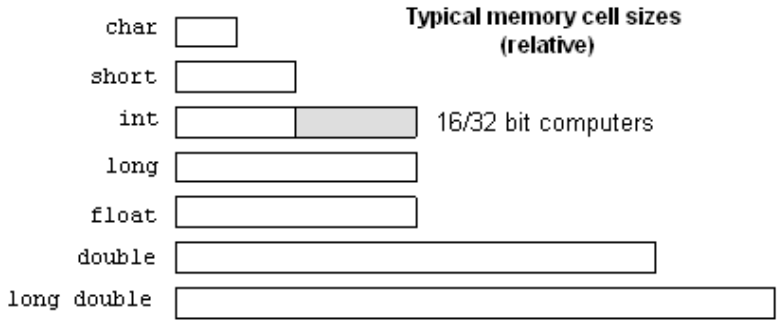
Data Types

Integral Types

An integral data type is a type that is fundamentally an integer. That is, it has no fractional portion. Integral types come in different sizes.

- The size determines how many bytes are required to store values.
- The size also determines how large that value can be.
- Integral types can be either *signed* or *unsigned*:
  - signed values can include positive and negative numbers (including zero)
  - unsigned values only include positive numbers (including zero)
- You usually want to use a data type that relates to the values it will contain.
  - If the data type is too large, there is wasted space.
  - If the data type is too small, there is a loss of data.

There are 6 different integer types (8 including **char**) and their sizes are dependent on the computer. Most of the computers and compilers we using are running 32-bit software. This is a diagram of the relative sizes:



This table shows the range of values for the integral types on a 32-bit computer:

Type	Bytes	Also called	Range of values (Binary)	Range of values (Decimal)
`signed char`	1	`char` (compiler-dependent)	-2^7 to $2^7 - 1$	-128 to 127
`unsigned char`	1	`char` (compiler-dependent)	0 to $2^8 - 1$	0 to 255

`signed short int`	2	`short`
		`short int`
		`signed short`
			-2^{15} to $2^{15} - 1$	-32,768 to 32,767

`unsigned short int`	2	`unsigned short`	0 to $2^{16} - 1$	0 to 65,535
`signed int`	4	`int` `signed`	-2^{31} to $2^{31} - 1$	-2,147,483,648 to 2,147,483,627
`unsigned int`	4	`unsigned`	0 to $2^{32} - 1$	0 to 4,294,967,295

`signed long int`	4	`long`
		`long int`
		`signed long`
			-2^{31} to $2^{31} - 1$	-2,147,483,648 to 2,147,483,627

unsigned long int	4	unsigned long	0 to 2^{32} - 1	0 to 4,294,967,295

This table includes the binary values:

Type	Binary Range	Decimal Range
signed char	10000000 to 01111111	-127 to 128
unsigned char	00000000 to 11111111	0 to 255
signed short	1000000000000000 to 0111111111111111	-32,768 to 32,767
unsigned short	0000000000000000 to 1111111111111111	0 to 65,535
signed int	10000000000000000000000000000000 to 01111111111111111111111111111111	-2,147,483,648 to 2,147,483,627
unsigned int	00000000000000000000000000000000 to 11111111111111111111111111111111	0 to 4,294,967,295
signed long	10000000000000000000000000000000 to 01111111111111111111111111111111	-2,147,483,648 to 2,147,483,627
unsigned long	00000000000000000000000000000000 to 11111111111111111111111111111111	0 to 4,294,967,295

A signed integer is 32 bits wide and can store values in the range: -2,147,483,648 to 2,147,483,627

A signed char is 8 bits wide and can store values in the range: -128 to 127

What happens when you try to store a value that is too large for the data type? With unsigned values, it just "wraps" back around to 0. Think of the bits being sort of like an odometer on a car. Once the odometer gets to 999999, it will "wrap" back around to 0. So, an unsigned char with a value of 255 will become 0:

```
 11111111
+       1
---------
000000000
```

With signed numbers, the result is undefined. It could be anything and do anything, including crashing the program. It's up to the particular compiler. The GNU gcc compiler does a sort of "wrapping" itself. The difference is that instead of going from the largest positive value back to 0, the bits go from the largest positive value to the smallest negative value. (e.g. 127 + 1 is -128).

Overflow example

Literal Constants

We know that a literal constant like 42 is an **`int`** and that a literal constant like 42.0 is a **`double`**.

- If a literal is too large for an **`int`**, its type will be **`long int`**.
- To force a smaller literal number to a particular type, append a suffix:
  - Append the letter 'F' to a floating point value to make it a **`float`**.

    ```
    426.0F
    ```

  - Append the letter 'L' to an integral value to make it a **`long int`**. (Never use lowercase.)

    ```
    426L vs. 426l
    ```

  - Append the letter 'U' to an integral value to make it an **`unsigned int`**.

    ```
    426U
    ```

  - You can combine them to make an **`unsigned long int`**. (The order doesn't matter)

    ```
    425UL or 426LU
    ```

- Also, be aware that in C, the type of a literal **`char`** is an **`int`**:

    ```
    char c = 'A';
    printf("sizeof(c)   is %2i\n", sizeof(c));   /* char variable */
    printf("sizeof('A') is %2i\n", sizeof('A')); /* char literal  */
    ```

  Output:

    ```
    sizeof(c)   is  1
    sizeof('A') is  4
    ```

Don't forget that when you are reading/writing (**`scanf`** and **`printf`**) shorts and longs that you need to use modifiers on the type. **`h`** for *short* and **`l`** (lowercase L) for *long*. Refer to them here.

Usually, we write literal integral values using decimal (base 10) notation. C provides two other forms: octal (base 8) and hexadecimal (base 16)

- Numbers in octal (base 8) can only use the digits 0..7
- Octal numbers can be disguished from decimal and hexadecimal by using a leading zero:

    ```
    01  014  077  01472  077634L  03421U
    ```

- Numbers in hexadecimal (base 16) use the digits 0..9 and then the letters A..F
- Hexadecimal (or just hex) numbers have a leading 0x (a zero followed by the letter X)

    ```
    0x10  0X10  0x14  0x17AF  0xFFFF  0xabF10CD8L  0xFFFFU
    ```

- Octal and hexadecimal are unsigned only.

Floating Point Types

Unlike the integral types, floating point types are not divided into signed and unsigned. All floating point types are signed only. Floating point numbers follow the IEEE-754 Floating Point Standard. Here's more information about floating point numbers than you'll probably ever need.

Fun with floating point numbers

Here are the approximate ranges of the IEEE-754 floating point numbers on Intel x86 computers:

Type	Size	Smallest Postive Value	Largest Positive Value	Precision
`float`	4	1.1754×10^{-38}	3.4028×10^{38}	6 digits
`double`	8	2.2250×10^{-308}	1.7976×10^{308}	15 digits
`long double`	10*	3.3621×10^{-4932}	1.1897×10^{4932}	19 digits

Some floating point constants. These are all of type **`double`**:

```
42.0  42.0e0  42.  4.2e1  4.2E+1  .42e2  420.e-1  42e0  42.E0
```

To indicate that the type is **`float`**, you must append the letter `f` or `F`:

```
42.0f  42.0e0f  42.F  4.2e1F  etc...
```

To indicate that the type is **`long double`**, you must append the letter `l` (lowercase 'L') or `L`:

```
42.0L  42.0e0L  42.l  4.2e1l  etc...
```

In practice, **NEVER** use the lowercase L (which looks very similar to the number one: 1), as it will certainly cause confusion. (See above.)

*Here are the sizes of floating point numbers on various C compilers:

GNU gcc	Borland	Microsoft
sizeof(42.0) is 8	sizeof(42.0) is 8	sizeof(42.0) is 8
sizeof(42.0F) is 4	sizeof(42.0F) is 4	sizeof(42.0F) is 4
sizeof(42.0L) is 12	sizeof(42.0L) is 10	sizeof(42.0L) is 8

Partial float.h listing

Another toy

The `typedef` Keyword

Suppose we want to add a boolean type to C. (There isn't one, so we typically use **`int`** in place of a boolean.) We've already done it using `#define`: here

To declare a variable, we simply do this:

```
int a;             /* Create an integer named a               */
unsigned char b;   /* Create an unsigned char named b         */
short int c;       /* Create a short integer named c          */
float d;           /* Create a float named d                  */
unsigned char * e; /* Create an unsigned char pointer named e */
```

These cause the compiler to allocate space for each variable, based on it's type.

If we want to create a new type (instead of a new variable), we add the `typedef` keyword:

```
typedef int a;             /* Create a new type named a */
typedef unsigned char b;   /* Create a new type named b */
```

```
typedef short int c;       /* Create a new type named c */
typedef float d;           /* Create a new type named d */
typedef unsigned char * e; /* Create a new type named e */
```

You can think of these type definitions as *aliases* for other types. To create a new variable of type `a`:

```
a i; /* Create an 'a' variable named i */
b j; /* Create a 'b' variable named j  */
```

Of course, thinks makes no sense whatsoever. For any real use, you need to give the typedefs meaningful names. Compare to `#define`:

```
  /* Create new types using typedef */
typedef int Bool;
typedef unsigned char BYTE;
typedef short int FAST_INT;
typedef float CURRENCY;
typedef unsigned char * PCHAR;
```

```
  /* Create new types using #define */
#define Bool int
#define BYTE unsigned char
#define FAST_INT short int
#define CURRENCY float
#define PCHAR unsigned char *
```

Examples:

```
Bool playing, paused;   /* Booleans for a DVD player    */
BYTE next, previous;    /* For scanning bytes in memory */
CURRENCY tax, discount; /* To calculate total price     */
PCHAR inbuf, outbuf;    /* To manipulate strings        */
```

Summary:

- Use `typedef` when you want to create an alias for another type (easier to change later)
- Use `typedef` to simplify the name of a type

```
  /* Each is an array of 10 unsigned char pointers */
unsigned char *a[10];
unsigned char *b[10];
unsigned char *c[10];
unsigned char *d[10];
```

This is the same:

```
typedef unsigned char *Strings[10]; /* Strings is a new type */

  /* An array of 10 unsigned char pointers */
Strings a, b, c, d;
```

- Unlike `#define`, the `typedef` keyword obeys the scope rules.

```
void foo(void)
{
  typedef int Bool; /* Is visible only in this function            */
  #define BOOL int  /* Is visible in every function below this one */

  if (/* whatever */)
  {
    typedef int INT32; /* Visible only in if */
    /* Other stuff */
  }
  /* Other stuff */
}
```

- `#define` is an unsophisticated "search and replace" by the preprocessor:

```
#define CPTR1 char *
typedef char * CPTR2;
```

```
CPTR1 p1, p2; /* What is the type of p1 and p2? */
CPTR2 p3, p4; /* What is the type of p1 and p2? */

printf("%i, %i\n", sizeof(p1), sizeof(p2));
printf("%i, %i\n", sizeof(p3), sizeof(p4));
```
