CSE 1310 - Introduction to Computers & Programming

Data Types & Number Systems

Alex Dillhoff

University of Texas at Arlington

Numbers in Memory

Data is represented in memory dependent on the **type**. The **type** also determines how much memory they require.

There are several types in C:

- Scalar
- Aggregate
- Functions
- Union
- Struct
- Void

Memory and C Programs

When a program is executed, two broad categories of data are placed in memory:

1. **Object code** - The instructions which are executed.
2. **Variables** - The individual data that are processed.

Representing Numbers

The lowest unit of memory is represented as a **bit**, which can either be 1 or 0.

The next largest unit of measurement for information is a **byte**, which consists of 8 bits.

Representing Numbers

Another unit of measurement for data is a **word**, which has a size dependent on a specific architecture.

Commonly, a **word** is designed to optimize at the hardware level. The size is usually chosen such that an entire instruction can be transferred in a single operation.

Sometimes the size represents the largest possible address size.

Representing Numbers

Any number can be conveniently represented as a combination of the multiples of each of the powers of the base.

Examples in base 10

- $212 = 2 * 10^2 + 1 * 10^1 + 2 * 10^0$
- $1650 = 1 * 10^3 + 6 * 10^2 + 5 * 10^1 + 0 * 10^0$
- $6 = 6 * 10^0$
- $21 = 2 * 10^1 + 1 * 10^0$

Representing Binary

Binary numbers can either be 0 or 1 for each power. They can be represented similarly to the approach taken in the previous slide.

Examples in base 2

- $2 = 1 * 2^1 + 0 * 2^0$
- $32 = 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$
- $10 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$
- $5 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0$

Converting from decimal to binary

Base Notation

When representing numbers from multiple systems, it is convenient to show the base of each number using a subscript.

- $127_{10} = 1111111_2$
- $5_{10} = 101_2$
- $256_{10} = 100000000_2$
- $3_{10} = 11_2$

Representing Hexadecimal

Hexadecimal numbers have digits that can be $0 - F$, reflecting a base of 16.

The counting sequence of hexadecimal is 0-9 then A-F.

Examples in base 16

- $F = 15 * 16^0$
- $80 = 8 * 16^1 + 0 * 16^0$
- $FF = 15 * 16^1 + 15 * 16^0$
- $A0E = 10 * 16^2 + 0 * 16^1 + 14 * 16^0$

Converting from decimal to hexadecimal

$128_{10} = ?_{16}$

Conversion: Divide by the base you are converting to. The remainder fills up the right-most digit.

$\frac{128}{16} = 8$ remainder 0

Converting from decimal to hexadecimal

$128_{10} = ?_{16}$

Conversion: Divide by the base you are converting to. The remainder fills up the right-most digit.

$\frac{128}{16} = 8$ remainder 0

Take the remaining value, 8, and divide again, placing the remainder in the next position.

$\frac{8}{16} = 0$ remainder 8

Result: $128_{10} = 80_{16}$

Converting from decimal to hexadecimal

$312_{10} = ?_{16}$

$\frac{312}{16} = 19$ remainder 8

Intermediate Result: 8_{16}

$\frac{19}{16} = 1$ remainder 3

Intermediate Result: 38_{16}

$\frac{1}{16} = 0$ remainder 1

Final Result: 138_{16}

Scalar Types in C

- C supports character, integer, and scalar types.
- Each type has a **minimum size**.
- Character and integer types can either be signed or unsigned.

Scalar Types in C

Integer types can represent a range of numbers dependent on their size.

For example, an integer type with a size in m bits can represent a range of $[-2^{m-1} - 1, 2^{m-1} - 1]$ for **signed** types and $[0, 2^m - 1]$ for **unsigned** types.

Scalar Types in C

https://en.wikipedia.org/wiki/C_data_types

Signed versus Unsigned Types

An `int` is a **signed** type, meaning it can represent both positive and negative numbers.

The minimum bit size of an `int` is 16 bits.

Signed Types

The left-most bit in a **signed** type is called the **sign bit.**

A 1 signifies a negative value, and a 0 is a positive value.

Examples

- $0111_2 = 7_{10}$
- $1111_2 = -7_{10}$ (or is it?)

Signed Types

When representing a negative number, the left-most bit is reserved as the **sign bit**.

If this bit is 1, then the number is negative.

Signed Types

Most platforms use a representation called **two's complement**.

Let's first look at **one's complement**.

One's Complement

Most platforms use a representation called **two's complement**.

Let's first look at **one's complement** using a 4-bit number.

One's Complement

Value	Binary	Negative
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000

One's Complement

If the left-most bit is used as the sign bit, then `0111` = 7.

What happens if we add a single bit?

One's Complement

`1111 = ?`

One's Complement

`1111 = -0`

One's Complement

One's complement is not ideal for basic arithmetic operations.

Consider `3 + (-2)` by evaluating it in binary.

One's Complement

$$\begin{array}{rrr} & 0011 & 3 \\ + & 1101 & -2 \\ \hline \end{array}$$

One's Complement

$$\begin{array}{rrr} & 0011 & 3 \\ + & 1101 & -2 \\ \hline & 0000 & 0 \end{array}$$

One's Complement

$$\begin{array}{rrr} & 0011 & 3 \\ + & 1101 & -2 \\ \hline & 0000 & 0 \end{array}$$

We have enough bits to represent the number 1... **what happened?**

Two's Complement

Two's complement addresses this shortcoming.

It was designed to work naturally with binary arithmetic operations.

Two's Complement

The only difference between one's complement and two's complement is that you add 1 after negating the bits.

Value	Binary	Negative
0	0000	0000
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001

Two's Complement

Consider `3 + (-2)` with `two's complement`.

Two's Complement

$$\begin{array}{rrr} & 0011 & 3 \\ + & 1110 & -2 \\ \hline \end{array}$$

One's Complement

$$
\begin{array}{rrr}
 & 0011 & 3 \\
+ & 1110 & -2 \\
\hline
 & 0001 & 1
\end{array}
$$

Example: to_bit_string.c
Example: rollover.c
Example: sizeof.c

Type Conversions

There are two approaches to converting a value from one type to another:

1. Automatic Type Conversion
2. Forced Type Conversion

Automatic Type Conversion

- Every expression has an associated type.
- Expressions resulting from logical or relational operators have type `int`.
- All values of `char` or `short` are promoted to `int` before processing.

Dominating Types

Automatic conversions involving mixed types acted upon by a binary operation generally follow the following prioritization:

1. `long double`
2. `double`
3. `float`
4. `unsigned long`
5. `long`
6. `unsigned`
7. `int`

Automatic Type Conversions

Further reading: Chapter 3.10

Example: auto_convert.c

Forced Type Conversions

Individual expressions and values can be cast to a different type using the following syntax:

Syntax

```
(type) var;
```

Example

```
float a = 3.1;
printf("a as an int is %d\n", (int) a);
```