Lecture 7: Types

Sarah Nadi
nadi@ualberta.ca
Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro. Most examples taken from K.N. King's book]

Agenda

- Integer types
- Floating types
- Character types
- Type conversion
- Type definitions
- The `sizeof` operator

Readings

- Textbook Chapter 7

Basic Built-in Types in C

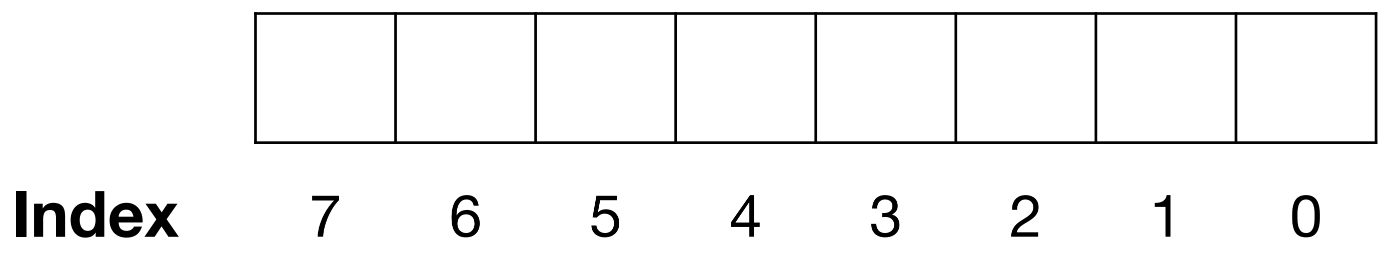
- Integer types (e.g., int)
- Floating point types (e.g., float)
- Boolean types are not built-in (remember how you need to stdbool.h to use `bool`)

Why are Types Important?

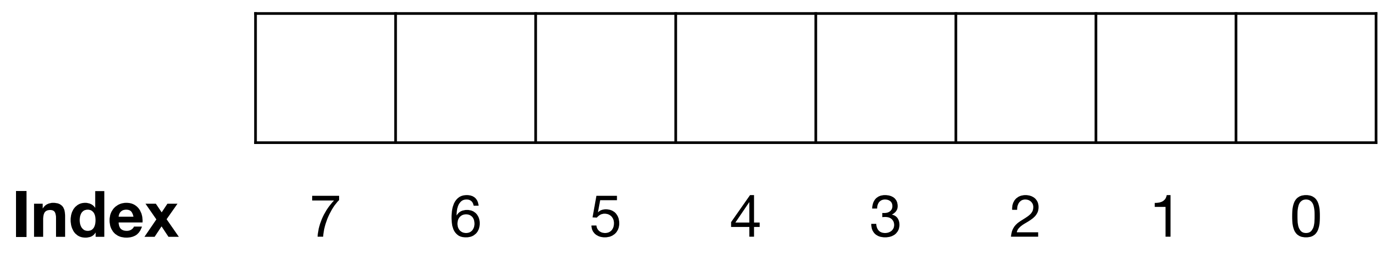
- All data is stored in binary form in memory
- The amount of “space” used for a variable depends on its type
- “Space” is measured in number of bits or bytes (1 byte = 8 bits)

Integer Types

Binary Representation

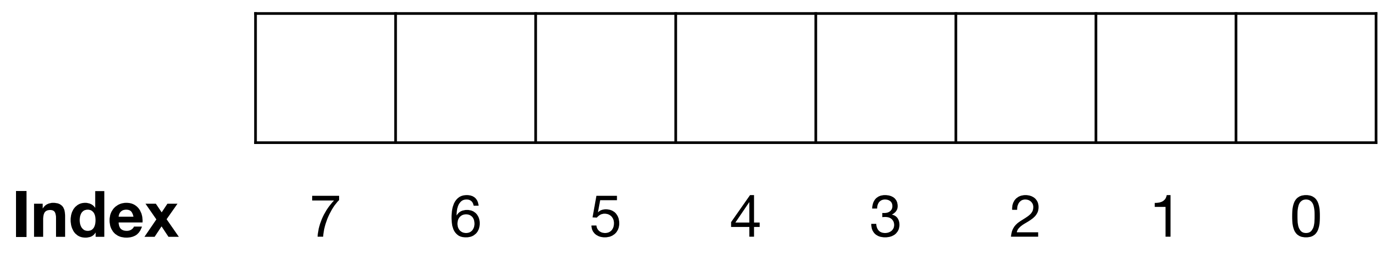


Binary Representation



Each bit can have a value of 0 or 1

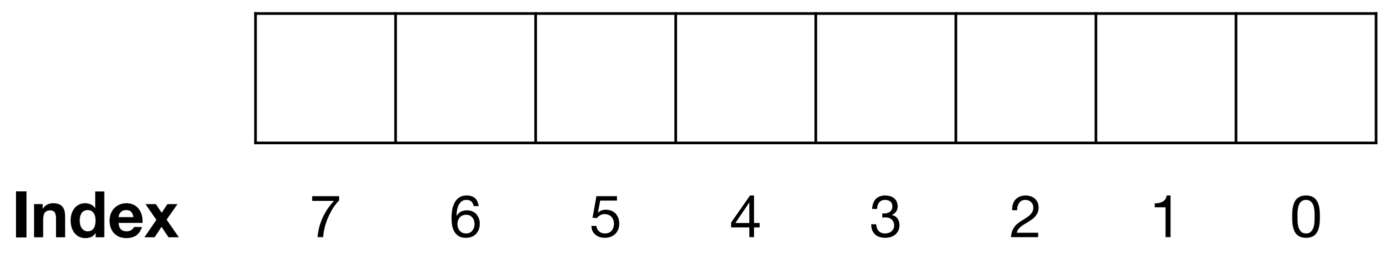
Binary Representation



Each bit can have a value of 0 or 1

Binary Number	**Equivalent Decimal Number**
1	1
10	2
10001	17

Binary Representation



Each bit can have a value of 0 or 1

Binary Number	Equivalent Decimal Number
1	1
10	2
10001	17

Given 8 bits (ignoring sign for now), the maximum binary number that can be stored is 11111111, which is 255 in decimal

Calculating the Maximum Number that Can be Stored

Given `n` bits, the maximum decimal number that can be stored in those bits is

$2^n - 1$

Calculating the Maximum Number that Can be Stored

Given `n` bits, the maximum decimal number that can be stored in those bits is $2^n - 1$

Number of bits	**Maximum Decimal Number**
1	1
2	3
8	255

… but this formula assumes we are using every single bit of those `n` bits to represent the number

Integer Types

- Two categories: signed (default) and unsigned
- As the name indicates, signed integers can store negative numbers while unsigned integers can store only positive numbers

Signed Integers

- The leftmost bit is reserved for the sign: 1 for negative numbers and 0 for positive numbers

- Assume we will use 1 byte to store a signed integer.

    - If the leftmost bit is used for the sign, then we only have 7 bits left.

    - Therefore, if a signed integer is represented through 8 bits, the range of decimal numbers it can hold is -128 to 127

sign bit							

Wait a minute.. why is it -128 and not -127?

[Example from http://kias.dyndns.org/comath/12.html#2c]

Wait a minute.. why is it -128 and not -127?

- To enable binary arithmetic, a technique called “2’s complement” is used

[Example from http://kias.dyndns.org/comath/12.html#2c]

Wait a minute.. why is it -128 and not -127?

- To enable binary arithmetic, a technique called “2’s complement” is used
- Suppose we want to subtract b=1101_2 from a=11011_2 using a 6-bit adder.

[Example from http://kias.dyndns.org/comath/12.html#2c]

Wait a minute.. why is it -128 and not -127?

- To enable binary arithmetic, a technique called “2’s complement” is used
- Suppose we want to subtract b=1101_2 from a=11011_2 using a 6-bit adder.
  - To do so, we need to calculate the 2’s complement of b (think of it as -b).

[Example from http://kias.dyndns.org/comath/12.html#2c]

Wait a minute.. why is it -128 and not -127?

- To enable binary arithmetic, a technique called “2’s complement” is used
- Suppose we want to subtract b=1101_2 from a=11011_2 using a 6-bit adder.
  - To do so, we need to calculate the 2’s complement of b (think of it as -b).
  - The first step is to calculate the 1’s complement by taking the inverse of every bit. Thus, 001101_2 (note how we added 0’s to change it into 6 bits) becomes 110010_2

[Example from http://kias.dyndns.org/comath/12.html#2c]

Wait a minute.. why is it -128 and not -127?

- To enable binary arithmetic, a technique called “2’s complement” is used
- Suppose we want to subtract b=1101_2 from a=11011_2 using a 6-bit adder.
  - To do so, we need to calculate the 2’s complement of b (think of it as -b).
  - The first step is to calculate the 1’s complement by taking the inverse of every bit. Thus, 001101_2 (note how we added 0’s to change it into 6 bits) becomes 110010_2
  - The 2’s complement is then obtained by adding 1 to the 1’s complement. Thus here, the 2’s complement of b would be 110011_2

[Example from http://kias.dyndns.org/comath/12.html#2c]

Wait a minute.. why is it -128 and not -127?

- To enable binary arithmetic, a technique called “2’s complement” is used
- Suppose we want to subtract b=1101_2 from a=11011_2 using a 6-bit adder.
  - To do so, we need to calculate the 2’s complement of b (think of it as -b).
  - The first step is to calculate the 1’s complement by taking the inverse of every bit. Thus, 001101_2 (note how we added 0’s to change it into 6 bits) becomes 110010_2
  - The 2’s complement is then obtained by adding 1 to the 1’s complement. Thus here, the 2’s complement of b would be 110011_2
  - Then we can perform subtraction by adding the 2’s complement of b to a:

[Example from http://kias.dyndns.org/comath/12.html#2c]

Wait a minute.. why is it -128 and not -127?

- To enable binary arithmetic, a technique called “2’s complement” is used
- Suppose we want to subtract b=1101_2 from a=11011_2 using a 6-bit adder.
  - To do so, we need to calculate the 2’s complement of b (think of it as -b).
  - The first step is to calculate the 1’s complement by taking the inverse of every bit. Thus, 001101_2 (note how we added 0’s to change it into 6 bits) becomes 110010_2
  - The 2’s complement is then obtained by adding 1 to the 1’s complement. Thus here, the 2’s complement of b would be 110011_2
  - Then we can perform subtraction by adding the 2’s complement of b to a:

```
  011011
 +110011
  ------
  001110
```

[Example from http://kias.dyndns.org/comath/12.html#2c]

Ok, so what does this have to do with the range of a signed int?

Let’s start with 0 and repeatedly add 1:

[Based on http://kias.dyndns.org/comath/13.html]

Ok, so what does this have to do with the range of a signed int?

Let's start with 0 and repeatedly add 1:

0	0	0	0	0	0	0	0	**0_{10}**

Ok, so what does this have to do with the range of a signed int?

Let's start with 0 and repeatedly add 1:

	0	0	0	0	0	0	0	0
+								1

0_{10}

Ok, so what does this have to do with the range of a signed int?

Let’s start with 0 and repeatedly add 1:

	0	0	0	0	0	0	0	0	$\mathbf{0_{10}}$
+								1	
=	0	0	0	0	0	0	0	1	$\mathbf{1_{10}}$

[Based on http://kias.dyndns.org/comath/13.html]

Ok, so what does this have to do with the range of a signed int?

Let's start with 0 and repeatedly add 1:

$$
\begin{array}{rcccccccccl}
 & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & & \mathbf{0_{10}} \\
+ & & & & & & & & 1 & & \\
= & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{1} & & \mathbf{1_{10}} \\
+ & & & & & & & & 1 & & \\
 & & & & & \mathbf{...} & & & & &
\end{array}
$$

[Based on http://kias.dyndns.org/comath/13.html]

Ok, so what does this have to do with the range of a signed int?

Let’s start with 0 and repeatedly add 1:

$$
\begin{array}{cccccccccl}
 & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \mathbf{0_{10}} \\
+ & & & & & & & & 1 & \\
= & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{1} & \mathbf{1_{10}} \\
+ & & & & & & & & 1 & \\
 & & & & \cdots & & & & & \\
= & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \mathbf{127_{10}}
\end{array}
$$

[Based on http://kias.dyndns.org/comath/13.html]

Ok, so what does this have to do with the range of a signed int?

Let's start with 0 and repeatedly add 1:

	0	0	0	0	0	0	0	0	**0_{10}**
+								1	
=	0	0	0	0	0	0	0	1	**1_{10}**
+								1	
				...					
=	0	1	1	1	1	1	1	1	**127_{10}**
+								1	

[Based on http://kias.dyndns.org/comath/13.html]

Ok, so what does this have to do with the range of a signed int?

Let's start with 0 and repeatedly add 1:

	0	0	0	0	0	0	0	0	**0_{10}**
+								1	
=	0	0	0	0	0	0	0	1	**1_{10}**
+								1	
				...					
=	0	1	1	1	1	1	1	1	**127_{10}**
+								1	
=	1	0	0	0	0	0	0	0	

[Based on http://kias.dyndns.org/comath/13.html]

Ok, so what does this have to do with the range of a signed int?

Let’s start with 0 and repeatedly add 1:

	0	0	0	0	0	0	0	0	**0_{10}**
+								1	
=	0	0	0	0	0	0	0	1	**1_{10}**
+								1	
				...					
=	0	1	1	1	1	1	1	1	**127_{10}**
+								1	
=	1	0	0	0	0	0	0	0	

If this was an unsigned integer, this value would be 128, but since this is a signed integer and the sign bit is 1, we know that this value must be a negative number

So what is the decimal value of the signed integer 10000000_2 ?

So what is the decimal value of the signed integer 10000000_2 ?

- To find the value of a negative signed integer, we need to calculate its 2’s complement

So what is the decimal value of the signed integer 10000000_2 ?

- To find the value of a negative signed integer, we need to calculate its 2’s complement
- First, the 1’s complement is 01111111_2, which is the equivalent of decimal +127. To find the 2’s complement, we would add 1 to the 1’s complement, which would give us 128. This means that the decimal equivalent of the signed integer 10000000_2 is -128.

So what is the decimal value of the signed integer 10000000_2 ?

- To find the value of a negative signed integer, we need to calculate its 2’s complement
- First, the 1’s complement is 01111111_2, which is the equivalent of decimal +127. To find the 2’s complement, we would add 1 to the 1’s complement, which would give us 128. This means that the decimal equivalent of the signed integer 10000000_2 is -128.
- To convince yourself further, try adding 1 to 10000000_2 and see what the decimal equivalent is (it should be -127 which matches our expectation)..

So what is the decimal value of the signed integer 10000000_2 ?

- To find the value of a negative signed integer, we need to calculate its 2’s complement

- First, the 1’s complement is 01111111_2, which is the equivalent of decimal +127. To find the 2’s complement, we would add 1 to the 1’s complement, which would give us 128. This means that the decimal equivalent of the signed integer 10000000_2 is -128.

- To convince yourself further, try adding 1 to 10000000_2 and see what the decimal equivalent is (it should be -127 which matches our expectation)..

- Using the same logic, 11111111_2 is equivalent to -1 since it’s 2’s complement is 1 (the 1’s complement of this number is 0 and adding 1 to it would give 1).

Summary of Signed Integers

- If a signed integer has n bits, it can represent decimal numbers in the range `-2`$^{n-1}$ `to (2`$^{n-1}$ `-1)`

Type	32 bit architecture		64 bit architecture	
	Size	**Number Range**	**Size**	**Number Range**
short int	16 bits (2 bytes)	-32,768 to 32,767	16 bits (2 bytes)	-32,768 to 32,767
unsigned short int	16 bits (2 bytes)	0 to 65,535	16 bits (2 bytes)	0 to 65,535
int	32 bits (4 bytes)	-2,147,483,648 to 2,147,483,647	32 bits (4 bytes)	-2,147,483,648 to 2,147,483,647
unsigned int	32 bits (4 bytes)	0 to 4,294,967,295	32 bits (4 bytes)	0 to 4,294,967,295
long int	32 bits (4 bytes)	-2,147,483,648 to 2,147,483,647	64 bits (8 bytes)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long int	32 bits (4 bytes)	0 to 4,294,967,295	64 bits (8 bytes)	0 to 18,446,744,073,709,551,615
long long int (c99 specific)	64 bits (8 bytes)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,	64 bits (8 bytes)	-9,223,372,036,854,775,808 to 9,223,372,036,854,77

Note that these are typical sizes, but they are not mandated in the C standard. To verify the implementation of the compiler you are using, you can check the limits.h file (see https://www.tutorialspoint.com/c_standard_library/limits_h.htm)

Integer Constants

- Constants are numbers that appear in your program
- They can be written in decimal (base 10), octal (base 8), or hexadecimal (base 16)
- Decimal constants contain digits between 0 and 9, but must not begin with 0 (e.g., 15, 255, 57834)
- Octal constants contain digits between 0 and 7 and **must** begin with 0 (e.g., 017, 055, 0101)
- Hexadecimal constants contain digits between 0 and 9 and letters between a and f (upper or lower case), and **always begin with 0x** (e.g., 0xff, 0xFF, 0x7ab)

Notes about Constants

- The different number systems are just alternates. They do not change how the integer is stored internally: always in binary
- The type of the constant depends on its size (the compiler tries assigning it the smallest type that can fit it)
- If you want to force the compiler to store the constant as a long integer, put L or l after it (e.g., 15L, 0x7ffl)
- If you want to indicate that a constant is unsigned, put U or u after it (e.g., 15U, 0377u)

Integer Overflow

- When arithmetic operations are formed on integers, the result might be too large to represent in the given type. If that happens, an *integer overflow* is said to occur
- When overflow occurs with signed integers, the result is undefined
- When overflow occurs with unsigned integers, the result is defined: we get the correct answer modulo `2n`, where `n` is the number of bits used to store the result. For example, if we add 1 to the unsigned 16-bit number 65,535, the result is guaranteed to be 0.

Reading and Writing Integers

```
unsigned int i;
scanf("%u", &i); // reads i in base 10
printf("%u", i); //writes i in base 10
scanf("%o", &i); //reads i in base 8
printf("%o", i); //prints i in base 8
scanf("%x", &i); //reads i in base 16
printf("%x", i); //prints i in base 16
```

```
short s;
scanf("%hd", &s); /*just put a letter h in front of
                    d, o, u, or x */
printf("%hd", s);
```

```
long l;

scanf("%ld", &l);
printf("%ld", l);
```

```
long long l; //c99 only

scanf("%lld", &l);
printf("%lld", l);
```

Floating Types

Floating Types

- Three formats (sizes according to IEEE floating-point standard):
  - `float`: single-precision, 32 bits (4 bytes)
  - `double`: double-precision, 64 bits (8 bytes)
  - `long double`: extended precision (typically 80 or 128 bits — rarely used)

General Form of Storing Floating Point Numbers

sign bit	k bits represent the exponent	remaining bits represent the fraction

General Form of Storing Floating Point Numbers

sign bit	k bits represent the exponent	remaining bits represent the fraction

- The actual value of the exponent is: `valueStored - (2`$^{\texttt{k-1}}$`-1)`. The second part of the equation is called the *bias*.

General Form of Storing Floating Point Numbers

sign bit	k bits represent the exponent	remaining bits represent the fraction

- The actual value of the exponent is: `valueStored - (2`$^{k-1}$ `-1`). The second part of the equation is called the *bias*.
- The bits that represent the fraction (a.k.a *significand* or *mantissa*) encode the part after the decimal point in normalized form.

General Form of Storing Floating Point Numbers

sign bit	k bits represent the exponent	remaining bits represent the fraction

- The actual value of the exponent is: `valueStored - (2`$^{k-1}$` -1)`. The second part of the equation is called the *bias*.
- The bits that represent the fraction (a.k.a *significand* or *mantissa*) encode the part after the decimal point in normalized form.
- `k` determines how large (or small) numbers can be, while the number of bits in the fraction part determines the precision

General Form of Storing Floating Point Numbers

sign bit	k bits represent the exponent	remaining bits represent the fraction

- The actual value of the exponent is: `valueStored - (2`$^{k-1}$` -1)`. The second part of the equation is called the *bias*.
- The bits that represent the fraction (a.k.a *significand* or *mantissa*) encode the part after the decimal point in normalized form.
- `k` determines how large (or small) numbers can be, while the number of bits in the fraction part determines the precision
- This link provides a nice set of concrete examples of converting floating point numbers from decimal to binary: http: // sandbox.mc.edu/~bennet/cs110/flt/dtof.html

General Technique for Converting a Decimal Number into Floating Point Binary Representation

1. Convert the value to binary by separately converting the integral part (i.e., number before the decimal) and the fractional part (i.e., number after the decimal)

2. Append "* 2^0" to the end of the binary number and normalize until the decimal point has only "1" in front of it. Keep track of the exponent since you will use it in step 4.

3.The mantissa is the value after the decimal point after normalizing in step 2. Place this mantissa in the mantissa field (omitting the leading 0 before the decimal point) and fill 0's to the right of the number as necessary

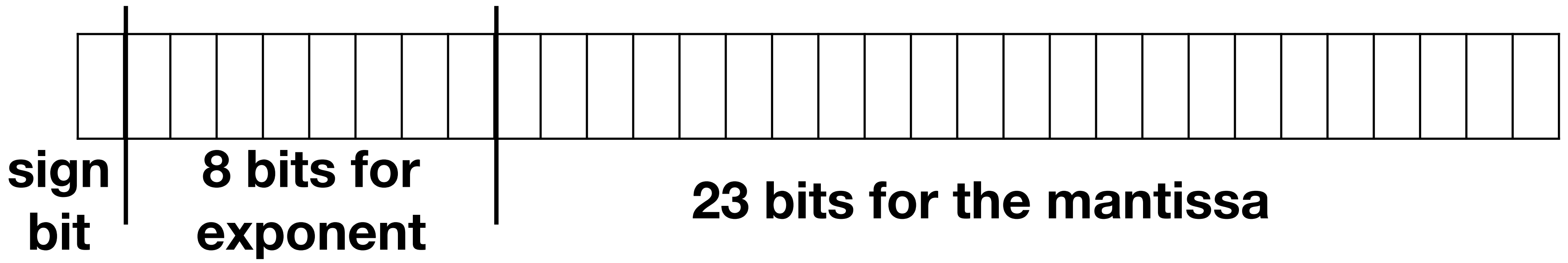
4. Add the bias to the exponent. Remember that bias is $2^{k-1} -1$, where k is the number of bits used to store the exponent

5. Convert the biased exponent into binary and place it in the exponent field

6. Set the sign bit

Hand-written examples

IEEE Floating-point Standard Single Precision (32 bits)



- Note that since the leading bit in the mantissa is always 1 (remember we move the decimal point till there is only a single 1 before it), it is not represented. Thus, there are technically 24 bits for the fraction, but only 23 stored. This extra bit is called the *hidden bit*.

Conversion Specifiers & Constants for Floating Types

- Conversion specifiers:
  - Seen before: `%m.pf`, `%m.pe`, and `%m.pg`
  - NEW: `%m.plf` and `%m.pLf` can be used for type `double` and type `long double`, respectively
  - `p`: number of digits after the decimal point or max. number of significant digits in case of the format specifier `g`
  - `m:` specifies the minimum field width to be used for printing
- Floating constants can be written in a variety of ways:
  - x = 57.0; x = 57.;
  - x = 5.7e+1; x = 57.0e0; x = .57e2;
  - x = 57E0; x = .570.0e-1;

Character Types

(already covered last week)

Character Type Recap

- C treats characters as small integers
- All operations on integers can be done with characters
- ASCII is the most popular character set. See http://www.asciitable.com/
- Use the conversion specifier `%c`
- A variable of type `char` is stored in 1 byte
- There are many libraries that have built-in functions for handling characters
- NEW: character types can be `signed` or `unsigned`. Typically, this should not matter unless you are intentionally using a char to store an integer.

Summary of Arithmetic Types

- Integer types
  - `char`
  - signed: `signed char, short int, int, long int, long long int`
  - unsigned: `unsigned char, unsigned short int, unsigned int, unsigned long int, unsigned long long int, _Bool`
- Floating types
  - real: `float, double, long double`
  - complex (won't cover): `float _Complex, double _Complex, long double _Complex`

Type Conversion

- *Implicit conversion*: automatically done by the compiler
- *Explicit conversion*: explicitly done by the programmer in the source code

Implicit Conversion

- *Implicit type conversion is* automatically done by the compiler occurs when:
    - operands in an arithmetic/logical expression are not of the same type
    - type of the right side of an assignment does not match the type of the variables on the left side
    - type of an argument in a function does not match that of the parameter
    - type of function return value does not match the return type

Implicit Conversion

Cont'd

- Rule of thumb: no loss of semantics or precision so the compiler chooses the “narrowest” type that will safely accommodate both values

- Usually the compiler converts the type of the narrower type to the type of the other operand

```
_Bool → char → short int → int → unsigned int
→ long int → unsigned long int

(→ ) float → double → long double
```

Explicit Conversions

- Happens through *casting*: `(type name) expression`
- The value of `expression` is converted to the `type name`. E.g.:

```
float f, frac_part;
frac_part = f - (int) f;
```

  - note that `(type name)` is regarded as a unary operator so it is applied first

`sizeof` Operator

- Checks how much memory is required to store values of a particular type: `sizeof(type name)`
- Examples:
  - `sizeof(int);`
  - `sizeof(100.0f);`
  - `sizeof(i);`
  - `sizeof(i+j);`
- It is a unary operator that returns a `size_t` integer (i.e., `unsigned long int`). The return value is the number of bytes storing the value.

Printing the value of `sizeof`

- The conversion specifier for `size_t` is typically `%lu` or `%zu` (used in C99 in case the value is greater than the capacity of an unsigned long int)
- Example:

```
int i;
sizeof(i); /* normally 4 but depends on the architecture */
printf("Size of int is %zu bytes\n", sizeof(int));
printf("Size of int is %lu bytes\n", (unsigned long) sizeof(int));
```

Calculating the Length of an Array using `sizeof`

- This works for any array regardless of its type

```
for(int i = 0; i < sizeof(array) / sizeof (array[0]); i++)
   //do something
```

Type Definitions

- `#define BOOL int`
  - This is a **macro**.
  - Every appearance of `BOOL` is simply replaced by `int`
  - This replacement happens before the compiling starts looking at the code
- Type definition: `typedef int Bool;`
  - this is a **C statement that the compiler understands**
  - `Bool` is a new data type, and its type is `int`
  - later, `Bool` can be used the same way as the built-in type `int` to declare variables
  - this can be useful for code readability and portability

Type Definitions *Cont'd*

```
typedef int Quantity;

int main(){
  Quantity q;
  q = 5;
}
```

- Later, you may just change the `typedef statement to typedef long Quantity;` to increase its size without changing anything else in your code.
- In the C library, there is actually a `typedef` to define `size_t` that we saw before: `typedef unsigned long int size_t;`

demo: types.c