Types and Representation
C++ Types

- The *type* of all data used in a C++ program must be specified
  - A data type is a description of the data being represented
    - That is, a set of possible values and a set of operations on those values
- There are many different C++ types
  - So far we’ve mostly seen *ints* and *floats*
    - Which represent different types of numbers
**Numeric Types**

- *Integers* are whole numbers
  - Like 1, 7, 23567, -478
  - There are numerous different integer types
    - `int`, `short`, `long`, `long long`, `unsigned int`
    - These types differ by size, and whether or not negative numbers can be represented

- *Floating point numbers* can have fractional parts
  - Such as 234.65, -0.322, 0.71, 3.14159
  - There are also different floating point types
    - `double`, `float`, `long double`
In addition to numbers it is common to represent text

- Text is made up of character sequences
  - Often not just limited to a-z and A-Z

The `char` type represents *single* characters

- To distinguish them from identifiers, characters are enclosed in single quotes
  - `char ch = 'a'`
- Remember that strings are enclosed by ""'s
Boolean values represent the logical values *true* and *false*

- These are used in conditions to make decisions

In C++ the `bool` type represents Boolean values

- The value 1 represents *true*, and the value 0 represents *false*
More About Types

- Numeric values (and bool values) do not have to be distinguished from identifiers
  - Identifiers cannot start with a number or a –
- Representing large amounts of text with single characters would be cumbersome
  - Text enclosed in ""'s is referred to as a string
    - A sequence of characters
    - More on this later ...
The purpose of a variable is to store data

The *type* of a variable describes what kind of data is being stored

- An *int* stores whole numbers
- A *char* stores a single character
- And so on

Why do we have to specify the type?
(Human) Language and Types

- When we communicate, we generally don't announce the type of data that we are using:
  - Since we understand the underlying meaning of the words we use in speech, and
  - The symbols we use are unambiguous.
- For example, do these common symbols represent words or numbers?
  - eagle
  - 256
A computer has a much more limited system for representation.

Everything has to be represented as a series of binary digits.

- That is 0s and 1s

What does this binary data represent?

- 0110 1110 1111 0101 0001 1101 1110 0011
- It could be an integer, a float, a string, ...
  - But there is no way to determine this just by looking at it
Numbers
A single binary digit, or bit, is a single 0 or 1
- bit = *binary digit*

Collections of bits of various sizes
- byte – eight bits, e.g. 0101 0010
- kB – kilobyte = \(2^{10}\) bytes = 1,024 bytes = 8,192 bits
- MB – megabyte = \(2^{20}\) bytes = 1,048,576 bytes
  - = 8,388,608 bits
- GB – gigabyte = \(2^{30}\) bytes = 1,073,741,824 bytes
  - = 8,589,934,592 bits
- TB – terabyte = \(2^{40}\) bytes
Modern computer architecture uses binary to represent numerical values
- Which in turn represent data whether it be numbers, words, images, sounds, ...

Binary is a numeral system that represents numbers using two symbols, 0 and 1
- Whereas decimal is a numeral system that represents numbers using 10 symbols, 0 to 9
Number Bases

- We usually count in base 10 (decimal)
  - But we don’t have to, we could use base 8, 16, 13, or 2 (binary)
- What number does 101 represent?
  - It all depends on the base (also known as the radix) used – in the following examples the base is shown as a subscript to the number
  - 101\textsubscript{10} = 101\textsubscript{10} (wow!)
    - i.e. 1*10\textsuperscript{2} + 0*10\textsuperscript{1} + 1*10\textsuperscript{0} = 100 + 0 + 1 = 101
  - 101\textsubscript{8} = 65\textsubscript{10}
    - i.e. 1*8\textsuperscript{2} + 0*8\textsuperscript{1} + 1*8\textsuperscript{0} = 64 + 0 + 1 = 65
  - 101\textsubscript{16} = 257\textsubscript{10}
    - i.e. 1*16\textsuperscript{2} + 0*16\textsuperscript{1} + 1*16\textsuperscript{0} = 256 + 0 + 16 = 257
  - 101\textsubscript{2} = 5\textsubscript{10}
    - i.e. 1*2\textsuperscript{2} + 0*2\textsuperscript{1} + 1*2\textsuperscript{0} = 4 + 0 + 1 = 5
More Examples

- What does $12,345_{10}$ represent?
  - $1 \times 10^4 + 2 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$

- Or $12,345_{16}$?
  - $1 \times 16^4 + 2 \times 16^3 + 3 \times 16^2 + 5 \times 16^1 + 5 \times 16^0 = 74,565_{10}$

- And what about $11,001_2$?
  - $1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 25_{10}$

- The digit in each column is multiplied by the base raised to the power of the column number
  - Counting from zero, the right-most column
# More Examples in Columns

<table>
<thead>
<tr>
<th>base 10</th>
<th>$10^4$ (10,000)</th>
<th>$10^3$ (1,000)</th>
<th>$10^2$ (100)</th>
<th>$10^1$ (10)</th>
<th>$10^0$ (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12,345</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>base 16</th>
<th>$16^4$ (65,536)</th>
<th>$16^3$ (4,096)</th>
<th>$16^2$ (256)</th>
<th>$16^1$ (16)</th>
<th>$16^0$ (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12,345</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>base 2</th>
<th>$2^4$ (16)</th>
<th>$2^3$ (8)</th>
<th>$2^2$ (4)</th>
<th>$2^1$ (2)</th>
<th>$2^0$ (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11,001</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Hexadecimal

Hexadecimal or base 16 has been used as an example of a number base:

- It is often used as a convenient way to represent numbers in a computer system.
- Since it is much more compact than binary.
- And is easy to convert from binary:
  - By converting every four bits to a single hex digit.
For hexadecimal numbers we need symbols to represent values between 10 and 15
- $0_{16}$ through $9_{16}$ represent $0_{10}$ through $9_{10}$
- In addition we need symbols to represent the values between $10_{10}$ and $15_{10}$
  - Since each is a single hex digit

We use letters
- $A_{16} = 10_{10}$
- $B_{16} = 11_{10}$
- ...
- $F_{16} = 15_{10}$
It is straightforward to convert from binary to hex and vice versa

- Consider the binary value 0110 1101
  - This single byte equals $109_{10}$
  - Or 6D in hex: $6_{10} \times 16_{10} = 96_{10} + 13_{10} = 109_{10}$

- But we don’t have to convert the entire value in this way
  - We can just convert every 4 bits to its hex representation
# Binary to Hex

## Base 2 Table

<table>
<thead>
<tr>
<th>base 2</th>
<th>$2^8$</th>
<th>$2^7$</th>
<th>$2^6$</th>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>...</td>
<td>0 or 128</td>
<td>0 or 64</td>
<td>0 or 32</td>
<td>0 or 16</td>
<td>0 or 8</td>
<td>0 or 4</td>
<td>0 or 2</td>
<td>0 or 1</td>
</tr>
<tr>
<td>$109_{10}$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

For $109_{10}$, the binary representation is

$$6 (0 + 2 + 4 + 0) \quad D (1 + 0 + 4 + 8)$$

to determine the hex digit sum the 4 corresponding binary digits

## Base 16 Table

<table>
<thead>
<tr>
<th>base 16</th>
<th>$16^2$</th>
<th>$16^1$ # of 16s</th>
<th>$16^0$ # of 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>...</td>
<td>0 to 240 ($16 \times 15$)</td>
<td>0 to 15</td>
</tr>
<tr>
<td>$109_{10}$</td>
<td>6</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>
There is an obvious way to represent whole numbers in a computer

- Just convert the number to binary, and record the binary number in the appropriate bits in RAM

However there are two broad sets of whole numbers

- **Unsigned** numbers
  - Which must be positive
- **Signed** numbers
  - Which may be positive or negative
Large Numbers

- Any variable type has a finite size
  - This size sets an upper limit of the size of the numbers that can be stored
  - The limit is dependent on how the number is represented
- Attempts to store values that are too large will result in an error
  - The exact kind of error depends on what operation caused the overflow
Just convert the number to binary and store it

What is the largest positive number that can be represented in 32 bits?

1111 1111 1111 1111 1111 1111 1111 1111

or 4,294,967,295\textsubscript{10}
Adding Unsigned Numbers

- Binary addition can be performed in the same way as decimal addition.
  - Though $1_2 + 1_2 = 10_2$
  - and $1_2 + 1_2 + 1_2 = 11_2$
- But how do we perform subtraction, and how do we represent negative numbers?
  - Since a – sign isn't a 1 or a 0 ...
Our only unit of storage is bits

So the fact that a number is negative has to somehow be represented as a bit value

- i.e. as a 1 or a 0

How would you do it?

- We could use one bit to indicate the sign of the number, *signed integer representation*
Signed Integer Representation

- Keep one bit (the left-most) to record the sign
  - 0 means – and 1 means +
- But this is not a good representation
  - It as two representations of zero
    - Which seems weird and requires logic to represent both
    - And wastes a bit pattern that could represent another value
  - It requires special logic to deal with the sign bit and
    - It makes implementing subtraction difficult and
  - For reasons related to hardware efficiency we would like to avoid performing subtraction entirely
There is an alternative way of representing negative numbers called *radix complement*

- That avoids using a negative sign!

To calculate the radix complement you need to know the maximum size of the number

- That is, the maximum number of digits that a number can have

- And express all numbers using that number of digits
  
  - e.g. with 4 digits express 23 as 0023
Negative numbers are represented as the *complement* of the positive number

- The complement of a number, \( N \), in \( n \) digit base \( b \) arithmetic is: \( b^n - N \)

Let’s look at two base 10 examples, one using 2 digit arithmetic, and one 3 digit arithmetic

- Complement of 24 in two digit arithmetic is:
  - \( 10^2 - 24 = 76 \)
- Complement of 024 in three digit arithmetic is:
  - \( 10^3 - 24 = 976 \)
Complement Subtraction

- Complements can be used to do subtraction
- Instead of subtracting a number add its complement
  - And ignore any number past the maximum number of digits
- Let’s calculate $49 - 17$ using complements:
  - We add $49$ to the complement of $17$
    - The complement of $17$ is $83$
  - $49 + 83 = 132$, ignore the $1$, and we get $32$
What did we just do?
- The complement of 17 in 2 digit arithmetic is \(100 - 17 = 83\)
- And we *ignored* the highest order digit
  - The 100 in 132 (from \(49 + 83 = 132\))
  - That is, we took it away, or subtracted it
- So in effect \(49 + \text{complement}(17)\) equals:
  - \(49 + (100 - 17) - 100\) or
  - \(49 - 17\)
But ...

- So it looks like we can perform subtraction by just doing addition (using the complement)
- But there might be a catch here – what is it?
  - To find the complement we had to do subtraction!

  DOH!

- but let’s go back to binary again
Two’s Complement

- In binary we can calculate the complement in a special way without doing any subtraction
  - Pad the number with os up to the number of digits
  - Flip all of the digits (1's become 0, 0's become 1's)
  - Add 1
- Let's calculate 6 – 2 in 4 digit 2's complement arithmetic then check that it is correct
  - Note: no subtraction will be used!
Calculate 6 – 2 in binary using 2’s complement in 4 digit arithmetic

The answer should be 4, or 0100 in binary

Remember that a number has to be padded with zeros up to the number of digits (4 in this case) before flipping the bits

<table>
<thead>
<tr>
<th>6 in binary</th>
<th>2 in binary</th>
<th>flip the bits</th>
<th>add 1</th>
<th>add it to 6</th>
<th>result</th>
<th>ignore left digit</th>
<th>=</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0010</td>
<td>1101</td>
<td>1110</td>
<td>+</td>
<td>1110</td>
<td>10100</td>
<td>0100</td>
</tr>
</tbody>
</table>
## 3 Bit 2’s Complement

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Flip Bits</th>
<th>Add 1</th>
<th>Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>111</td>
<td>(1)000</td>
</tr>
<tr>
<td>001</td>
<td>+1</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>010</td>
<td>+2</td>
<td>101</td>
<td>110</td>
</tr>
<tr>
<td>011</td>
<td>+3</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>100</td>
<td>-4</td>
<td>011</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td>-3</td>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
<td>001</td>
<td>010</td>
</tr>
<tr>
<td>111</td>
<td>-1</td>
<td>000</td>
<td>001</td>
</tr>
</tbody>
</table>
32 bit 2’s Complement

- 32 bits is $2 \times 31$ bits
  - We can use $31 - 1$ bits for the positive numbers,
  - 31 bits for the negative numbers, and
  - 1 bit for zero
- A range of $2,147,483,647$ to $-2,147,483,648$
  - $2^{31} - 1$ positive numbers,
  - $2^{31}$ negative numbers
  - and 0
To add two numbers $x + y$
- If $x$ or $y$ is negative calculate its 2’s complement
- Add the two numbers
- Check for overflow
  - If both numbers have the same sign,
  - But the result is a different sign then,
  - There is overflow and the resulting number is too big to be represented!

To subtract one number from another $x - y$
- Calculate $x + 2$’s complement ($y$)
Signed Integers

- Here are examples of signed integer types
  - short (16 bits)
    - $-32,768$ to $+32,767$
  - int (32 bits)
    - $-2,147,483,648$ to $+2,147,483,647$
  - long (64 bits)
    - $-9,223,372,036,854,775,808$ to $+9,223,372,036,854,775,807$
Floating Point Numbers

- It is not possible to represent every floating point number within a certain range
  - Why not?
- Floating point numbers can be represented by a *mantissa* and an *exponent*
  - e.g. $1.24567 \times 10^3$, or 1,245.67
  - mantissa = 0.124567
  - exponent = 4
Representing Floats

- The mantissa and exponent are represented by some number of bits
  - Dependent on the size of the type
  - The represented values are evenly distributed between 0 and 0.999...
- For example, a 32 bit (4 byte) float
  - mantissa: 23 bits
  - exponent: 8 bits
  - sign bit: 1 bit
There are only two Boolean values
  - True
  - False
Therefore only one bit is required to represent Boolean data
  - 0 represents false
  - 1 represents true
Character Representation

- How many characters do we need to represent?
  - A to Z, a to z, 0 to 9, and
  - `!@#$%^&*()-=_+[]{}\|;':"",./?
- So how many bits do we need?
  - $26 + 26 + 10 + 31 = 93$
Each character can be given a different value if represented in an 8 bit code
- so $2^8$ or 256 possible codes
- This code is called ASCII
  - American Standard Code for Information Interchange
  - e.g. m = 109, D = 68, o = 111
Unicode

- Unicode is an alternative to ASCII
- It can represent thousands of symbols
- This allows characters from other alphabets to be represented
Strings

- To represent a string use a sequence made up of the codes for each character.
- So the string representing Doom would be:
  - 01000100 01101111 01101111 01101101
    - 01000100₂ = 68₁₀ which represents D
    - 01101111₂ = 111₁₀ which represents o
    - 01101111₂ = 111₁₀ which represents o
    - 01101101₂ = 109₁₀ which represents m
- There is more to this as we will discover later.
Remember this number from the last slide?
- 01000100 01101111 01101111 01101101
- That represented “Doom”
  - Or did it?
- Maybe it is actually an integer!
  - In which case it represents the number 1,148,153,709

This is why C++ needs to keep track of types to know what the bits actually represent
More Representation

- What about colours?