Notes 5.0: Arithmetic types, constants, printf()

COMP9021 Principles of Programming

School of Computer Science and Engineering
The University of New South Wales

2014 session 1

Numbers, booleans, characters

Data objects we might want to use and operate on include:

- Natural numbers, such as 0 or 234213423452452352523.
- Integers, such as 67, -1233, or -3667456234534574632.
- Real numbers, such as 43.456346, π, e or $\frac{1}{3}$.
- Complex numbers.
- The boolean values true and false.
- Characters, listed in ascii.pdf.

To represent the former data objects, C provides the following types.

Types (1)

- For natural numbers,
  - `unsigned short int`,
  - `unsigned int`,
  - `unsigned long int` and
  - `unsigned long long int`

  where `int` can be left implicit.
- For integers,
  - `signed short int`,
  - `signed int`,
  - `signed long int` and
  - `signed long long int`

  where `signed` or `int` can be left implicit (possibly both, except of course for `signed int`).
- For real numbers,
  - `float`,
  - `double` and
  - `long double`.

Types (2)

For complex numbers,

- `float _Complex`,
- `double _Complex` and
- `long double _Complex`

with `complex` available as a macro to use in place of `_Complex`, provided that the `complex.h` header file from the standard library be included.

- For booleans, `_Bool`, with `bool` available as a macro, provided that the `stdbool.h` header file from the standard library be included.
- For characters,
  - `char`,
  - `signed char` and
  - `unsigned char`.

Types for natural numbers

The program bytes_and_natural_numbers.c indicates how many bytes are allocated to each of the data types meant to represent natural numbers, on the machine where the program is run.

If, on a particular machine, n is the number of bytes allocated to such a type, then that type encompasses all natural numbers from 0 up to $2^{8n} - 1$, on that particular machine, as confirmed by ranges_and_natural_numbers.c.

In order to be portable, a program should not use the actual number of bytes that is allocated to any type meant to represent natural numbers, but only the requirements set by the standard that

- an `unsigned short` should be at least 16 bits wide, an `unsigned long` at least 32 bits wide, and an `unsigned long long` at least 64 bits wide;
- the range of a type is included in the range of a larger type.

Internal representation of natural numbers

The internal representation of natural numbers is given by their representation in base 2.

On machines where 16 bits are allocated to an `unsigned short int`, particular representations include:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	: 0
0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	: 138
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	: 65535

Types for integers

The program bytes_and_integers.c indicates how many bytes are allocated to each of the data types meant to represent integers, on the machine where the program is run.

If, on a particular machine, n is the number of bytes allocated to such a type, then that type is guaranteed to encompass all integers from $-2^{8n-1} + 1$ up to $2^{8n-1} - 1$, on that particular machine, as confirmed by ranges_and_integers.c.

In order to be portable, a program should not use the actual number of bytes that is allocated to any type meant to represent integers, but only the requirements set by the standard that

- a `short` should be at least 16 bits wide, a `long` at least 32 bits wide, and a `long long` at least 64 bits wide;
- the range of a type is included in the range of a larger type.

Type `_Bool` is not a type of its own: it is synonymous to `int`.

Internal representation of integers (1)

The internal representation of integers is usually given by their twos-complement notation, even though other representations are possible, such as one-complement notation or sign magnitude notation; that notation allows one to also represent -2^{8n-1} on a machine where n bytes are allocated to the type.

The leftmost bit is set to 0 for a positive number, and set to 1 for a negative number.

A representation of a negative number N can be obtained from the representation of the absolute value of N by changing all 0s to 1s, all 1s to 0s, and then adding 1. For instance, if we had one byte to represent -119 in twos-complement notation, we would proceed as follows.

- Represent 119:
- Change 0s to 1s and 1s to 0s:
- Add 1:

0	1	1	1	0	1	1	1
1	0	0	0	1	0	0	0
1	0	0	0	1	0	0	1

Internal representation of integers (2)

On machines where 16 bits are allocated to a `signed short int`, particular representations include:

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	: -32768
1	1	1	1	1	1	1	1	0	1	1	1	0	1	1	0	: -138
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	: -1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	: 0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	: 1
0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	: 138
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	: 32767

Types for real numbers

The program bytes_and_real_numbers.c indicates how many bytes are allocated to each of the data types meant to represent real numbers (which of course can only be rational), on the machine where the program is run.

Some of the information that characterises the types meant to represent real numbers is displayed by ranges_and_real_numbers.c.

The standard requires that the range of a type is included in the range of a larger type.

Internal representation of normalised real numbers

Take a nonzero rational number r that in base 2, is of the form $\pm 1.b_1 \ldots b_{23} 2^e$ with $b_1, \ldots, b_{23}$ in $\{0, 1\}$ and e in $\{-126, \ldots, 127\}$.

r can be represented in single precision 32 bits, using the leftmost bit to represent the sign (0 for plus, 1 for minus), the 23 rightmost bits to store $b_1, \ldots, b_{23}$, and the remaining 8 bits (in-between) to represent e.

There are $2^8 - 2$ possible values for e. The eight bits dedicated to representing e code $e + 2^{8-1} - 1$. Hence

- | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | represents $e = -126$ $(-126 + 2^7 - 1 = 1)$
- | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | represents $e = 127$ $(127 + 2^7 - 1 = 254)$

For instance, since $1.25 \times 2^{-3} = 0.15625$, the binary representation of 0.25 is 0.01, and $-3 + 2^7 - 1 = 124$, the representation of 0.15625 is

0	0	1	1	1	1	1	0	0	0	1	0	0	0	·····	0	0

Internal representation of denormalised real numbers (1)

It would be possible to extend the previous representation of $r = \pm 1.b_1 \ldots b_{23} 2^e$ to e being either -127 or 128.

Rather than representing a number r of the form $\pm 1.b_1 \ldots b_{23} 2^e$ with $e = -127$ following the principles applied to the case where $e \in \{-126, \ldots, 127\}$, precision is sacrificed for range: an alternative representation, namely

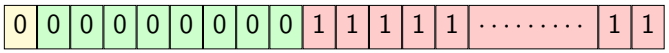
$$\pm 0.b_1 \ldots b_{23} 2^{-126}$$

with $b_1, \ldots, b_{23}$ in $\{0, 1\}$ is used, that allows one to represent smaller and smaller numbers with less and less precision.

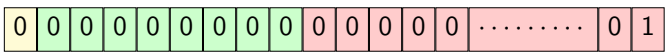
Such numbers are said to be denormalised, unless $b_1, \ldots, b_{23}$ are all equal to 0, in which case r is (positive or negative) 0.

Internal representation of denormalised real numbers (2)

So the largest positive denormalised number, represented as



is equal to $2^{-126} \times \frac{2^{-1}-2^{-24}}{2^{-1}} \approx 1.18 \times 10^{-38}$, while the smallest positive denormalised number, represented as



is equal to $2^{-126} \times 2^{-23} = 2^{-149} \approx 1.4 \times 10^{-45}$.

Internal representation of infinities and nonnumbers

Rather than representing a number r of the form $\pm 1.b_1 \dots b_{23} 2^e$ with $e = 128$ following the principles applied to the case where $e \in \{-126, \dots, 127\}$, there are 2 cases:

- All of b_1, ..., b_{23} are equal to 0, in which case r is $+\infty$ (positive infinity) or $-\infty$ (negative infinity).
- At least one of b_1, ..., b_{23} is not equal to 0, in which case r is considered to be too large and treated as NaN (not a number).

The same overall principles apply to the double precision 64 bits representation of normalised rational numbers of the form

$$\pm 1.b_1 \dots b_{52} 2^e,$$

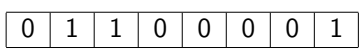
with b_1, ..., b_{52} in $\{0, 1\}$ and $e \in \{-1022, \dots, 1023\}$, denormalised rational numbers of the form $\pm 0.b_1 \dots b_{52} 2^{-1022}$ with b_1, ..., b_{52} in $\{0, 1\}$, infinities and nonnumbers.

Types for characters

The program bytes_and_characters.c indicates how many bytes are allocated to each of the data types meant to represent characters, on the machine where the program is run.

Type `char` is neither signed nor unsigned, but plain, though this is usually equivalent to one or the other.

A character is represented by its integer code in the underlying character set, usually the ASCII character set. For instance, character `a` is usually coded as 97 and represented as



Constants that represent integers (1)

There are three ways to represent an integer by a constant.

- In decimal, with a sequence of digits that does not start with 0.
  Example: `22`, `91`, `84`.
- In octal, with a sequence of digits smaller than 8 that starts with 0.
  Example: `022`, `0041`, `074`.
- In hexadecimal, with a sequence of digits and lower or upper case letters from a to f that starts with `0x` or `0X`.
  Example: `0x22`, `0Xabc`, `0Xa9`.

There is no constant to represent negative integers: `-89` is not a constant, but an expression consisting of the unary minus operator applied to the constant `89`. Also, `+89` is not a constant, but an expression consisting of the unary plus operator applied to the constant `89`.

The header file `stdbool.h` defines `true` and `false` as macros for `1` and `0`, respectively, in accordance with the fact that `_Bool` is synonymous to `int`.

Constants that represent integers (2)

A constant of this kind can

- fit into an `int`, in which case it is of type `int`, or
- be too big to fit into an `int` but not too big to fit into a `long`, in which case it is of type `long`, or
- be too big to fit into a `long` but not too big to fit into a `long long`, in which case it is of type `long long`, or
- be too big to fit into a `long long`, in which case something happens that we happily ignore.

Moreover, a constant of this kind can

- be suffixed with `l` or `L`, in which case it is of type `long` if it fits in a `long`, and of type `long long` if it does not fit in a `long` but fits in a `long long`, or
- be suffixed with `ll` or `LL`, in which case it is of type `long long` if it fits in a `long long`.

Constants that represent integers (3)

For instance, on machines where 4 bytes are allocated to `int`s and `long`s and 8 bytes are allocated to `long long`s,

- `1` is of type `int`;
- `1l` is of type `long`;
- `1ll` is of type `long long`;
- `2147483647` (equal to $2^{31} - 1$) is of type `int`;
- `21474836471` is of type `long`;
- `214748364711` is of type `long long`;
- `2147483648` is of type `long long`;
- `21474836481` is of type `long long`;
- `214748364811` is of type `long long`.

Enumerated types

An enumerated type is a sequence of values of type `int` represented by enumeration constants.

- The value of the first enumeration constant is 0, or an explicit value.
- The value of any other enumeration constant is 1 plus the value of the previous one, or an explicit value.

Values are represented by constant expressions, possibly involving previous enumeration constants. The enumeration constants are separated by commas, and a comma can follow the last constant. For instance,

- ```
  enum rating {strongly_disagree = 1, disagree,
               mod_disagree, mod_agree, agree, strongly_agree};
  ```
  assigns 1, 2, 3, 4, 5, 6 to the enumeration constants;
- ```
  enum count {zero, one, two, five = 5, seven = 7,
              null = 0, once = zero + 1, twice, thrice,};
  ```
  assigns 0, 1, 2, 5, 7, 0, 1, 2, 3 to the enumeration constants.

Constants that represent natural numbers

What has been said about constants that represent integers applies mutatis mutandis to constants that represent natural numbers, except that `u` or `U` is added after the digits in the representation, with `unsigned` added to the names of the types.

For instance,

- `1u` and `1U` are of type `unsigned int`.
- `1lu`, `1lU`, `1Lu`, `1LU`, `1ul`, `1uL`, `1Ul`, `1UL` are of type `unsigned long`.
- `0XFUL` is of type `unsigned long`, and represents the natural number 15.

Constants that represent real numbers (1)

The syntactic form of constants that represent real numbers is depicted in float.pdf

Examples of such constants are

```
1.56E+12    2.87e-3    3.14159
.2          4e16       .8E-5      100.
```

Again, a leading a + or - sign is interpreted as an operator; hence in particular, no constant represents a negative real number.

Constants that represent real numbers (2)

A constant of this kind can

- fit into a `double`, in which case it is of type `double`, or
- be too big or too small to fit into a `double` but not too big or too small to fit into a `long double`, in which case it is of type `long double`, or
- be too big or too small to fit into a `long double`, in which case something happens that we happily ignore.

Moreover, a constant of this kind can

- be suffixed with `f` or `F`, in which case it is of type `float` if it fits in a `float`, of type `double` if it does not fit in a `float` but fits in a `double`, and of type `long double` if it does not fit in a `double` but fits in a `long double`, or
- be suffixed with `l` or `L`, in which case it is of type `long double` if it fits in a `long double`.

Constants that represent characters (1)

There are four ways to represent a character by a constant of type `int`.

- By putting the character between single quotes, except for a number of characters such as the new line character, the single quote or the backslash.
- By using the decimal value of the code of the character.
- By putting single quotes around a backslash followed by the octal value of the code of the character.
- By putting single quotes around a backslash followed by either `x` or `X` followed by the hexadecimal value of the code of the character.

For instance, upper case C, whose ascii code has a decimal value of 67, can be represented by one of

```
'C'      67      '\103'      '\x43'      '\X43'
```

The first representation is usually much preferable.

Constants that represent characters (2)

For some characters, an escape sequence rather than the character itself can be put between single quotes:

- `\a` for `BEL` (alert, which can be an auditive or visual signal)
- `\b` for `BS` (backspace)
- `\f` for `FF` (formfeed, which advances to start of next page)
- `\n` for `LF` (new line, which moves to beginning of next line)
- `\r` for `CR` (carriage return, which moves to beginning of current line)
- `\v` for `VT` (vertical tab, which moves to next vertical tab position)
- `\'` for the single quote
- `\\` for the backslash

The horizontal tab can be inserted directly between single quotes (in Emacs, by typing `\C-q` and then hitting the tab key), or one can use the escape sequence `\t`.

An example of use of escape sequences

The program salary.c illustrates the use of escape sequences, though the demonstration is more convincing when the program is run from the command line rather than from within Emacs...

The escape sequence `\b` occurs in the string constant

```
Enter your \"year\ly\" salary: $_____\b\b\b\b\b\b
```

which, as any string constant, is delimited by a pair of double quotes.

The string constant contains two occurrences of double quotes that, in order not to be confused with a double quote that delimits a string constant, are escaped. A character that taken individually, does not need to be escaped, can be escaped either with no useful effect, as the escaped `l` in `year\ly`, or with a useful effect as the escaped double quotes around `year\ly`, provided of course that the escaped character is not one of the special escape sequences (as would be the case if the occurrence of `a` within `year\ly` was escaped...).

Functions to manipulate characters (1)

The standard library provides two useful functions to process characters.

- `getchar()` is a function that takes no argument; each time it is called it reads the next character from the text stream associated with the standard input (by default the keyboard) and returns the value of type `int` that represents that character.
- `putchar()` is a function that takes one integer as (unique) argument; it prints the character represented by the value of its argument on the text stream associated with the standard output (by default the screen).

To process input stored in a file *`file`*, the contents of the file can be redirected to standard input with the `<` redirection operator: the program can then be run by typing `a.out <`*`file`* from the command line or by typing `r <`*`file`* from within Emacs.

EOF

When input is typed in from the keyboard, the end of input is indicated by typing `Contrl D` on Unix machines and `Contrl Z` on Windows machines. When input is redirected from a file, the `Contrl D` or `Contrl Z` still indicate the end of input.

`EOF` is a constant, defined as a macro in `stdio.h`, that indicates End Of File; on most systems it is equal to `-1`.

To process input, most programs display a structure of the form:

```
int c;
while ((c = getchar()) != EOF) {
    ...
}
```

If `c` is declared as a variable of type `char` rather than as a variable of type `int`, then this statement will cause the program to loop on systems where `char`s are represented as unsigned numbers.

Functions to manipulate characters (2)

Including the `ctype.h` header file allows one to use a number of functions that can determine whether a given character belongs to certain classes of characters; these functions are organised as represented in ctype.pdf.

`ctype.h` also allows one to convert a lowercase alphabetic character to its uppercase counterpart, and the other way around, thanks to the functions `toupper()` and `tolower()`, respectively (the character is unchanged if it is not a lowercase alphabetic character or an uppercase alphabetic character, respectively).

All these functions take an `int` as single argument and return an `int`.

Functions to manipulate complex numbers

The program complex.c illustrates the manipulation of complex numbers,

- using `I`, `J` or `j` to denote the imaginary number i and
- using the `creal()` and `cimag()` functions to access the real and imaginary parts, respectively, of a complex number,

all uses that require including the `complex.h` header file.

A complex number is considered to be infinite when one part is infinite, even if the other part is `NaN` (not a number).

A program to experiment

show_bits.c

Depending on the architecture, the bytes will be displayed in one of two orders: small endians or big endians.

For instance, if 4 bytes are allocated to an `int` then `9` will be displayed as

```
00000000 00000000 00000000 00001001
```

in big-endian order and as

```
00001001 00000000 00000000 00000000
```

in small-endian order.

The printf() function

A number can be represented by many different constants. The `printf()` function allows one to use the representation considered to be most appropriate, to add certain amount of space before or after that representation, as well as a certain number of `0`s before the representation.

A program that uses the `printf()` function, or more generally any function from the standard library that performs output formatting or that parses formatted input, must include the `stdio.h` header file, namely, must have in its preamble the line:

```
#include <stdio.h>
```

Function `printf()` takes a variable number of arguments: a format string that possibly contains conversion specifications, and for each such conversion specification, the name of a data item to be printed out.

The printf() function: the format string

Apart from (optional) conversion specifications, the format string may contain characters, represented literally or by escape sequences, output as expected, except for the `%` character, that has to be represented as `%%` in the format string.

Each conversion specification starts with `%`, and

- ends in a conversion letter
- that is possibly preceded by a size modifier
- that is possibly preceded by a dot followed by a natural number that determines the precision of the representation
- that is possibly preceded by a natural number that determines the minimum field width of the representation
- that is possibly preceded by a number of flags.

The printf() function: the flags

The flags include

- `-` to left justify rather than right justify the value within the field width;
- `0` to pad with `0` rather than with a space to meet the requirements of the field width;
- `+` to always produce a sign, either `+` or `-`;
- a space to always produce either the sign `-` or a space.
- `#` to produce an initial `0` when displaying integers represented in base 8, or to produce an initial `0x` or `0X` when displaying integers represented in base 16.

Displaying integers

The conversion letter is `d` for a signed representation in base 10, `u` for an unsigned representation in base 10, `o` for an unsigned representation in base 8, and `x` (with `a`–`f` used to represent the digit values between 10 and 15) or `X` (with `A`–`F` used to represent those digit values) for an unsigned representation in base 16.

There is no size modifier for data items of type `int` or `unsigned`. The size modifiers `h`, `l` and `ll` are used for data items of type `short`, `long` and `long long`, respectively.

The displayed sequence of digits is as short as possible, but no shorter than the precision, if given.

If no precision is requested and left justification is not requested, then leading `0`s are used for padding if this is required by the minimum field width and the `0` flag.

The program display_integers.c illustrates some of the possibilities.

Displaying real numbers (1)

The conversion letter is `f`, `e`, `E`, `g` or `G`. The `f` conversion letter uses no exponent. The `e` and `E` conversion letters are for scientific notation, with one nonzero digit before the decimal point and a power of 10. The `g` and `G` conversion letters determine a choice between either representation. With `e` and `g`, the exponent if any is output as `e` followed by a signed integer; with `E` and `G`, it is output as `E` followed by a signed integer.

There is no size modifier for data items of type `float`. The size modifiers `l` and `L` are used for data items of type `double` and `long double`, respectively.

The default precision for the `f`, `g`, `G`, `e` and `E` conversion letters is 6.

With the `f`, `e` and `E` conversion letters, precision gives the number of digits to be printed after the decimal point; with the `g` and `G` modifiers, it gives the number of significant digits with trailing `0`s removed, and with the decimal point removed if no nonzero digit follows.

Displaying real numbers (2)

The `g` and `G` conversion letters use the `e` and `E` format, respectively, for large numbers if the precision required would make the integral part of the number incorrect, and for small numbers if the precision required would make the whole representation longer. For instance,

```
printf("%g\n", 123456789.0);
```

prints `1.23457e+08`, whereas

```
printf("%g %g %g\n", 3.14159e-3, 3.14159e-4, 3.14159e-5);
```

prints `0.00314159 0.000314159 3.14159e-05` (note that `3.14159e-04` takes as much space as `0.000314159`).

If left justification is not requested, then leading `0`s are used for padding if this is required by the minimum field width and the `0` flag.

The program display_reals.c illustrates some of the possibilities.