Functions

Basics of Programming 1

G. Horváth, A.B. Nagy, Z. Zsóka, P. Fiala, A. Vitéz

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

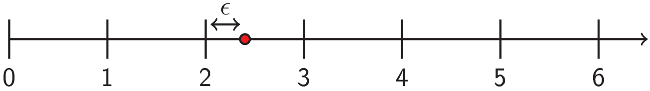
24 September, 2025

Content

Chapter 1

Arithmetic types of C

Representation accuracy of integer types

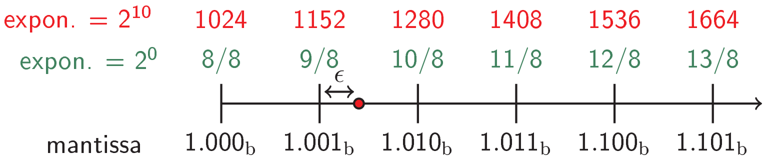


Absolute accuracy of number representation

It is the maximal ϵ error of representing an arbitrary real number with the closest integer

- The absolute accuracy of representing with integer types is 0.5

Representation accuracy of floating-point numbers



- in this example
  - The (absolute) representation accuracy of the mantissa is 1/16
  - If the exponent is 2^0, the representation accuracy is 1/16
  - If the exponent is 2^{10}, the representation accuracy is $2^{10}/16 = 64$
- There is no absolute, only relative accuracy, that is, in this present case, 3 bits.

Consequences of finite number representation

- As the floating-point number representation is not accurate, we must not check the equality of results of operations!

$$\frac{22}{7} + \frac{3}{7} \neq \frac{25}{7}$$

  instead

$$\left| \frac{22}{7} + \frac{3}{7} - \frac{25}{7} \right| < \varepsilon$$

- The exponent will magnify the rounding error of the finite long mantissa, thus the large numbers are much less accurate than small numbers. The errors of the large numbers can "eat up" the small ones:

$$A + a - A \neq a$$

Consequences of the binary representation of numbers

- A decimal finite number might not be finite in binary form, eg.:

$$0{,}1_{\mathrm{d}} = 0{,}0\overline{0011}_{\mathrm{b}}$$

- How many times will be this cycle repeated?

```
double d;
for (d = 0.0; d < 1.0; d = d+0.1) /* 10? 11? */
{
   ...
}
```

- The good solution is:

```
double d;
double eps = 1e-3; /* what is the right eps for here? */
for (d = 0.0; d < 1.0-eps; d = d+0.1) /* 10 times */
{
   ...
}
```

Chapter 2

Arithmetic types of C

Representing characters – The ASCII table

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 128 characters, that can be indexed with numbers 0x00–0x7f

Code	00	10	20	30	40	50	60	70
+00	NUL	DLE	␣	0	@	P	`	p
+01	SOH	DC1	!	1	A	Q	a	q
+02	STX	DC2	"	2	B	R	b	r
+03	ETX	DC3	#	3	C	S	c	s
+04	EOT	DC4	$	4	D	T	d	t
+05	ENQ	NAK	%	5	E	U	e	u
+06	ACK	SYN	&	6	F	V	f	v
+07	BEL	ETB	'	7	G	W	g	w
+08	BS	CAN	(	8	H	X	h	x
+09	HT	EM	)	9	I	Y	i	y
+0a	LF	SUB	*	:	J	Z	j	z
+0b	VT	ESC	+	;	K	[	k	{
+0c	FF	FS	,	<	L	\	l		
+0d	CR	GS	-	=	M	]	m	}
+0e	SO	RS	.	>	N	^	n	~
+0f	SI	US	/	?	O	_	o	DEL

Storing, printing and reading characters

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Characters (indexes of the ASCII table) are stored in `char` type
- Printing of the elements of the ASCII table is done with `%c` format code.

```
char ch = 0x61; /* hex 61 = dec 97 */
printf("%d: %c\n", ch, ch);
ch = ch+1; /* its value will be hex 62 = dec 98  */
printf("%d: %c\n", ch, ch);
```

- Output of the program

```
97:  a
98:  b
```

- Does it mean we have to learn the ASCII-codes to be able to print characters?

Character constants

- A character placed between apostrophes is equivalent to its ASCII-code

```
char ch = 'a'; /* 0x61 ASCII-code is copied to ch */
printf("%d: %c\n", ch, ch);
ch = ch+1;
printf("%d: %c\n", ch, ch);
```

```
97: a
98: b
```

- Beware! `'0'` $\neq 0$!

```
char n = '0'; /* 0x30 ASCII-code is copied to ch !!! */
printf("%d: %c\n", n, n);
```

```
48: 0
```

Character constants

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Special character constants – that would be hard to type. . .

0x00	`\0`	null character (NUL)
0x07	`\a`	bell (BEL)
0x08	`\b`	backspace (BS)
0x09	`\t`	tabulator (HT)
0x0a	`\n`	line feed (LF)
0x0b	`\v`	vertical tab (VT)
0x0c	`\f`	form feed (FF)
0x0d	`\r`	carriage return (CR)
0x22	`\"`	quotation mark
0x27	`\'`	apostrophe
0x5c	`\\`	backslash

Character or integer number?

- In C language characters are equivalent to integer numbers
- It will be decided only at the moment of displaying how an integer value is printed: as a number or as a character (`%d` or `%c`)
- We can perform the same operations on characters as on integers (adding, subtracting, etc....)
- But what is the point in adding-subtracting characters?

Operations with characters

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Let's write a program, that reads characters as long as a new line character has not arrived. After this the program should print out the sum of the read (scanned) digits.

```
char c;
int sum = 0;
do
{
  scanf("%c", &c);                /* reading */
  if (c >= '0' && c <= '9')       /* if numerical digit */
    sum = sum + (c-'0');          /* summing */
}
while (c != '\n');                /* stop condition */
printf("The sum is: %d\n", sum);
```

```
The airplane has landed at 12:35 this afternoon
The sum is: 11
```

Chapter 3

Functions

Segmentation – motivation

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Let's create a program, that prints out the sum of the squares of all positive numbers, that are smaller than 12! $(1^2 + 2^2 + \cdots + 11^2)$

```c
#include <stdio.h> /* for printf */

int main(void)
{
  int i, sum; /* aux. variable and sum of squares*/

  sum = 0;                          /* initialization */
  for (i = 1; i < 12; i = i+1) /* i = 1,2,...,11 */
    sum = sum + i*i;                /* summing */

  printf("The square sum: %d\n", sum);
  return 0;
}
```

link

Segmentation – motivation


```c
int main(void) {
  int i, sum1, sum2, sum3;

  sum1 = 0;            /* for 12 */
  for (i = 1; i < 12; i = i+1)
    sum1 = sum1 + i*i;

  sum2 = 0;            /* for 24 */
  for (i = 1; i < 24; i = i+1)
    sum2 = sum2 + i*i;

  sum3 = 0;            /* for 30 */
  for (i = 1; i < 30; i = i+1)
    sum3 = sum3 + i*i;

  printf("%d, %d, %d\n",
    sum1, sum2, sum3);
  return 0;
}
```

link

Let's create a program, that will perform the previous tasks with numbers 12, 24 and 30!

Our solution

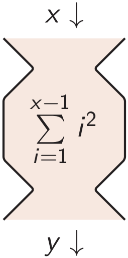
- was made by Copy+Paste+correct
- many possibilities for mistakes, errors
- long program
- it is hard to manage

Is it possible in a more smarter way?

Functions

The function

- Standalone program segment
- For operations that occur frequently
- We can run it (call it) with different arguments
- Calculates something, and gives back the result for the program that called it



Functions – solution

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
int squaresum(int n) /* function definition */
{
  int i, sum = 0;
  for (i = 1; i < n; i = i+1)
    sum = sum + i*i;
  return sum;
}

int main(void) /* main program */
{
  int sum1, sum2, sum3;

  sum1 = squaresum(12); /* function call */
  sum2 = squaresum(24);
  sum3 = squaresum(30);

  printf("%d, %d, %d\n", sum1, sum2, sum3);
  return 0;
}
```

link

Function definition

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Syntax of a function definition

```
<type of return value>
<function identifier> (<list of formal parameters>)
<block>
```

```c
int squaresum(int n)
{
  int i, sum = 0;
  for (i = 1; i < n; i = i+1)
    sum = sum + i*i;
  return sum;
}
```

Function definition

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Type of the return value:

- The type of the calculated value

```
double average(int a, int b)
{
  return 0.5 * (a + b);
}
```

- or `void` (empty), if the function does not calculate anything

```
void print_point(double x, double y)
{
  printf("(%.3f, %.3f)", x, y); /* (2.000, 4.123) */
}
```

- because sometimes we don't care about the calculated value, only about the "side effect" (secondary effect).

A remark: Primary and secondary effects

Primary the function calculates and gives back the return value

Secondary the function "performs some more things" (prints on screen, writes to file, plays an MP3, launches a missile. . .)

- Some programming languages make a clear distinction between different program segments:
  - **function** where the primary effect is the important
  - **procedure** no primary effect, but the secondary effect is important
- In C language there is only function. Procedures are represented by functions with empty (`void`) return value.
- Generally, we should try to separate the primary and secondary effects!

Function definition

Formal list of parameters

- Comma-separted list of declaration of parameters one-by-one, so we can reference them inside the function

```c
double volume(double x, double y, double z)
{
  return x*y*z;
}
```

- The number of parameters can be 0, 1, 2, . . . as much as you want (127 🙂)
- If there are 0 parameters, we denote it with `void`

```c
double read_next_positive(void)
{
  double input;
  do scanf("%lf", &input) while (input <= 0);
  return input;
}
```

Function definition

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

The `return` statement

- it gives a return value, it terminates the execution of the function's block, and returns to the point of calling
- there can be more of it, but it will cause to (terminate and) return to the point of calling at the first execution.

```
double distance(double a, double b)
{
  double dist = b - a;
  if (dist < 0)
    return -dist;
  return dist;
}
```

- it can also occur in a `void`-type function `return;`

Function call

```
double distance(double a, double b)
{
   ...
}
```

Syntax of a function call

<function identifier> (<actual argument expr.>)

```
double x = distance(2.0, 3.0); /* x will be 1.0 */
```

```
double a = 1.0;
double x = distance(2.5-1.0, a); /* x will be 0.5 */
```

```
double pos = read_next_positive(); /* empty () */
```

The main program as a function


```
int main(void) /* now we understand, what this is */
{
  ...
  return 0;
}
```

The main program is also a function

- it is called by the operation system at the start of the program
- it does not get any arguments (we will change this later)
- it returns with integer (`int`) value
  - Traditionally, if execution was OK, it gives 0-t, otherwise an error code

```
Process returned 0 (0x0)   execution time:  0.001 s
press ENTER to continue.
```

Mechanism of function call

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
/* Area of a rectangle */
int area(int x, int y)
{
  int S;
  S = x * y;
  return S;
}

/* Main program */
int main(void)
{
  int a, b, T;
  a = 2;                /* base */
  b = 3;            /* height */
  T = area(a, b); /* area */
  return 0;
}
```

register: 0

Mechanism of function call

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Passing parameters by value

- Functions receive the value of the actual argument expressions.
- Parameters can be used as variables, that have an initial value assigned at the point of calling.
- Functions may modify the values of the parameters, but this has no effect on the calling program segment.

Visibility and life-cycle of variables

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Local variables

1. parameters of functions
2. variables declared inside a function

- They are created when entering into the function, and are erased when returning from the function.
- They are invisible for program segments outside of the function. (also for the calling segment!)

Global variables – only for emergency cases!

Variables declared outside of functions (even outside of `main()`)

- They exist throughout the life-cycle of the program.
- They are visible for everyone and can be modified by anyone!
- In case of conflicts, the local variable masks out the global one.

Riddle

What will the following program print on the screen?

```
#include <stdio.h>

int a, b;

void func(int a)
{
  a = 2;
  b = 3;
}

int main(void)
{
  a = 1;
  func(a);
  printf("a: %d, b: %d\n", a, b);
  return 0;
}
```

link

Operations with characters

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Let's write a function, that converts the lowercase letters of the English alphabet to uppercase, but leaves all other characters unchanged.

```
char toupper(char c)
{
  if (c >= 'a' && c <= 'z') /* if lowercase */
  {
    return c - 'a' + 'A';
  }
  return c;
}
```

A complex task

Let's create a C program, that asks two integer numbers from the user (`low` < `high`), and lists all prime numbers between these two numbers..

- Pseudo-code of the solution broken into segments:

mainprogram

```
IN: low, high
FOR EACH i
  between low and high
  IF primetest(i) TRUE
    OUT: i
```

primetest(p)

```
FOR EACH i
  between 2 and root of p
  IF i divides p
    return FALSE
return TRUE
```

- Notice the role of the two i and p

Complex task – solution

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
#include <stdio.h> /* scanf, printf */

int low, high; /* global variables */

void read(void) /* inputting function */
{
  printf("Give a small and a larger number!\n");
  scanf("%d%d", &low, &high);
}

int isprime(int p) /* primetest function. */
{
  int i;
  for (i=2; i*i<=p; i=i+1) /* i from 2 to root of p */
    /* if p is dividable by i, not a prime */
    if (p%i == 0)
      return 0;
  return 1; /* if we get here, it is a prime */
}
```

Complex task – solution

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c

int main()
{
  int i;

  read(); /* we read the limits with a function */

  printf("Primes between %d and %d:\n", low, high);
  for (i=low; i<=high; i=i+1)
  {
    if (isprime(i)) /* we test with a function */
      printf("%d\n", i);
  }

  return 0;
}
```

link

Design principles

- Functions and programs communicate via parameters and return values.
- Except when this is their special task, functions
  - do not print on the screen,
  - do not read from keyboard,
  - do not use global variables.

Chapter 4

Type conversion

What is that?

In some cases the C-program needs to convert the type of our expressions.

```
long func(float f) {
  return f;
}

int main(void) {
  int i = 2;
  short s = func(i);
  return 0;
}
```

In this example: `int` → `float` → `long` → `short`

- `int` → `float` rounding, if the number is large
- `float` → `long` may cause overflow, rounding to integer
- `long` → `short` may cause overflow

Converting types

- Basic principle
  - preserve the value, if possible
- In case of overflow
  - the result is theoretically undefined

- Conversion with one operand (we have seen that)
  - at assignment of value
  - at calling a function (when actualising the formal parameters)
- Conversion with two operands (eg. `2/3.4`)
  - evaluating an operation

Conversion with two operands

The conversion of the two operands to the same, common type happens according to these rules

operand one	the other operand	common, new type
`long double`	anything	`long double`
`double`	anything	`double`
`float`	anything	`float`
`unsigned long`	anything	`unsigned long`
`long`	anything (`int`, `unsigned`)	`long`
`unsigned`	anything (`int`)	`unsigned`
`int`	anything (`int`)	`int`

Type conversions

Example for conversion

```
int a = 3;
double b = 2.4;
a = a*b;
```

1. $3 \rightarrow 3.0$
2. $3.0 * 2.4 \rightarrow 7.2$
3. $7.2 \rightarrow 7$

Thank you for your attention.