Southern Technical University
Technical Institute / Qurna
Dep. of Computer Systems Techniques

Second class
Subject : Data Structures
Lecturer : Israa Mahmood Hayder
Lecture no.2

اسلوب تمثيل الهياكل البسيطة

(Primitive Data Structures)

--الاسبوع الثاني -

الثاني - الثالث	* إسلوب تمثيل هياكل البيانات البسيطة *primitive data structures* representation. - الأعداد الصحيحة Integer . - الأعداد الحقيقية Real . - الرموز Characters. - السلاسل الرمزية Strings . - المؤشرات Pointers . - البيانات المنطقي Logical Data

B// Rationale (مبررات الوحدة) :-

The primitive data structure is the simplest and very important type to start with and to know how they are represented, how much memory size required to represent each of them, and the functions that used with these types in C++ language.

C// Central Ideas (الفكرة المركزية):-

- Introduction to Primitive data structures:
- Integers
- Real Numbers
- Characters

D// Objectives (أهداف الوحدة):-

After studying this unit , the student will be able to:-

- Realize how the integers are represented using:
  1) sign and magnitude 2) 1's complement 3) 2's complement
- Represent the real numbers using:

1) Fixed point method 2) Floating point method

QUIZ :

Circle the correct answer:

1- The number (-9) in 1's complement and module 32 is equal to:

a)22 b) 15 c)6

2-How many bits the fraction part requires to be represented in fixed point method module 16: a) 24 bit b)5 bit c)7 bit

3- Sign and magnitude require:

a) 32 bit b) 16 bit c) 8 bit

4- In 32 bit floating point, exponent takes: a) 1 bit b) 8 bits c)23 bits

5-To represent the short data in memory we require:

a) 2bytes b) 1 byte c) 1bit

3- The Text (عرض الوحدة النمطية):

1- Introduction to Primitive data structures:

Abstract data types are divided into two categories, primitive data types and user-defined data types. A primitive data type is defined by the programming language, such as the data types you learned about in the previous unit. Some programmers call these built-in data types.

You determine the amount of memory to reserve by determining the appropriate abstract data type group to use and then deciding which abstract data type within the group is right for the data.

There are four data type groups:

- **Integer** Stores whole numbers and signed numbers. Great for storing the number of dollars in your wallet when you don't need a decimal value.
- **Floating-point** Stores real numbers (fractional values). Perfect for storing bank deposits where pennies (fractions of a dollar) can quickly add up to a few dollars.
- **Character** Stores a character. Ideal for storing names of things.
- **Stri**ng
- **Boolean** Stores a true or false value. The correct choice for storing a yes or no or true or false response to a question.
- **Pointers**

A // Methods for representing integers:

The *integer* abstract data type group consists of four abstract data types used to reserve memory to store whole numbers: byte , short , int , and long as described in previous unit. Depending on the nature of the data, sometimes an integer must be stored using a positive or negative sign, such as a +10 or –5. Other times an integer is assumed to be positive so there isn't any need to use a positive sign. An integer that is stored with a sign is called a *signed number;* an integer that isn't stored with a sign is called an *unsigned number*.

Declaration in C++: Int x;

An integer is a whole number which may be negative. The number must therefore be encoded in such a way as to tell if it is positive or negative, and to follow the rules of addition. The trick involves using an encoding method called *twos complement*.

- **A positive integer or zero** will be represented in binary (base 2) as a natural number, except that the highest-weighted bit (the bit on the far left) represents the plus or minus sign. So for a positive integer or zero, this bit must be set to 0 (which corresponds to a plus sign, as 1 is a minus sign). Thus, if a natural number is encoded using 4 bits, the largest number possible will be 0111 (or 7 in decimal).

Generally, the largest positive integer encoded using *n* bits will be $2^{n-1}-1$.

- **A negative integer** is encoded using twos complement.

The principle of *twos complement*:
Choose a negative number.

- Take its absolute value (its positive equivalent)
- It is represented in base 2 using n-1 bits
- Each bit is switched with its complement (i.e. the zeroes are all replaced by ones and vice versa)
- Add 1

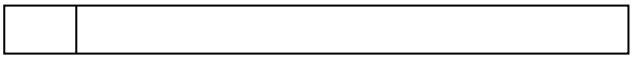
Note that by adding a number and its twos complement the result is 0

Let's see this demonstrated in an example:
We want to encode the value -5 using 8 bits. To do so:

- write 5 in binary: 00000101
- switch it to its complement: 11111010
- add 1: 11111011
- the 8-bit binary representation of -5 is 11111011

Comments:
The highest-weighted bit is 1, so it is indeed a negative number.
If you add 5 and -5 (00000101 and 11111011) the sum is 0 (with remainder 1).

1- *Sign and magnitude*: used to represent the signed numbers.



Sign magnitude

Sign: 0- positive , 1- negative

Number of bits in magnitude ($M = R^n$)
n: الاس (power)
R: الاساس (Base)

Ex1 Express the integer (-9) in sign and magnitude and module 16.

$M = R^n$

$16 = 2^n$ → 16= 2^n then we need to use 4 bits for representing the number+1bit for sign.:

-9 = | 1 | 1 0 0 1 |

Ex2 Express the integer (-7) in sign and magnitude and module 32.

$M = R^n$

$32 = 2^5$ =(need 5 bits) → (-7)= | 1 | 0 0 1 1 1 |

Disadvantage:

1- Need additional bit for sign
2- Need time to determine the operation (addition/subtraction)
3- Need address and substructure machine

2- ***2's complement***: Do not use sign bit, and it reduce subtraction and time.

$$2\text{'s complement(-x)} = M - |x|$$

Ex1 Express the integer (-2) in 2's complement and module 16.

$16\text{-}2=(10000)_2 - (10)_2 = (1110)_2$ or $16\text{-}2=14 = (1110)_2$

Ex2 Express the integer (-38) in 2's complement and module 32.

-38 mod 32= -6
$32\text{-}(6) = 26 = (11010)_2$

Ex3 Express the integer (-47) in 2's complement and module 16.

-47 mod 16= -15
$16\text{-}(15) = 1 = (0001)_2$

Ex4 Evaluate the operation (3+4) , (3-4) and (-3+4) in 2's complement and module 16.

a) $3+4 = (0011)_2 + (0100)_2 = (0111)_2 = 7$
b) $3\text{-}4 = (0011)_2 + 2\text{'s compl}(4) = (0011)_2 + (1100)_2$
$= (1111)_2$

التحقيق : 3-4 =-1 ← 1111→0000+1=0001
لان الناتج سالب ناخذ المتمم

3- **1's complement:** Do not use sign bit, and it reduce subtraction and time.

$$1\text{'s complement(-x)} = M - |x| - 1 \quad \text{or} \quad M + X - 1$$

Ex1 Express the integer (-9) in 1's complement and module 16.

1's compl(-9)= 16-9-1= 6= $(110)_2$

Ex2 Express the integer (-18) in 1's complement and module 32.

1's compl(-18)= 32-18-1 = 13 =$(01101)_2$

Ex3 Evaluate the operation (3-4) in 1's complement and module 16.

3-4 = $(0011)_2$ + 1's compl(4) = $(0011)_2$ + $(1011)_2$
=$(1110)_2$

التحقيق : 3-4 =-1 ← 1's compl(1110) = 0001
لان الناتج سالب ناخذ المتمم

B// Methods for representing Real Numbers:

1- Fixed Point Method: It uses 16- bit as follows

4-bits Sign **7-bits Magnitude** **5-bits Fraction**

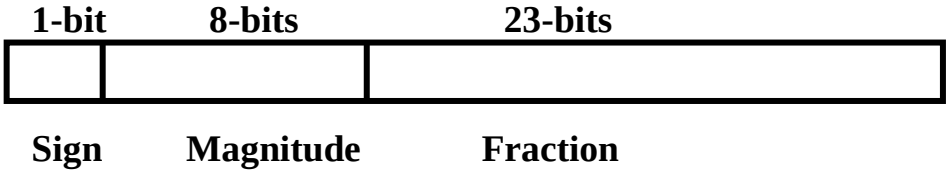
2- Floating-point method:

The term *floating-point* refers to the way decimals are referenced in memory. There are two parts of a floating-point number: the real number, which is stored as a whole number, and the position of the decimal point within the whole number. This is why it is said that the decimal point "floats" within the number. The float abstract data type is used for real

numbers that require single precision, such as United States currency. *Single precision* means the value is precise up to 7 digits to the right of the decimal.

Declaration in C++:
Float x;

Floating point uses 32 bits for representing too large or too small numbers.



The goal is to represent a number with a decimal point in binary (for example, *101.01*, which is not read *one hundred one point zero one* because it is in fact a binary number, i.e. *5.25* in decimal) using the form 1.XXXXX... * 2^n (in our example, 1.0101*2^2). IEEE standard 754 defines how to encode a real number.
This standard offers a way to code a number using 32 bits, and defines three components:

- the plus/minus sign is represented by one bit, the highest-weighted bit (furthest to the left)
- the exponent is encoded using 8 bits immediately after the sign
- the mantissa (the bits after the decimal point) with the remaining 23 bits

Thus, the coding follows the form:
seeeeeeeemmmmmmmmmmmmmmmmmmmmmm

- the **s** represents the sign bit.
- each **e** represents an exponent bit
- each **m** represents a mantissa bit

However, there are some restrictions for exponents:

- the exponent 00000000 is forbidden
- the exponent 11111111 is forbidden However, they are sometimes used to report errors. This numeric configuration is called *NaN*, for *Not a number*.
- 127 (01111111) must be added to the exponent in order to convert the decimal to a real number in binary. The exponents, therefore, can range from -254 to 255
- Thus, the formula for expressing real numbers is:
- (-1)^S * 2^(E - 127) * (1 + F)
- where:
- S is the sign bit and so 0 is understood as positive (-1^0=1).

- E is the exponent to which 127 must be added to obtain the encoded equivalent.
- F is the fraction part, the only one which is expressed, and which is added to 1 to perform the calculation.

Here is an **example**:
The value **525.5** is to be encoded.

- 525.5 is positive, so the first bit will be 0.
- Its representation in base 2 is: 1000001101.1
- By normalising it, we get: $1.0000011011*2^{\wedge 9}$
- Adding 127 to the exponent, which is 9, gives 136, or in base 2: 10001000
- The mantissa is composed of the decimal part of 525.5 in normalised base 2, which is 0000011011.
- As the mantissa must take up 23 bits, zeroes must be added to complete it: 00000110110000000000000
- The binary representation of 525.5 under IEEE standard 754 is therefore:
  0 1000 1000 00000110110000000000000
  0100 0100 0000 0011 0110 0000 0000 0000 (4403600 in hexadecimal)

Here is another **example**, this time using a negative real number :
The value **-0.625** is to be encoded.

- The s bit is 1, as 0.625 is negative.
- 0.625 is written in base 2 as follows: 0.101
- We want to write it in the form 1.01 x 2-1
- Consequently, the exponent is worth 1111110 as 127 - 1 = 126 (or 1111110 in binary)
- The mantissa is 01000000000000000000000 (only the digits after the decimal point are represented, as the whole number is always equal to 1)
- The binary representation of the number 0.625 under IEEE standard 754 is:
  1 1111 1110 01000000000000000000000
  1111 1111 0010 0000 0000 0000 0000 0000 (FF 20 00 00 in hexadecimal)

Quiz1:

Represent the number $(101.11100)_2$ in memory

3-Types of floating point Binary reals

1- Single precession (32 bit) floating point:

1 Bit for sign
8 bit for exponent
23 bit for fractional part

2- Single precession (64 bit) floating point:

1 Bit for sign
11 bit for exponent
52 bit for fractional part

3- Single precession (80 bit) floating point:

1 Bit for sign
16 bit for exponent
63 bit for fractional part

If number =

0 00000000 00000000… → 0
1 00000000 00000000… → -0
0 11111111 00000000 … → ∝
1 11111111 00000000 … → -∝
x 11111111 1xxxxxxx … → QNot a name
x 11111111 0xxxxxxx … → Snot a name

4-Numerical Functions in C++

Sin(), cos(), abs() tan(), atan()
Log(), log10(), power(), sqrt(), rnd()

Quiz2:

1- Wirite program to print the size of the primitive types using the function sizeof().
2- Evaluate 4-12 in module 16 using 1's complement method.
3- Express 11000110.1 using 32 bit floating point method.

References:

1- http://en.kioskea.net/s/representation-of-real-numbers-and-integers (URL)

٢- سلسلة ملخصات شوم للبرمجة بلغة C++، الجزء الاول،الطبعة الاولى ،جون ر.هيوبارد، الدار الدولية للاستثمارات الثقافية، ٢٠٠٠ .

٣- هياكل البيانات / الطبعة الثانية، تاليف د.عصام الصفار،اصدارات السفير للنشر/ بغداد، ٢٠٠١.

٤- الحقيبة التعليمية مادة "هياكل البيانات "، اعداد : نفارت الياس يوسف ،المعهد التقني كركوك