CSE1142 – C Data Types & Operators

Sanem Arslan Yılmaz

Some of the slides are from:
CMPE150 – Boğaziçi University
Deitel & Associates

Agenda

- Variables
  - What are they used for?
  - Rules for identifier names
- Standart Data Types
  - Integer
  - Floating-point numbers
  - Characters
- Constants
- Enumerated type
- Operators
- Type casting

Variables

- Operations (such as addition, subtraction, etc.) operate on operands.
- You need some space to store the value of each operand.
- A variable provides storage space for a value.

Variables

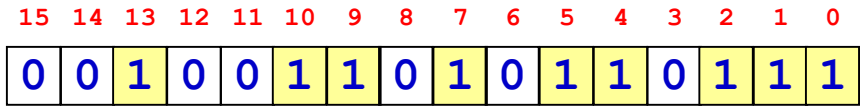
- **IMPORTANT:** The value of a variable can never be empty. The value is represented via multiple bits, each of which is either 0 or 1. So, the variable always has a value.

- When a local variable is defined, its initial value is undefined. In other words, it has an arbitrary value.

- So, make sure that the variable has a valid value before you perform any operation based on that value.

Variables

- Each variable consists of multiple bits. E.g.:



➔ $2^{13}+2^{10}+2^9+2^7+2^5+2^4+2^2+2^1+2^0=9911$

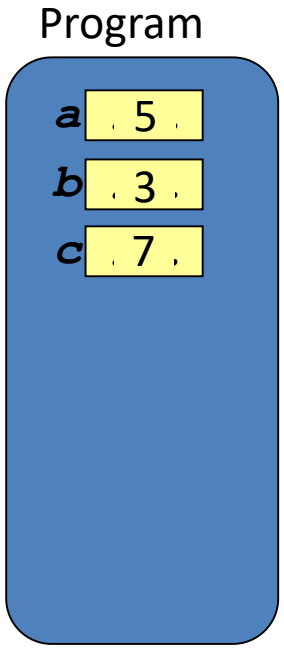
- Thus, every value is actually stored as a sequence of bits (1s and 0s) in the computer.
- The number of bits is called the size of the variable.
- The size of a variable depends on the type of the variable, the hardware, the operating system, and the compiler used.
  - So, in your programs NEVER make assumptions about the size of a variable.
  - The size may change due to the factors listed above, and your program will not work.

Variables

```
#include <stdio.h>

int main
{
    int a, b, c;

    a=10;
    b=3;
    c=a-b;
    a=b+2;
}
```



Rules for identifier names

- While defining names for variables (and also functions, user-defined types, and constants in the future) you should obey the following rules:
  - The first character of a name must be a letter or underscore (‘_’).
  - The remaining characters must be letters, digits, or underscore.
  - Only the first 31 characters are significant.
  - Avoid reserved words such as int, float, char, etc. as identifier names.
- However, it is better to avoid starting identifier names with underscore.
- Also remember that C language is case-sensitive.
- It is a very good practice to use meaningful names.

Standard data types

- You have to specify the type of a variable when you define it.

- There are three standard data types:
  - Integer (i.e., whole numbers)
  - Float (i.e., real or floating-point numbers)
  - Characters

Standard integer types

TYPE	SIZE	VALUE RANGE
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 or 8 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 or 8 bytes	0 to 4,294,967,295
long long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8 bytes	0 to 18,446,744,073,709,551,616

Integers

- There are variations of **`int`** such as **`long int`**, **`short int`**, **`unsigned int`**.
  - For each one of these types, you may ignore the word "**`int`**" and use **`long`**, **`short`**, and **`unsigned`**, respectively.

- The sizes of these types are ordered as follows:

  short int ≤ int ≤ long int

Integers

- Syntax:

```
int variable_list;
```

where `variable_list` is a comma-separated list of variable names. Each variable name may be followed by an optional assignment operator and a value for initialization.

Eg: `int a, b=10, c;`

- Integer is a class of variable types. The most basic one is `int`.
- The size may change, but the leftmost bit is used for the sign. The remaining bits represent the value in binary.
- Though the size of an int variable may vary, it is always limited, i.e., it contains a limited number of bits. Therefore, the maximum and minimum values that can be represented by an int variable is limited.

Representation of integers

- Integers are encoded using *2's complement representation*
- Leftmost bit represents the *sign* (0 for positive, 1 for negative).
- If positive, value is the numeric value of the remaining binary pattern.
- If negative, value is the numeric value of the complement(*) of the remaining binary pattern, plus one.

(*) complement: Switch 0's to 1's, and vice versa.

Representation of integers

- Suppose the bit pattern is: 0 100 0001
  - Sign bit is 0 ⇒ positive
  - Absolute value is $100\ 0001_B = 65_D$ (D: Decimal; B: Binary)
  - The represented value is $+65_D$

- Suppose the bit pattern is: 1 000 0001
  - Sign bit is 1 ⇒ negative
  - Absolute value is the complement of $000\ 0001_B$ plus 1, i.e., $111\ 1110_B + 1_B = 127_D$
  - Hence, the represented value is -127_D

Standard integer types

- If an integer type is stored in **N** bits
  - Signed range: $-2^{N}-1$ and $2^{N-1}-1$
  - Unsigned range: 0 to $2^{N}-1$.

- The ***sizeof()*** function returns the number of bytes occupied by each type.

```
#include <stdio.h>
main(){
        // sizeof's return type is long,
        // so we use the %lu format specifier.
        printf("size of char = %lu\n", sizeof(char));
        printf("size of int = %lu\n", sizeof(int));
        printf("size of long = %lu\n", sizeof(long));
}
```

The overflow problem

- What if we go over the allowed range?

```
#include<stdio.h>
int main()
{
        int a;
        a = 2147483647; // largest int
        printf("%d\n", a+1);
        return 0;
}
```

- The program prints -2,147,483,648, the smallest **int** value.
- The variable *overflows*.

The overflow problem

- To see the reason of the overflow, we look at the 2's complement representation of the numbers.

```
+2,147,483,647 =
0 111 1111 1111 1111 1111 1111 1111 1111
+1
-----------------------------------------------
1 000 0000 0000 0000 0000 0000 0000 0000
= -2,147,483,648
```

The overflow problem

- Caused the crash of the Ariane 5 rocket in 1996.
  - https://hownot2code.com/2016/09/02/a-space-error-370-million-for-an-integer-overflow/

- *Gangnam Style* overflows INT_MAX, forces YouTube to go 64-bit in 2014
  - https://www.bbc.com/news/world-asia-30288542

- Moral: Think about the possible magnitude of your numbers; choose your data type accordingly

Floating-point numbers

- Syntax:

```
float variable_list;
```

- Float type is used for real numbers.

- Note that all integers may be represented as floating-point numbers, but not vice versa.

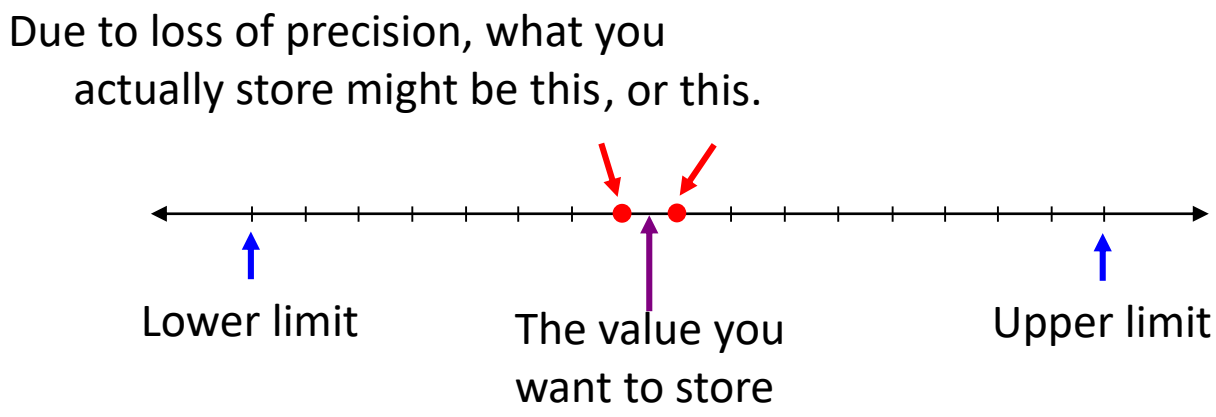
Floating-point numbers

- Similar to integers, floats also have their limits: maximum and minimum values are limited as well as the precision.

TYPE	STORAGE SIZE	VALUE RANGE	PRECISION
float	4 bytes	1.2E-38 to 3.4E+38	6 decimal places
double	8 bytes	2.3E-308 to 1.7E+308	15 decimal places
long double	10 bytes	3.4E-4932 to 1.1E+4932	19 decimal places

- The sizes of these types are ordered as follows:

  float ≤ double ≤ long double



Characters

- Syntax:

```
char variable_list;
```

- Character is the only type that has a fixed size in all implementations: 1 byte.

- All letters (uppercase and lowercase, separately), digits, and signs (such as +, -, !, ?, $, £, ^, #, comma itself, and many others) are of type character.

Characters

- Since every value is represented with bits (0s and 1s), we need a mapping for all these letters, digits, and signs.

- This mapping is provided by a table of characters and their corresponding integer values.
  - The most widely used table for this purpose is the ASCII table.

Characters

- The ASCII table contains the values for 256 values (of which only the first 128 are relevant for you).
- Each row of the table contains one character.
- The row number is called the ASCII code of the corresponding character.

Characters

- Never memorize the ASCII codes. They are available in all programming books and the Internet. (Eg: http://www.ascii-code.com)

- What is important for us is the following three rules:
  - All lowercase letters (a,b,c,...) are consecutive.
  - All uppercase letters (A,B,C,...) are consecutive.
  - All digits are consecutive.

ASCII table (partial)

ASCII code	Symbol	ASCII code	Symbol	ASCII code	Symbol	ASCII code	Symbol
...	...	66	B	84	T	107	k
32	blank	67	C	85	U	108	l
37	%	68	D	86	V	109	m
42	*	69	E	87	W	110	n
43	+	70	F	88	X	111	o
...	...	71	G	89	Y	112	p
48	0	72	H	90	Z	113	q
49	1	73	I	...	...	114	r
50	2	74	J	97	a	115	s
51	3	75	K	98	b	116	t
52	4	76	L	99	c	117	u
53	5	77	M	100	d	118	v
54	6	78	N	101	e	119	w
55	7	79	O	102	f	120	x
56	8	80	P	103	g	121	y
57	9	81	Q	104	h	122	z
...	...	82	R	105	i	...	...
65	A	83	S	106	j		

Characters

- A character variable actually stores the ASCII value of the corresponding letter, digit, or sign.

- I/O functions (printf(), scanf(), etc.) do the translation between the image of a character displayed on the screen and the ASCII code that is actually stored in the memory of the computer.

Characters

- Note that `a` and `A` have different ASCII codes (`97` and `65`).
- You could also have a variable with name `a`. To differentiate between the variable and the character, we specify all characters in single quotes, such as `'a'`. Variable names are never given in quotes.
  - Example:
```
char ch;
ch='a';
```
- Note that using double quotes makes it a string (to be discussed later in the course) rather than a character. Thus, `'a'` and `"a"` are different.
- Similarly, `1` and `'1'` are different. Former has the value `1` whereas the latter has the ASCII value of `49`.

Characters

- Example: Consider the code segment below.

```
char ch;
ch='A';
printf("Output is %c", ch);
```

- The string in printf() is stored as

```
79,117,116,112,117,116,32,105,115,32,37,99
```

  which are the ASCII codes of the characters in the string.

- When printf() is executed, it first replaces **`37,99`** (**`%c`**) with **`65`** (**`A`**), and then displays the corresponding characters on the screen.

Characters

```
#include<stdio.h>
int main()
{
        char c1, c2, c3;
        c1='a'; c2='b'; c3='\n';

        // Display characters
        printf("%c%c%c", c1, c2, c3);

        // Display numeric codes
        printf("%d %d %d", c1, c2, c3);
        return 0;
}
```

Output:

```
ab
97 98 10
```

Constants

- Syntax:

```
#define constant_name constant_value
```

- As the name implies, variables store values that vary while constants represent fixed values.

- Note that there is no storage when you use constants. Actually, when you compile your program, the compiler replaces the constant name with the value you defined.

- The pre-processor replaces every occurrence of **`constant_name`** with everything that is to the right of **`constant_name`** in the definition.
  - Note that there is no semicolon at the end of the definition.

- Conventionally, we use names in uppercase for constants.

Constants

```
#include <stdio.h>
#define N 10
int main()
{
        int friends = 5;
        printf("I have %d apples\n", N);
        printf("We each get %d apples\n", N/friends);
}
```

The preprocessor replaces all occurrences of N with 10
The compiler sees:

```
. . .
printf("I have %d apples\n", 10);
printf("We each get %d apples\n", 10/friends);
. . .
```

Enumerated type

- Used to define your own types.

- Syntax:

```
enum type_name {
        item_name=constant_int_value, ...
} variable_list;
```

- By default, the value of the first item is 0, and it increases by one for consecutive items. However, you may change the default value by specifying the constant value explicitly.

- Eg:

```
enum boolean {FALSE,TRUE} v1, v2;
enum days {SUN,MON,TUE,WED,THU,FRI,SAT};
enum {one=1,five=5,six,seven,ten=10,eleven} num;
```

Operators

- Assignment operator (=)
  - The value of the expression on the RHS is assigned (copied) to the LHS.
  - It has right-to-left associativity.

```
a = b = c = 10;
```

is the same as

```
a = (b = (c = 10));
```

makes all three variables `10`.

Assignment and type conversion

- When a variable of a narrower type is assigned to a variable of wider type, no problem.
  - Eg:
```
int a=10;
float f;
f=a; // f is 10.00
```

- However, there is loss of information in reverse direction.
  - Eg:
```
float f=10.9;
int a;
a=f; // a is 10
```

Operators

- Arithmetic operators (+,-,*,/,%)
  - General meanings are obvious.
  - + (addition)
  - - (subtraction)
  - * (multiplication)
  - / (division)
  - % (remainder)

```
int main()
{
        int a, b, c;
        a = 13; b = 5;
        c = a + b; // c is 18
        c = a - b; // c is 8
        c = a * b; // c is 65
        c = a / b; // c is 2
        c = a % b; // c is 3
}
```

Operators

- What is important is the following:
  - If one of the operands is of a wider type, the result is also of that type.
  - Eg: Result of int+float is float.
  - Result of float+double is double.

- Remainder operator (%) accepts only integer operands.

Integer and float division

- Division of two integers returns the integer part of the fraction.
  - 5 / 4 returns 1, not 1.25

- Division of one floating-point and an integer, or of two floating-point values returns a floating-point result.
  - 5.0 / 4 returns 1.25

Integer and float division

- Suppose we divide two integer variables and desire a floating-point result.

- We can either multiply one operand with 1.0:

```
int a, b; float c;
a = 5; b = 4;
c = 1.0 * a / b;
```

- Or, we can *cast* one operand temporarily into a float

```
c = (float)a / b;
```

Operators

- Logic operators (&&, ||, !)
  - Logic operators take integer class operands.
    - Zero == false.
    - Anything non-zero == true.
  - "`&&`" does a logical-AND operation. (True only if both operands are true.)
  - "`||`" does a logical-OR operation. (False only if both operands are false.)
  - "`!`" does a negation operation. (Converts true to false, and false to true.)

Operators

- Logic operators follow the logic rules

a	b	a && b	a || b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

- The order of evaluation is from left to right
- As usual parenthesis overrides default order

Operators

```
#include<stdio.h>
int main(){
        int a, b, c;
        a = 0; b = 1; c = 5;
        printf("%d AND %d = %d\n", a, b, a && b);
        printf("%d AND %d = %d\n", b, c, b && c);
        printf("%d OR %d = %d\n", a, b, a || b);
        printf("%d OR %d = %d\n", b, c, b || c);
}
```

Output:

```
0 AND 1 = 0
1 AND 5 = 1
0 OR 1 = 1
1 OR 5 = 1
```

Operators

- If the first operand of the "`&&`" operator is **false**, the second operand is not evaluated at all (since it is obvious that the whole expression is false).
  - Eg: In the expression below, if the values of `a, b` and `c` are initially `1, 0` and `1`, respectively,

    ```
    a == b && (c=2)
    ```

    then the second operand is not evaluated at all, so `c` keeps its value as `1`.

- Similarly, if the first operand of the "`||`" operator is **true**, the second operand is not evaluated at all.

Augmented operators

- A shortcut for

```
a = a + b;
```

  is

```
a += b;
```

- Similarly

```
a -= b;
```
 is
```
a = a - b;
```

```
a *= b;
```
 is
```
a = a * b;
```

```
a /= b;
```
 is
```
a = a / b;
```

```
a %= b;
```
 is
```
a = a % b;
```

- Convenient in loops, where variables are periodically updated.

Operators

- Pre/Post increment/decrement operators (++, --)
  - The operator ++ increments the value of the operand by 1.
    - If the operator comes BEFORE the variable name, the value of the variable is incremented before being used, i.e., the value of the expression is the incremented value. This is pre-increment.
    - In post-increment, the operator is used after the variable name, and incrementation is performed after the value is used, i.e., the value of the expression is the value of the variable before incrementation.

Operators

- Eg:

```
a=10;          c=10,
b=++a;         d=c++;
```

Both `a` and `c` will be come `11`, but `b` will be `11` while `d` is `10`.

Comparison Operators

- Comparison operators (==,!=,<,<=,...)
  - Return 1 if the relation is true, 0 if false.
  - IMPORTANT: When you compare two float values that are supposed to be equal mathematically, the comparison may fail due to the loss of precision discussed before.

Comparison Operators

Symbol	Usage	Meaning
==	x == y	is x equal to y?
!=	x != y	is x not equal to y?

>	x > y	is x greater than y?
<	x < y	is x less than y?
>=	x >= y	is x greater than or equal to y?
<=	x <=y	is x less than or equal to y?

Operators

- Bitwise operators (&, |, ^, <<, >>, ~)
  - Bitwise operators take integer class operands.
    - For the logic operators, the variable represents a single logical value, true or false.
    - For the bitwise operators, each bit of the variable represents true or false.
  - `&`, `|`, and `^` perform bitwise-AND, -OR, -XOR, respectively.
  - `<<` and `>>` perform left- and right-shifts.
  - "`~`" takes bitwise one's complement.

Operators

Operation	Result
5 & 10 (0000 0101 & 0000 1010)	0 (0000 0000)
5 && 10 (0000 0101 && 0000 1010)	1 (0000 0001)
5 | 10 (0000 0101 | 0000 1010)	15 (0000 1111)
8 ^ 10 (0000 0111 ^ 0000 1010)	13 (0000 1101)
7 << 2 (0000 0111 << 0000 0010)	28 (0001 1100)
7 >> 2 (0000 0111 >> 0000 0010)	1 (0000 0001)
~5 (~0000 0101)	-6 (in two's complement) (1111 1010)

Expressions and precedence

- Some operators are evaluated before others:
  4+5*2 is 14, not 18.

- Highest precedence: Parentheses
  (4+5)*2 == 18

- *, /, % are evaluated before +, -
  4 * 5 – 2 + 6 / 3 – 2 == (4*5) – 2 + (6/3) – 2
  == 18

Operators with equal precedence

- Within themselves, *, /, % are evaluated left-to-right.
  4 * 5 / 2 == 10, while 4*(5/2) == 8 (integer division)

- Called "left-to-right associativity".

- Comparison operators have lower precedence than arithmetic operators.
  2 + 3 < 7 – 1 is the same as (2+3) < (7-1)

- Logical operators have lower precedence than comparison operators.
  x < 3 && x > 1 is the same as (x < 3) && (x > 1)

C operator precedence table

Operator	Associativity
() [] . ->	left-to-right
++ -- + - ! ~ (type) * & sizeof	right-to-left
* / %	left-to-right
+ -	left-to-right
<< >>	left-to-right
< <= > >=	left-to-right
== !=	left-to-right
&	left-to-right
^	left-to-right
|	left-to-right
&&	left-to-right
||	left-to-right
?:	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	right-to-left
,	left-to-right

Operator precedence, simplified

- No need to memorize the table.

- Remember simple rules
  - Parentheses are evaluated first.
  - *, /, % come before +, -.
  - Comparisons come after arithmetic.
  - Assignments come last.

- When in doubt about other operators, use parentheses.
  - Instead of

```
x + 1 < 3 && y / 2 == 5 || x + 2*y > 10
```

    write

```
( ( x + 1 < 3 ) && (y / 2 == 5) ) || (x + 2*y > 10)
```

Type casting

- Also called *conversion* or *type conversion*.
- It does NOT change the type of a variable. It is not possible to change the type of a variable.
- What casting does is to convert the type of a value.

Type casting

- Eg:
```
int a=10, b=3;
float f, g;
f=a/b;
g=(float)a/b;
```

- The type of `a` does not change; it is still and integer. However, in the expression `(float)a/b`, the value of `a`, which is `10`, is converted to float value of `10.0`, and then it is divided by `b`, which is `3`. Thus, we perform float division and `g` becomes `3.3333...`

- On the other hand, we perform an integer division for `f`, so it becomes `3`.