Chapter 3 | Data Types

CS185

Data Types

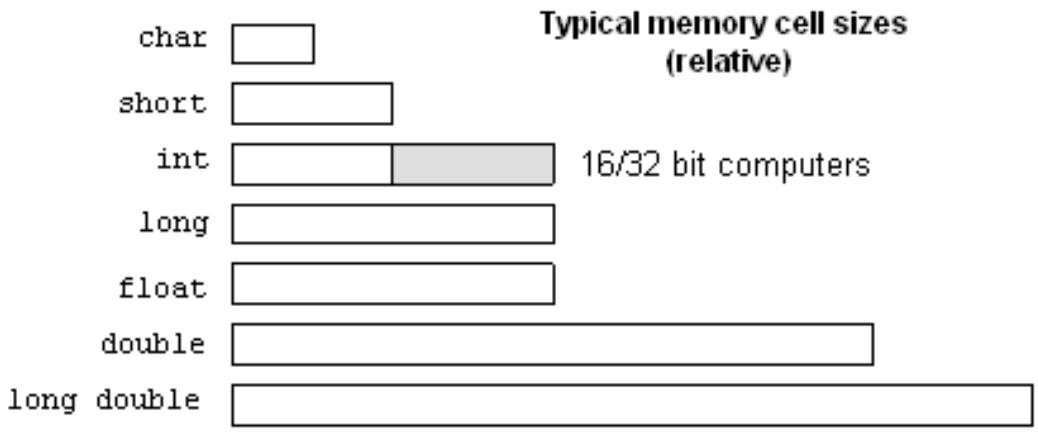
Integral Types

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 are using follow the relative sizes below:



This table shows the range of values for the integral types:

Type	Also called	Bytes
signed char	char (compiler dependent)	1
unsigned char	char (compiler dependent)	1
signed short int	short short int signed short	Not smaller than char. At least 16 bits.
unsigned short int	unsigned short	Same as short
signed int	int signed	Not smaller than short. At least 16 bits.
unsigned int	unsigned	Same as int
signed long int	long long int signed long	Not smaller than int. At least 32 bits.
usigned long int	unsigned long	Same as long
signed long long int	long long long long int signed long long	Not smaller than long. At least 64 bits.
usigned long long int	unsigned long long	Same as long long

One rule has to be followed (in C and C++):

```
sizeof(char) <= sizeof(short int) <= sizeof(int) <= sizeof(long int)
```

This table shows how to compute the range of any data type:

Data Type	Range Equation
Any signed data type with size "n" bits	2^{n-1} to $(2^{n-1} - 1)$
Any unsigned data type with size "n" bits	0 to $(2^n - 1)$

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 Microsoft 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).

Code	`#include <iostream>`
	
	`int main(void)`
	`{`
	`  unsigned char uc = -1;`
	`  std::cout << "uc = " << (int)uc << std::endl;`
	
	`  signed char sc = 127;`
	`  sc++;`
	`  std::cout << "sc = " << (int)sc << std::endl;`
	
	`  system("pause");`
	`}`

Output	uc = 255 sc = -128

Floating Point Types

Unlike the integral types, floating point types are not divided into signed and unsigned. All floating point types are signed only.

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: `l`), as it will certainly cause confusion. (See above.)

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

GNU g++	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`

Boolean Type

The boolean type, known in C++ as bool, can only represent one of two states, **true** or **false**.

```
bool b = true;  /* Create an integer named b and set its value to true */
```

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
```

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 disguised 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.

The `typedef` Keyword

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                   */
```

These cause the compiler to allocate space for each variable, based on its type.

If we want to create a new type (instead of a new variable), we use 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 */
```

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, things makes no sense whatsoever. For any real use, you need to give the typedefs meaningful names.

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

Examples:

```
BYTE next, previous;    /* For scanning bytes in memory */
CURRENCY tax, discount; /* To calculate total price     */
```

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;
```

- 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 */
}
```