits
- Discard w bits: w bits

Standard Multiplication Function
- Ignores high order w bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same

---

**Power-of-2 Multiply with Shift**

Operation
- u << k gives u \times 2^k
- Both signed and unsigned

Operands: w bits
- True Product: w bits
- Discard w bits: w bits

Examples
- u << 3
- u << 5 - u << 3
- Most machines shift and add faster than multiply
- Compiler generates this code automatically

---

**Signed Power-of-2 Divide with Shift**

Quotient of Signed by Power of 2
- u >> k gives [u / 2^k]
- Uses arithmetic shift

Quotient of Unsigned by Power of 2
- Correct Power-of-2 Divide

Division Computed Hex Binary

---

**Correct Power-of-2 Divide**

Quotient of Negative Number by Power of 2
- Want [u / 2^k] (Round Toward 0)
- Compute as (u + 1<<(k-1)) / 2^k
- In C: (x + (1<<k)-1) >> k
- Biases dividend toward 0

Case 1: No rounding

Dividend: [u / 2^k]

---

**Operands**

- u << 8
- y >> 8
- y >> 4
- y >> 1
- x >> k

**Examples**

- 59.4257813
- 950.8125
- 7606.5
- 15213
- 00000000

---
Representing Strings

Strings in C
- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character “0” has code 0x30
  - Digit / has code 0x30+/4
- String should be null-terminated
  - Final character = 0

Representing Strings

char S[6] = "15213";

What’s next

Recitation on Monday
- autolab, fish machines, datalab
- get started on lab #1 before then...

Floating point (Tues): representations and arithmetic

Word-Oriented Memory Organization

Addresses Specify Byte Locations
- Address of first byte in word
- Addresses of successive words differ by word size
  - e.g., 4 (32-bit) or 8 (64-bit)

Examining Data Representations

Code to Print Byte Representation of Data

typedef unsigned char *pointer;
void show_bytes(pointer *pointer, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("0x%02x\n", *(pointer+i));
}

Print directives:
- %p: Print pointer
- %x: Print Hexadecimal
**show_bytes Execution Example**

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

```
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffcba 0x00
0x11ffffcbb 0x00
```

**Representing Integers**

```c
int A = 15213;
int B = --15213;
long long int C = 15213;
```

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3B 6D

**Decimal:** 15213

**Binary:** 0011 1011 0110 1101

**Hex:** 3B 6D

**IA32, x86-64**

```
A
C
```

**Sun**

```
A
C
```

Two's complement representation (Covered later)

**Representing Pointers**

```c
int B = -15213;
int *P = &B;
```

**Deciphering Numbers**

- Value:
  - 0x12ab
- Pad to 4 bytes:
  - 0x000012ab
- Split into bytes:
  - 00 00 12 ab
- Reverse:
  - ab 12 00 00

**Application of Boolean Algebra**

Applied to Digital Systems by Claude Shannon

1937 MIT Master's Thesis

- Reason about networks of relay switches
  - Encode closed switch as 1, open switch as 0

```
A&B

Connection when

A&B | ~A&B

= A*B
```

**Integer C Puzzles**

- Assume 32-bit word size, two's complement integers
- For each of the following C expressions, either:
  - Argue that is true for all argument values
  - Give example where not true

- \( x < 0 \) \( \Rightarrow \) \([x*2] < 0\)
- \( x > 0 \) \( \Rightarrow \) \([x<<30] < 0\)
- \( x > -1 \)
- \( x <= 0 \)
- \( x >> 3 == x/8 \)
- \( x & (x-1) != 0 \)
Values for Different Word Sizes

<table>
<thead>
<tr>
<th>W</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

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

C Programming
- Include <limits.h>
- K&R App. B11
- Declares constants, e.g.,
  - ULONG_MAX
  - LONG_MAX
  - LONG_MIN
- Values platform-specific

Unsigned & Signed Numeric Values

<table>
<thead>
<tr>
<th>X</th>
<th>B2U(X)</th>
<th>B2T(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Equivalence
- Same encodings for nonnegative values

Uniqueness
- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding

⇒ Can Invert Mappings
- U2B(x) | = | B2U(-1(x))
- T2B(x) | = | B2T(-1(x))

Relation between Signed & Unsigned

Two’s Complement

x
T2U
T2B

Unsigned

Maintain Same Bit Pattern

w = 1
x
w
w

x

| 1 | 1 | ··· | 1 | 1 |
| 1 | 1 | ··· | 1 | 0 |

Large negative weight
Large positive weight

When should I use unsigned?

Don’t Use Just Because Number Nonzero
- Easy to make mistakes
  unsigned i;
  for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
- Can be very subtle
  #define DELTA sizeof(int)
  int i;
  for (i = CNT; i-DELTA >= 0; i-= DELTA)
    . . .

Do Use When Performing Modular Arithmetic
- Multiprecision arithmetic
- Working right up to limit of word size

Negating with Complement & Increment

Claim: Following Holds for 2’s Complement

-x + 1 = -x

Complement
- Observation: -x + x = 1111 11.. = -1

x

Increment
- -x + x + 1 = -x + x
- -x + 1 = -x

Warning: Be cautious treating int’s as integers

Comp. & Incr. Examples

x = 15213

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>1011110111011011</td>
</tr>
<tr>
<td>x + 1</td>
<td>15214</td>
<td>1011110111011100</td>
</tr>
<tr>
<td>x + 2</td>
<td>15215</td>
<td>1011110111011101</td>
</tr>
<tr>
<td>x + 3</td>
<td>15216</td>
<td>1011110111011110</td>
</tr>
</tbody>
</table>

0

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>-1</td>
<td>FF FFE</td>
<td>11111111 11111111 11111111</td>
</tr>
<tr>
<td>-2</td>
<td>FF FE</td>
<td>11111111 11111111 11111110</td>
</tr>
<tr>
<td>-3</td>
<td>FF FD</td>
<td>11111111 11111111 11111101</td>
</tr>
<tr>
<td>-4</td>
<td>FF FC</td>
<td>11111111 11111111 11111100</td>
</tr>
<tr>
<td>-5</td>
<td>FF FB</td>
<td>11111111 11111111 11111011</td>
</tr>
<tr>
<td>-6</td>
<td>FF FA</td>
<td>11111111 11111111 11111010</td>
</tr>
<tr>
<td>-7</td>
<td>FF F9</td>
<td>11111111 11111111 11111001</td>
</tr>
<tr>
<td>-8</td>
<td>FF F8</td>
<td>11111111 11111111 11111000</td>
</tr>
<tr>
<td>-9</td>
<td>FF F7</td>
<td>11111111 11111111 11110111</td>
</tr>
<tr>
<td>-10</td>
<td>FF F6</td>
<td>11111111 11111111 11110110</td>
</tr>
<tr>
<td>-11</td>
<td>FF F5</td>
<td>11111111 11111111 11110101</td>
</tr>
<tr>
<td>-12</td>
<td>FF F4</td>
<td>11111111 11111111 11110100</td>
</tr>
<tr>
<td>-13</td>
<td>FF F3</td>
<td>11111111 11111111 11110011</td>
</tr>
<tr>
<td>-14</td>
<td>FF F2</td>
<td>11111111 11111111 11110010</td>
</tr>
<tr>
<td>-15</td>
<td>FF F1</td>
<td>11111111 11111111 11110001</td>
</tr>
</tbody>
</table>
Mathematical Properties

Modular Addition Forms an Abelian Group
- Closed under addition
  \[ 0 \leq \text{UAdd}(u, v) \leq 2^w - 1 \]
- Commutative
  \[ \text{UAdd}(u, v) = \text{UAdd}(v, u) \]
- Associative
  \[ \text{UAdd}(\text{UAdd}(u, v), w) = \text{UAdd}(u, \text{UAdd}(v, w)) \]
- 0 is additive identity
  \[ \text{UAdd}(u, 0) = u \]
- Every element has additive inverse
  - Let \( \text{UComp}(u) = 2^w - u \)
  - \( \text{UAdd}(u, \text{UComp}(u)) = 0 \)

Mathematical Properties of TAdd

Isomorphic Algebra to UAdd
- \( \text{TAdd}(u, v) = \text{U2T}(\text{UAdd}(T2U(u), T2U(v))) \)
- Since both have identical bit patterns

Two's Complement Under TAdd Forms a Group
- Closed, Commutative, Associative, 0 is additive identity
- Every element has additive inverse
  \[ \text{TComp}(u) = \begin{cases} -u & u \neq \text{TMin} \setminus \text{TMin} \\ u = \text{TMin} \setminus \text{TMin} \end{cases} \]

Multiplication

Computing Exact Product of \( w \)-bit numbers \( x, y \)
- Either signed or unsigned

Ranges
- Unsigned: \( 0 \leq \text{min}(x, y) \leq (2^w - 1)^2 \leq 2^{2w} - 2^{w+1} + 1 \)
  - Up to 2\( w \) bits
- Two's complement min: \( \text{min}(x, y) \leq (-2^{w-1})(2^w - 1) = -2^{2w-3} + 2^{w-1} \)
  - Up to 2\( w \) bits
- Two's complement max: \( \text{max}(x, y) \leq (-2^{w-1})^2 = 2^{2w-2} \)
  - Up to 2\( w \) bits, but only for \( \text{TMin} \)

Maintaining Exact Results
- Would need to keep expanding word size with each product computed
- Done in software by “arbitrary precision” arithmetic packages

Compiled Multiplication Code

C Function
```c
int mul12(int x)
{
    return x*12;
}
```

Compiled Arithmetic Operations
```c
leal (%eax,%eax,2), %eax
sall $2, %eax
```

Explanation
- C compiler automatically generates shift/add code when multiplying by constant

Compiled Unsigned Division Code

C Function
```c
unsigned udiv8(unsigned x)
{
    return x/8;
}
```

Compiled Arithmetic Operations
```c
shrl $3, %eax
```

# Logical shift
- Return \( x >> 3 \)
- Uses logical shift for unsigned

For Java Users
- Logical shift written as >>>

Compiled Signed Division Code

C Function
```c
int idiv8(int x)
{
    return x/8;
}
```

Compiled Arithmetic Operations
```c
testl %eax, %eax
js L4
```

# Arithmetic shift
- Return \( x >> 3 \)
- Uses arithmetic shift for int

For Java Users
- Arith. shift written as >>
Properties of Unsigned Arithmetic

Unsigned Multiplication with Addition Forms
- Commutative Ring
  - Addition is commutative group
  - Closed under multiplication
  - Multiplication Commutative
    \[ \text{UMult}_w(u, v) = \text{UMult}_w(v, u) \]
  - Multiplication is Associative
    \[ \text{UMult}_w(t, (\text{UMult}_w(u, v))) = (\text{UMult}_w(t, u)), \text{UMult}_w(t, v)) \]
  - 1 is multiplicative identity
    \[ \text{UMult}_w(u, 1) = u \]
  - Multiplication distributes over addition
    \[ \text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v)) \]

Properties of Two’s Comp. Arithmetic

Isomorphic Algebras
- Unsigned multiplication and addition
- Two’s complement multiplication and addition

Both Form Rings
- Isomorphic to ring of integers mod \(2^w\)

Comparison to Integer Arithmetic
- Both are rings
- Integers obey ordering properties, e.g.,
  \[ u > 0 \Rightarrow u + v > 0 \]
  \[ u > 0, v > 0 \Rightarrow u \cdot v > 0 \]
- These properties are not obeyed by two’s comp. arithmetic

Integer C Puzzles Revisited

- \(x < 0 \Rightarrow ((x*2) < 0)\)
- \(ux >> 0\)
- \(x & 7 == 7 \Rightarrow (x<<30) < 0\)
- \(ux > -1\)
- \(x > y \Rightarrow -x < -y\)
- \(x + x >> 0\)
- \(x > 0 \Rightarrow y > 0 \Rightarrow x + y > 0\)
- \(x >> 0\)
- \(x <= 0 \Rightarrow -x >= 0\)
- \(x >> 3 == x/8\)
- \(x & (x-1) != 0\)

Initialization

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