1 Hexadecimal

Often when programming we want to describe a specific bit pattern. Hexadecimal, base 16, is often used for this. The reason hexadecimal is used over other representations is because one hexadecimal digit corresponds to four binary digits. Representations in other bases do not necessarily have that correspondence because the correspondence is only present when the number of digits is a power of two. Additionally, a fundamental unit used in computers is a byte. Bytes are almost always eight bits in size and therefore are two hexadecimal characters. Technically a byte is the smallest addressable unit of memory, so other sizes could exist and historically have existed, but such systems haven't existed since the 1970s at the latest. In situations where it could vary, or extreme precision is needed like a standards document, the word octet may be used to refer to eight bits.

Hexadecimal consists of the standard base-10 digits of 0-9 followed by the letters A-F, with A = 10 and F = 15.

2 Word Size

Every computer architecture has a *word size*. Often one can tell the word size of a processor by looking at the width of a register. On some architectures specialized registers have width that differs from the remainder of registers, so it is not a foolproof definition. We will cover registers later in the course. An alternative definition of word size is the maximum virtual address space. Unfortunately there are many architectures that violate this definition as well, many of them for embedded systems. Even the common IA-32 architecture (better known as 32-bit x86) violated this using Physical Address Extensions (PAE), but operating system support for it was limited.

Practically the processors you will see today have word size of 8, 16, 32, or 64 bits. The use of 8 and 16 tends to be limited to embedded systems in the form of microcontrollers (a special type of processor that integrates memory and input/output subsystems). 32 bit systems exist fewer places today. They used to be prevalent in desktops and laptops, but those started to change to 64 bits about ten years ago. Devices like phones still routinely have 32 bit processors, but those are also transitioning to 64 bits.

A good rule of thumb to determine the word size of a processor is to look at the width of a pointer. Pointers will be covered in more detail later in the course, but for now think of them as memory addresses.

If a system has a 32-bit word size, then it can address 2^{32} bytes $=$ 4 gigabytes $\approx 4 \times 10^9$ bytes. A 64-bit word size can address 2^{64} bytes $=$ 16 exabytes $\approx 1.84 \times 10^{19}$ bytes.

3 Sizes of Fundamental Integral Types of C++

Type Specifier	Equivalent Type	Width in bits by data model				
		C++ Standard	LP32	ILP32	LLP64	LP64
char signed char unsigned char	char	one character	8	8	8	8
short short int signed short signed short int	short int	≥16	16	16	16	16
unsigned short unsigned short int	unsigned short int					
int signed signed int	int	≥ 16	16	32	32	32
unsigned unsigned int	unsigned int					
long long int signed long signed long int	long int	≥ 32	32	32	32	64
unsigned long unsigned long int	unsigned long int					
long long long long int signed long long signed long long int	long long int	≥ 64	64	64	64	64
unsigned long long unsigned long long int	unsigned long long int					

Notes:

- In addition to the sized specified above, the C++ standard requires

  $$1 = \text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)} \leq \text{sizeof(long long)}.$$

- The types unsigned char and signed char are not the same and could in theory make a difference. In practice chars are almost always by default unsigned.
- For C++14 and newer, the minimum size of a char is 8 bits.
- The type long long did not exist until C++11.
- The LP32 model is used by the Win16 API.
- The ILP32 model is used by the Win32 API and 32 bit UNIX-like operating systems (Linux, OS X, etc).
- The LLP64 model is used by the Win64 API.
- The LP64 is used by 64-bit UNIX-like operating systems (Linux, OS X, etc).

4 Unsigned Encoding

For a vector of bits $\langle b_{w-1}, b_{w-2}, \ldots, b_0 \rangle$, the corresponding integer is given by

$$\sum_{i=0}^{w-1} b_i 2^i.$$

This means that for w bits, the numbers 0 through $2^w - 1$, inclusive, are representable.

5 Twos Complement Encoding

For a vector of bits $\langle b_{w-1}, b_{w-2}, \ldots, b_0 \rangle$ in twos complement, the corresponding integer is given by

$$-b_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i.$$

This means that for w bits, the numbers -2^{w-1} through $2^{w-1} - 1$, inclusive, are representable.

Twos complement is almost universally to encode signed types.

6 Math with integers

When working with unsigned numbers, the overflow behavior is the same as if you were working modulo 2^w. For unsigned types this is the defined operation in C++.

For signed integers, the C++ leaves this behavior as undefined. In general it is advisable to avoid undefined behavior.

Division truncates the result, which is the same as taking the floor for unsigned integers. For signed integers, this takes the floor for positive values, and the ceiling for negative values. This is also known as rounding towards zero.

7 Size of Fundamental Floating Point Types in C++

In C++, as with most programming languages, floating point behavior usually specified by the IEEE 754 standard (with the latest version being IEEE 754-2008). They are commonly referred to as "IEEE Floating Point numbers". The standard can also be referred to as ISO/IEC/IEEE 60559:2011, which is identical to IEEE 754-2008.

The `float` data type is most often a 32-bit IEEE 754 floating point number.

The `double` data type is most often a 64-bit IEEE 754 floating point number.

The `long double`, the extended double, is a special data type. On IA-32 and IA-64 (x86 and x86-64) this is often a special 80-bit x87 floating point. In practice these are rarely used since they are not portable between processor architectures.

The general form of an IEEE 754 is

$$(-1)^s \times m \times 2^e,$$

where s is the sign bit, m is the *mantissa* (also known as the significand or coefficient), and e is the exponent. The sign bit is exactly one bit and indicates whether the number is positive or negative. The mantissa is in fractional binary representation. The exponent is in a biased integral representation.

	Size in Bits		
	Sign Bit	Mantissa	Exponent
float	1	23	8
double	1	52	11

In memory the sign bit is in the most significant place, followed by the exponent, with the mantissa occupying the least significant portion.

7.1 Fractional Binary Representation

For a vector of bits $\langle b_m, b_{m-1}, \ldots, b_1, b_0, b_{-1}, \ldots, b_{-n}\rangle$ in twos complement, the corresponding integer is given by

$$\sum_{i=-n}^{m} b_i 2^i.$$

Note that for the first m bits, this is the same as an integral representation. For the last n bits, this corresponds to values small than 1 in fractional powers of two, such as $\frac{1}{2}$, $\frac{1}{4}$, etc.

7.2 Biased Integral Representation

A biased integral representation is a means of encoding positive and negative values which is distinct from twos complement.

For a vector of bits $\langle b_{w-1}, b_{w-2}, \ldots, b_0\rangle$ in biased integral representation, the corresponding integer is given by

$$e - 2^{w-1} - 1,$$

where the term $2^{w-1} - 1$ is the bias.

In practice this means that a float has exponent sizes of -126 through $+127$, inclusive, and a double has exponent sizes of -1022 through $+1023$, inclusive.

7.3 Normalized Values

This is the most common case. It occurs when the exponent is neither all zeros, nor all ones. In this case the exponent is interpreted in biased integral representation and the mantissa is interpreted as a fractional value in the range $[1, 2)$, as $1.b_{n-1} \cdots b_1 b_0$. This is often known as an *implied leading one representation.* It is used because it allows for an additional bit of representation since the exponent can be adjusted to compensate for the leading one.

7.4 Denormalized Values

When the exponent field is all zeros, the value is in *denormalized* form. These are also known as *subnormals.* In this case the exponent is interpreted in a biased integral representation and the mantissa is interpreted as a fractional value in the range $[0, 1)$, as $0.b_{n-1} \cdots b_1 b_0$.

The purpose of having denormalized values is two fold. First, it allows for representing exact 0. It cannot be represented as a normalized value because of the leading one representation. Second, it allows for numbers near 0, which enables *gradual underflow.*

7.5 Special Values

The final category of values is when the exponent field is all ones. When the mantissa is all zeros, then the value is either $+\infty$ or $-\infty$, depending on the sign bit. If the mantissa is not all zeros, the result is called NaN, short for "Not A Number." These are used to represent the result of things like $\sqrt{-1}$, $0/0$, and other undefined or unrepresentable numbers.

7.5.1 NaN

The IEEE 754 standard specifies two types of NaN, signaling and quiet, sometimes represented as sNaN and qNaN, respectively. Generally the distinction between these two is not of practical importance.

C++ in particular treats all NaN as quiet.

7.6 Rounding

The IEEE 754 standard specifies five types of rounding:

1. Round to nearest, ties to even
2. Round to nearest, ties away from zero
3. Round toward zero
4. Round toward $+\infty$
5. Round toward $-\infty$

In practice the rounding type used is generally not important. Prior to C++11, controlling the environment was compiler specific. For C++11 and newer, it can be controlled by the functions in the header `cfenv`.

7.7 Equality

As will be explored in the second systems homework, equality of floating point numbers is often a mistake. That is the code

```
double a, b;
// compute a and b
if (a == b) {
  // do something
}
```

will often fail.

7.8 Absolute Error

One way of fixing this is to call two numbers equal if the absolute difference is small enough. Mathematically this is

$$|a - b| < \epsilon.$$

While this can work, it is difficult to do well since the choice of ϵ is very application specific.

7.9 Relative Error

A better way is to consider the relative difference between the numbers. Mathematically this is

$$\left|\frac{A-B}{B}\right| < \epsilon.$$

This suffers from a different problem. Why did we select to divide by B instead of A? This choice could affect the result.

7.10 Combining Both

While still problematic a better option is to combine both absolute and relative errors.

```
bool AlmostEqualRelativeOrAbsolute(float A, float B,
     float maxRelativeError, float maxAbsoluteError) {
    if (fabs(A - B) < maxAbsoluteError)
        return true;

    float relativeError;

    if (fabs(B) > fabs(A))
        relativeError = fabs((A - B) / B);
    else
        relativeError = fabs((A - B) / A);

    if (relativeError <= maxRelativeError)
        return true;

    return false;
}
```

From `http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm`. This is still not ideal, but is generally good enough. Keep in mind whenever comparing floats, or working with them, you need to be aware of how to compare equality and use an appropriate method for your application.