Arithmetic types in C

Basics of Programming 1

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

18 September, 2024

Content

Chapter 1

Arithmetic types of C

Types – Introduction

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Type is

- Set of values
- Operations

Types – Introduction

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Type is

- Set of values
- Operations

In a real computer – the set of values is limited

- We can not represent arbitrary large numbers

Types – Introduction

Type is

- Set of values
- Operations

In a real computer – the set of values is limited

- We can not represent arbitrary large numbers
- We can not represent numbers with arbitrary accuracy
  $\pi \neq 3.141592654$

Types – Introduction

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Type is

- Set of values
- Operations
- Representation

In a real computer – the set of values is limited

- We can not represent arbitrary large numbers
- We can not represent numbers with arbitrary accuracy
  $\pi \neq 3.141592654$
- We must know the limits of what can be represented, in order to store our data

Types – Introduction

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Type is

- Set of values
- Operations
- Representation

In a real computer – the set of values is limited

- We can not represent arbitrary large numbers
- We can not represent numbers with arbitrary accuracy
  $\pi \neq 3.141592654$
- We must know the limits of what can be represented, in order to store our data
  - without any loss of information or

Types – Introduction

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Type is

- Set of values
- Operations
- Representation

In a real computer – the set of values is limited

- We can not represent arbitrary large numbers
- We can not represent numbers with arbitrary accuracy
  $\pi \neq 3.141592654$
- We must know the limits of what can be represented, in order to store our data
  - without any loss of information or
  - with an acceptable level of information loss, without wasting memory

Types of C language

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- void
- scalar
  - arithmetic
    - integer: integer, character, enumerated
    - floating-point
  - pointer
- function
- union
- compound
  - array
  - structure

Types of C language

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- void
- scalar
  - arithmetic
    - integer: integer, character, enumerated
    - floating-point
  - pointer
- function
- union
- compound
  - array
  - structure
- Today we will learn about them

Binary representation of integers

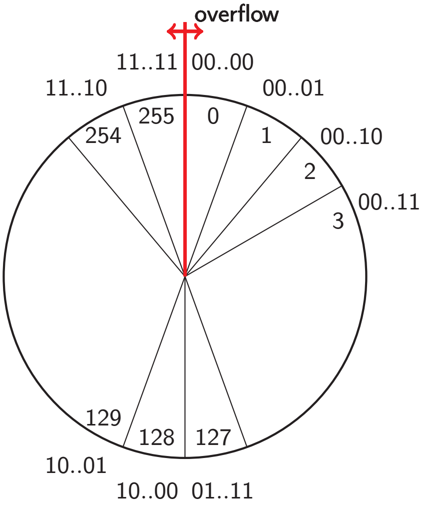
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Binary representation of unsigned integers stored in 8 bits

dec	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	hex
0	0	0	0	0	0	0	0	0	0x00
1	0	0	0	0	0	0	0	1	0x01
2	0	0	0	0	0	0	1	0	0x02
3	0	0	0	0	0	0	1	1	0x03
⋮	⋮							⋮	⋮
127	0	1	1	1	1	1	1	1	0x7F
128	1	0	0	0	0	0	0	0	0x80
129	1	0	0	0	0	0	0	1	0x81
⋮	⋮							⋮	⋮
253	1	1	1	1	1	1	0	1	0xFD
254	1	1	1	1	1	1	1	0	0xFE
255	1	1	1	1	1	1	1	1	0xFF

The overflow

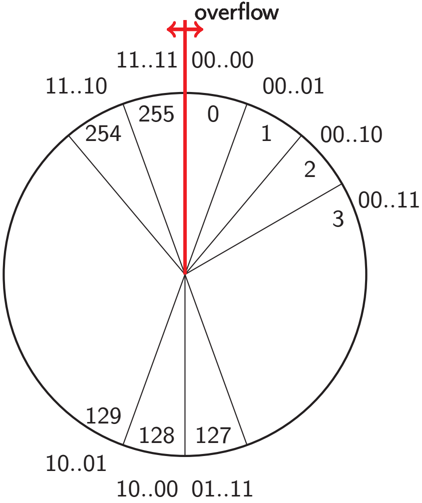
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- In case of unsigned integers stored in 8 bits

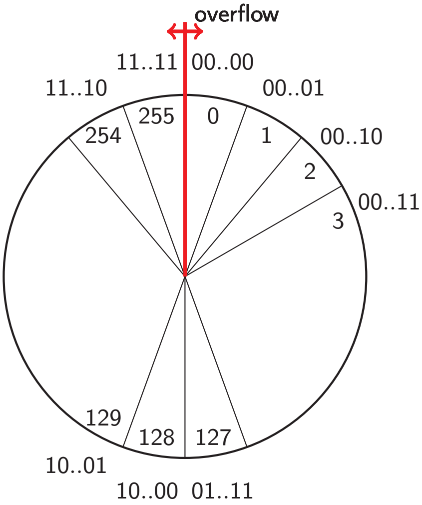
The overflow

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- In case of unsigned integers stored in 8 bits
  - 255+1 = 0

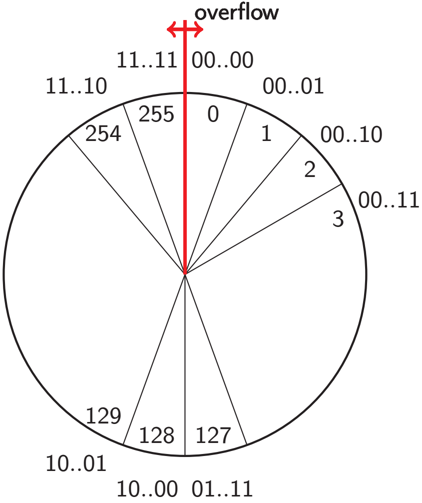
The overflow



- In case of unsigned integers stored in 8 bits
  - 255+1 = 0
  - 255+2 = 1

The overflow

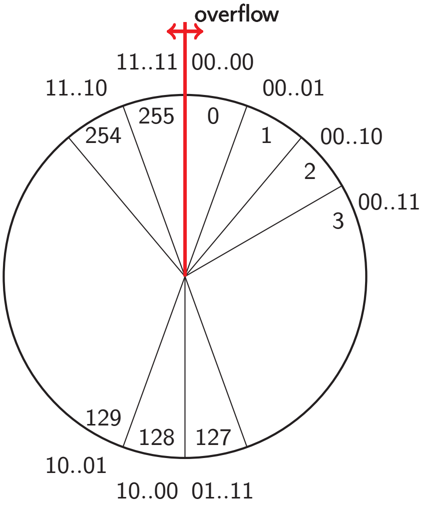
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- In case of unsigned integers stored in 8 bits
  - 255+1 = 0
  - 255+2 = 1
  - 2–3 = 255

The overflow

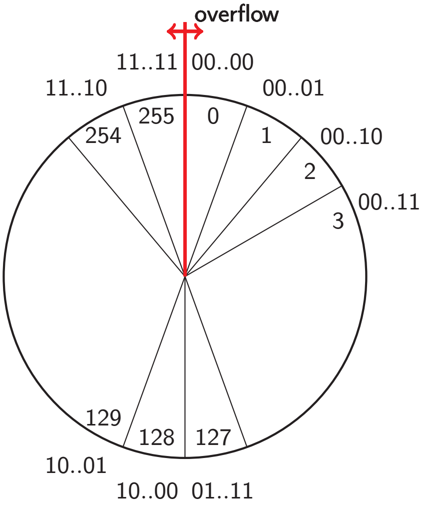
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- In case of unsigned integers stored in 8 bits
  - 255+1 = 0
  - 255+2 = 1
  - 2–3 = 255
- "modulo 256 arithmetic"

The overflow

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- In case of unsigned integers stored in 8 bits
  - 255+1 = 0
  - 255+2 = 1
  - 2−3 = 255
- "modulo 256 arithmetic"
  - We always see the remainder of the result divided by 256

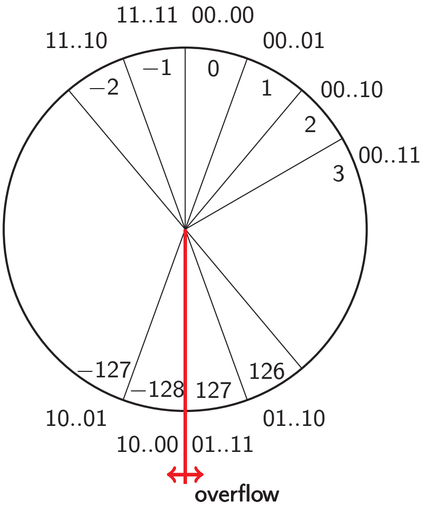
Two's complement representation of integers

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Two's complement representation of signed integers stored in 8 bits

dec	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	hex
0	0	0	0	0	0	0	0	0	0x00
1	0	0	0	0	0	0	0	1	0x01
2	0	0	0	0	0	0	1	0	0x02
3	0	0	0	0	0	0	1	1	0x03
⋮	⋮							⋮	⋮
127	0	1	1	1	1	1	1	1	0x7F
−128	1	0	0	0	0	0	0	0	0x80
−127	1	0	0	0	0	0	0	1	0x81
⋮	⋮							⋮	⋮
−3	1	1	1	1	1	1	0	1	0xFD
−2	1	1	1	1	1	1	1	0	0xFE
−1	1	1	1	1	1	1	1	1	0xFF

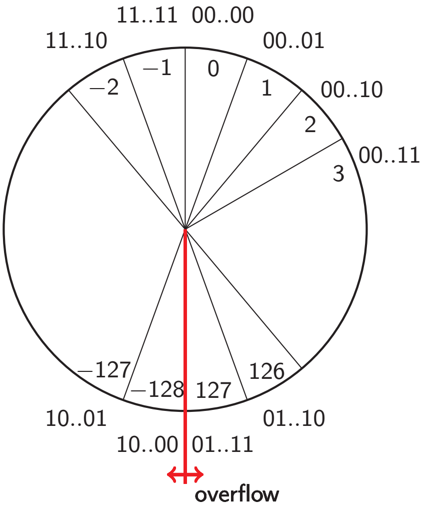
The overflow



- In case of signed integers stored in 8 bits

The overflow

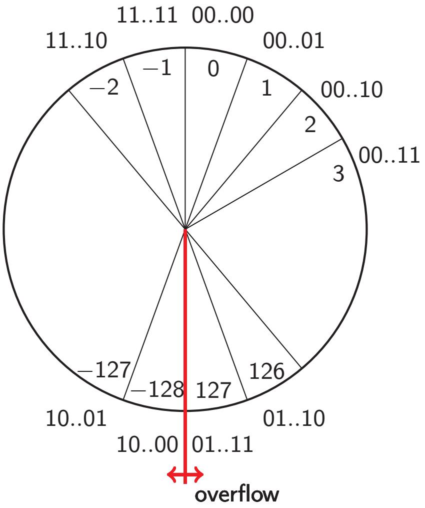
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- In case of signed integers stored in 8 bits
  - 127+1 = -128

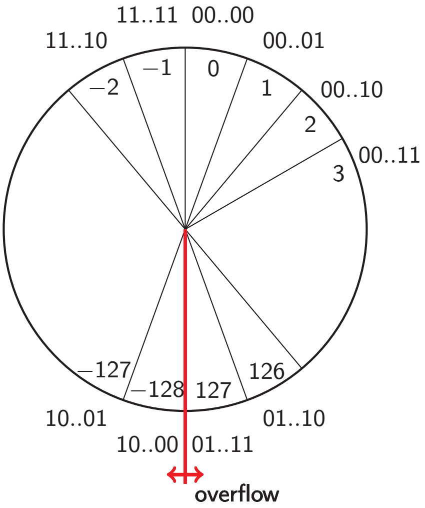
The overflow

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- In case of signed integers stored in 8 bits
  - 127+1 = -128
  - 127+2 = -127

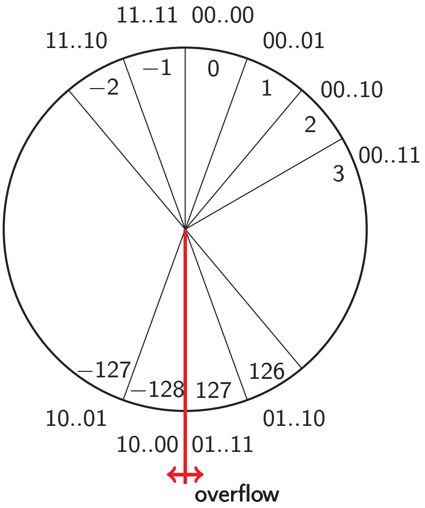
The overflow



- In case of signed integers stored in 8 bits
  - 127+1 = -128
  - 127+2 = -127
  - -127-3 = 126

The overflow

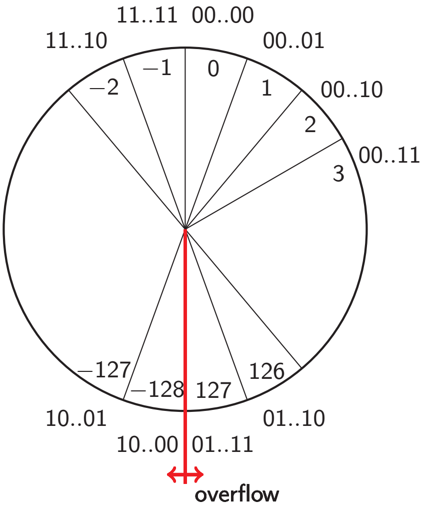
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- In case of signed integers stored in 8 bits
  - 127+1 = -128
  - 127+2 = -127
  - -127-3 = 126
- on the other hand

The overflow

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- In case of signed integers stored in 8 bits
  - 127+1 = -128
  - 127+2 = -127
  - -127-3 = 126
- on the other hand
  - 2-3 = -1

Integer types in C

type	bit[1]	<limits.h>		printf
`signed char`	8	`CHAR_MIN`	`CHAR_MAX`	`%hhd`[2]
`unsigned char`	8	`0`	`UCHAR_MAX`	`%hhu`[2]
`signed short int`	16	`SHRT_MIN`	`SHRT_MAX`	`%hd`
`unsigned short int`	16	`0`	`USHRT_MAX`	`%hu`
`signed int`	32	`INT_MIN`	`INT_MAX`	`%d`
`unsinged int`	32	`0`	`UINT_MAX`	`%u`
`signed long int`	32	`LONG_MIN`	`LONG_MAX`	`%ld`
`unsigned long int`	32	`0`	`ULONG_MAX`	`%lu`
`signed long long int`[2]	64	`LLONG_MIN`	`LLONG_MAX`	`%lld`
`unsigned long long int`[2]	64	`0`	`ULLONG_MAX`	`%llu`

[1] Typical values, the standard only determines the minimum

[2] since the C99 standard

Declaration of integers

- Defaults
  - The `signed` sign-specifier can be omitted

```c
int i;          /* signed int */
long int l;     /* signed long int */
```

Declaration of integers

- Defaults
  - The `signed` sign-specifier can be omitted

```c
int i;            /* signed int */
long int l;       /* signed long int */
```

  - If there is sign- or length-modifier, the `int` can be omitted.

```c
unsigned u;        /* unsigned int */
short s;           /* signed short int */
```

Integer types

- An example on how to use the previous table: a program that runs for a very long time[3]

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

int main(void)
{ /* almost all long long int */
  long long i;

  for (i = LLONG_MIN; i < LLONG_MAX; i = i+1)
    printf("%lld\n", i);

  return 0;
}
```

link

[3] provided that `long long int` is 64 bit long, the program runs for 585 000 years if the computer prints 1 million numbers per second

Integer constants

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Specifying integer constants

```c
int i1=0, i2=123, i4=-33;              /* decimal */
int o1=012, o2=01234567;               /* octal */
int h1=0x1a, h2=0x7fff, h3=0xAa1B     /* hexadecimal */

long l1=0x1al, l2=-33L;                /* l or L */

unsigned u1=33u, u2=45U;               /* u or U */
unsigned long ul1=33uL, ul2=1231U;     /* l and u */
```

Integer constants

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Specifying integer constants

```
int i1=0, i2=123, i4=-33;             /* decimal */
int o1=012, o2=01234567;              /* octal */
int h1=0x1a, h2=0x7fff, h3=0xAa1B   /* hexadecimal */

long l1=0x1al, l2=-33L;               /* l or L */

unsigned u1=33u, u2=45U;              /* u or U */
unsigned long ul1=33uL, ul2=123lU; /* l and u */
```

- If neither u or l is specified, the first type that is big enough is taken:
  1. `int`
  2. `unsigned int` – in case of hexa and octal constants
  3. `long`
  4. `unsigned long`

Why do we need to know the limits of number representations?

Let's determine the following value!

$$\binom{15}{12} = \frac{15!}{12! \cdot (15-12)!}$$

(What is the number of possibilities of selecting 12 out of 15 different chocolates?)

Why do we need to know the limits of number representations?

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Let's determine the following value!

$$\binom{15}{12} = \frac{15!}{12! \cdot (15-12)!}$$

(What is the number of possibilities of selecting 12 out of 15 different chocolates?)

- The value of the numerator is $15! = 1\ 307\ 674\ 368\ 000$

Why do we need to know the limits of number representations?

Let's determine the following value!

$$\binom{15}{12} = \frac{15!}{12! \cdot (15-12)!}$$

(What is the number of possibilities of selecting 12 out of 15 different chocolates?)

- The value of the numerator is $15! = 1\ 307\ 674\ 368\ 000$
- The value of the denominator is $12! \cdot 3! = 2\ 874\ 009\ 600$

Why do we need to know the limits of number representations?

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Let's determine the following value!

$$\binom{15}{12} = \frac{15!}{12! \cdot (15-12)!}$$

(What is the number of possibilities of selecting 12 out of 15 different chocolates?)

- The value of the numerator is $15! = 1\ 307\ 674\ 368\ 000$
- The value of the denominator is $12! \cdot 3! = 2\ 874\ 009\ 600$
- None of them can be represented as a 32 bits `int`!

Why do we need to know the limits of number representations?

Let's determine the following value!

$$\binom{15}{12} = \frac{15!}{12! \cdot (15-12)!}$$

(What is the number of possibilities of selecting 12 out of 15 different chocolates?)

- The value of the numerator is $15! = 1\,307\,674\,368\,000$
- The value of the denominator is $12! \cdot 3! = 2\,874\,009\,600$
- None of them can be represented as a 32 bits `int`!
- But with simplifying the expression

$$\frac{15 \cdot 14 \cdot 13}{3 \cdot 2 \cdot 1} = \frac{2730}{6} = 455$$

all parts can be calculated without any problem, even on 12 bits.

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);
```

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
```

Storing, printing and reading characters

- 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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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);
```

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
```

Character constants

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

```c
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'` $\neq 0$!

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

Character constants

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

```c
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'` $\neq 0$!

```c
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?

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- In C language characters are equivalent to integer numbers

Character or integer number?

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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`)

Character or integer number?

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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.. . .)

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.

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.

```c
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);
```

Operations with characters

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.

```c
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
```

Operations with characters

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

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.

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

Floating-point types

- Normal form

$$23.2457 = (-1)^{0} \cdot 2.3245700 \cdot 10^{+001}$$
$$-0.001822326 = (-1)^{1} \cdot 1.8223260 \cdot 10^{-003}$$

Floating-point types

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Normal form

$$23.2457 = (-1)^{0} \cdot 2.3245700 \cdot 10^{+001}$$

$$-0.001822326 = (-1)^{1} \cdot 1.8223260 \cdot 10^{-003}$$

Representation of the normal form

- Floating-point fractional = sign bit + mantissa + exponent

Floating-point types

- Normal form

$$23.2457 = (-1)^{0} \cdot 2.3245700 \cdot 10^{+001}$$
$$-0.001822326 = (-1)^{1} \cdot 1.8223260 \cdot 10^{-003}$$

Representation of the normal form

- Floating-point fractional = sign bit + mantissa + exponent
  1. sign bit: 0–positive, 1–negative

Floating-point types

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Normal form

$$23.2457 = (-1)^{0} \cdot 2.3245700 \cdot 10^{+001}$$
$$-0.001822326 = (-1)^{1} \cdot 1.8223260 \cdot 10^{-003}$$

Representation of the normal form

- Floating-point fractional = sign bit + mantissa + exponent
  1. sign bit: 0–positive, 1–negative
  2. mantissa: unsigned integer (without the decimal comma), because of normalization, the first digit is ≥ 1

Floating-point types

- Normal form

$$23.2457 = (-1)^{0} \cdot 2.3245700 \cdot 10^{+001}$$
$$-0.001822326 = (-1)^{1} \cdot 1.8223260 \cdot 10^{-003}$$

Representation of the normal form

- Floating-point fractional = sign bit + mantissa + exponent
  1. sign bit: 0–positive, 1–negative
  2. mantissa: unsigned integer (without the decimal comma), because of normalization, the first digit is ≥ 1
  3. exponent (or order, characteristic): signed integer

Floating-point types

- Binary normal form

$$5.0 = 1.25 \cdot 4 = (-1)^{0} \cdot 1.0100_{\mathrm{b}} \cdot 2^{010_{\mathrm{b}}}$$

0	**0100**	**010**

[4] the leading bit is implicit

Floating-point types

- Binary normal form

$$5.0 = 1.25 \cdot 4 = (-1)^{0} \cdot 1.0100_{\mathrm{b}} \cdot 2^{010_{\mathrm{b}}}$$

0	**0100**	**010**

Representation of binary normal form

- Floating-point fractional = sign bit + mantissa + exponent

[4] the leading bit is implicit

Floating-point types

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Binary normal form

$$5.0 = 1.25 \cdot 4 = (-1)^{0} \cdot 1.0100_{\mathrm{b}} \cdot 2^{010_{\mathrm{b}}}$$

0	**0100**	**010**

Representation of binary normal form

- Floating-point fractional = sign bit + mantissa + exponent
  1. sign bit: 0–positive, 1–negative

[4] the leading bit is implicit

Floating-point types

- Binary normal form

$$5.0 = 1.25 \cdot 4 = (-1)^{0} \cdot 1.0100_{\mathrm{b}} \cdot 2^{010_{\mathrm{b}}}$$

0	**0100**	**010**

Representation of binary normal form

- Floating-point fractional = sign bit + mantissa + exponent
  1. sign bit: 0–positive, 1–negative
  2. mantissa: unsigned integer (without the **binary comma**), because of normalization, the first digit is $= 1$, so we don't store it[4].

[4] the leading bit is implicit

Floating-point types

- Binary normal form

$$5.0 = 1.25 \cdot 4 = (-1)^{0} \cdot 1.0100_{\mathrm{b}} \cdot 2^{010_{\mathrm{b}}}$$

0	**0100**	**010**

Representation of binary normal form

- Floating-point fractional = sign bit + mantissa + exponent
  1. sign bit: 0–positive, 1–negative
  2. mantissa: unsigned integer (without the **binary comma**), because of normalization, the first digit is = 1, so we don't store it[4].
  3. exponent: signed integer

[4] the leading bit is implicit

Floating-point types in C

- Floating-point types of C

type	typical values bits	mantissa	exponent	printf/scanf
`float`	32 bits	23 bits	8 bits	`%f`
`double`	64 bits	52 bits	11 bits	`%f`/`%lf`
`long double`	128 bits	112 bits	15 bits	`%Lf`

Floating-point types in C

- Floating-point types of C

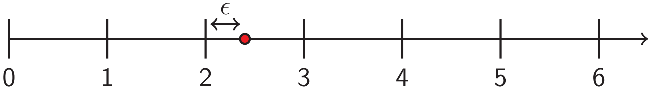
type	typical values bits	mantissa	exponent	printf/scanf
`float`	32 bits	23 bits	8 bits	`%f`
`double`	64 bits	52 bits	11 bits	`%f`/`%lf`
`long double`	128 bits	112 bits	15 bits	`%Lf`

- Floating-point constants

```c
float       f1=12.3f , f2=12.F , f3=.5f , f4=1.2e-3F ;
double      d1=12.3  , d2=12.  , d3=.5  , d4=1.2e-3  ;
long double l1=12.3l , l2=12.L , l3=.5l , l4=1.2e-3L ;
```

- In C we use decimal point and not a comma!

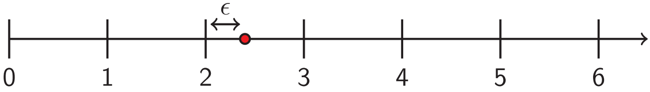
Representation accuracy of integer types



Absolute accuracy of number representation

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

Representation accuracy of integer types

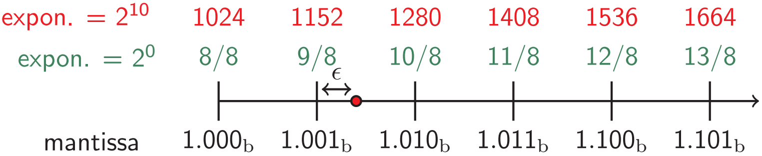


Absolute accuracy of number representation

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

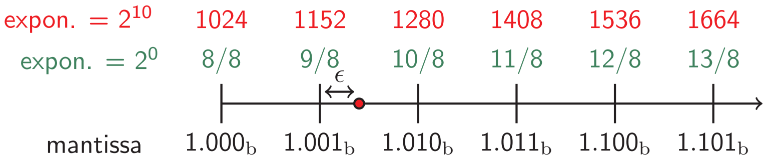
- The absolute accuracy of representing with integer types is 0.5

Representation accuracy of floating-point numbers



Representation accuracy of floating-point numbers

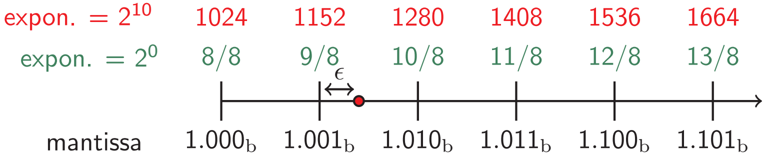
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- in this example
  - The (absolute) representation accuracy of the mantissa is $1/16$

Representation accuracy of floating-point numbers

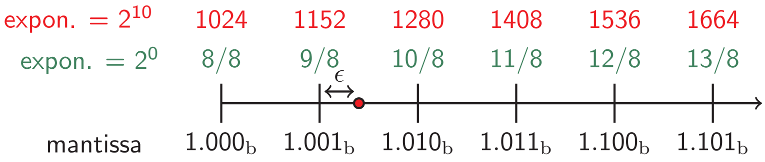
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- in this example
  - The (absolute) representation accuracy of the mantissa is $1/16$
  - If the exponent is 2^0, the representation accuracy is $1/16$

Representation accuracy of floating-point numbers

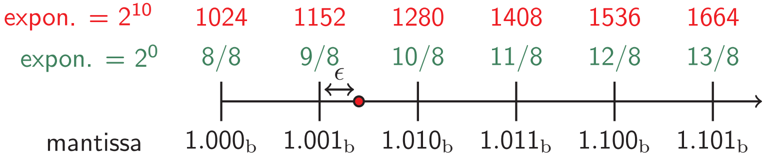
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- 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 $2^{10}/16 = 64$

Representation accuracy of floating-point numbers

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES



- 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

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

Consequences of finite number representation

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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.:

$$0.1_{\mathrm{d}} = 0.0\overline{0011}_{\mathrm{b}}$$

Consequences of the binary representation of numbers

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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? */
{
   ...
}
```

Consequences of the binary representation of numbers

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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

Implicit 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

Converting types

- Basic principle
  - preserve the value, if possible

Converting types

- Basic principle
  - preserve the value, if possible
- In case of overflow

Converting types

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

Converting types

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

- Conversion with one operand (we have seen that)

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

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)

Converting types

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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`)

Converting types

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

The conversion of the two operands to the same, common type happens according to these rules (from top to bottom)

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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Example for conversion

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

Type conversions

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Example for conversion

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

1 $3 \rightarrow 3.0$

Type conversions

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Example for conversion

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

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

Type conversions

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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.