Data Types

- Basic data types. There are four basic data types:
  - `char` one byte character in the local character set (typically, ASCII code)
  - `int` an integer, in the natural representation of the host machine
  - `float` single precision floating point
  - `double` double precision floating point

Modifiers for Data Types

- The basic `int` type can be qualified by prefixing `short` and `long`. We can use `short` in place of `short int` and `long` in place of `long int`.
- The type `long double` can also be used and represents extended double precision value but it is implementation dependent.
- `unsigned` or `signed`. For example `unsigned int` would have a range of $[0..2^{32}-1]$ machine where `int` is of size 4 bytes.. And `signed int` would have a range of $[-2^{31}...2^{31}-1]$

How to store signed integers? (1)

- Storing unsigned integers is simple in binary. For example, for an unsigned 3-bit value, we have

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

- Note that a k-bit integer is stored as $X_{k-1}X_{k-2}\ldots X_2X_1X_0$ where X_{k-1} is the most significant bit (MSB) and X_0 is the least significant bit (LSB).
- In general, for an unsigned k-bit number, we can store the range 0 to $2^k - 1$. But what about negative numbers?

How to store signed integers? (2)

- We could use sign-magnitude representation, where the first bit is the sign (0 for positive, 1 for negative) and the rest is the magnitude.

000	+0
001	+1
010	+2
011	+3
100	-0
101	-1
110	-2
111	-3

- Problems?
  - Two zeros...
  - Arithmetic operations are complicated... Try 1 + (-1)

How to store signed integers? (3)

- The 2's complement representation solves both problems!
  - Store positive numbers directly in binary
  - For negative numbers, write the number in binary ignoring the sign. Then complement the number by flipping 0's to 1's and vice versa. Finally add 1 to get the 2's complement

000	+0
001	+1
010	+2
011	+3
100	-4
101	-3
110	-2
111	-1

- Note the positive integers always start with a zero and negative integers always start with 1 in the 2's complement representation!
- Note that 2's complement for a k-bit number is the same as subtracting the number from 2^k.
- This simplifies arithmetic. Try 1 + (-1)

8-bit two's-complement integers

Bits	Unsigned value	2's complement value
0111 1111	127	127
0111 1110	126	126
0000 0010	2	2
0000 0001	1	1
0000 0000	0	0
1111 1111	255	-1
1111 1110	254	-2
1000 0010	130	-126
1000 0001	129	-127
1000 0000	128	-128

Data Type Sizes

Basic data types. The following are typical sizes but beware that the sizes are *machine dependent!*

- `short` 2 bytes signed
- `int` 4 bytes signed (but 2 bytes on some systems... $\langle$*argh*!$\rangle$)
- `long` 8 bytes signed (but is 4 bytes on older systems)
- `char` 1 byte ASCII code (unlike 2 byte Unicode in Java)
- `float` 4 bytes IEEE 754 format (same as in Java)
- `double` 8 bytes IEEE 754 format (same as in Java)

We can use the `sizeof` operator to determine the size (in bytes) of any type.

Determining types on a system

```
/* C-examples/intro/width.c */
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("size of char = %d \n", sizeof(char));
    printf("size of short = %d  \n", sizeof(short));
    printf("size of unsigned short  = %d  \n", sizeof(unsigned short));
    printf("size of int = %d  \n", sizeof(int));
    printf("size of unsigned int = %d  \n", sizeof(unsigned int));
    printf("size of long = %d  \n", sizeof(long));
    printf("size of unsigned long = %d  \n", sizeof(unsigned long));
    printf("size of float = %d  \n", sizeof(float));
    printf("size of double = %d  \n", sizeof(double));
    printf("size of long double = %d  \n", sizeof(long double));
    return 0;
}
```

Determining types on a system (contd.)

Here is the output on onyx.

```
[amit@onyx C-examples]: types
size of char = 1
size of short = 2
size of unsigned short  = 2
size of int = 4
size of unsigned int = 4
size of long = 8
size of unsigned long = 8
size of float = 4
size of double = 8
size of long double = 16
[amit@onyx C-examples]:
```

Range of Data Type Values

- The header file `<limits.h>` defines limits on integer types whereas the header file `<float.h>` defines limits on floating point types.
- **Exercise 2-1 (modified)**. Write a program to determine the ranges of char, short , int, and long variables, both signed and unsigned, by printing appropriate values from standard headers. Determine the ranges of the various floating-point types.
- **Solution**. See example C-examples/intro/range.c.

Typical ranges for C data types

Output of range program on the onyx server.

TYPE	MIN	MAX
char	-128	127
unsigned char	0	255
short	-32768	32767
unsigned short	0	65535
int	-2147483648	2147483647
unsigned int	0	4294967295
long	-9223372036854775808	9223372036854775807
unsigned long	0	18446744073709551615
float	1.175494e-38	3.402823e+38
double	2.225074e-308	1.797693e+308
long double	3.362103e-4932	1.189731e+4932

Other Modifiers

- `const` Constant or read-only. Similar to `final` in Java.
- `static` Similar to static in Java but not the same. Here is an example for its use in a C function. It creates a private persistent variable.

  ```
  int foo(int x)
  {
      static int y=0;   /* value of y is saved */
      y = x + y + 7;  /* across invocations of foo */
      return y;
  }
  ```

  The `static` modifier has another meaning that we will see later.
- `extern`. The variable is declared external to the source file.
- `volatile`. Ask the compiler to not apply optimizations to the variable.
- `register`. Variables declared with qualifier `register` are (if possible) stored in fast registers of the machine. It can only be used for variables declared inside a function and the address operator & cannot be applied to such variables.

Constants

- Use suffix L or l for a long int constant and the suffix U or u for an unsigned constant.

```
int i = 1234;
long j = 2147483648L;
unsigned short k = 65535; /* 2^16 - 1 */
unsigned long m = 123456789UL;
```

- Real numbers are double by default. Use suffix F or f for a float constant and the suffix L or l for long double constant.

```
double x = 1E+25;
float y =  1.14F;
```

- Integers can be specified in octal or hexadecimal instead of decimal. A leading 0 in an integer constant means octal; a leading 0x or 0X means hexadecimal. These can have a U or L suffix like other integer constants.

```
int n1 = 037; /* 31 in decimal */
int n2 = 0x1F; /* 31 in decimal */
```

Enumeration constants

- A enumeration is a list of constant integer values.

  ```
  enum bool { false, true };
  enum bool flag; // a Boolean flag variable
  ```

  where the first name in the list has the value 0, the next 1 and so on unless explicit values are specified. if not all values are specified, unspecified values continue the progression from the last specified value.

  ```
  enum escape { BACKSPACE = '\b', TAB = '\t', NEWLINE = '\n',
                                                      RETURN = '\r' };
  enum month { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL,
                                  AUG, SEP, OCT, NOV, DEC };
                          /* FEB is 2, MAR is 3 and so on */
  ```

- Names in different enumerations must be distinct. Values need not be distinct in the same enumeration.

Type Conversion

- "Narrower" types can be converted to "wider" type without losing information. E.g., an int can be assigned to a long, a float to a double
- A "wider" type can be assigned to a "narrower" type without casting but information may be lost.

```
int m; long n = 10000000000;
float x; double y = 2E300;
m = n;
x = y;
printf("%ld %d %le %e\n", n, m, y, x);
```

  Note that the compiler gives no warning (even with -Wall flag) on the above. However, using the -Wconversion flag does give a warning. See the example `C-examples/intro/conv.c`
- Forced casting works using the following syntax (similar to Java):

  (type-name) expression

Reading Assignment and Exercises

- Read Chapter 2 of the C book (skipping Section 2.9 on bitwise operators for now).
- Attempt Exercises 2-2, 2-4 and 2-10.