Simple Data Types in C

Alan L. Cox
alc@rice.edu
Objectives

Be able to explain to others what a data type is

Be able to use basic data types in C programs

Be able to see the inaccuracies and limitations introduced by machine representations of numbers
What do you see?
What do you see?
Last one
Everything is Just a Bunch of Bits

Bits can represent many different things
  • Depends on interpretation

You and your program must keep track of what kind of data is at each location in the computer’s memory
  • E.g., program data types
Big Picture

Processor works with finite-sized data
All data implemented as a sequence of bits
  • Bit = 0 or 1
  • Represents the level of an electrical charge

\[\text{Byte} = 8 \text{ bits}\]

\[\text{Word} = \text{largest data size handled by processor}\]
  • 32 bits on most older computers
  • 64 bits on most new computers
Data types in C

Only really four basic types:

- char
- int (short, long, long long, unsigned)
- float
- double

Size of these types on CLEAR machines:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
</tbody>
</table>

Sizes of these types vary from one machine to another!
Characters (char)

Roman alphabet, punctuation, digits, and other symbols:

- Encoded within one byte (256 possible symbols)
- ASCII encoding (man ascii for details)

In C:

```c
char a_char = 'a';
char newline_char = '\n';
char tab_char = '\t';
char backslash_char = '\\';
```
### ASCII

Special control characters

<table>
<thead>
<tr>
<th>0 NUL</th>
<th>1 SOH</th>
<th>2 STX</th>
<th>3 ETX</th>
<th>4 EOT</th>
<th>5 ENQ</th>
<th>6 ACK</th>
<th>7 BEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 BS</td>
<td>9 HT</td>
<td>10 NL</td>
<td>11 VT</td>
<td>12 NP</td>
<td>13 CR</td>
<td>14 SO</td>
<td>15 SI</td>
</tr>
<tr>
<td>16 DLE</td>
<td>17 DC1</td>
<td>18 DC2</td>
<td>19 DC3</td>
<td>20 DC4</td>
<td>21 NAK</td>
<td>22 SYN</td>
<td>23 ETB</td>
</tr>
<tr>
<td>24 CAN</td>
<td>25 EM</td>
<td>26 SUB</td>
<td>27 ESC</td>
<td>28 FS</td>
<td>29 GS</td>
<td>30 RS</td>
<td>31 US</td>
</tr>
<tr>
<td>32 SP</td>
<td>33 !</td>
<td>34 &quot;</td>
<td>35 #</td>
<td>36 $</td>
<td>37 %</td>
<td>38 &amp;</td>
<td>39 '</td>
</tr>
<tr>
<td>40 (</td>
<td>41 )</td>
<td>42 *</td>
<td>43 +</td>
<td>44 ,</td>
<td>45 -</td>
<td>46 .</td>
<td>47 /</td>
</tr>
<tr>
<td>48 0</td>
<td>49 1</td>
<td>50 2</td>
<td>51 3</td>
<td>52 4</td>
<td>53 5</td>
<td>54 6</td>
<td>55 7</td>
</tr>
<tr>
<td>56 8</td>
<td>57 9</td>
<td>58 :</td>
<td>59 ;</td>
<td>60 &lt;</td>
<td>61 =</td>
<td>62 &gt;</td>
<td>63 ?</td>
</tr>
<tr>
<td>64 @</td>
<td>65 A</td>
<td>66 B</td>
<td>67 C</td>
<td>68 D</td>
<td>69 E</td>
<td>70 F</td>
<td>71 G</td>
</tr>
<tr>
<td>72 H</td>
<td>73 I</td>
<td>74 J</td>
<td>75 K</td>
<td>76 L</td>
<td>77 M</td>
<td>78 N</td>
<td>79 O</td>
</tr>
<tr>
<td>80 P</td>
<td>81 Q</td>
<td>82 R</td>
<td>83 S</td>
<td>84 T</td>
<td>85 U</td>
<td>86 V</td>
<td>87 W</td>
</tr>
<tr>
<td>88 X</td>
<td>89 Y</td>
<td>90 Z</td>
<td>91 [</td>
<td>92 \</td>
<td>93 ]</td>
<td>94 ^</td>
<td>95 _</td>
</tr>
<tr>
<td>96 `</td>
<td>97 a</td>
<td>98 b</td>
<td>99 c</td>
<td>100 d</td>
<td>101 e</td>
<td>102 f</td>
<td>103 g</td>
</tr>
<tr>
<td>104 h</td>
<td>105 i</td>
<td>106 j</td>
<td>107 k</td>
<td>108 l</td>
<td>109 m</td>
<td>110 n</td>
<td>111 o</td>
</tr>
<tr>
<td>112 p</td>
<td>113 q</td>
<td>114 r</td>
<td>115 s</td>
<td>116 t</td>
<td>117 u</td>
<td>118 v</td>
<td>119 w</td>
</tr>
<tr>
<td>120 x</td>
<td>121 y</td>
<td>122 z</td>
<td>123 {</td>
<td>124</td>
<td>125 }</td>
<td>126 ~</td>
<td>127 DEL</td>
</tr>
</tbody>
</table>
Characters are just numbers

What does this function do?

```c
char fun(char c)
{
    char new_c;
    if ((c >= 'A') && (c <= 'Z'))
        new_c = c - 'A' + 'a';
    else
        new_c = c;
    return (new_c);
}
```
Fundamental problem:
  • Fixed-size representation can’t encode all numbers

Standard low-level solution:
  • Limit number range and precision
    • Usually sufficient
    • Potential source of bugs

Signed and unsigned variants
  • unsigned modifier can be used with any sized integer (short, long, or long long)
Signed Integer

1...2...
...1,306...1,307...
...32,767...-32,768...
...-32,767...-32,766...

BAAA
BAAA
BAAA BAAA
BAAA
Integer Representations

Why one more negative than positive?
Math for $n$ bits:

Define $x = x_{n-1} \cdots x_0$

$$B2U(x) = \sum_{i=0}^{n-1} 2^i x_i$$

$$B2T(x) = -2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i$$

sign bit

0=non-negative

1=negative
**Integer Ranges**

**Unsigned**

\[ \text{UMin}_n \ldots \text{UMax}_n = 0 \ldots 2^n - 1: \]

<table>
<thead>
<tr>
<th>Bits</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>

\[ \text{unsigned int} \]

<table>
<thead>
<tr>
<th>Bits</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
</tbody>
</table>

\[ \text{int} \]

\[ \text{unsigned long int} \]

**2’s Complement**

\[ \text{TMin}_n \ldots \text{TMax}_n = -2^{n-1} \ldots 2^{n-1}-1: \]

- **Note:** C numeric ranges are platform dependent!

`#include <limits.h>` **to define** ULONG_MAX, UINT_MIN, INT_MAX, ...
Detecting Overflow in Programs

Some high-level languages (ML, Ada, ...):

- Overflow causes exception that can be handled

C:

- Overflow causes no special event
- Programmer must check, if desired

E.g., given a, b, and c = UAdd\(_n\)(a, b) – overflow:

Claim: Overflow iff c < a (Or similarly, iff c < b)

Proof: Know 0 \leq b < 2^n
If no overflow, \[ c = (a + b) \mod 2^n = a + b \geq a + 0 = a \]
If overflow, \[ c = (a + b) \mod 2^n = a + b - 2^n < a \]
Overflow

 unsigned int x = 2123456789U;
 unsigned int y = 3123456789U;
 unsigned int z;

 z = x + y;

 z is 951,946,282 not 5,246,913,578 as expected

 int x = 2123456789;
 int y = 3123456789;
 int z;

 z = x + y;

 y is not a valid positive number (sign bit is set)!
 It’s -1,171,510,507

 z is still 951,946,282
However, ...

#include <assert.h>

void procedure(int x) {
...
    assert(x + 10 > x);

Should this assertion ever fail?

Do a web search for “GCC bug 30475”.

The C language definition does not assume 2’s complement as the underlying implementation.

The behavior of signed integer overflow is undefined.
You say, “What should I do?”

```
#include <limits.h>

/* Safe, signed add. Won’t overflow. */
int
safe_add(int x, int y)
{
    if (y < 0)
        return (safe_sub(x, -y));
    if (INT_MAX - y < x)
        return (INT_MAX); /* Don’t overflow! */
    return (x + y);
}
```
Bit Shifting as Multiplication

*Shift left* \((x \ll 1)\) multiplies by 2:

\[
\begin{align*}
0 & 0 & 1 & 1 & = 3 \\
0 & 1 & 1 & 0 & = 6 \\
1 & 1 & 0 & 1 & = -3 \\
1 & 0 & 1 & 0 & = -6
\end{align*}
\]

Works for unsigned, 2’s complement

Can overflow

In decimal, same idea multiplies by 10: e.g., 42 \(\rightarrow\) 420
Bit Shifting as Division

**Logical shift right** \((x \gg 1)\) divides by 2 for unsigned:

\[
\begin{align*}
01111 & = 7 \\
0011 & = 3
\end{align*}
\]

Always rounds down!

\[
\begin{align*}
1001 & = 9 \\
0100 & = 4
\end{align*}
\]

**Arithmetic shift right** \((x \gg 1)\) divides by 2 for 2’s complement:

\[
\begin{align*}
0111 & = 7 \\
0011 & = 3 \\
1100 & = -4
\end{align*}
\]

Always rounds down!
Bit Shifting for Multiplication/Division

Why useful?

- Simpler, thus faster, than general multiplication & division
- Standard compiler optimization

Can shift multiple positions at once:

- Multiplies or divides by corresponding power-of-2
- a << 5  a >> 5
A Sampling of Integer Properties

For both unsigned & 2’s complement:

Mostly as usual, e.g.:
- 0 is identity for +, -
- 1 is identity for \( \times, \div \)
- +, -, \( \times \) are associative
- +, \( \times \) are commutative
- \( \times \) distributes over +, -

Some surprises, e.g.:
- \( \div \) doesn’t distribute over +, -
- \( \neg (a,b > 0 \implies a + b > a) \)

Why should you care?
- Programmer should be aware of behavior of their programs
- Compiler uses such properties in optimizations
Beware of Sign Conversions in C

Beware implicit or explicit conversions between unsigned and signed representations!

One of many common mistakes:

```c
unsigned int u;
...
if (u > -1) ... ?
```

Always false(!) because -1 is converted to unsigned, yielding UMax_n

What’s wrong? ?
Non-Integral Numbers: How?

Fixed-size representations
- Rational numbers (i.e., pairs of integers)
- Fixed-point (use integer, remember where point is)
- Floating-point (scientific notation)

Variable-size representations
- Sums of fractions (e.g., Taylor-series)
- Unbounded-length series of digits/bits
Floating-point

Binary version of scientific notation

1.001101110 × 2^5 = 100110.1110
= 32 + 4 + 2 + 1/2 + 1/4 + 1/8
= 38.875

-1.011 × 2^{-3} = -.001011
= - (1/8 + 1/32 + 1/64)
= -.171875

binary point
FP Overflow & Underflow

Fixed-sized representation leads to limitations

Large positive exponent.
Unlike integer arithmetic, overflow → imprecise result ($\pm\infty$), not inaccurate result.

Round to $-\infty$  
Round to $+\infty$

Zero

Negative Overflow  Expressible  Negative Underflow  Positive Overflow  Expressible  Positive Overflow

Large negative exponent
Round to zero
FP Representation

$1.001101110 \times 2^5$

**Fixed-size representation**

- Using more significand bits $\Rightarrow$ increased precision
- Using more exponent bits $\Rightarrow$ increased range

*Typically, fixed # of bits for each part, for simplicity*
FP Representation: IEEE 754

Current standard version of floating-point

Single-precision (float)
- One word: 1 sign bit, 23 bit fraction, 8 bit exponent
- Positive range: $1.17549435 \times 10^{-38} \ldots 3.40282347 \times 10^{+38}$

Double-precision (double)
- Two words: 1 sign bit, 52 bit fraction, 11 bit exponent
- Positive range: $2.2250738585072014 \times 10^{-308} \ldots 1.7976931348623157 \times 10^{+308}$

Representation details covered in ELEC 220
Lots of details in B&O Chapter 2.4
IEEE 754 Special Numbers

+0.0, -0.0
+∞, -∞

NaN: “Not a number”

\[ (+1.0 \times 10^{+38})^2 = +\infty \quad +0.0 \div +0.0 = \text{NaN} \]
\[ +1.0 \div +0.0 = +\infty \quad +\infty - +\infty = \text{NaN} \]
\[ +1.0 \div -0.0 = -\infty \quad \sqrt{-1} = \text{NaN} \]
FP vs. Integer Results

```c
int    i = 20 / 3;
float  f = 20.0 / 3.0;
```

True mathematical answer: $20 \div 3 = 6 \frac{2}{3}$

- $i = \, ? \, 6$  \hspace{1cm} Integer division ignores remainder
- $f = \, ? \, 6.666667$  \hspace{1cm} FP arithmetic rounds result
FP vs. Integer Results

```
int  i = 1000 / 6;
float f = 1000.0 / 6.0;
```

True mathematical answer: $1000 \div 6 = 166 \frac{2}{3}$

$i = ?$  166  Integer division ignores remainder

$f = ?$  166.666672  FP arithmetic rounds result

**Surprise!**

Arithmetic in binary, printing in decimal - doesn’t always give expected result
**FP ↔ Integer Conversions in C**

```c
#include <limits.h>
#include <stdio.h>

int main(void)
{
    unsigned int ui = UINT_MAX;
    float f = ui;
    printf("ui: %u\nf: %f\n", ui, (double)f);
}
```

Surprisingly, this program print the following. Why?

```
ui: 4294967295
f: 4294967296.000000
```
FP ↔ Integer Conversions in C

```c
int i = 3.3 * 5;
float f = i;
```

True mathematical answer: $3.3 \times 5 = 16 \frac{1}{2}$

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>3.3</td>
<td>Integer → FP: Can lose precision, rounds if necessary. 32-bit int fits in double-precision FP.</td>
</tr>
<tr>
<td>$f$</td>
<td>5.0</td>
<td>FP → integer: Truncate fraction. If out of range, undefined – not error.</td>
</tr>
</tbody>
</table>

Converts 5 → 5.0 - Truncates result $16 \frac{1}{2}$ → 16
FP Behavior

Programmer must be aware of accuracy limitations!
Dealing with this is a subject of classes like CAAM 453

$$(10^{10} + 10^{30}) + -10^{30} = ? \quad 10^{10} + (10^{30} + -10^{30})$$

$$10^{30} - 10^{30} = ? \quad 10^{10} + 0$$

$$0 \neq \quad 10^{10}$$

Operations not associative!

$$(1.0 + 6.0) ÷ 640.0 = ? \quad (1.0 ÷ 640.0) + (6.0 ÷ 640.0)$$

$$7.0 ÷ 640.0 = ? \quad .001563 + .009375$$

$$0.010937 \neq \quad .010938$$

$\times, ÷$ not distributive across $+,-$
What about other types?

Booleans
- A late addition to C

Strings
- We’ll cover these in a later class

Enumerated types
- A restricted set of integers

Complex Numbers
- Not covered
Booleans

One bit representation

- 0 is false
- 1 is true

One byte or word representation

- Inconvenient to manipulate only one bit
- Two common encodings:
  - 0000...0000 is false
  - 0000...0001 is true
  - all other words are garbage

- Wastes space, but space is usually cheap
Booleans in C

```
#include <stdbool.h>

bool bool1 = true;
bool bool2 = false;
```

bool added to C in 1999

Many programmers had already defined their own Boolean type

- To avoid conflict bool is disabled by default
C’s Common Boolean Operations

C extends definitions to integers

- Booleans are encoded as integers
  - 0 == false
  - non-0 == true
- Logical AND: 0 && 4 == 0 3 && 4 == 1 3 && 0 == 0
- Logical OR: 0 || 4 == 1 3 || 4 == 1 3 || 0 == 1
- Logical NOT: ! 4 == 0 ! 0 == 1

&& and || short-circuit

- Evaluate 2nd argument only if necessary
- E.g., 0 && error-producing-code == 0
Enumerated Types

E.g., a Color = red, blue, black, or yellow

- Small (finite) number of choices
- Booleans & characters are common special cases
- Pick arbitrary bit patterns for each

Not enforced in C

- Actually just integers
- Can assign values outside of the enumeration
- Could cause bugs
Enumerated Types in C

Pre-C99 Boolean definition:
```c
enum Bool { false = 0, true = 1 };    
typedef enum Bool bool;             
bool my_bool = true;                 
```

Alternative style:
```c
enum AColor { COLOR_RED, COLOR_WHITE, 
              COLOR_BLACK, COLOR_YELLOW };    
typedef enum AColor color_t;         
color_t my_color = COLOR_RED;        
```

The new type name is "enum Color"
First Assignment

Carefully read the assignments web page
  • Honor code policy
  • Slip day policy

All assignments will be announced on Owlspace and posted on the web page

Assignments are due at 11:55PM on the due date, unless otherwise specified

Assignments must be done on CLEAR servers
Next Time

Arrays and Pointers in C