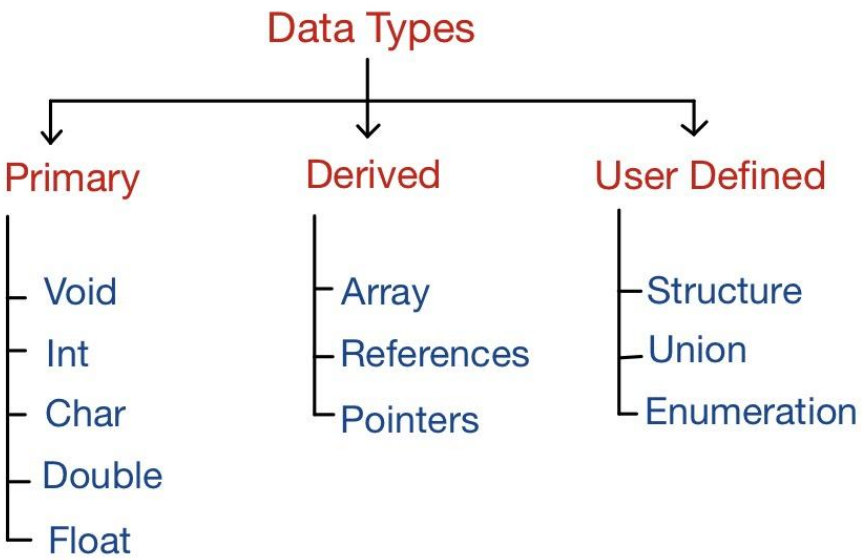
Basic Types

Pedro Ribeiro

DCC/FCUP

2024/2025



Basic Types

- We have seen two basic types of the C language: `int` and `float`
- Let's now introduce more types:
  - unsigned integers
  - integers of different sizes
  - floating point types with more precision
  - characters

Integers in C

- Values of type `int` *have a sign*: they can be negative, positive or zero
- We use `unsigned int` for *unsigned* integers: only positive or zero

```
int i;           // with sign
unsigned int j; // without sign
```

- We can abbreviate the declaration to unsigned:

```
unsigned j; // unsigned int
```

Integer Sizes

- The `int` type is typically represented using 32-bits (may be smaller on 8 8- and 16-bit CPUs)
- We can use `long` and `short` attributes to specify larger or smaller sizes
- Together with `unsigned` we have 6 different types of integers:

`short int`	`unsigned short int`
`int`	`unsigned int`
`long int`	`unsigned long int`

- The order of the attributes doesn't matter and we can omit `int`

Integer Sizes

- We know that `short` $\leq$ `int` $\leq$ `long`
- However, C language does not specify the exact limits for sizes
- For each implementation: the *header* file `<limits.h>` defines the limits for each type

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

int main(void) {
  printf("SHRT_MIN = %d\n", SHRT_MIN);
  printf("SHRT_MAX = %d\n", SHRT_MAX);

  printf("INT_MIN = %d\n", INT_MIN);
  printf("INT_MAX = %d\n", INT_MAX);

  printf("LONG_MIN = %ld\n", LONG_MIN); // %ld to format long
  printf("LONG_MAX = %ld\n", LONG_MAX);

  return 0;
}
```

Integer Sizes

- Running on Mooshak machine (Linux, X86_64):

```
SHRT_MIN = -32768
SHRT_MAX = 32767
INT_MIN = -2147483648
INT_MAX = 2147483647
LONG_MIN = -9223372036854775808
LONG_MAX = 9223372036854775807
```

- On that machine:
  - `short` uses 2 bytes (16 bits) $(2^{15} - 1 = 32\,767)$
  - `int` uses 2 bytes (16 bits) $(2^{31} - 1 = 2\,147\,483\,647)$
  - `long` uses 2 bytes (16 bits) $(2^{63} - 1 = 9\,223\,372\,036\,854\,775\,807)$

- If you are curious, current computer use two's complement to represent integers

Integer constants

- Integer constants in a program are usually `int`
- If the value is too large then they are `long int`
- We can specify `long` constants ending with `L` or `l` and `unsigned` with `U` or `u`:

```
          17 // int
      -1000L // long int
      2500UL // unsigned long int
```

- If we assign a constant of a different type from the variable, there is an **implicit type conversion**:

```
long int i = 17; // 17 -> 17L
```

Reading and writing integers

- To read or write `long` or `unsigned` integers we must use specific formats in `scanf` and `printf`.
  - `%u` - unsigned decimal integer
  - `%ld` - decimal integer long
  - `%lu` - decimal integer unsigned long

- Example:

```
long x;

// read and write long integer
scanf("%ld", &x);
printf("%ld\n", x);
```

Floating point

- In computing, real numbers are typically represented using floating point arithmetic
  - a *significand* (a.k.a. *mantissa*, a signed sequence of a fixed number of digits in some base) multiplied by an *integer power* of that base
- Two basic types for floating point in C:
  - `float` for single precision
  - `double` for double precision
- The language standard does not define the precision of these types
- Modern implementations usually follow a standard called IEEE 754

type	smallest positive	largest value	precision
`float`	$\approx 1.17 \times 10^{-38}$	$\approx 3.40 \times 10^{38}$	6 digits
`double`	$\approx 2.22 \times 10^{-308}$	$\approx 1.79 \times 10^{308}$	15 digits

- `double` is used for most applications
- `float` is only used if precision is not important or to save memory

Floating point constants

- We can write floating-point constants in various ways:

  `57.0` `57.` `57E0` `5.7e1`

- The constant has a *mantissa* and optionally an *exponent*

- The exponent indicates the power of 10 that multiplies the mantissa and is prefixed with the letter `E` (or `e`).
  - `5.7e1` is 5.7×10^{1}
  - `5.7E-3` is 5.7×10^{-3}

Reading and writing floating point

- `printf` and `scanf` specifiers:
  - `%f` - float
  - `%lf` - double
- If we want to write with k digits after decimal point we should use `%.kf` or `%.klf`

Example:

```c
double d;

scanf("%lf", &d); // for the output below, suppose the user
                  // introduces the value 1.23456789

printf("%lf\n", d);    // write double
printf("%.2lf\n", d); // use 2 digits after decimal point
printf("%.3lf\n", d); // user 3 digits after decimal point
```

```
1.234568
1.23
1.235
```

Explicit conversions

- We can explicitly convert from one type to another (*cast*) using: `(type) expr`

  Example:

```c
int k = 2, n = 3;
printf("%f\n", (float)k/(float)n); // 0.66666
printf("%f\n", (float)k/n);        // 0.66666
printf("%f\n", (float)(k/n));      // 0.00000
```

- We should perform conversions before operations that could cause an *overflow*:

```c
int i = 1500;
long j;
j = (long)(i*i*i); // overflow?
j = (long)i*i*i; // OK
```

Characters

- The `char` type is used to represent characters
- Characters are represented by their numeric code
- The language standard does not define the encoding
- The most common code is ASCII
  (*American Standard Code for Information Interchange*)
- ASCII contains 128 symbols and only the letters of the Latin alphabet without accents
- UTF-8 is an extension of ASCII that supports other alphabets and accents (e.g. Portuguese)
  - for the purposes of this class, we'll only use plain ASCII
    (using UTF-8 in C is harder)

ASCII Table

- See for instance Wikipedia or ascii-code

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Characters

- A variable of type `char` holds any character:

```
char ch;
ch = 'A';
```

- character constants are delimited by single (not double) quotes:

```
'A'    // uppercase letter A
'a'    // lowercase letter A
'?'    // question mark
' '    // space
```

- Characters in C are treated as *"small"* integers (typically 1 byte).
- A constant is simply the number corresponding to the character code Examples (in ASCII):

```
'A'   // 65
'a'   // 97
' '   // 32
'0'   // 48
```

Operations on characters

- We can perform arithmetic operations on characters
- They correspond to operations on the associated numeric codes
  Examples (assuming ASCII):

```
int i;
char ch;

i = 'a';      // i is 97
ch = 65;      // ch has the code for 'A'
ch = ch + 1; // ch has the code for 'B'
ch++;        // ch has the code for 'C'
```

- We can also compare characters like any other number
- To make programs portable (and readable), we should use constants and not concrete code.
  Example: to test whether a character is between A and Z

```
if (ch >= 'A' && ch <= 'Z') ... // OK!
if (ch >=  65 && ch <=  90) ...   // Don't use this...
```

Escape sequences

- A constant character is usually just a symbol enclosed in single quotes
- Some special characters can't be written like this
- For these cases we can use escape sequences

  Some examples:

`\n`	line change	`\t`	horizontal tab
`\\`	backslash	`\'`	single quotation mark

- We delimit escapes with single quotes, e.g.:

```
char ch = '\n'; // line change
```

Printing and reading characters

- We can use the `%c` format to read or write characters using `scanf` and `printf`

  Example to print the ASCII table (only printable characters):

  `ascii_table.c` (source)

```c
#include <stdio.h>

int main(void) {
  char ch;

  for (ch = 32; ch<127; ch++) {
    if (ch % 8 == 0) {
      printf("\n"); // change line
    }
    printf("%3d: %c ", ch, ch); // %3d prints right alined with 3 "places"
  }
  printf("\n");

  return 0;
}
```

Printing and reading characters

- Output of the `ascii_table.c` program:

```
 32:     33: !   34: "   35: #   36: $   37: %   38: &   39: ’
 40: (   41: )   42: *   43: +   44: ,   45: -   46: .   47: /
 48: 0   49: 1   50: 2   51: 3   52: 4   53: 5   54: 6   55: 7
 56: 8   57: 9   58: :   59: ;   60: <   61: =   62: >   63: ?
 64: @   65: A   66: B   67: C   68: D   69: E   70: F   71: G
 72: H   73: I   74: J   75: K   76: L   77: M   78: N   79: O
 80: P   81: Q   82: R   83: S   84: T   85: U   86: V   87: W
 88: X   89: Y   90: Z   91: [   92: \   93: ]   94: ^   95: _
 96: ‘   97: a   98: b   99: c  100: d  101: e  102: f  103: g
104: h  105: i  106: j  107: k  108: l  109: m  110: n  111: o
112: p  113: q  114: r  115: s  116: t  117: u  118: v  119: w
120: x  121: y  122: z  123: {  124: |  125: }  126: ~
```

Printing and reading characters

- For chars it is more common to use the following `stdio` functions:

```
char ch

ch = getchar(); // read a character
putchar(ch);    // write a character
```

- Let's have a closer look at `getchar`:

```
int getchar(void);
```

  - Consumes the next character from the *standard input* (*stdin*) and returns its numeric code.
  - If there are no more characters returns `EOF` (*end-of-file*)
    - ★ `EOF` is constant defined in `stdio.h` (typically `-1`)
    - ★ on the terminal: Control-D signals the end of the input
- The result is an `int` and not a `char`
- It will usually be a *"small"* integer (the code of one character)

Example

- Program to count the number of lines in the standard input:

`count_lines_v1.c` (source)

```c
#include <stdio.h>

int main(void) {
    int ch;                 // code of a character
    int lines = 0;          // number of lines
    do {
        ch = getchar();     // read a character
        if(ch == '\n')      // new line?
            lines++;
    } while(ch != EOF);
    printf("%d lines\n", lines);

    return 0;
}
```

Example

- More "idiomatic" version of the same idea:

`count_lines_v2.c` (source)

```c
#include <stdio.h>

int main(void) {
    int ch;             // code of a character
    int lines = 0;      // number of lines
    while ((ch=getchar()) != EOF) {
      if(ch == '\n')  // new line?
         lines++;
    }
    printf("%d lines\n", lines);

    return 0;
}
```

Library functions

- The header `ctype.h` includes some predefined functions for manipulating characters (see documentation):
  - test whether a character is an uppercase letter, lowercase letter, numeral, etc.
  - converting lowercase letters to uppercase and vice versa
- These are simple but frequently used definitions

```c
#include <ctype.h>

int islower(int ch); // lowercase letter a..z
int isupper(int ch); // capital letter A..Z
int isalpha(int ch); // a..z or A..Z
int isdigit(int ch); // decimal digit 0..9

int tolower(int ch); // convert to lower case
int toupper(int ch); // convert to uppercase
```

- **Warning:** only works for simple ASCII - e.g. letters without accents.

Example

- Counting letters and digits from standard input:

`count_letters_digits.c` (source)

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

int main(void) {
  int ch;
  int letters = 0, digits = 0;

  while((ch=getchar()) != EOF) {
    if (isalpha(ch))
       letters++;
    else if(isdigit(ch))
       digits++;
  }
  printf("%d, %d\n", letters, digits);

  return 0;
}
```