| CS13002 Programming and Data Structures | Spring semester |
|---|---|

# Variables and simple data types

The first abstraction a high-level language (like C) offers is a way of structuring data. A machine's memory is a flat list of memory cells, each of a fixed size. The abstraction mechanism gives special interpretation to collections of cells. Think of a collection of blank papers glued (or stapled) together. A piece of blank paper is a piece of paper, after all. However, when you see the neatly bound object, you leap up in joy and assert, "Oh, that's my note book!" This is abstraction. Papers remain papers and their significance in a note book is in no way diminished. A special meaning of the collection is a thing that is rendered by the abstraction. There is another point here -- usage convenience. You would love to take class notes in a note book instead of in loose sheets. A note book is abstract in yet another sense. You call it a note book irrespective of the size and color of the papers, of whether there are built-in lines on the papers, of what material is used to manufacture the papers, etc.

The basic unit for storage of digital data is called a **bit**. It is an object that can assume one of the two possible values "0" and "1". Depending on how one is going to implement a bit, the values "0" and "1" are defined. If a capacitor stands for a bit, you may call its state "0" if the charge stored in it is less than 0.5 Volt, else you call its state "1". For a switch, "1" may mean "on" and "0" then means "off". Let us leave these implementation details to material scientists and VLSI designers. For us it is sufficient to assume that a computer comes with a memory having a huge number of built-in bits.

A single bit is too small a unit to be adequately useful. A collection of bits is what a practical unit for a computer's operation is. A **byte** (also called an **octet**) is a collection of eight bits. Bigger units are also often used. In many of today's computers data are transfered and processed in chunks of 32 bits (4 bytes). Such an operational unit is often called a **word**. Machines supporting 64-bit words are also coming up and are expected to replace 32-bit machines in near future.

## Basic data types

Bytes (in fact, bits too) are abstractions. Still, they are pretty raw. We need to assign special meanings to collections of bits in order that we can use those collections to solve our problems. For example, a matrix inversion routine deals with matrices each of whose elements is a real (or rational or complex) number. We then somehow have to map memory contents to numbers, to matrices, to pairs of real numbers (complex numbers), and so on. Luckily enough, a programmer does not have to do this mapping himself/herself. The C compiler already provides the abstractions you require. It is the headache of the compiler how it would map your abstract entities to memory cells in your machine. You, in your turn, must understand the abstraction level which is provided to you for writing programs in C.

For the time being, we will look at the basic data types supported by C. We will later see how these individual data types can be glued together to form more structured data types. Back to our note book example. A paper is already an abstraction, it's not any collection of electrons, protons and neutrons. So let us first understand what a paper is and what we can do with a piece of paper. We will later investigate how we can manufacture note books from papers, and racks from note books, and book-shelfs from racks, drawers, locks, keys and covers.

### Integer data types

Integers are whole numbers that can assume both positive and negative values, i.e., elements of the set:

```
{ ..., -3, -2, -1, -, 1, 2, 3, ... }
```

This set is infinite, both the ellipses extending *ad infinitum*. C's built-in integer data types do not assume all possible integral values, but values between a minimum bound and a maximum bound. This is a pragmatic and historical definition of integers in C. The reason for these bounds is that C uses a fixed amount of memory for each individual integer. If that size is 32 bits, then only $2^{32}$ integers can be represented, since each bit has only two possible states.

| Integer data type | Bit size | Minimum value | Maximum value |
|---|---|---|---|
| `char` | 8 | $-2^7$=-128 | $2^7$-1=127 |
| `short int` | 16 | $-2^{15}$=-32768 | $2^{15}$-1=32767 |
| `int` | 32 | $-2^{31}$=-2147483648 | $2^{31}$-1=2147483647 |
| `long int` | 32 | $-2^{31}$=-2147483648 | $2^{31}$-1=2147483647 |
| `long long int` | 64 | $-2^{63}$=-9223372036854775808 | $2^{63}$-1=9223372036854775807 |
| `unsigned char` | 8 | 0 | $2^8$-1=255 |
| `unsigned short int` | 16 | 0 | $2^{16}$-1=65535 |

| | | | |
|---|---|---|---|
| `unsigned int` | 32 | 0 | $2^{32}$-1=4294967295 |
| `unsigned long int` | 32 | 0 | $2^{32}$-1=4294967295 |
| `unsigned long long int` | 64 | 0 | $2^{64}$-1=18446744073709551615 |

**Notes**

- The term `int` may be omitted in the long and short versions. For example, `long int` can also be written as `long`, `unsigned long long int` also as `unsigned long long`.
- ANSI C prescribes the exact size of `int` (and `unsigned int`) to be either 16 bytes or 32 bytes, that is, an `int` is either a `short int` or a `long int`. Implementers decide which size they should select. Most modern compilers of today support 32-bit `int`.
- The `long long` data type and its `unsigned` variant are not part of ANSI C specification. However, many compilers (including gcc) support these data types.

## Float data types

Like integers, C provides representations of real numbers and those representations are finite. Depending on the size of the representation, C's real numbers have got different names.

| Real data type | Bit size |
|---|---|
| float | 32 |
| double | 64 |
| long double | 128 |

## Character data types

We need a way to express our thoughts in writing. This has been traditionally achieved by using an alphabet of symbols with each symbol representing a sound or a word or some punctuation or special mark. The computer also needs to communicate its findings to the user in the form of something written. Since the outputs are meant for human readers, it is advisable that the computer somehow translates its bit-wise world to a human-readable script. The Roman script (mistakenly also called the English script) is a natural candidate for the representation. The Roman alphabet consists of the lower-case letters (a to z), the upper case letters (A to Z), the numerals (0 through 9) and some punctuation symbols (period, comma, quotes etc.). In addition, computer developers planned for inclusion of some more control symbols (hash, caret, underscore etc.). Each such symbol is called a **character**.

In order to promote interoperability between different computers, some standard encoding scheme is adopted for the computer character set. This encoding is known as **ASCII** (abbreviation for **American Standard Code for Information Interchange**). In this scheme each character is assigned a unique integer value between 32 and 127. Since eight-bit units (bytes) are very common in a computer's internal data representation, the code of a character is represented by an 8-bit unit. Since an 8-bit unit can hold a total of $2^8$=256 values and the computer character set is much smaller than that, some values of this 8-bit unit do not correspond to visible characters. These values are often used for representing invisible control characters (like line feed, alarm, tab etc.) and extended Roman letters (inflected letters like ä, é, ç). Some values are reserved for possible future use. The ASCII encoding of the printable characters is summarized in the following table.

| Decimal | Hex | Binary | Character | | Decimal | Hex | Binary | Character |
|---|---|---|---|---|---|---|---|---|
| 32 | 20 | 00100000 | SPACE | | 80 | 50 | 01010000 | P |
| 33 | 21 | 00100001 | ! | | 81 | 51 | 01010001 | Q |
| 34 | 22 | 00100010 | " | | 82 | 52 | 01010010 | R |
| 35 | 23 | 00100011 | # | | 83 | 53 | 01010011 | S |
| 36 | 24 | 00100100 | $ | | 84 | 54 | 01010100 | T |
| 37 | 25 | 00100101 | % | | 85 | 55 | 01010101 | U |
| 38 | 26 | 00100110 | & | | 86 | 56 | 01010110 | V |
| 39 | 27 | 00100111 | ' | | 87 | 57 | 01010111 | W |
| 40 | 28 | 00101000 | ( | | 88 | 58 | 01011000 | X |
| 41 | 29 | 00101001 | ) | | 89 | 59 | 01011001 | Y |
| 42 | 2a | 00101010 | * | | 90 | 5a | 01011010 | Z |
| 43 | 2b | 00101011 | + | | 91 | 5b | 01011011 | [ |
| | | | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 44 | 2c | 00101100 | , | | 92 | 5c | 01011100 | \ |
| 45 | 2d | 00101101 | - | | 93 | 5d | 01011101 | ] |
| 46 | 2e | 00101110 | . | | 94 | 5e | 01011110 | ^ |
| 47 | 2f | 00101111 | / | | 95 | 5f | 01011111 | _ |
| 48 | 30 | 00110000 | 0 | | 96 | 60 | 01100000 | ` |
| 49 | 31 | 00110001 | 1 | | 97 | 61 | 01100001 | a |
| 50 | 32 | 00110010 | 2 | | 98 | 62 | 01100010 | b |
| 51 | 33 | 00110011 | 3 | | 99 | 63 | 01100011 | c |
| 52 | 34 | 00110100 | 4 | | 100 | 64 | 01100100 | d |
| 53 | 35 | 00110101 | 5 | | 101 | 65 | 01100101 | e |
| 54 | 36 | 00110110 | 6 | | 102 | 66 | 01100110 | f |
| 55 | 37 | 00110111 | 7 | | 103 | 67 | 01100111 | g |
| 56 | 38 | 00111000 | 8 | | 104 | 68 | 01101000 | h |
| 57 | 39 | 00111001 | 9 | | 105 | 69 | 01101001 | i |
| 58 | 3a | 00111010 | : | | 106 | 6a | 01101010 | j |
| 59 | 3b | 00111011 | ; | | 107 | 6b | 01101011 | k |
| 60 | 3c | 00111100 | < | | 108 | 6c | 01101100 | l |
| 61 | 3d | 00111101 | = | | 109 | 6d | 01101101 | m |
| 62 | 3e | 00111110 | > | | 110 | 6e | 01101110 | n |
| 63 | 3f | 00111111 | ? | | 111 | 6f | 01101111 | o |
| 64 | 40 | 01000000 | @ | | 112 | 70 | 01110000 | p |
| 65 | 41 | 01000001 | A | | 113 | 71 | 01110001 | q |
| 66 | 42 | 01000010 | B | | 114 | 72 | 01110010 | r |
| 67 | 43 | 01000011 | C | | 115 | 73 | 01110011 | s |
| 68 | 44 | 01000100 | D | | 116 | 74 | 01110100 | t |
| 69 | 45 | 01000101 | E | | 117 | 75 | 01110101 | u |
| 70 | 46 | 01000110 | F | | 118 | 76 | 01110110 | v |
| 71 | 47 | 01000111 | G | | 119 | 77 | 01110111 | w |
| 72 | 48 | 01001000 | H | | 120 | 78 | 01111000 | x |
| 73 | 49 | 01001001 | I | | 121 | 79 | 01111001 | y |
| 74 | 4a | 01001010 | J | | 122 | 7a | 01111010 | z |
| 75 | 4b | 01001011 | K | | 123 | 7b | 01111011 | { |
| 76 | 4c | 01001100 | L | | 124 | 7c | 01111100 | \| |
| 77 | 4d | 01001101 | M | | 125 | 7d | 01111101 | } |
| 78 | 4e | 01001110 | N | | 126 | 7e | 01111110 | ~ |
| 79 | 4f | 01001111 | O | | 127 | 7f | 01111111 | DELETE |

Table : The ASCII values of the printable characters

C data types are necessary to represent characters. As told earlier, an eight-bit value suffices. The following two built-in data types are used for characters.

```
char
unsigned char
```

Well, I mentioned earlier that these are integer data types. I continue to say so. These are both integer and character data types. If you want to interpret a `char` value as a character, you see the character it represents. If you want to view it as an integer, you see the ASCII value of that character. For example, the upper case A has an ASCII value of 65. An eight-bit value representing the character A automatically represents the integer 65, because to the computer A is recognized by its ASCII code, not by its shape, geometry or sound!

## Pointer data types

Pointers are addresses in memory. In order that the user can directly manipulate memory addresses, C provides an abstraction of addresses. The memory location where a data item resides can be accessed by a pointer to that particular data type. C uses the special character * to declare pointer data types. A pointer to a `double` data is of data type `double *`. A pointer to an `unsigned long int` data is of type `unsigned long int *`. A character pointer has the data type `char *`. We will study pointers more elaborately later in this course.

# Constants

Having defined data types is not sufficient. We need to work with specific instances of data of different types. Thus we are not much interested in defining an abstract class of objects called integers. We need specific instances like 2, or -496, or +1234567890. We should not feel extravagantly elated just after being able to define an abstract entity called a house. We need one to live in.

Specific instances of data may be constants, i.e., values that do not change during the execution of programs. For example, the mathematical pi remains constant throughout every program, and expectedly throughout our life-time too. Similarly, when we wrote `1.0/n` to compute reciprocals, we used the constant `1.0`.

Constants are written much in the same way as they are written conventionally.

## Integer constants

An integer constant is a non-empty sequence of decimal numbers preceded optionally by a sign (+ or -). However, the common practice of using commas to separate groups of three (or five) digits is not allowed in C. Nor are spaces or any character other than numerals allowed. Here are some valid integer constants:

```
332
-3002
+15
-00001020304
```

And here are some examples that C compilers do not accept:

```
3 332
2,334
- 456
2-34
12ab56cd
```

You can also express an integer in base 16, i.e., an integer in the **hexadecimal** (abbreviated **hex**) notation. In that case you must write either `0x` or `0X` before the integer. Hexadecimal representation requires 16 digits 0,1,...,15. In order to resolve ambiguities the digits 10,11,12,13,14,15 are respectively denoted by `a,b,c,d,e,f` (or by `A,B,C,D,E,F`). Here are some valid hexadecimal integer constants:

```
0x12ab56cd
-0X123456
0xABCD1234
+0XaBCd12
```

Since different integer data types use different amounts of memory and represent different ranges of integers, it is often convenient to declare the intended data type explicitly. The following suffixes can be used for that:

| Suffix | Data type |
|---|---|
| `L` (or `l`) | `long` |
| `LL` (or `ll`) | `long long` |
| `U` (or `u`) | `unsigned` |
| `UL` (or `ul`) | `unsigned long` |
| `ULL` (or `ull`) | `unsigned long long` |

Here are some specific examples:

```
4000000000UL
123U
-0x7FFFFFFFl
0x123456789abcdef0ULL
```

## Real constants

Real constants can be specified by the usual notation comprising an optional sign, a decimal point and a sequence of digits. Like integers no other characters are allowed. Here are some specific examples:

```
1.23456
1.
.1
-0.12345
+.4560
```

And here are some non-examples (invalid real constants):

```
.
- 1.23
1 234.56
1,234.56
1.234.56
```

Real numbers are sometimes written in the *scientific notation* (like $3.45 \times 10^{67}$). The following expressions are valid for writing a real number in this fashion:

```
3.45e67
+3.45e67
-3.45e-67
.00345e-32
1e-15
```

You can also use `E` in place of `e` in this notation.

### Character constants

Character constants are single printable symbols enclosed within single quotes. Here are some examples:

```
'A'
'7'
'@'
' '
```

There are some special characters that require you to write more than one printable characters within the quotes. Here is a list of some of them:

| Constant | Character | ASCII value |
|---|---|---|
| `'\0'` | Null | 0 |
| `'\b'` | Backspace | 8 |
| `'\t'` | Tab | 9 |
| `'\n'` | New line | 13 |
| `'\''` | Quote | 39 |
| `'\\'` | Backslash | 92 |

Since characters are identified with integers in the range -127 to 128 (or in the range 0 to 255), you can use integer constants in the prescribed range to denote characters. The particular sequence `'\xuv'` (synonymous with `0xuv`) lets you write a character in the hex notation. (Here `u` and `v` are two hex digits.) For example, `'\x2b'` is the integer 43 in decimal notation and stands for the character '+'.

### Pointer constants

Well, there are no pointer constants actually. It is dangerous to work with constant addresses. You may anyway use an integer as a constant address. But doing that lets the compiler issue you a warning message. Finally, when you run the program and try to access memory at a constant address, you are highly likely to encounter a frustrating mishap known as "Segmentation fault". That's a deadly enemy. Try to avoid it as and when you can!

Incidentally, there is a pointer constant that is used widely. This is called `NULL`. A `NULL` pointer points to nowhere.

## Variables

Constants are not always sufficient to reflect reality. Though I am a constant human being and your constant PDS teacher, I am not a constant teacher for you or this classroom. Your or V1's teacher changes with time, though at any particular instant it assumes a constant value. A variable data is used to portray this scenario.

A variable is specified by a name given to a collection of memory locations. Named variables are useful from two considerations:

- Variables bind particular areas in the memory. You can access an area by a name and not by its explicit address. This abstraction simplifies a programmer's life dramatically. (If you want to tell a story to your friend about your pet, you would like to use its name instead of holding the constant object all the time in front of your friend's bored eyes.)
- Names promote parameterized computation. You change the value of a variable and obtain a different output. For example, the polynomial $2a^2+3a-4$ evaluates to different values, as you plug in different values for the variable a.

Of course, the particular name a is symbolic here and can be replaced by any other name (b,c etc.), but the formal naming of the parameter allows you to write (and work with) the function symbolically.

## Naming conventions

C does not allow any sequence of characters as the name of a variable. This kind of practice is not uncommon while naming human beings too. However, C's naming conventions are somewhat different from human conventions. To C, a legal name is any name prescribed by its rules. There is no question of aesthetics or meaning or sweet-sounding-ness.

You would probably not name your (would-be) baby as "123abc". C also does not allow this name. However, C allows the name "abc123". One usually does not see a human being with this name. But then, have you heard of "Louis XVI"?

Well, you may rack your brain for naming your baby. Here are C's straightforward rules.

- Any sequence of alphabetic characters (lower-case a to z and upper-case A to Z) and numerals (0 through 9) and underscore (_) can be a valid name, provided that:
    - The name does not start with a numeral.
    - The name does not coincide with one of C's *reserved words* (like `double`, `unsigned`, `for`). These words have special meanings to the compilers and so are not allowed for your variables.
    - The name does not coincide with the same name of another entity (declared in the same scope).
    - The name does not contain any character other than those mentioned above.
- C's naming scheme is **case-sensitive**, i.e., `teacher`, `Teacher`, `TEACHER`, `TeAcHeR` are all different names.
- C does not impose any restriction on what name goes to what type of data. The name `fraction` can be given to an `int` variable and the name `submerged` can be given to a `float` variable.
- There is no restriction on the minimum length (number of characters) of a name, as long as the name is not the empty string.
- Some compilers impose a restriction on the maximum length of a name. Names bigger than this length are truncated to the maximum allowed leftmost part. This may lead to unwelcome collisions in different names. However, this upper limit is usually quite large, much larger than common names that we give to variables.

In C, names are given to other entities also, like functions, constants. In every case the above naming conventions must be adhered to.

## Declaring variables

For declaring one or more variables of a given data type do the following:

- First write the data type of the variable.
- Then put a space (or any other white character).
- Then write your comma-separated list of variable names.
- At the end put a semi-colon.

Here are some specific examples:

```
int m, n, armadillo;
int platypus;
float hi, goodMorning;
unsigned char _u_the_charcoal;
```

You may also declare pointers simultaneously with other variables. All you have to do is to put an asterisk (*) before the name of each pointer.

```
long int counter, *pointer, *p, c;
float *fptr, fval;
double decker;
double *standard;
```

Here `counter` and `c` are variables of type `long int`, whereas `pointer` and `p` are pointers to data of type `long int`. Similarly, `decker` is a `double` variable, whereas `standard` is a pointer to a `double` data.

### Initializing variables

Once you declare a variable, the compiler allocates the requisite amount of memory to be accessed by the name of the variable. C does not make any attempt to fill that memory with any particular value. You have to do it explicitly. An uninitialized memory may contain any value (but it must contain some value) that may depend on several funny things like how long the computer slept after the previous shutdown, how much you have browsed the web before running your program, or may be even how much dust has accumulated on the case of your computer.

We will discuss in the next chapter how variables can be assigned specific values. For the time being, let us investigate the possibility that a variable can be initialized to a *constant* value at the time of its declaration. For achieving that you should put an equality sign immediately after the name followed by a constant value before closing the declaration by a comma or semicolon.

```
int dint = 0, hint, mint = -32;
char *Romeo, *Juliet = NULL;
float gloat = 2e-3, throat = 3.1623, coat;
```

Here the variable `dint` is initialized to 0, `mint` to -32, whereas `hint` is not initialized. The `char` pointer Romeo is not initialized, whereas Juliet is initialized to the `NULL` pointer.

Notice that uninitialized (and unassigned) variables may cause enough sufferings to a programmer. Take sufficient care!

### Names of constants

So far we have used immediate constants that are defined and used in place. In order to reuse the same immediate constant at a different point in the program, the value must again be explicitly specified.

C provides facilities to name constant values (like variables). Here we discuss two ways of doing it.

#### Constant variables

Variables defined as above are read/write variables, i.e., one can both read their contents and store values in them. Constant variables are read-only variables and can be declared by adding the reserved word `const` before the data type.

```
const double pi = 3.1415926535;
const unsigned short int perfect1 = 6, perfect2 = 28, perfect3 = 496;
```

These declarations allocate space and initialize the variables like variable variables, but don't allow the user to alter the value of `PI`, `perfect1` etc. at a later time during the execution of the program.

#### `#define`'d constants

These are not variables. These are called macros. If you `#define` a value against a name and use that name elsewhere in your program, the name is literally substituted by the C preprocessor, before your code is compiled. Macros do not reside in the memory, but are expanded well before any allocation attempt is initiated.

```
#define PI 3.1415926535
#define PERFECT1 6
#define PERFECT2 28
#define PERFECT3 496
```

Look at the differences with previous declarations. First, only one macro can be defined in a single line. Second, you do not need the semicolon or the equality sign for defining a macro.

Parameterized macros can also be defined, but unless you fully understand what a macro means and how parameters are handled in macros, don't use them. Just a wise tip, I believe! You can live without them.

## Typecasting

An integer can naturally be identified with a real number. The converse is not immediate. However, we can adopt some convention regarding conversion of a real number to an integer. Two obvious candidates are truncation and rounding. C opts for truncation.

In order to convert a value `<val>` of any type to a value of `<another_type>` use the following directive:

```
(<another_type>)<val>
```

Here `<val>` may be a constant or a value stored in a named variable. In the examples below we assume that piTo4 is a `double` variable that stores the value 97.4090910340.

| Typecasting command | Output value |
|---|---|
| `(int)9.8696044011` | The truncated integer 9 |
| `(int)-9.8696044011` | The truncated integer -9 |
| `(float)9` | The floating-point value 9.000000 |
| `(int)piTo4` | The integer 97 |
| `(char)piTo4` | The integer 97, or equivalently the character 'a' |
| `(int *)piTo4` | An integer pointer that points to the memory location 97. |
| `(double)piTo4` | The same value stored in `piTo4` |

Typecasting also applies to expressions and values returned by functions.

# Representation of numbers in memory

## Binary representation

Computer's world is binary. Each computation involves manipulating a series of bits each realized by some mechanism that can have two possible states denoted "0" and "1". If that is the case, integers, characters, floating point numbers need also be represented by bits. Here is how this representation can be performed.

For us, on the other hand, it is customary to have 10 digits in our two hands and consequently 10 digits in a number system. The decimal system is natural. Not really, it is just the convention. From our childhood we have been taught to use base ten representations to such an extent that it is difficult to conceive of alternatives, in fact to even think that any natural number greater than 1 can be a legal base for number representation. (There also exists an "exponentially big" *unary representation* of numbers that uses only one digit better called a "symbol" now.)

### Binary expansion of integers

Let's first take the case of non-negative integers. In order to convert such an integer n from the decimal representation to the binary representation, one keeps on dividing n by 2 and remembering the intermediate remainders obtained. When n becomes 0, we have to write the remainders in the reverse sequence as they are generated. That's the original n in binary.

| | n | Remainder |
|---|---|---|
| | 57 | |
| Divide by 2 | 28 | 1 |
| Divide by 2 | 14 | 0 |
| Divide by 2 | 7 | 0 |
| Divide by 2 | 3 | 1 |
| Divide by 2 | 1 | 1 |
| Divide by 2 | 0 | 1 |

$57 = (111001)_2$

For computers, we usually also specify a size t of the binary representation. For example, suppose we want to represent 57 as an `unsigned char`, i.e., as an 8-bit value. The above algorithm works fine, but we have to

- either insert the requisite number of leading zero bits,
- or repeat the "divide by 2" step exactly t times without ever looking at whether the quotient has become 0.

| | n | Remainder |
|---|---|---|
| | 57 | |
| Divide by 2 | 28 | 1 |
| Divide by 2 | 14 | 0 |
| Divide by 2 | 7 | 0 |
| Divide by 2 | 3 | 1 |
| Divide by 2 | 1 | 1 |

| | | |
|---|---|---|
| Divide by 2 | 0 | 1 |
| Divide by 2 | 0 | 0 |
| Divide by 2 | 0 | 0 |

$57 = (00111001)_2$

What if the given n is too big to fit in a t-bit place? Now also you can "divide by 2" exactly t times and read the t remainders backward. That will give you the least significant t bits of n. The remaining more significant bits will simply be ignored.

| | **n** | **Remainder** |
|---|---|---|
| | 657 | |
| Divide by 2 | 328 | 1 |
| Divide by 2 | 164 | 0 |
| Divide by 2 | 82 | 0 |
| Divide by 2 | 41 | 0 |
| Divide by 2 | 20 | 1 |
| Divide by 2 | 10 | 0 |
| Divide by 2 | 5 | 0 |
| Divide by 2 | 2 | 1 |

$657 = (\ldots 10010001)_2$

**Signed magnitude representation of integers**

Now we add provision for sign. Here is how this is conventionally done. In a t-bit signed representation of n:

- The most significant (leftmost) bit is reserved for the sign. "0" means positive, "1" means negative.
- The remaining t-1 bits store the (t-1)-bit representation of the magnitude (absolute value) of n (i.e., of |n|).

**Example:** The 7-bit binary representation of 57 is $(0111001)_2$.

- The 8-bit signed magnitude representation of 57 is $(00111001)_2$.
- The 8-bit signed magnitude representation of -57 is $(10111001)_2$.

**Back to decimal**

Given an integer in unsigned or signed representation, its magnitude and sign can be determined. For the sign, the most significant bit is consulted. For the magnitude, a sum of appropriate powers of 2 is calculated.

Let the magnitude be stored in *l* bits. The bits are numbered 0,1,. . .,*l*-1 from right to left. The i-th position (from the right) corresponds to the power $2^i$. One simply adds the powers of 2 corresponding to those positions that hold 1 bits in the binary representation.

| **Signed integer** | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|
| **Position** | Sign | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **Contribution** | + | 0 | $2^5$ | $2^4$ | $2^3$ | 0 | 0 | $2^0$ |

$+(2^5+2^4+2^3+2^0) = +(32+16+8+1) = +57$

| **Signed integer** | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|
| **Position** | Sign | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **Contribution** | - | 0 | 0 | $2^4$ | 0 | 0 | 0 | $2^0$ |

$-(2^4+2^0) = -(16+1) = -17$

| **Unsigned integer** | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|
| **Position** | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **Contribution** | $2^7$ | 0 | 0 | $2^4$ | 0 | 0 | 0 | $2^0$ |

$2^7+2^4+2^0 = 128+16+1 = 145$

**Notes:**

- The t-bit unsigned representation can accommodate integers in the range 0 to $2^t$-1.
- The t-bit signed magnitude representation can accommodate integers in the range $-(2^{t-1}-1)$ to $+(2^{t-1}-1)$.
- In the signed magnitude representation 0 has two renderings: +0 = 0000...0 and -0=1000...0.

## 1's complement representation

1's complement of a t-bit sequence $(a_{t-1}a_{t-2}...a_0)_2$ is the t-bit sequence $(b_{t-1}b_{t-2}...b_0)_2$, where for each i we have $b_i = 1 - a_i$, i.e., $b_i$ is the bit-wise complement of $a_i$. Here $(b_{t-1}b_{t-2}...b_0)_2 = 2^t-1-(a_{t-1}a_{t-2}...a_0)_2$.

The t-bit 1's complement representation of an integer n is a t-bit signed representation with the following properties:

- The most significant (leftmost) bit is the sign bit, 0 if n is positive, 1 if n is negative.
- The remaining t-1 bits are used to stand for the absolute value |n|.
  - If n is positive, these t-1 bits hold the (t-1)-bit binary representation of |n|.
  - If n is negative, these t-1 bits hold the (t-1)-bit 1's complement of |n|.

**Example:** The 7-bit binary representation of 57 is $(0111001)_2$. The 7-bit 1's complement of 57 is $(1000110)_2$.

- The 1's complement representation of +57 is $(00111001)_2$.
- The 1's complement representation of -57 is $(11000110)_2$.

**Notes:**

- The t-bit 1's complement representation can accommodate integers in the range $-(2^{t-1}-1)$ to $+(2^{t-1}-1)$.
- 0 has two representations: +0 = $(0000...0)_2$ and -0 = $(1111...1)_2$.

## 2's complement representation

The t-bit 2's complement of a positive integer n is 1 plus the t-bit 1's complement of n. Thus one first complements each bit in the t-bit binary expansion of n, and then adds 1 to this complemented number. If $n = (a_{t-1}a_{t-2}...a_0)_2$, then its t-bit 1's complement is $(b_{t-1}b_{t-2}...b_0)_2$ with each $b_i = 1 - a_i$, and therefore the 2's complement of n is $n' = 1+(b_{t-1}b_{t-2}...b_0)_2 = 1+(2^t-1)-n = 2^t-n$. In order that $n'$ fits in t-bits we then require $0<=n'<=2^t-1$, i.e., $1<=n<=2^t$.

The t-bit 2's complement representation of an integer n is a t-bit signed representation with the following properties:

- The most significant (leftmost) bit is the sign bit, 0 if n is positive, 1 if n is negative.
- The remaining t-1 bits are used to stand for the absolute value |n|.
  - If n is positive, these t-1 bits hold the (t-1)-bit binary representation of |n|.
  - If n is negative, these t-1 bits hold the (t-1)-bit 2's complement of |n|.

**Example:** The 7-bit binary representation of 57 is $(0111001)_2$. The 7-bit 1's complement of 57 is $(1000110)_2$, so the 7-bit 2's complement of 57 is $(1000111)_2$.

- The 2's complement representation of +57 is $(00111001)_2$.
- The 2's complement representation of -57 is $(11000111)_2$.

**Notes:**

- The t-bit 2's complement representation can accommodate integers in the range $-2^{t-1}$ to $+(2^{t-1}-1)$.
- 0 has only one representation: $(0000\ldots0)_2$.
- The 2's complement representation simplifies implementation of arithmetic (in hardware).

**Example:** The different 8-bit representations of signed integers are summarized in the following table:

| Decimal | Signed magnitude | 1's complement | 2's complement |
|---|---|---|---|
| +127 | 01111111 | 01111111 | 01111111 |
| +126 | 01111110 | 01111110 | 01111110 |
| +125 | 01111101 | 01111101 | 01111101 |
| . . . | . . . | . . . | . . . |
| +3 | 00000011 | 00000011 | 00000011 |
| +2 | 00000010 | 00000010 | 00000010 |
| +1 | 00000001 | 00000001 | 00000001 |
| 0 | 00000000 or 10000000 | 00000000 or 11111111 | 00000000 |
| -1 | 10000001 | 11111110 | 11111111 |
| -2 | 10000010 | 11111101 | 11111110 |
| -3 | 10000011 | 11111100 | 11111101 |
| . . . | . . . | . . . | . . . |
| -126 | 01111110 | 10000001 | 10000010 |
| -127 | 01111111 | 10000000 | 10000001 |
| -128 | No rep | No rep | 10000000 |

## Hexadecimal and octal representations

Similar to binary (base 2) representation, one can have representations of integers in any base B>=2. In computer science two popular bases are 16 and 8. The representation of an integer in base 16 is called the **hexadecimal representation**, whereas that in base 8 is called the **octal representation** of the integer.

For any base B, the base B representation of n can be obtained by successively dividing n by B until the quotient becomes zero. One then writes the remainders in the reverse sequence as they are generated. Since division by B leaves remainders in the range 0,1,. . .,B-1, one requires these many digits for the base B representation. If B=8, the natural (octal) digits are 0,1,. . .,7. For B=16, we have a problem; we now require 16 digits 0,1,. . .,15. Now it is difficult to distinguish, for example, between 13 as a digit and 13 as the digit 1 followed by the digit 3. We use the symbols a,b,c,d,e,f (also in upper case) to stand for the hexadecimal digits 10,11,12,13,14,15.

**Example:** Hexadecimal representation

| | n | Remainder |
|---|---|---|
| | 413657 | |
| Divide by 16 | 25853 | 9 |
| Divide by 16 | 1615 | 13 |
| | | |

| | | |
|---|---|---|
| Divide by 16 | 100 | 15 |
| Divide by 16 | 6 | 4 |
| Divide by 16 | 0 | 6 |

413657 = 0x64fd9

**Example:** Octal representation

| | n | Remainder |
|---|---|---|
| | 413657 | |
| Divide by 8 | 51707 | 1 |
| Divide by 8 | 6463 | 3 |
| Divide by 8 | 807 | 7 |
| Divide by 8 | 100 | 7 |
| Divide by 8 | 12 | 4 |
| Divide by 8 | 1 | 4 |
| Divide by 8 | 0 | 1 |

$413657 = (1447731)_8$

Since 16 and 8 are powers of two, the hexadecimal and octal representations of an integer can also be computed from its binary representation. For the hexadecimal representation, one generates groups of successive 4 bits starting from the right of the binary representation. One may have to add a requisite number of leading 0 bits in order to make the leftmost group contain 4 bits. One 4 bit integer corresponds to an integer in the range 0,1,. . .,15, i.e., to a hexadecimal digit. For the octal representation, grouping should be made three bits at a time.

**Example:** The binary representation of 413657 is $(1100100111111011001)_2$. Arranging this bit-sequence in groups of 4 gives:

```
110 0100 1111 1101 1001
```

Thus 413657 = 0x64fd9, as calculated above.

The grouping with three bits per group is:

```
1 100 100 111 111 011 001
```

Thus $413657 = (1447731)_8$.

## IEEE floating point standard

Now it's time for representing real numbers in binary. Let us first review our decimal intuition. Think of the real number:

$n = 172.93 = 1.7293 \times 10^2 = 0.17293 \times 10^3$

By successive division by 2 we can represent the integer part 172 of n in binary. For the fractional part 0.93 we use repeated multiplication by two in order to get the bits after the binary point. After each multiplication, the integer part of the product generates the next bit in the representation. We then replace the old fractional part by the fractional part of the product.

| | Integral part | Remainder |
|---|---|---|
| | 172 | |
| Divide by 2 | 86 | 0 |
| Divide by 2 | 43 | 0 |
| Divide by 2 | 21 | 1 |
| Divide by 2 | 10 | 1 |
| Divide by 2 | 5 | 0 |
| Divide by 2 | | |

| | Fractional part | Integral part |
|---|---|---|
| | 0.93 | |
| Multiply by 2 | 0.86 | 1 |
| Multiply by 2 | 0.72 | 1 |
| Multiply by 2 | 0.44 | 1 |
| Multiply by 2 | 0.88 | 0 |
| Multiply by 2 | 0.76 | 1 |
| Multiply by 2 | | |

| | | |
|---|---|---|
| | 2 | 1 |
| Divide by 2 | 1 | 0 |
| Divide by 2 | 0 | 1 |

$172 = (10101100)_2$

| | | |
|---|---|---|
| | 0.52 | 1 |
| Multiply by 2 | 0.04 | 1 |
| Multiply by 2 | 0.08 | 0 |
| Multiply by 2 | 0.16 | 0 |
| Multiply by 2 | 0.32 | 0 |

$0.93 = (0.1110111000\ldots)_2$

$$172.93 = (10101100.1110111000\ldots)_2 = (1.01011001110111000\ldots)_2 \times 2^7 = (0.101011001110111000\ldots)_2 \times 2^8$$

It turns out that the decimal fraction 0.93 does not have a terminating binary expansion. So we have to approximate the binary expansion (after the binary point) by truncating the series after a predefined number of bits. Truncating after ten bits gives the approximate value of n to be:

$$\begin{aligned} & (1.0101100111)_2 \times 2^7 \\ = & (2^0 + 2^{-2} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-10}) \times 2^7 \\ = & 2^7 + 2^5 + 2^3 + 2^2 + 2^{-1} + 2^{-2} + 2^{-3} \\ = & 128 + 32 + 8 + 4 + 0.5 + 0.25 + 0.125 \\ = & 172.875 \end{aligned}$$

This example illustrates how to store approximate representations of real numbers using a fixed amount of bits. If we write the expansion in the **normal form** with only one 1 bit (and nothing else) to the left of the binary point, then it is sufficient to store only the fractional part (0101100111 in our example) and the exponent of 2 (7 in the example). This is precisely what is done by the **IEEE 754 floating-point format**. This is a 32-bit representation of signed floating point numbers. The 32 bits are used as follows:

| **31** | **30** | **29** | **...** | **24** | **23** | **22** | **21** | **...** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|
| S | $E_7$ | $E_6$ | ... | $E_1$ | $E_0$ | $M_{22}$ | $M_{21}$ | ... | $M_1$ | $M_0$ |

The meanings of the different parts are as follows:

- S is the sign bit, 0 represents positive, and 1 negative.
- The eight bits $E_7E_6\ldots E_1E_0$ represent the exponent. For usual numbers it is allowed to lie in the range 1 to 254.
- The rightmost 23 bits $M_{22}M_{21}\ldots M_1M_0$ represent the mantissa (also called significand). It is allowed to take any of the $2^{23}$ values between 0 and $2^{23}$-1.

**Normal numbers**

The normal number that this 32-bit value stores is interpreted as:

$$(-1)^S \times (1.M_{22}M_{21}\ldots M_1M_0)_2 \times 2^{[(E_7E_6\ldots E_1E_0)_2 - 127]}$$

The biggest real number that this representation stores corresponds to

```
0 11111110 1111111 11111111 11111111
```

which is approximately $2^{128}$, i.e., $3.403 \times 10^{38}$. The smallest positive value that this format can store corresponds to

```
0 00000001 0000000 00000000 00000000
```

which is

$$1.00000000000000000000000 \times 2^{-126} = 2^{-126},$$

i.e., nearly $1.175 \times 10^{-38}$.

**Denormal numbers**

The IEEE standard also supports a **denormal form**. Now all the exponent bits $E_7E_6\ldots E_1E_0$ must be 0. The 32-bit value is now interpreted as the number:

$$(-1)^S \times (0.M_{22}M_{21}\ldots M_1M_0)_2 \times 2^{-126}$$

The maximum positive value that can be represented by the denormal form corresponds to

```
0 00000000 1111111 11111111 11111111
```

which is

$0.11111111111111111111111 \times 2^{-126} = 2^{-126} - 2^{-149}$.

This is obtained by subtracting 1 from the least significant bit position of the smallest positive integer representable by a normal number. Denormal numbers therefore correspond to a *gradual underflow* from normal numbers.

The minimum positive value that can be represented by the denormal form corresponds to

```
0 00000000 0000000 00000000 00000001
```

which is $2^{-149}$, i.e., nearly $1.401 \times 10^{-45}$.

**Special numbers**

Recall that the exponent bits were not allowed to take the value 1111 1111 (255 in decimal). This value corresponds to some special numbers. These numbers together with some other special ones are listed in the following table.

| 32-bit value | Interpretation |
|---|---|
| 0 1111 1111 0000000 00000000 00000000 | +Inf |
| 1 1111 1111 0000000 00000000 00000000 | -Inf |
| 0 1111 1111 Any nonzero 23-bit value | NaN |
| 1 1111 1111 Any nonzero 23-bit value | NaN |
| 0 0000 0000 0000000 00000000 00000000 | +0 |
| 1 0000 0000 0000000 00000000 00000000 | -0 |
| 0 0111 1111 0000000 00000000 00000000 | +1.0 |
| 1 0111 1111 0000000 00000000 00000000 | -1.0 |
| 0 1000 0000 0000000 00000000 00000000 | +2.0 |
| 1 1000 0000 0000000 00000000 00000000 | -2.0 |
| 0 1000 0000 1000000 00000000 00000000 | +3.0 |
| 1 1000 0000 1000000 00000000 00000000 | -3.0 |
| 0 1111 1110 1111111 11111111 11111111 | $2^{255} - 2^{231}$ |
| 0 0000 0001 0000000 00000000 00000000 | $2^{-126}$ |
| 0 0000 0000 1111111 11111111 11111111 | $2^{-126} - 2^{-149}$ |
| 0 0000 0000 0000000 00000000 00000001 | $2^{-149}$ |

# Introduction to arrays

Arrays are our first example of *structured data*. Think of a book with pages numbered 1,2,. . .,400. The book is a single entity, has its individual name, author(s), publisher, bla bla bla, but the contents of its different pages are (normally) different. Moreover, Page 251 of the book refers to a particular page of the book. To sum up, individual pages retain their identities and still we have a special handy bound structure treated as a single entity. That's again abstraction, but this course is mostly about that.

Now imagine that you plan to sum 400 integers. Where will you store the individual integers? Thanks to your ability to declare variables, you can certainly do that. Declare 400 variables with 400 different names, initialize them individually and finally add each variable separately to an accumulating sum. That's gigantic code just for a small task.

Arrays are there to help you. Like your book you now have a single name for an entire collection of 400 integers. Declaration is small. Codes for initialization and addition also become shorter, because you can now access the different elements of the collection by a unique index. There are built-in C constructs that allow you do parameterized (i.e., indexed) tasks repetitively.

## Declaring arrays

Simple! Just as you did for individual data items, write the data type, then a (legal) name and immediately after that the size of the array within square brackets. For example, the declaration

```
int intHerd[400];
```

creates an array of name `intHerd` that is capable of storing 400 `int` data. A more stylistic way to do the same is illustrated now.

```
#define HERD_SIZE 400
int intHerd[HERD_SIZE];
```

Here are two other arrays, the first containing 123 `float` data, the second 1024 `unsigned char` data.

```
float flock[123];
unsigned char crowd[1024];
```

You can intersperse declaration of arrays with those of simple variables and pointers.

```
unsigned long man, society[100], woman, *ptr;
```

This creates space for two `unsigned long` variables `man` and `woman`, an array called `society` with hundred `unsigned long` data, and also a pointer named `ptr` to an `unsigned long` data.

Note that all individual elements of a single array must be of the same type. You cannot declare an array some of whose elements are integers, the rest floating-point numbers. Such heterogeneous collections can be defined by other means that we will introduce later.

## Accessing individual array elements

Once an array A of size s is declared, its individual elements are accessed as `A[0],A[1],...,A[s-1]`. It is very important to note that:

Array indexing in C is zero-based.

This means that the "first" element of A is named as `A[0]` (not `A[1]`), the "second" as `A[1]`, and so on. The last element is `A[s-1]`.

Each element `A[i]` is of data type as provided in the declaration. For example, if the declaration goes as:

```
int A[32];
```

each of the elements `A[0],A[1],...,A[31]` is a *variable* of type `int`. You can do on each `A[i]` whatever you are allowed to do on a single `int` variable.

C does not provide automatic range checking.

If an array A of size s is declared, the element `A[i]` belongs to the array (more correctly, to the memory locations allocated to A) if and only if 0 <= i <= s-1. However, you can use `A[i]` for other values of i. No compilation errors (nor warnings) are generated for that. Now when you run the program, the executable attempts to access a part of the memory that is not allocated to your array, nor perhaps to (the data area allocated to) your program at all. You simply do not know what resides in that part of the memory. Moreover, illegal memory access may lead to the deadly "segmentation fault". C is too cruel at certain points. Beware of that!

## Initializing arrays

Arrays can be initialized during declaration. For that you have to specify constant values for its elements. The list of initializing values should be enclosed in curly braces. For the declaration

```
int A[5] = { 51, 29, 0, -34, 67 };
```

`A[0]` is initialized to 51, `A[1]` to 29, `A[2]` to 0, `A[3]` to -34 and `A[4]` to 67. Similarly, for the declaration

```
char C[8] = { 'a', 'b', 'h', 'i', 'j', 'i', 't', '\0' };
```

`C[0]` gets the value `'a'`, `C[1]` the value `'b'`, and so on. The last (7th) location receives the null character. Such null-terminated character arrays are also called **strings**. Strings can be initialized in an alternative way. The last declaration is equivalent to:

```
char C[8] = "abhijit";
```

Now see that the trailing null character is missing here. C automatically puts it at the end. Note also that for individual characters, C uses single quotes, whereas for strings, it uses double quotes.

If you do not mention sufficiently many initial values to populate an entire array, C uses your incomplete list to initialize the array locations at the lower end (starting from 0). The remaining locations are initialized to zero. For example, the initialization

```
int A[5] = { 51, 29 };
```

is equivalent to

```
int A[5] = { 51, 29, 0, 0, 0 };
```

If you specify an initialization list, you may omit the size of the array. In that case, the array will be allocated exactly as much space as is necessary to accommodate the initialization list. You must, however, provide the square brackets to indicate that you are declaring an array; the size may be missing between them.

```
int A[] = { 51, 29 };
```

creates an array A of size 2 with `A[0]` holding the value 51 and `A[1]` the value 29. This declaration is equivalent to

```
int A[2] = { 51, 29 };
```

but not to

```
int A[5] = { 51, 29 };
```

There are a lot more things that pertain to arrays. You may declare multi-dimensional arrays, you may often interchange arrays with pointers, and so on. But it's now too early for these classified topics. Wait until your experience with C ripens.

---

Course home