Computer Programming

Data types. Expressions. Conditional Statements

Robert Varga

Technical University of Cluj-Napoca
Computer Science Department

Course 2

Contents

Contents

Primitive data types

The primitive data types in C can separated into two classes:

- integer types
  - char, short int, int, long long int and their unsigned versions
  - all of them use two's complement (C2) for negative values
- real types
  - float, double, long double
  - all of them use the sign bit, exponent bits, mantissa (fraction) bits representation = floating point representation

Characters

- characters in C are treated as integers
- when printing them on the screen their value is mapped from a number to the actual character according to the ASCII table
- the order of characters is based on their integer values
- we can use arithmetic operations on characters, like 'a'+1, 'a'-32
- some categories:
    - special and whitespace characters - from 0 to 32
        - newline = 10
        - space = 32
    - digits - from 48 to 59
    - upper-case letters - from 65 to 90
    - lower-case letters - from 97 to 122

ASCII table

	32	100000	@	64	1000000	`	96	1100000
!	33	100001	A	65	1000001	a	97	1100001
"	34	100010	B	66	1000010	b	98	1100010
#	35	100011	C	67	1000011	c	99	1100011
$	36	100100	D	68	1000100	d	100	1100100
%	37	100101	E	69	1000101	e	101	1100101
&	38	100110	F	70	1000110	f	102	1100110
'	39	100111	G	71	1000111	g	103	1100111
(	40	101000	H	72	1001000	h	104	1101000
)	41	101001	I	73	1001001	i	105	1101001
*	42	101010	J	74	1001010	j	106	1101010
+	43	101011	K	75	1001011	k	107	1101011
,	44	101100	L	76	1001100	l	108	1101100
-	45	101101	M	77	1001101	m	109	1101101
.	46	101110	N	78	1001110	n	110	1101110
/	47	101111	O	79	1001111	o	111	1101111
0	48	110000	P	80	1010000	p	112	1110000
1	49	110001	Q	81	1010001	q	113	1110001
2	50	110010	R	82	1010010	r	114	1110010
3	51	110011	S	83	1010011	s	115	1110011
4	52	110100	T	84	1010100	t	116	1110100
5	53	110101	U	85	1010101	u	117	1110101
6	54	110110	V	86	1010110	v	118	1110110
7	55	110111	W	87	1010111	w	119	1110111
8	56	111000	X	88	1011000	x	120	1111000
9	57	111001	Y	89	1011001	y	121	1111001
:	58	111010	Z	90	1011010	z	122	1111010
;	59	111011	[	91	1011011	{	123	1111011
<	60	111100	\	92	1011100	¦	124	1111100
=	61	111101	]	93	1011101	}	125	1111101
>	62	111110	^	94	1011110	~	126	1111110
?	63	111111	_	95	1011111	⌂	127	1111111

ASCII codes of important characters.

Shows character, code in decimal and code in binary in three columns

Overflow

- for integer types on n bits:
  - if we go above the maximum limit, **overflow** happens
  - only the last n (least significant) bits of the value are retained
  - if we go below the minimum limit, *underflow* happens
  - the behavior is similar
- for real types:
  - if we go above the maximum limit, overflow happens
  - values above the representable maximum are transformed into a special value called *infinity*
  - values below the representable minimum are transformed into a special value called *minus infinity*
  - there is another special value called *not-a-number* reserved for cases when the result is uncertain: infinity - infinity, square root of a negative number

Overflow - Example 1

```c
//unsigned integer types
unsigned int a = 4294967295;
a = a + 7;
printf("%u\n", a);
```

- a contains the largest representable value $= 2^{32} - 1$
- a+1 would be equal to 1 followed by 32 bits of 0 in binary
- a+7 is equal to $2^{32} + 6$, bit 32 is ignored
- prints 6

Overflow - Example 2

```c
//signed integer types
int b = 2147483646;
b = b + 10;
printf("%d\n", b);
```

- b contains the largest representable value $= 2^{31} - 1$
- b $= 0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{C2}$
- $b + 1 = 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{C2}$
- prints $-2^{31} + 9$
- overflow behavior is not standard for signed types because older processors used different representations for negative numbers, but for C2 this is the expected behavior

Overflow - Example 3

```c
//real types
float c = 1e20f;
c = c*c;
printf("%f\n", c);
```

- c is initialized with 10^{20} using scientific notation
- after the multiplication, c would be 10^{40}
- however, this exceeds the maximum limit for float
- prints infinity

Contents

Expressions

- an **expression** can be defined as:
  - a literal or a variable, like 7, or 'a', or x
  - an unary operator acting on an expression, like -7 or +x
  - a binary operator acting on two expressions, like x+y or z*14
  - a ternary operator acting on three expressions, like x ? 0 : 1
- a literal is a constant number, character or string
- it is like a formula from mathematics
- every expression has a type
- most mathematical formulas can be transcribed as C expressions almost identically

Types of operators

- arithmetic
  - + - * / %
  - the result is a number
  - there is no power operator
- relational
  - > >= < <= == !=
  - the result is true (1) or false (0)
  - == is for checking equality, != is for checking not equal
- logical
  - ! && ||
  - the result is true (1) or false (0)
  - ! is for logical negation, && is for logical and, || is for logical or

Evaluating expressions

- if the expression is a literal or a single variable, evaluating it is simple: the value is equal to the value of the literal/variable and the type is kept
- if the expression contains operands of the same type, the type is kept
- if the expression contains operands of different types, the operands of the smaller types are automatically promoted to the larger type
- this automatic conversion is called **implicit conversion**
- type order from small to large:
  char → unsigned char → short int → unsigned short int → int → unsigned int → long long int → unsigned long long int → float → double → long double

Evaluating expressions - special case

- if all operands are integers smaller than int, they are automatically promoted to int when they appear in arithmetic expressions
- unsigned types are promoted to unsigned int
- this happens even when all operands are chars or shorts

```
char x = 120;
char y = 110;
int z = x + y;
printf("%d", z); //230
```

Cast operator

- we can explicitly convert an expression to a specific type using the cast operator
- prepend the expression with (type), where type is the desired target type
- this is called **explicit conversion**
- Example 1: `float x = 1 / (float)2;`
  - the right operand is converted to float, before the division operation, the left operand is promoted to float, x will store 0.5f
- Example 2: `float x = (float)5;`
  - the right operand is converted to float from int
  - this conversion would also happen automatically
- Example 3: `int x = (float)1;`
  - the right operand is converted to float and then converted to int (implicitly)

From float to int

- a floating point value that is transformed into a integer type via implicit or explicit conversion is *truncated*
- *truncation* throws away decimals after the decimal point
- this is different behavior from the whole-part or *floor* function which returns the closest integer, always rounding downwards
- there exists the *ceil* function in math.h for rounding upwards
- and the *round* function for rounding to the nearest integer, when fractional part is 0.5 it rounds away from 0
- these 4 are all different behaviors

From float to int - examples

x	(int)x	floor(x)	ceil(x)	round(x)
1.2	1	1	2	1
1.5	1	1	2	2
1.7	1	1	2	2
2	2	2	2	2
-1.2	-1	-2	-1	-1
-1.5	-1	-2	-1	-2
-1.7	-1	-2	-1	-2
-2	-2	-2	-2	-2

Differences between explicit conversion (truncation) and rounding functions from math.h. Counterintuitively, functions return floating point numbers

Arithmetic operations - Examples

```c
//unary operators
-a
+-6

//binary operators
int x = 6-7;
int y = 1/2;
float z = 1/2;
float u = 1.f/2;
float v = 1/2.f;
float w = 1/2.0;
float q = (float)1/2;

int a = -1/2;
int b = -7/4.0;
int c = 5%3;
int d = -5%3;
```

Arithmetic operations - Examples

```c
//unary operators
-a
+-6

//binary operators
int x = 6-7;
int y = 1/2;
float z = 1/2;
float u = 1.f/2;
float v = 1/2.f;
float w = 1/2.0;
float q = (float)1/2;

int a = -1/2;
int b = -7/4.0;
int c = 5%3;
int d = -5%3;
```

- changes the sign of a
- the + unary operator is just cosmetic

- x is -1
- y is 0, both operands are ints
- z is still 0, conversion only after the division
- u is 0.5f, right operand is promoted to float
- v is also 0.5f, left operand is promoted to float
- w is converted from double to float
- q is 0.5f, use cast operator for conversion

- a is 0
- b is -1
- c is 2
- d is -2

Assignment and incrementation

- the result of an expression `x = y` is the value assigned to x
- the C language defines the pre- and post-increment operators
- `++i` increases i by one and evaluates to the new value
- `i++` increases i by one and evaluates to the old value
- there exist pre- and post-decrement operators, defined similarly (`--i` and `i--`)
- most arithmetic operators can be compounded with the assignment
- example: `x += 5` signifies x becomes $x + 5$

Division by zero

- it is not possible to divide integers by 0
- this will result in a run-time error
- it is possible to divide floating-point numbers by 0
- this will result in $\pm\infty$
- operations for which the value cannot be determined are evaluated to not-a-number
- examples: infinity - infinity, infinity / infinity, square root of negative number

Priority and associativity

- every operator has a priority or precedence
- when evaluating complicated expressions, the operators with the lowest priority are evaluated first
- if there are multiple operators with the same priority, their order of evaluation is determined by their type of associativity: from left to right or from right to left
- evaluation order can be changed by employing parentheses ()
- when using uncommon operators the it is best practice to enforce evaluation order with parentheses
- the priority makes sense: first do simple operations like $++$, then arithmetic multiplication, then arithmetic addition, then comparisons, then logical and end with assignment

Priority and associativity - simplified table

Priority	Operators	Associativity
1	post ++ −−	→
2	pre ++ −− (type) unary + -	←
3	* / %	→
4	binary + -	→
6	< <= > >=	→
7	== !=	→
11	&&	→
12	‖ !=	→
14	= and its compound versions	←

Some operators have been omitted.

A complete table can be found here.

Relational and logical operators

- the C language originally had no primitive type for boolean[1]
- every non-zero value is considered true and all types of 0 are considered false
- the relational (comparison) and logical operators return true (1) or false (0)
- the logical operators && and || implement short-circuiting:
  - e1 || 1 || e3 || e4 ... - is evaluated as true, and the expressions e3 and those after it are not evaluated at all
  - e1 && 0 && e3 && e4 ... - is evaluated as false, and the expressions e3 and those after it are not evaluated at all

[1] the C99 standard introduces the _Bool type and aliases bool, true, false in the stdbool.h header

Common mistakes

```
float x = 1/2;


int a = 2;
float y = 1 / 2.0 * a;


int a = 5, b;
int c = -(a = 1) + (b = a + 2);
```

- x will hold 0, the conversion happens after integer division
- y will hold a/2, first, one is divided by two, then the result is multiplied by a
- c most likely will hold 6, avoid expressions that have secondary effects (change other variables)

Contents

Conditional statements

- conditional or selection statements allow the program to select a particular execution path from a set of alternatives
- on a lower level this is achieved by jumping to specific lines from the code
- along with repetitive (iteration) and jump statements, they permit the implementation of any algorithm

If statement

- the if statement checks the true-value of an expression and enters a branch if it is true
- an optional alternative branch can be given via the else statement in case the expression is false
- code execution continues normally after the if statement
- if a branch consists of multiple instructions they need to be grouped using {}

```
if(expression)
    statements_true;
else
    statements_false;
```

Program 2.1 - Simple if statement

```c
#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);
    if (x%2)
        printf("odd");
    else
        printf("even");
    return 0;
}
```

- the expression from the if is true when it is different from 0 = when the remainder is one = when x is odd
- in C any value different from zero is considered true

Program 2.2 - Cascaded if statements

```c
#include <stdio.h>

int main() {
    char c;
    scanf("%c", &c);
    if ('a' <= c && c <= 'z')
        printf("lower-case");
    else if ('A' <= c && c <= 'Z')
        printf("upper-case");
    else
        printf("other");
    return 0;
}
```

- cascaded if statements
- used when we have more than two different branches
- we can compare characters directly
- the last else ensures that one of the branches executed
- each else matches with the previous if statement

Common mistakes

```
if (a = 1)
  printf("true")



if (a = 0)
  printf("false")



if (a == 1);
  printf("after if")
```

- the expression inside the if does not check equality, it assigns the value 1 to a and evaluates to 1 (true)
- the expression inside the if does not check equality, it assigned the value 0 to a and evaluates to 0 (false)
- putting a semicolon after the expression terminates the if statement
- the last print statement will always execute

Ternary operator - mini if

- the C language has a ternary conditional operator
- it requires three operands:

  `expression_test ? expression_true : expression_false`
- this works just like an if statement: if the test expression is true the whole expression takes on the value from the first branch (between the ? and : symbols), otherwise the whole expression takes on the value of the second branch (after the :)
- equivalent to

  `if (expression_test)expression_true else expression_false`
- it is recommended only when the instructions are short and the code remains easy to follow
- example: `mx = x > y ? x : y;`

Switch statement

- the switch statement compares an integer expression against possibly multiple values and branches on match
- it can be implemented with cascaded if statements, but in some cases it can produce more compact and readable code
- the statements are executed starting from the first constant expression that matches the expression
- default is optional, it matches with any expression, it is checked last

```
switch(expression){
    case constant-expression_1 : statements_1; [break;]
    ...
    case constant-expression_n : statements_n; [break;]
    default: statements;
}
```

Program 2.3 - Switch statement example

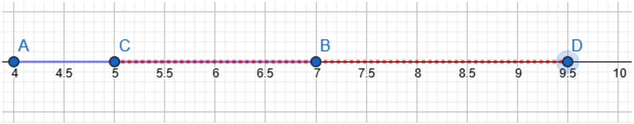
```c
#include <stdio.h>

int main() {
  char c;
  scanf("%c", &c);
  switch(c){
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u': puts("vowel"); break;
    default: puts("consonant");
  }
  return 0;
}
```

- switch works with integral types, char is one of them
- all statements are executed after the first matching case, unless break is called
- what does this print if c is a digit?

Study problem - Interval intersection

- you are given two intervals on the real line
- determine the size of the interval that is their intersection
- let [A, B] and [C, D] denote the two intervals



- we want to find the length of the common portion
- there is always a valid answer
- if the intervals do not intersect, the length of their intersection is 0

Study problem - Interval intersection

- the endpoints of the intervals can be arranged in different orders:
  - $24 = 4!$ ways - in general
  - $6 = \binom{4}{2}$ ways - if we impose $A <= B$ and $C <= D$
- the left starting point of the intersection must be the larger of A and C
- the right ending point of the intersection must be the smaller of B and D
- if these two values are in the wrong order there is no intersection

Program 2.4 - Interval intersection

```c
#include <stdio.h>

int main()
{
  float A, B, C, D;
  scanf("%f%f", &A, &B);
  scanf("%f%f", &C, &D);
  float L = A;
  if (C > L)
    L = C;
  float R = B;
  if (D < R)
    R = D;
  printf("%f\n", R>L ? R-L : 0);
  return 0;
}
```

- let L denote the left starting point of the intersection = the maximum of A and C
- let R denote the right ending point of the intersection = the minimum of B and D
- the length is R-L if they are in the correct order, otherwise 0 - use the ternary conditional operator