Ministry of Higher Education & Scientific Research
Al-Furat Al-Awsat Technical University
Karbala Technical Institute
Department of Computer System Techniques.

Data Structures

Second Stage

Definition of Data Structures

Assist. Prof. Dr. Wathiq Laftah Al-Yaseen

2022-2023

In this lesson we will learn …

- Definition of data structures.
- Basic concept of data structures.
- Data structure types.
- Select of a particular data structure.

Data Structure

Data structure is the structural representation of logical relationships between elements of data. In other words a data structure is a way of organizing data items by considering its relationship to each other.

Data structure mainly specifies the structured organization of data, by providing accessing methods with correct degree of associativity. Data structure affects the design of both the structural and functional aspects of a program.

Algorithm + Data structure = Program

Basic Concepts of Data Structures

- **Data**: Is the fact that we can see and deal with in our daily life like; book, car, 1245, ….etc.

- **Information**: Is a collection of words, numbers, dates, or communicated material that have meaning.

Data is the raw material while the information is the processed form of data.

$$\text{Data} \xrightarrow{\text{Processing}} \text{Information}$$

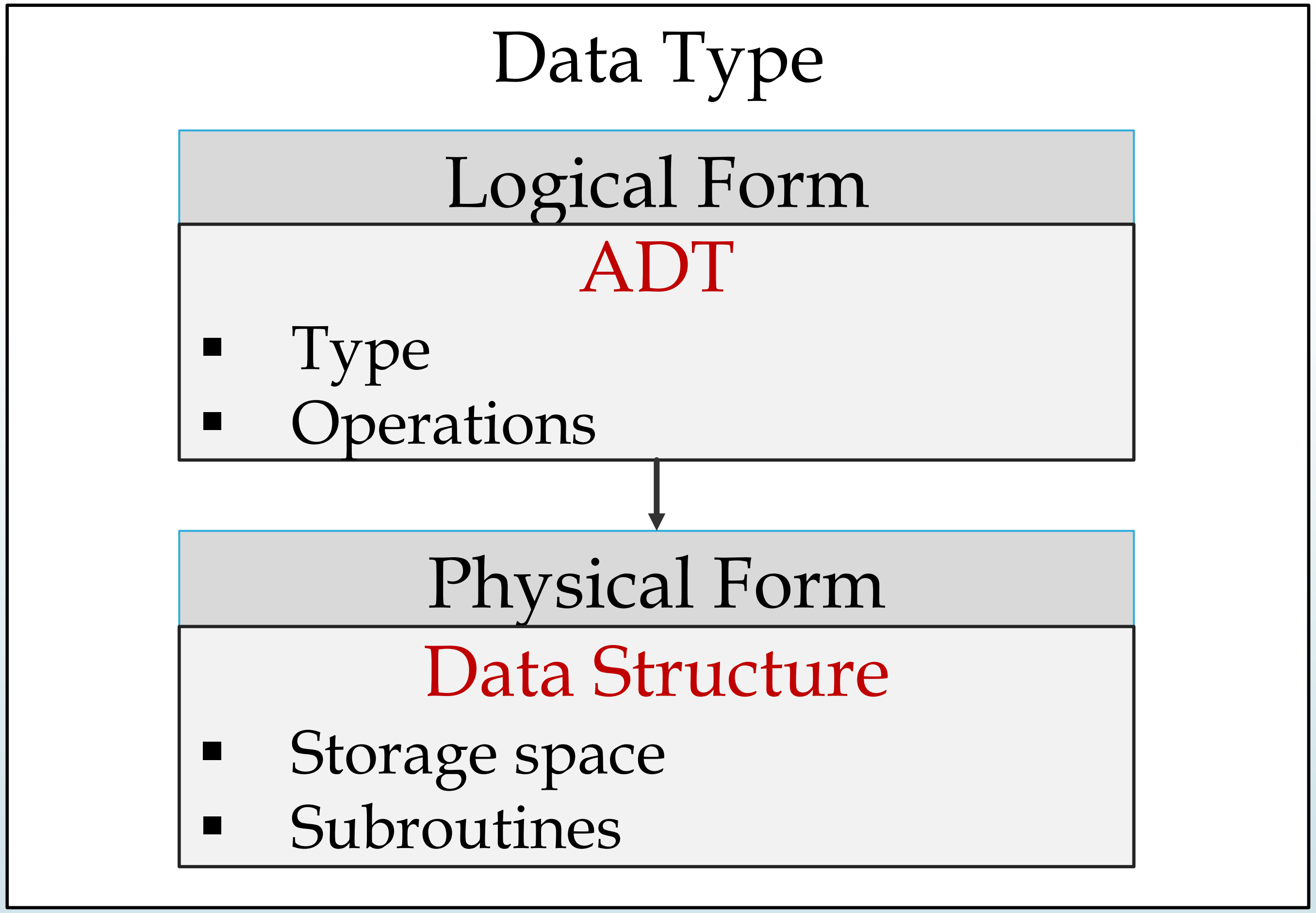
- **Type**: is a collection of values. For example, the **Boolean** type consists of the values **True** and **False**. The **integers** also form a type. An **integer** is a simple type because its values contain no subparts. A **bank account record** will typically contain several pieces of information such as **name**, **address**, **account number**, and **account balance**. Such a record is an example of an **composite type**.

- **Data item**: is a piece of information or a record whose value is drawn from a type. For example, the value **True** is a **data item** from the **Boolean** type.

- **Data type**: is a type together with a collection of operations to manipulate the **type**. A **data type** consists of two parts, a set of **data items** and **fundamental operations** on this set. We can see that the **integer type** consists of values (whole numbers in some defined range) and **operations** (addition, subtraction, multiplication and division, etc.).

- **Abstract data type (ADT)**: is the realization of a **data type** as a software component.

Data Structure is the implementation for an ADT.



This figure illustrate the relationship between ADT and data structures. The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

Problems and Programs

The selection of a particular data structure will help the programmer to design more efficient programs that will be used to solve complexity problems.

The programmers have to strive hard to solve these problems. If the problem is analyzed and divided into sub problems, the task will be much easier.

The simplest and most straightforward way of solving a problem may not be sometimes the best one. Moreover, there may be more than one program to solve a problem.

Analysis of Programs

To choice of a particular program depends on following performance analysis and measurements.

- Space complexity

The amount of memory that program needs to run completion. The space needed by program consist of:

1. Instruction space. $\longrightarrow$ Fixed space.
2. Data space.
   - Constant and simple variables. $\longrightarrow$ Fixed space.
   - Fixed size structural variables, like array and structures.
   - Dynamically space. Such as strings.

Analysis of Programs

3. Environment stack space.

Needed to store the information to resume the suspended functions. The information that be saved are:

- Return address of the called functions.
- Values of all the parameters which are send and return between functions and called points.

Analysis of Programs

- Time complexity

The amount of time that program needs to run completion.

The exact time will depend on the implementation of the algorithm, programming language, optimizing the capabilities of the compiler used, the CPU speed, other hardware characteristics/specifications and so on.

Data Structure Types

Programming languages support defined and use data items by providing formulas to single-value items such as (integer, real, character, and boolean) and the multi-value items need to different data structures like (array, record, ..., etc.).

C++ supports the following data types:

Data Structure Types

Data Structure

- **Primitive D.S. (Primary)**
  - Integer type. (int, short, long)
  - Floating point type. (float, double, long double)
  - Char.
  - Boolean.
  - Pointers.
- **Non-Primitive D.S.**
  - **Non-Linear**
    - Graph.
    - Tree.
  - **Linear**
    - **Non-Linked List**
      - Array.
      - Stack.
      - Queue.
    - **Linked List**
      - Simple LL.
      - Double LL.
      - Circular LL.

Selection of Data Structure

Before any data can be stored in memory, you must tell the computer how much space to reserve for data by using an abstract data type.

Memory is reserved by using a data type in a declaration statement. The form of a declaration statement varies depending on the programming language you use. Here is a declaration statement for C++:

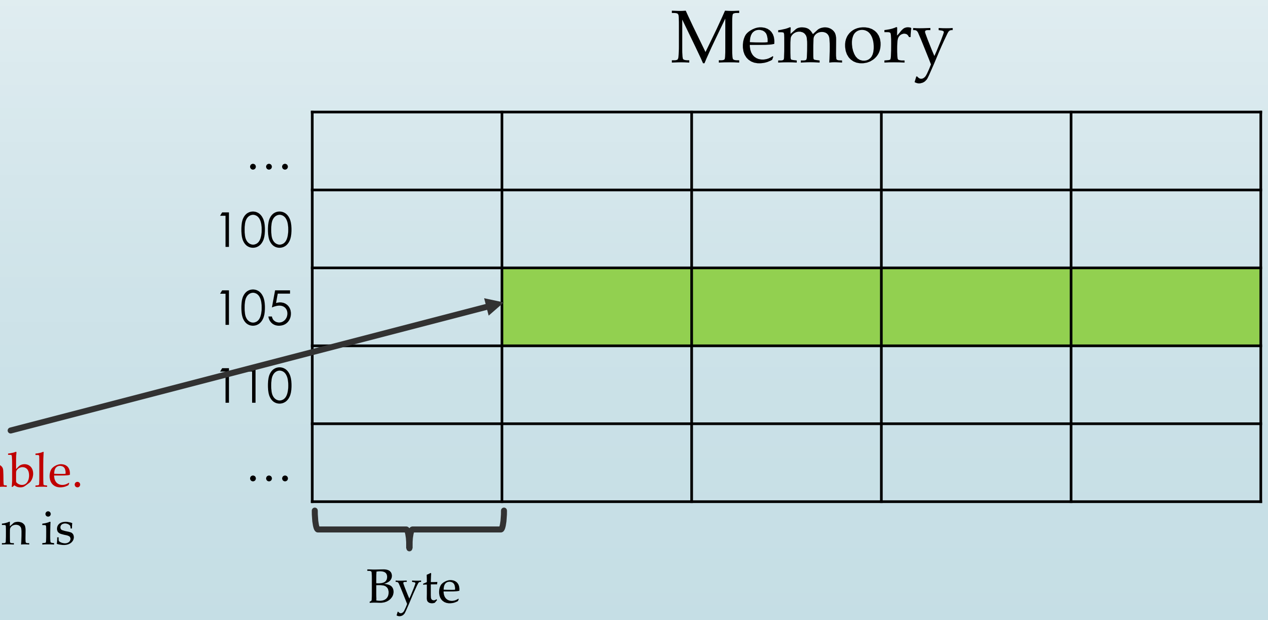
```
int myVariable;
```

Selection of Data Structure

We need to choose the suitable abstract data type for data that we want stored in memory, then use the abstract data type in a declaration statement to declare a variable. A variable is a reference to the memory location that we reserved using the declaration statement.

For Example:

int X;



4 bytes reserved to X variable.
The address of this location is saved in X.

Types of primitive data types

Data Type	Size (Byte)	Range of Values	Group
Byte	1	–128 to 127	Integers
short	2	–32,768 to 32,767	Integers
int	4	–2,147,483,648 to 2,147,483,647	Integers
long	8	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Integers
char	2	65,536 (Unicode)	Characters
float	4	3.4e-038 to 3.4e+038	Floating-point
double	8	1.7e-308 to 1.7e+308	Floating-point
boolean	1	0 or 1	Boolean

Ministry of Higher Education & Scientific Research
Al-Furat Al-Awsat Technical University
Karbala Technical Institute
Department of Computer System Techniques.

Data Structures

Second Stage

Primitive Data Structures Representation

Assist. Prof. Dr. Wathiq Laftah Al-Yaseen

2022-2023

In this lesson we will learn …

Primitive Data Structures Representation

- Integer numbers
- Floating point numbers
- Characters and Strings
- Pointers
- Logical Data

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 we learned in lesson (1), int, float, etc. Some programmers call these built-in data types.

There are four data type groups:

- **Integer** stores whole numbers and signed numbers; age, temperature, etc.
- **Floating-point** stores real numbers (fractional values); average, deposit, etc.
- **Character** stores a character; names, address, etc.
- **Boolean** stores a true or false value; sex, .

Integer representation

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.

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;**

Positive Integer or Zero

It will be represented in binary as a natural number, except that the most significant 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 $2^{n-1}-1$.

Negative Integer

There are two methods to represent and encoding the negative integer numbers.

- **Sign-Magnitude representation.**
- **Two's complement representation.**

Sign-Magnitude

The Most Significant Bit is used to represent the sign. **'1'** is used for **'-'** (negative sign), and **'0'** is used for **'+'** (positive sign).

The format of a Sign-Magnitude number in 8-bits is:

Position	7	6	5	4	3	2	1	0
Bits	MSB							LSB

$$\underbrace{\text{7}}_{\text{Sign}}\ \underbrace{\text{6 5 4 3 2 1 0}}_{\text{Magnitude}}$$

We can notice, the sign of number need for 1 bit, while the value of number need to 7 bits.

Sign-Magnitude

In order to represent the negative integer number by Sign-Magnitude method, we need to apply the following steps.

1. Take absolute value of number (Its positive equivalent).

2. Convert number to binary (Represent using n-1 bits).

3. Flip the most significant bit to 1.

Sign-Magnitude

Example (1): Express the integer number (-7) by using sign-magnitude representation with 4-bits.

1. Take the absolute of number.

$|-7| = 7$

2. Represent the resulted number in binary.

$(7)_{10} \longrightarrow (0111)_2$

3. Flip the MSB to 1.

$(0111)_2 \longrightarrow (1111)_2$

$$\boxed{\therefore (-7)_{10} = (1111)_2}$$

Sign-Magnitude

Example (2): Express the integer number (-9) by using sign-magnitude representation with 8-bits.

1. Take the absolute of number.

$$|-9| = 9$$

2. Represent the resulted number in binary.

$$(9)_{10} \longrightarrow (00001001)_2$$

3. Flip the MSB to 1.

$$(00001001)_2 \longrightarrow (10001001)_2$$

$$\boxed{\therefore (-9)_{10} = (10001001)_2}$$

Sign-Magnitude

Disadvantage:

1. Need additional bit for sign.
2. Not convenient for arithmetic.

   In decimal

   $(-4)_{10} + (1)_{10} = (-3)_{10}$

   In Sign-Magnitude

   $(1100)_2 + (0001)_2 = (1101)_2 = (-5)_{10} \neq (-3)_{10}$
3. They are two representations of zero.

   $(00000000)_2 = (+\ 0)_{10}$

   $(10000000)_2 = (-\ 0)_{10}$

Two's Complement (2's Complement)

The principle of *two's complement* are:

Choose a negative number.

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

Two's Complement (2's Complement)

Example (3): Express the integer number (-4) by using 2's complement representation with 4-bits.

1. Take its absolute value.

   $|-4| = 4$

2. It is represented in binary.

   $(4)_{10} \longrightarrow (0100)_2$

3. Switched each bit with its complement.

   $(0100)_2 \longrightarrow (1011)_2$

4. Add 1.

   $(1011)_2 + (0001)_2 = (1100)_2$

$$\boxed{\therefore (-4)_{10} = (1100)_2}$$

Two's Complement (2's Complement)

Example (4): Evaluate the operation (3+4) , (3-4) and (-3+4) in 2's complement with using 4-bits.

$3+4 = (0011)_2 + (0100)_2 = (0111)_2 = 7$

$3-4 = (0011)_2 - (0100)_2 = (0011)_2 + [2\text{'s complement of } (-4)]$
$\quad = (0011)_2 + (1100)_2 = (1111)_2 = -1$

$-3+4 = [2\text{'s complement of } (-3)] + (0100)_2 = (1101)_2 + (0100)_2$
$\quad = (0001)_2 = 1$

Two's Complement (2's Complement)

Advantages:

1. Only has one value for zero.
2. Convenient for arithmetic.

Summary of integer representation using 4-bits

Content of Memory	Unsigned	Sign-Magnitude	2's Complement
0000	0	+0	+0
0001	1	+1	+1
0010	2	+2	+2
0011	3	+3	+3
0100	4	+4	+4
0101	5	+5	+5
0110	6	+6	+6
0111	7	+7	+7
1000	8	-0	-8
1001	9	-1	-7
1010	10	-2	-6
1011	11	-3	-5
1100	12	-4	-4
1101	13	-5	-3
1110	14	-6	-2
1111	15	-7	-1

Real Representation

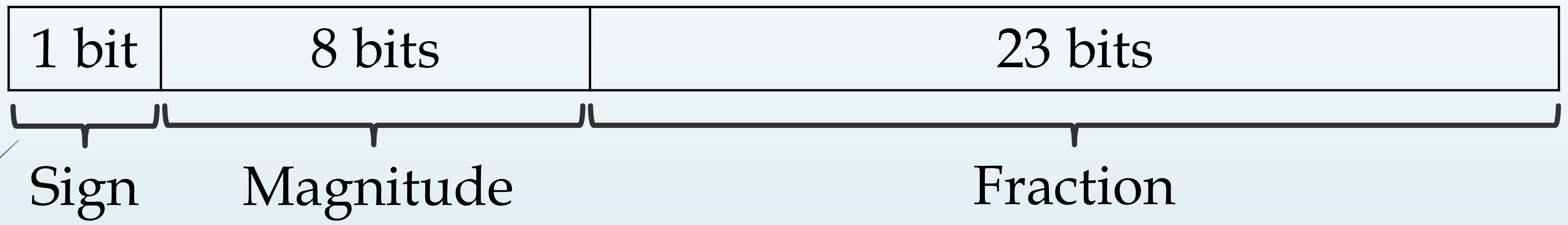
Computers which work with real arithmetic use a system called *floating point*. Floating point describes a system for representing real numbers which supports a wide range of values.

Declaration in C++: **float x;**

The term "floating point" refers to the fact that the decimal point can be placed anywhere relative to the significant digits of the number.
The most commonly encountered representation is that defined by the **IEEE 754 Standard (Institute for Electrical and Electronics Engineers)**.

Real Representation

The structure of floating point (*float* in C++) that uses 32 bits (4 bytes) as follows:



The goal is to represent a real number with a decimal point in binary by using a ***normalization method*** with form $1.XXX\ldots \times 2^{\text{exponent}}$.

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)

Real Representation

IEEE standard 754 offers a way to encoding a number using 32 bits, and defines three components:

- The positive/negative sign is represented by one bit, the most significant 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 encoding of real number follows the form:

s-eeeeeeee-mmmmmmmmmmmmmmmmmmmmmmm

Where s: sign; e: exponent; m: mantissa

Decimal floating point to IEEE standard representation

1. Compute the binary equivalent of the integer part and the fractional part.
2. Normalize the number by moving the decimal point to the right of the leftmost one, in order to get the form:

$$1.\underbrace{XXX\ldots.}_{\text{Mantissa}} \times 2^{\text{exponent}}$$

3. Add 127 to the exponent in order to convert the decimal into a real number in binary.
4. Store the results from 1 to 3 as follows:

Sign	Exponent (from step 3)	Mantissa (from step 2)

Real Representation

Example (5): Find the IEEE representation of $(4.25)_{10}$ using 32 bit.

1. The binary equivalent of $(4.25)_{10}$ is $(100.01)_2$.

$0.25 \times 2 = 0.5$

$0.5 \times 2 = 1.0$

$\therefore\ (0.25)_{10} = (01)_2$

2. Normalize the $(100.01)_2$ to $[(1.0001)_2 \times 2^2]$.

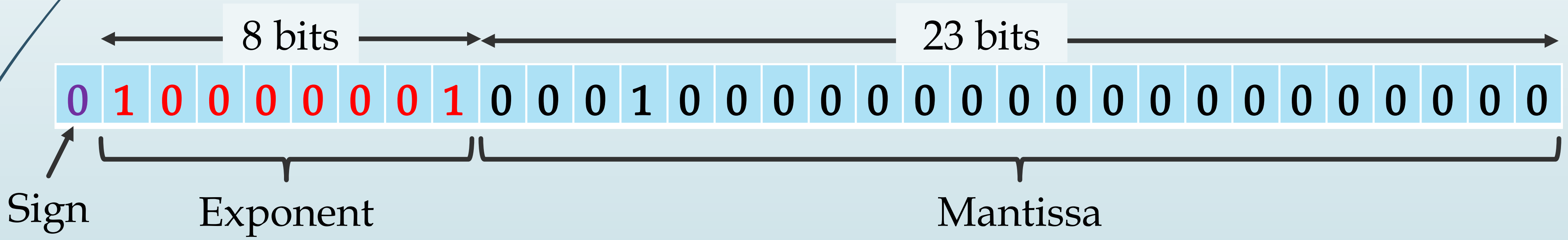
$$(100.01)_2 \longrightarrow (1.\underbrace{0001}_{\text{Mantissa}})_2 \times 2^2 \longrightarrow \text{No. of shifting = Exponent}$$

Real Representation

3. Add 127 to exponent with convert to binary.

$$\text{Exponent} + 127 = 2+127 = (129)_{10} \longrightarrow (10000001)_2$$

4. Store the results.



This results will be saved in memory in Hexadecimal as:

$(40\ 88\ 00\ 00)_{16}$

Real Representation

Example (6): Find the IEEE representation of $(-24.75)_{10}$ using 32 bit.

1. The binary equivalent of $(24.75)_{10}$ is $(11000.11)_2$.

2. Normalize the $(11000.11)_2$ to $[(1.100011)_2 \times 2^4]$.

3. Add 127 to 4; $127+4 = (131)_{10} \longrightarrow (10000011)_2$

4. Store the results.

Sign | Exponent | Mantissa

1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Saved in memory in Hexadecimal as $(C1\ C6\ 00\ 00)_{16}$

Website link: IEEE 754 Converter
https://www.h-schmidt.net/FloatConverter/IEEE754.html

Characters Representation

A *character* abstract data type is represented as an integer value that corresponds to a character set. A *character set* assigns an integer value to each character, punctuation, and symbol used in a language.

For example, the letter ***A*** is stored in memory as the value **65**, which corresponds to the letter ***A*** in a character set. The computer knows to treat the value 65 as the letter *A* rather than the number 65 because memory was reserved using the char abstract data type.

Characters Representation

There are two character sets used in programming, the American Standard Code for Information Interchange (ASCII) and Unicode.

ASCII is the oldest character sets and uses one byte to represent a maximum of 256 characters. However, a serious problem was evident after years of using ASCII. Many languages such as Russian, Arabic, Japanese, and Chinese have more than 256 characters in their language.

A new character set called Unicode was developed to resolve this problem. Unicode uses two bytes to represent each character.

Ministry of Higher Education & Scientific Research
Al-Furat Al-Awsat Technical University
Karbala Technical Institute
Department of Computer System Techniques.

Data Structures

Second Stage

Arrays

Assist. Prof. Dr. Wathiq Laftah Al-Yaseen

2022-2023

In this lesson we will learn …

Compound Data Structures

- Arrays.
- One-Dimension Array.
- Two-Dimension Array.
- Represent of Array by Rows method.
- Represent of Array by Columns method.

Arrays

An ***array*** is a compound data structures that can hold several values, all of one data type.

An ***array*** is a fixed size sequential collection of elements of identical types.

The declaration statement in C++ is:

```
Data_Type name_of_array[array size];
```

The elements in a one dimensional array are indexed by the integers (**0** to ***n*-1**), where ***n*** is the size of array.

Arrays

The amount of memory that be used by one dimensional array depends on the array's *data type* and the *number of elements (size of array)*.

Total array's size in memory = data type size × number of elements

For example,

```
short age[6];
```

The *age* array, defined here, is a one dimensional array that holds six short integer values.

A short integer in C++ uses **2 Bytes** of memory, so the *age* array would reserve **12 Bytes**.

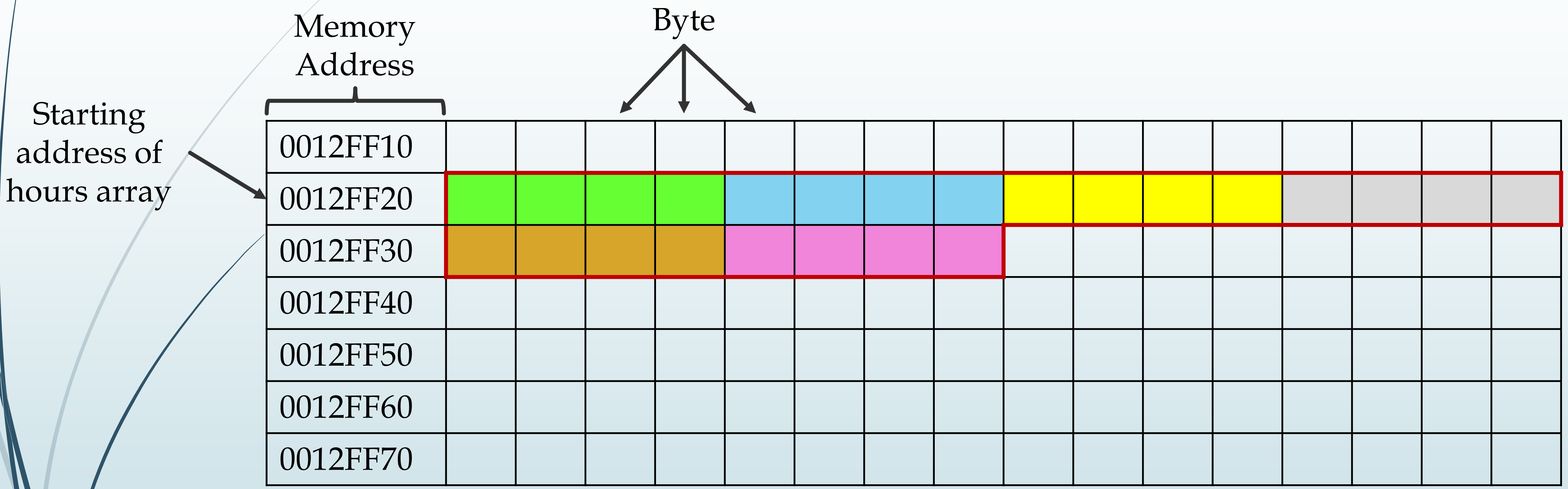
Represent one dimensional array in memory

The main memory of a computer can be thought as a huge one dimensional array of memory locations.

Each basic memory location consumes **1 Byte**, e.g. the **1KB** of main memory can be thought as an array with **1,024** elements, each element = 1 Byte.

In C/C++, a one dimensional array is a continuous block of memory. In our example, **int hours[6];** it will be represented in memory as:

Represent one dimensional array in memory



🟩 hours[0]	⬜ hours[3]	
🟦 hours[1]	🟧 hours[4]	
🟨 hours[2]	🟪 hours[5]	

The starting address of hours array in Hexadecimal is: 0012FF20

What is the address memory of the second element?

Represent one dimensional array in memory

In general, if we given starting address (*A*) of one dimensional array of type (*T*), we can find the ***k***-th element at address:

$$\boxed{\textbf{Element address}(k) = A + k \times \text{Size of } (T)}$$

Example (1): Assume ***B*** is one dimensional array declared as integer array, and the starting address is (**800**). Find the address of the element ***B*(3)**.

$A = 800;$ $k = 3;$ $\text{Size of } (T) = \text{Size of (int)} = 4 \text{ Byte};$

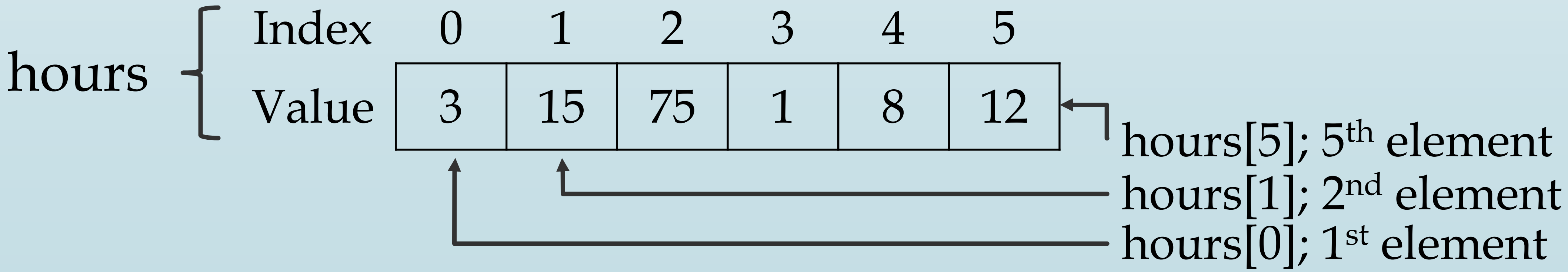
$$\text{Element address}(3) = A + k \times \text{Size of (T)}$$
$$= 800 + 3 \times 4 = 80C$$

Represent one dimensional array in C++

Example (2):

```
        int hours[6];
or      const int SIZE = 6;
        int hours[SIZE];
```

-The type of array is ***integer***.
-The name of this array is ***hours***.
-The number inside the brackets is the ***array's size***.

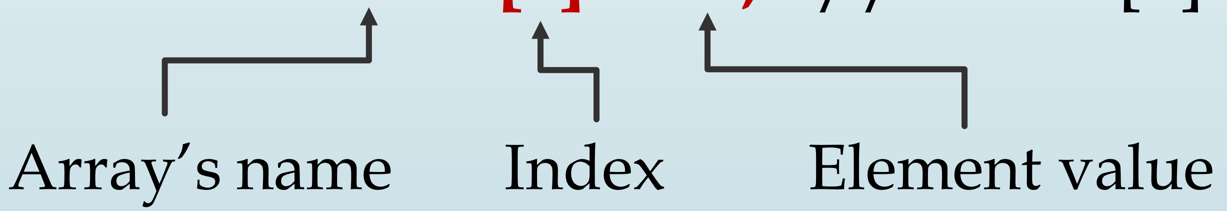


Represent one dimensional array in C++

The individual elements of one dimensional array are assigned unique indices. These indices are used to access the elements.

The following statement stores the integer value 50 in **hours[3]**. Note that, this is the fourth element of array that has index 3.

```
hours[3] = 50;   // hours[3] now hold 50.
```



Represent one dimensional array in C++

Why need to use array type?

Consider the following issue:
"We have a list of 1000 students' marks of an integer type"

Can we imagine how long we have to write the declaration part by using default variable declaration?

We will declare something like the following:

```
int  studMark0, studMark1, studMark2, ..., studMark999;
```

By using an array, we just declare like this:

```
int  studMark[1000];
```

Initialize one dimensional array in C++

Arrays can be initialized at the time of declaration when their initial values are known in advance.

For example;

```
int hours[6]={10,5,2,6,14,1};
```

If the number of values to be initialized is less than the size of the array, then the elements will be initialized to zero automatically.

For example;

```
int hours[6]={10,5,2};
```

Consider the declaration along with the initialization.

For example; `char b[]={'C','O','M','P','U'};`

One Dimensional Array in C++

```
// This program stores employee work hours in an int array.
// It uses one for loop to input the hours and another for loop to display them.

#include <iostream.h>
int main() {
     const int NUM_EMPLOYEES = 6;
     int hours[NUM_EMPLOYEES]; // Holds hours worked for 6 employees
     cout << "Enter the hours worked by " << NUM_EMPLOYEES << " employees:";
     for (int count = 0; count < NUM_EMPLOYEES; count++)
          cin >> hours[count];

     cout << "The hours we entered are:";
     for (int count = 0; count < NUM_EMPLOYEES; count++)
          cout << " " << hours[count];
     cout << endl;
     return 0;
}
```

Processing one dimensional array contents

Individual element of one dimensional array is processed like any other type of variable.

For example, the following statement multiplies **hours[3]** by the variable **rate**:

```
pay = hours[3] * rate;
```

Moreover, the following are examples of pre-increment and post-increment operations on array elements:

```
int score[5] = {7, 8, 9, 10, 11};
++score[2]; // Pre-increment operation on the value in score[2]
score[4]++; // Post-increment operation on the value in score[4]
```

Copy one array to another

Cannot simply assign one array to another array. To copy the contents of one array to another, we must assign each element of the first array, one at a time, to the corresponding element of the second array. The following code segment uses a for loop to do this.

```
const int SIZE = 6;
int arrayA[SIZE] = {10, 20, 30, 40, 50, 60};
int arrayB[SIZE] = { 2, 4, 6, 8, 10, 12};
for (int index = 0; index < SIZE; index++)
    arrayA[index] = arrayB[index];
```

Comparing two arrays

As we cannot copy one array to another with a single statement, we also cannot compare the contents of two arrays with a single statement.

That is, we cannot use the **==** operator with the names of two arrays to determine whether the arrays are equal.

To compare the contents of two arrays, we must compare their individual elements. For example, look at the following code:

Comparing Two Arrays

```
const int SIZE = 5;
int arrayA[SIZE] = {5, 10, 15, 20, 25};
int arrayB[SIZE] = {5, 10, 15, 20, 25};
bool arraysEqual = true;
int count = 0;
while ((count < SIZE) && (arrayA[count] == arrayB[count])){
    count++;
}
if (count >= SIZE)
    cout << "The arrays are equal.\n";
else
    cout << "The arrays are not equal.\n";
```

Two Dimensional Array

To define a two dimensional array, two size declarators are required: The first one is for the number of rows (1st dimension) and the second one is for the number of columns (2st dimension).

The general form is:

```
Data_Type name_of_array[1st dim][2nd dim];
```

For processing data in a two dimensional array, each element has two indexes: one for its row and another for its column.

Represent two dimensional array in memory

The amount of memory that be used by a two dimensional array is based on the array's data type and the number of elements (no of rows × no of columns).

Size of array in memory = data type size × no of element

Example: The stdMarks array, is a 2-dim array that holds four students (4 rows) and three marks for each student (3 columns), which mean it is holds (4 × 3) 12 integer values:

```
int stdMarks[4][3];
```

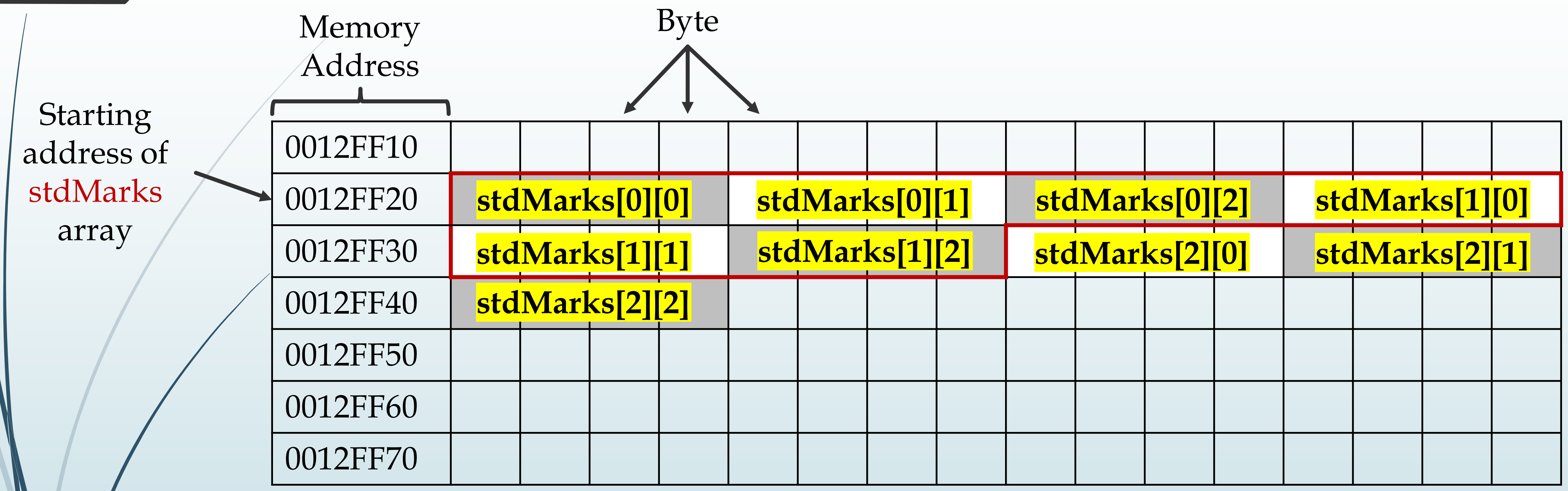
On a typical PC, a integer uses 4 Bytes of memory, so the stdMarks array would occupy (12 × 4) 48 Bytes.

Represent two dimensional array in memory

The main memory of a computer can be thought as a huge one dimensional array. Each basic memory location consumes 1 byte. E.g. the 1KB of main memory can be thought as an array with 1,024 elements, each element = 1 byte.

In C/C++, a two dimensional array is a continuous block of memory. In our example, int stdMarks[4][3] it will be represented in memory as:

Represent two dimensional array in memory



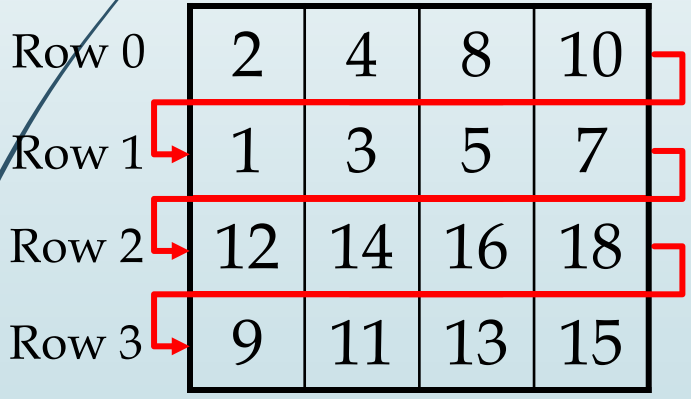
Example Array Size Declarators

Array declaration	Num. of elements	Size of each element	Size of the array
char A[3][4];	(3×4)=12	2 Byte	(12×2)=24 Bytes
short int B[10][5];	50	2 Byte	100 Bytes
int marks[4][10];	40	4 Byte	160 Bytes
float temp[5][5];	25	4 Byte	100 Bytes

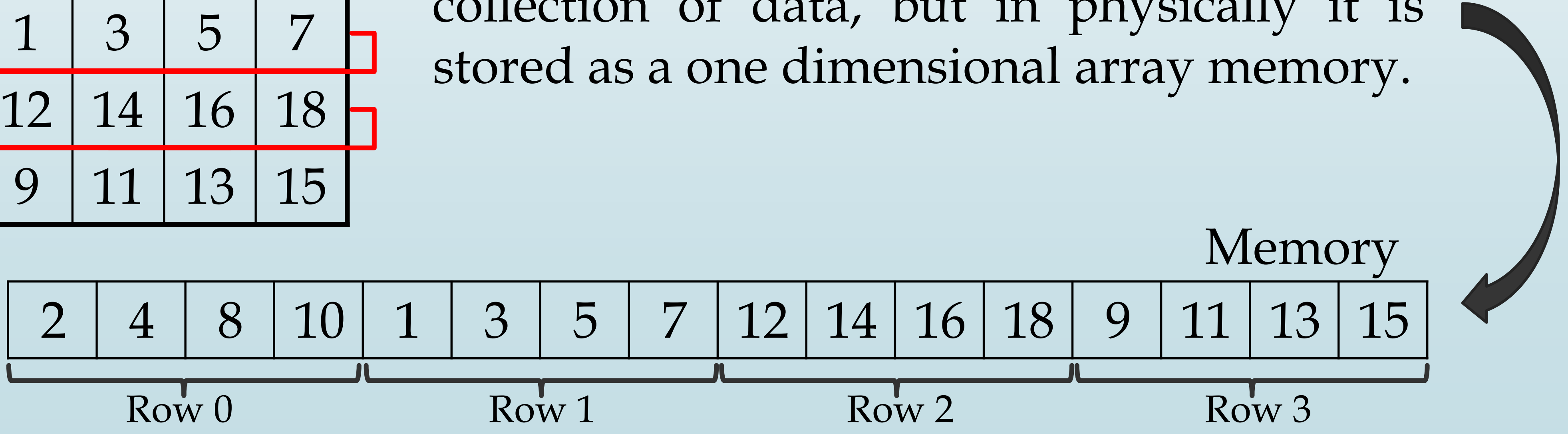
Represent two dimensional array in memory

The store of two dimensional array in memory can be represented in either of two methods:

1. Row-major order: Elements travel across rows, then across columns, Ex: int hours[4][4];



Logically it is viewed as a two-dimensional collection of data, but in physically it is stored as a one dimensional array memory.



Represent two dimensional array in memory

In general, if we have a starting address (A) of two dimensional array (R×C) of data type (T), we can find the V[i][j] element at address:

Address(V[i][j]) = A + i * (no of elements column (C)) * size(T) + j * size(T)

Example: Assume **C** is array declared as **char C[2][3];** and the starting address is $(100)_{16}$. Find the address of the element C(1,2)?

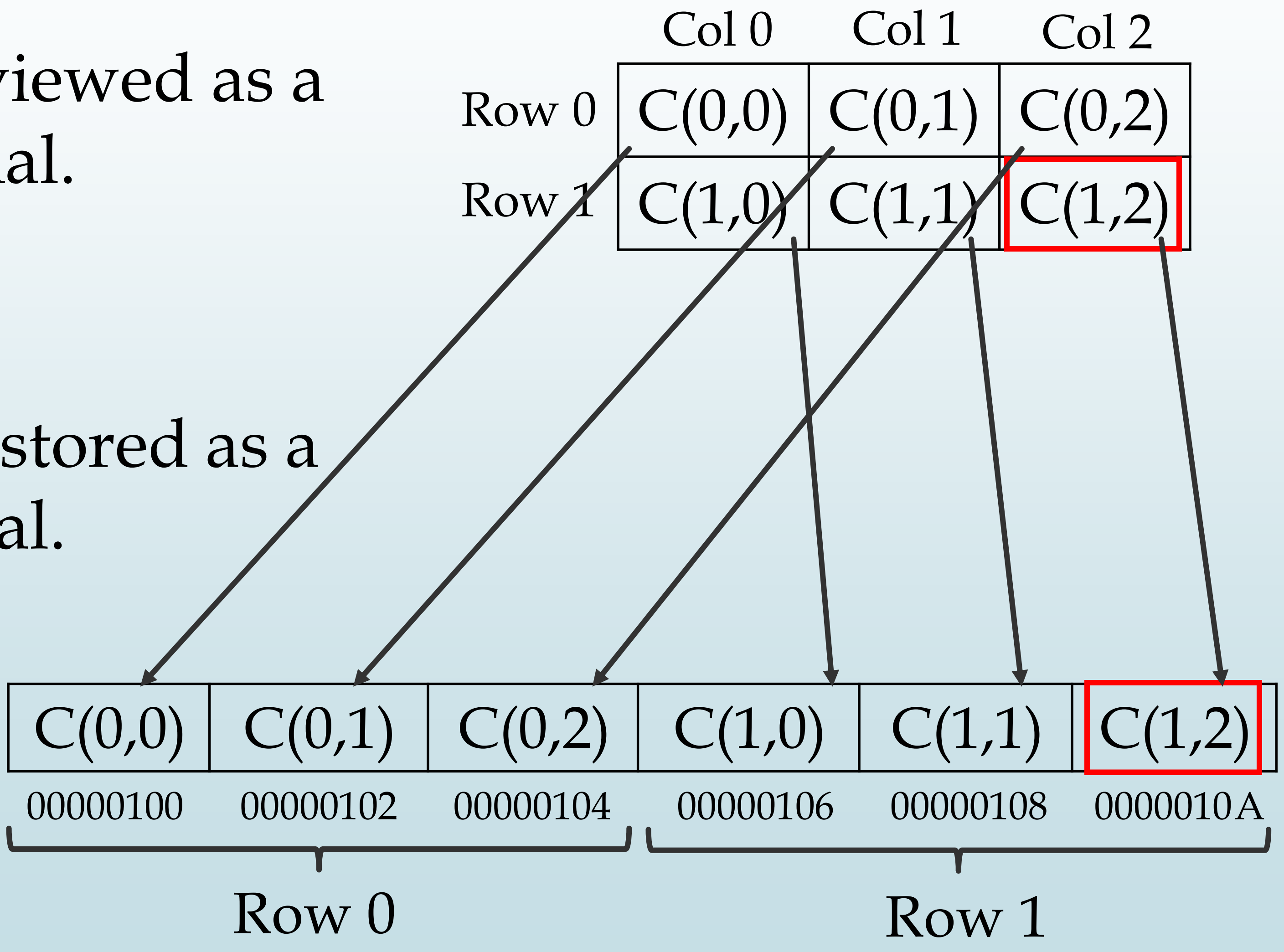
- Starting address (A) = $(100)_{16}$
- Row's index of element(i)= 1
- Column's index of element(j)= 2
- Number of elements in a column (C)= 3
- Size(T) = size (char) = 2 Byte

Address(C[1][2]) = 100 + 1 * 3 * 2 + 2 * 2 = $(10A)_{16}$

Represent two dimensional array in memory

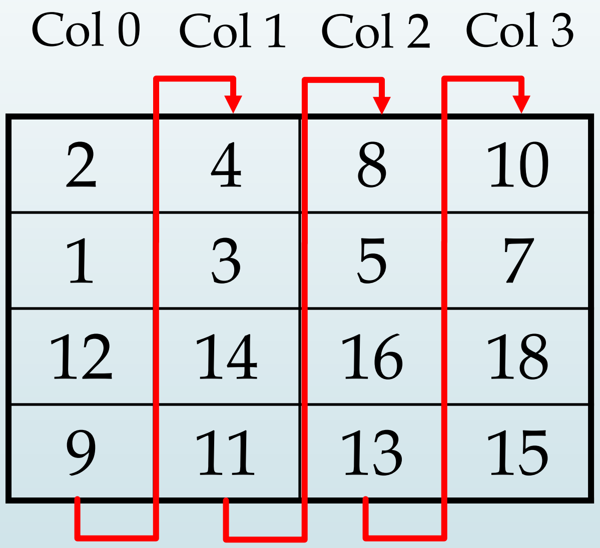
Logically it is viewed as a two dimensional.

Physically it is stored as a one dimensional.



Represent two dimensional array in memory

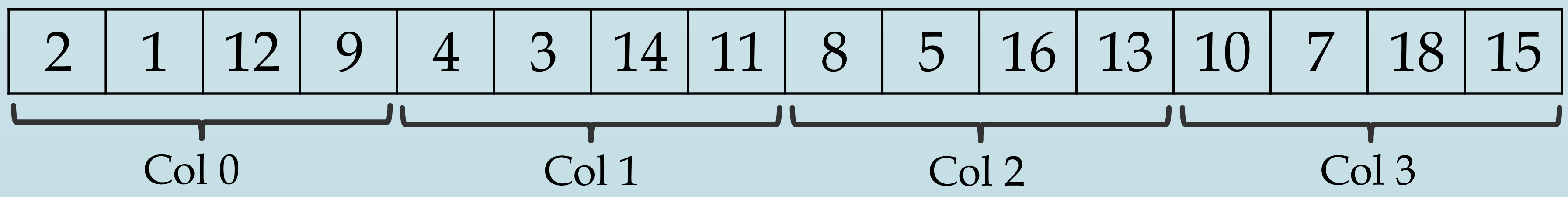
2. Column-major order: Elements travel across columns, then across rows, Ex: int hours[4][4];



Col 0	Col 1	Col 2	Col 3
2	4	8	10
1	3	5	7
12	14	16	18
9	11	13	15

Logically it is viewed as a two-dimensional collection of data, but in physically it is stored as a one dimensional array memory.

Memory



2	1	12	9	4	3	14	11	8	5	16	13	10	7	18	15
Col 0				Col 1				Col 2				Col 3			

Represent two dimensional array in memory

In general, if we have a starting address (A) of two dimensional array (R×C) of data type (T), we can find the V[i][j] element at address:

Address(V[i][j]) = A + j * (no of elements row (R)) * size(T) + i * size(T)

Example: Assume **C** is array declared as **char C[2][3];** and the starting address is $(100)_{16.}$ Find the address of the element C(0,2)?

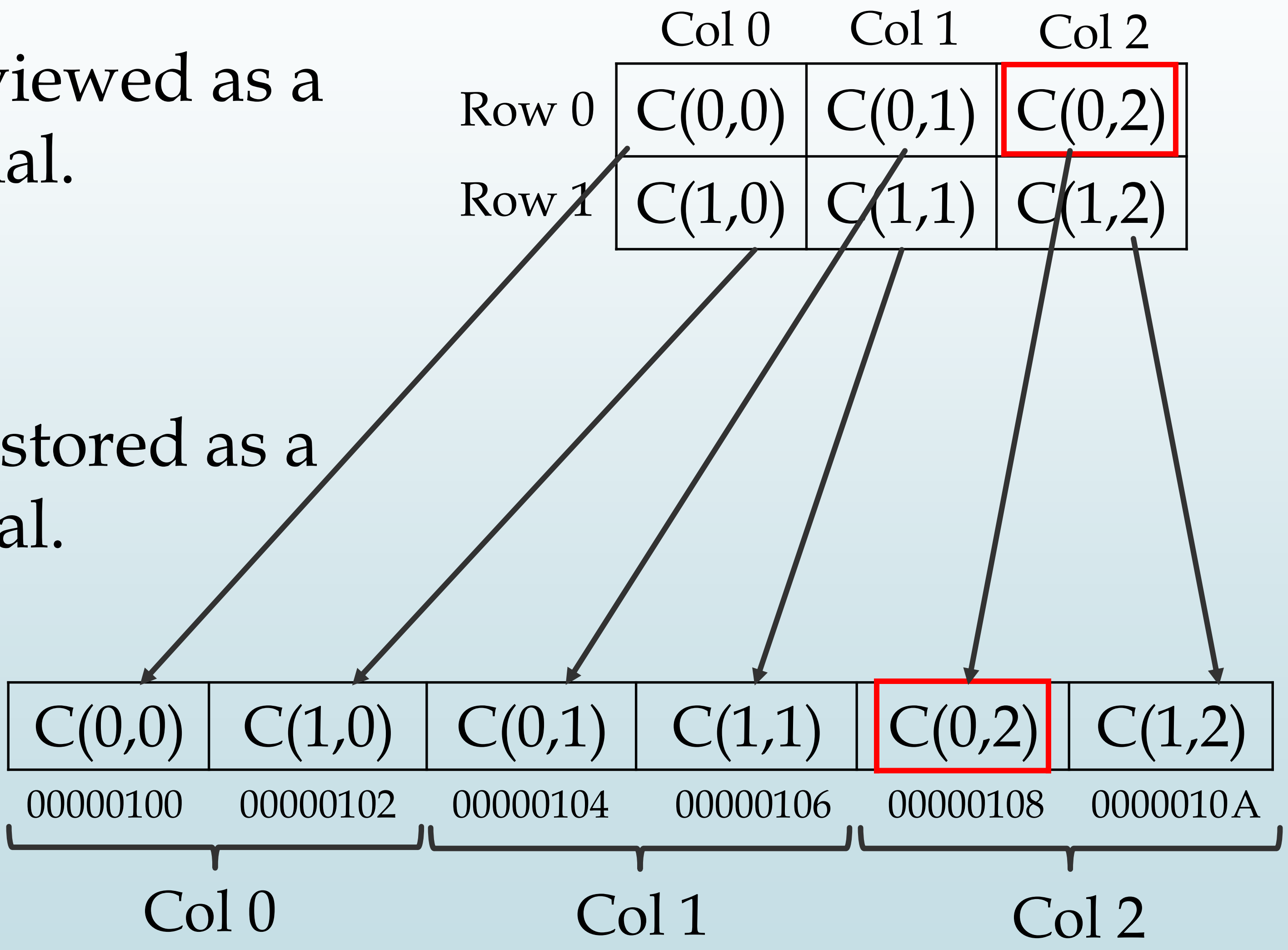
- Starting address (A) = $(100)_{16}$
- Row's index of element(i)= 0
- Column's index of element(j)= 2
- Number of elements in a row (R)= 2
- Size(T) = size (char) = 2 Byte

Address(C[0][2]) = 100 + 2 * 2 * 2 + 0 * 2 = $(108)_{16}$

Represent two dimensional array in memory

Logically it is viewed as a two dimensional.

Physically it is stored as a one dimensional.



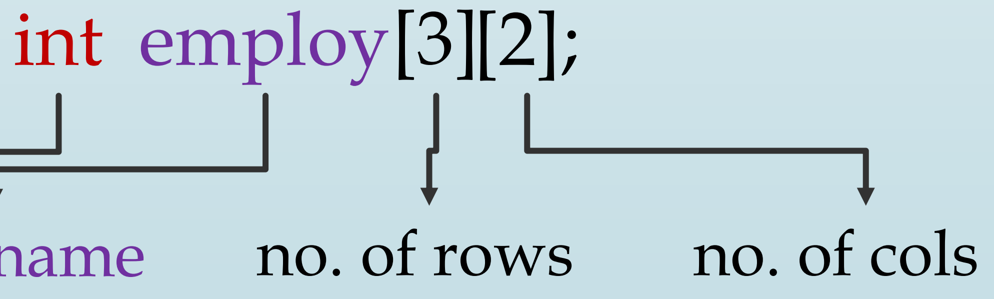
Represent two dimensional array in C++

To create a two dimension array, we use a declaration statement in page 17 of this slides.

A two dimension array declaration should indicate four things:

1. The data type of value that be stored in each element.
2. The name of the array.
3. The number of elements in row (1st dim size) of array.
4. The number of elements in column (2st dim size) of array.

Example:

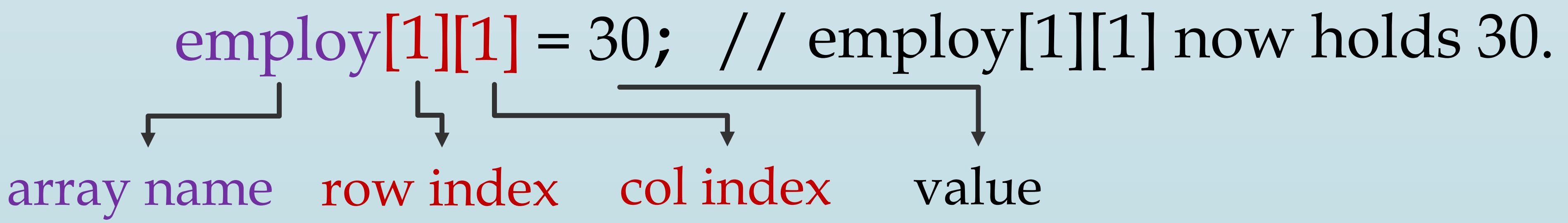


```
int employ[3][2];
```

Represent two dimensional array in C++

- The individual elements of a two dimensional array are assigned unique pair indexes one for row and another for column. These indexes are used to access the elements.
- Even though an entire array has only one name, the elements may be accessed and used as individual variables.

The following statement stores the integer 30 in employ[1][1]. Note that this is the fourth array element.



```
employ[1][1] = 30;   // employ[1][1] now holds 30.
```

Initialize two dimensional array in C++

- **Initializing all locations of array**

Array can be initialized at the time of declaration when their initial values are known in advance.

```
int employ[3][2]={{60,59},{68,48},{88,78}};
```

- **Partial array initialization**

If the number of values to be initialized is less than the size of the array, then the elements will be initialized to zero automatically.

```
int employ[3][2]={{60,59},{68}};
```

Exercise: Try initialize array:

```
int employ[3][2]={60,59,68,48,88,78};
```

What did you find out?

Read and write two dimensional array in C++

Write a C++ program to input *m* of marks to *n* students.

```
#include <iostream>

int main(){
    const int ROW_SIZE = 10, COLUMN_SIZE = 10;
    int stdMarks[ROW_SIZE][COLUMN_SIZE];
    int num_students, num_marks;
    cout << "Enter the number of students : ";
    cin >> num_students;
    cout << "Enter the number of marks : ";
    cin >> num_marks;
```

Read and write two dimensional array in C++

```
    for (int i = 0; i < num_students; i++){
        cout << “Input the marks to the student (“ << i+1 << “)\n”;
        for (int j = 0; j < num_marks; j++){
            cout << “Input the mark “ << j+1 <<“ :  “;
            cin >> stdMarks[i][j];
        }
    }
    for (int i = 0; i < num_students; i++){
        cout << “These marks for the student number (“ << i << “)”;
        for (int j = 0; j < num_marks; j++)
            cout << “  “ << stdMarks[i][j];
    }
    return 0;
}
```

Questions

- If we have a square two dimensional array, write a C++ program to output the main and secondary diagonal.
- Exchange the main diagonal of square array with secondary diagonal.
- Write a C++ program to multiple two dimensional array.
- Write a C++ program to compare between two dimensional array.
- Write a C++ program to find the min and max value in two dimensional array.
- Write a program in C++ to output the unique values of array.
- Write a program in C++ to sort one dimensional array.

Ministry of Higher Education & Scientific Research
Al-Furat Al-Awsat Technical University
Karbala Technical Institute
Department of Computer System Techniques.

Data Structures

Second Stage

Records

Assist. Prof. Dr. Wathiq Laftah Al-Yaseen

2022-2023

In this lesson we will learn …

- Introducing record.
- Represent Record in C++.
- Initialization of Record in C++.
- Inputting and Displaying Record in C++.
- Processing Record contents in C++.
- Copying Record to Another.
- Array of Records in C++.
- Array within Record in C++.
- Nested Record in C++.

Introducing record

A record is an integrated set of information about a single item (such as person, place, etc.) and the fields defined the individual components of the record.

In some situations, we need to group elements of different types in order to represent one item. For example, a set of fields might be a person's (name, age, address, and phone number). Together, all those fields respect to one person make up a record.

	name	age	address	phone number
Record	**Ahmed Ali**	**20**	**Karbala**	**07801234567**
	string	int	string	long

$$\underbrace{\text{string} \quad \text{int} \quad \text{string} \quad \text{long}}_{\text{Different types}}$$

Represent record in C++

- record called a struct (structure).
- fields called members.
- The declaration of structure must start with the keyword struct followed by the structure's name, then structure's member variables are declared within braces.
- General form for declaring structure in C++:

```
struct structName {
    dataType1 memberName1;
    dataType2 memberName2;
        ⋮
};
```

Represent record in C++

Example:

```
struct studentRec{
        char firstName[10], lastName[10];
        int Age;
        float Average;
};
```

After declared studentRec as a structure we can use it as any type to define variables:

```
        studentRec student;
```

Represent record in C++

The individual member of the record must be accessed by the name of the record followed by the dot operator (.) and then the name of the member.

```
cin >> student. firstName;
cin >> student. lastName;
student.Age = 18;
```

Initialization of record in C++

- Like normal variables, structures can be initialized at the time of declaration.
- Initialization of structure is almost similar to initializing array as:

```
struct Employee {
    int Id;
    char Name[25];
    int Age;
    long Salary;
};
Employee emp = {2, “Haider”, 35, 35000};
```

Inputting and displaying record in C++

- Data in a structure variable must be read one member at a time.
- Contents of a structure must be written one member at a time.

Example: This program reads personal information and print it?

```
#include <iostream>

#include <conio>
int main(){
     struct personRec{
          char lastName[10];
          char firstName[10];
          int age;
     };
```

Inputting and displaying record in C++

```
    personRec person;
    cout << “Enter first name: “; cin >> person.firstName;
    cout << “Enter last name: “;  cin >> person.lastName;
    cout << “Enter age: “;          cin >> person.age;
    cout << “\n\n Hello “ << person.firstName << “ “;
    cout << person.lastName << “. How are you?\n”;
    cout << "\n Congratulations on reaching the age of “ << person.age;
}
```

Processing record contents in C++

- Individual a record elements (members) are processed like any other type of variables. For example, when you have the following structure:

```
struct studentRec{
    float av;
    int mark1, mark2, mark3;
};
studentRec student1, student2;
student1.mark1 = 50;
student1.mark2 = 60;
student1.mark3 = 70;
```

- The following statement calculate the average of student1 and store it in av member.

```
student1.av = (student1.mark1+student1.mark2+student1.mark3)/3;
```

Copying record to another

- To copy the contents of one record to another, you can use assignment statement, the following example use the records student1 and student2 as them were declared in previous slide:

```
student2 = student1;
cout << student2.mark1;
cout << student2.mark2;
cout << student2.mark3;
cout << student2.av;
```

- When you run this code segment all printed values it will be the same values that relate to the Student1.

Array of records in C++

At first, we need to declare a structure such as:

```
struct studentRec{
        char firstName[10], lastName[10];
        int Age;
        float Average;
};
```

Then specify an array of that type:

```
studentRec studentArray[10];
```

Access elements of the array that has structure elements:

```
for (int i = 0; i < 10; i++)
    cin >> studentArray[i].firstName;
```

Array of records in C++

Example: Write a program in C++ to read information for group of students, each student has name, average, and class, therefore, this program will calculate the number of students which are success with print their information?

```
#include <iostream>
#include <conio>
int main() {
        struct studentRec {
              char stdName[10];
              float avg;
              int class;
        };
```

Array of records in C++

```
const int Size = 20;
studentRec arrStudent[Size];
int numSuccess = 0, n;
cout << "Enter Number Students: ";
cin >> n;
for (int i = 0; i < n; i++){
    cout << "\n Enter student name: ";
    cin >> arrStudent[i].stdName;
    cout << "Enter student average: ";
    cin >> arrStudent[i].avg;
    cout << "Enter student  class: ";
    cin >> arrStudent[i].class;
}
cout << "\n The information of success students";
```

Array of records in C++

```
    for (int i = 0; i < n; i++)
        if (arrStudent[i].avg > 49){
            numSuccess = numSuccess + 1;
            cout << "Student name: "<< arrStudent[i].stdName << endl;
            cout << "Student average: " << arrStudent[i].avg << endl;
            cout << "Student class: " << arrStudent[i].class << endl;
        }
    cout << "\n The number success students is : " << numSuccess;
}
```

Array within record in C++

- As we know, record is collection of different data type. Like normal data type, it can also define an array as well.
- The general form for array within structure is:

```
 struct structName {
        dataType1 memberName1;     // normal variable
        dataType2 arrayName[size];  // array variable
           ⋮
};
```

Array within record in C++

Example: Write a program in C++ to read one student information has (Number, Name, 3 marks), then find and print his total marks with average?

```
#include <iostream>
void main() {
    struct Student {
        int Num;
        char Name[25];
        int Marks[3];       // array of marks
        int Total;
        float Avg;
    };
```

Array within record in C++

```
    Student S;
    cout << "Enter Student Number: ";
    cin >> S.Num;
    cout << "Enter Student Name: ";
    cin >> S.Name;
    S.Total = 0;
    for (int i = 0; i < 3; i++){
        cout << Enter Marks " << i+1 << ": ";
        cin >> S.Marks[i];
        S.Total = S.Total + S.Marks[i];
    }
    S.Avg = S.Total /3;
    cout << "Total: " << S.Total << endl;
    cout << "Average: " << S.Avg;
}
```

Nested record in C++

- When a structure contains another structure, it is called nested structure.
- Syntax for structure within structure or nested structure is

```
 struct structName1 {
        dataType1 memberName1;
        dataType2 memberName2;
        ⋮
 }
struct structName2 {
        dataType1 memberName1;
        structName1 memberName2;
        ⋮
};
```

Nested record in C++

For example, we have two structures named Address and Employee. To make Address nested to Employee, we have to define Address structure before and outside Employee structure and create an object of Address structure inside Employee structure.

```
 struct Address {
     char HouseNo[25], City[25], PinCode[25];
};
struct Employee{
     int Id; char Name[25]; float Salary;
    Address Add;
};
```

Nested record in C++

Here example shows how we can access the member in nested structure, by using the structures that previously declared (Address and Employee).

```
cin >> Employee.Id;
cin >> Employee.Name;
cin >> Employee.Salary;
cin >> Employee.Add.HouseNo;
cin >> Employee.Add.City;
cin >> Employee.Add.PinCode;
```

Ministry of Higher Education & Scientific Research
Al-Furat Al-Awsat Technical University
Karbala Technical Institute
Department of Computer System Techniques.

Data Structures

Second Stage

Functions & Subroutines

Assist. Prof. Dr. Wathiq Laftah Al-Yaseen

2022-2023

In this lesson we will learn …

- Functions.
- Subroutines.
- Arguments passed by value and by reference.

Functions

Functions allow to structure programs in segments of code to perform individual tasks.

In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

```
type name ( parameter1, parameter2, ...) {
        statements
}
```

Where:

- type is the type of the value returned by the function.
- name is the identifier by which the function can be called.

Functions

- parameters (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma. Each parameter looks very much like a regular variable declaration (for example: int x), and in fact acts within the function as a regular variable which is local to the function. The purpose of parameters is to allow passing arguments to the function from the location where it is called from.
- statements is the function's body. It is a block of statements surrounded by braces { } that specify what the function actually does.

Functions

Let's have a look at an example:

```
#include <iostream>
using namespace std;
int addition (int a, int b) {
     int r;
     r=a+b;
     return r;
}

int main () {
      int z;
      z = addition (5,3);
      cout << "The result is " << z;
}
```

Functions

A function can actually be called multiple times within a program, and its argument is naturally not limited just to literals:

```
#include <iostream>
using namespace std;
int subtraction (int a, int b) {
    int r;
    r=a-b;
    return r;
}
int main () {
    int x=5, y=3, z;
    z = subtraction (7,2); cout << "The first result is " << z << '\n’;
    cout << "The second result is " << subtraction (7,2) << '\n’;
    cout << "The third result is " << subtraction (x,y) << '\n’;
    z= 4 + subtraction (x,y);
    cout << "The fourth result is " << z << '\n’;
}
```

Here in this example:
Call subtraction
function 4 times.

Subroutines

The syntax shown above for functions:

```
type name ( argument1, argument2 ...) {
      statements
}
```

Requires the declaration to begin with a type. This is the type of the value returned by the function. But what if the function does not need to return a value? In this case, the type to be used is *void* and is called subroutine. So the general syntax of subroutine is:

```
void  name (argument1, argument2, …){
       statements;
}
```

Subroutines

For example, a subroutine that simply prints a message may not need to return any value:

```
#include <iostream>
using namespace std;

void printmessage (string txt) {
      cout << txt;
}

int main () {
      printmessage (“Hello World”);
}
```

Cannot see return statement !

Arguments passed by value and by reference

In the subroutine seen earlier, arguments have always been passed *by value*. This means that, when calling a subroutine, what is passed to the subroutine are the values of these arguments on the moment of the call, which are copied into the variables represented by the subroutine parameters. For example, take:

```
int x=5, y=3;
addition ( x, y );
```

In this case, subroutine addition is passed 5 and 3, which are copies of the values of x and y, respectively. These values (5 and 3) are used to initialize the variables set as parameters in the subroutine's definition, but any modification of these variables within the subroutine has no effect on the values of the variables x and y outside it, because x and y were themselves not passed to the subroutine on the call, but only copies of their values at that moment.

Arguments passed by value and by reference

In certain cases, though, it may be useful to access an external variable from within a subroutine. To do that, arguments can be passed *by reference*, instead of *by value*. For example, the subroutine duplicate in this code duplicates the value of its three arguments, causing the variables used as arguments to actually be modified by the call.

To gain access to its arguments, the subroutine declares its parameters as *references*. In C++, references are indicated with an ampersand (&) following the parameter type, as in the parameters taken by example of duplicate.

Arguments passed by value and by reference

```
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c) {
        a*=2;
        b*=2;
        c*=2;
}

int main () {
      int x=1, y=3, z=7;
      duplicate (x, y, z);
      cout << "x=" << x << ", y=" << y << ", z=" << z;
      return 0;
}
```

Arguments passed by value and by reference

When a variable is passed *by reference*, what is passed is no longer a copy, but the variable itself, the variable identified by the subroutine parameter, becomes somehow associated with the argument passed to the subroutine, and any modification on their corresponding local variables within the subroutine are reflected in the variables passed as arguments in the call.

In fact, a, b, and c become aliases of the arguments passed on the subroutine call (x, y, and z) and any change on a within the subroutine is actually modifying variable x outside the function. Any change on b modifies y, and any change on c modifies z. That is why when, in the example, subroutine duplicate modifies the values of variables a, b, and c, the values of x, y, and z are affected.

Ministry of Higher Education & Scientific Research
Al-Furat Al-Awsat Technical University
Karbala Technical Institute
Department of Computer System Techniques.

Data Structures

Second Stage

Stack

Assist. Prof. Dr. Wathiq Laftah Al-Yaseen

2022-2023

In this lesson we will learn …

- What is the stack?
- Operations on the stack.
- Insertion algorithm (Push).
- Deletion algorithm (Pop).

Stack-Linear Data Structure

A stack is a method that be grouped the items together by placing one item on top of another item and then removing items one at a time from the top of the stack.

When we hear the term “stack” used outside the context of computer programming, we might envision a stack of dishes in our kitchen. This organization is structured in a particular way: the newest dish is on top and the oldest is on the bottom of the stack.

Each dish in a stack is accessed using LIFO: last in-first out. The only way to access each dish is from the top of the stack. If we want the third dish (the third oldest on the stack), then we must remove the first two dishes from the top of the stack. This places the third dish at the top of the stack making it available to be removed.

Stack-Linear Data Structure

Stack Definition is a linear linked list that permits the insertion or deletion to occur at one end only. Such a linear list is referred to as last in-first out (LIFO) list.

The most and least accessible element in a stack are known as the top and the bottom of the stack. The insertion operation is referred to as Push, the deletion as Pop.

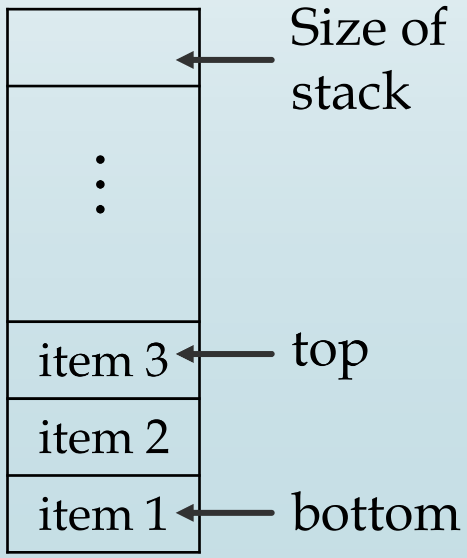
Operations on Stack

1. Push

Programmers use the term "push" to mean placing an item on a stack. Push is the direction that data is being added to the stack … Here's what actually happens. The new value is assigned to the next available array element and the index of that array element becomes the top of the stack. The program increments the current index of the stack by 1.

Stack is represented in C++ as follows:

```
stackType stackName[size of stack];
int top;
```



Operations on Stack

For examples:

```
typeStack stack[6];
int top = 0;
```

The C++ subroutine to push item in stack is as follow:

```
void push(typeStack item){
    if ((top+1) < size){
        top++;
        stack[top] = item;
    }
    else
        cout << "The stack is overflow !"
}
```

Operations on Stack

2. Pop

Popping is the reverse process of pushing: it removes an item from the stack. It is important to understand that popping an item off the stack doesn't copy the item. Once an item is popped from the stack, the item is no longer available on the stack, although the value remains in the array.

When you pop new value from the stack, we decrement the index at the top of the stack. That is, we make its index 2 instead of 3. This makes pop the new value at the top of the stack..

Operations on Stack

The C++ subroutine to pop item from stack is as follow:

```
typeStack pop(){
    if (top > 0){
        typeStack item = stack[top];
        top = top – 1;
        return item;
    }
    else{
        cout << “The stack is empty !”
        return -1;
    }
}
```

Ministry of Higher Education & Scientific Research
Al-Furat Al-Awsat Technical University
Karbala Technical Institute
Department of Computer System Techniques.

Data Structures

Second Stage

Queue

Assist. Prof. Dr. Wathiq Laftah Al-Yaseen

2022-2023

In this lesson we will learn …

- What is the queue?
- Operations on the queue.
- Insertion algorithm (Insert).
- Deletion algorithm (Remove).

The Queue

Queue is work on the principal of First-In-First-Out (FIFO), it means first entered item remove first. Queue have two ends front and rear, from front you can insert element and from rear you can delete element.

The concept of queue can be understood by our real life problems. For example a customer come and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service center.

In a multitasking operating system, the CPU cannot run all jobs at once, so jobs must be batched up and then scheduled according to some policy. Again, a queue might be a suitable option in this case.

The Queue

Definition: It is a homogeneous collection of elements in which new elements are added at one end called *rear*, and the existing elements are deleted from other end called *front*.

The basic operations that can be performed on queue are:

1. Insert (or add) an element to the queue.
2. Delete (or remove) an element from a queue.

Insert operation will add an element to queue, at the rear end, by incrementing the array index. Remove operation will delete from the front end by incrementing the array index and will assign the deleted value to a variable.

Total number of elements present in the queue is front-rear+1, when implemented using arrays.

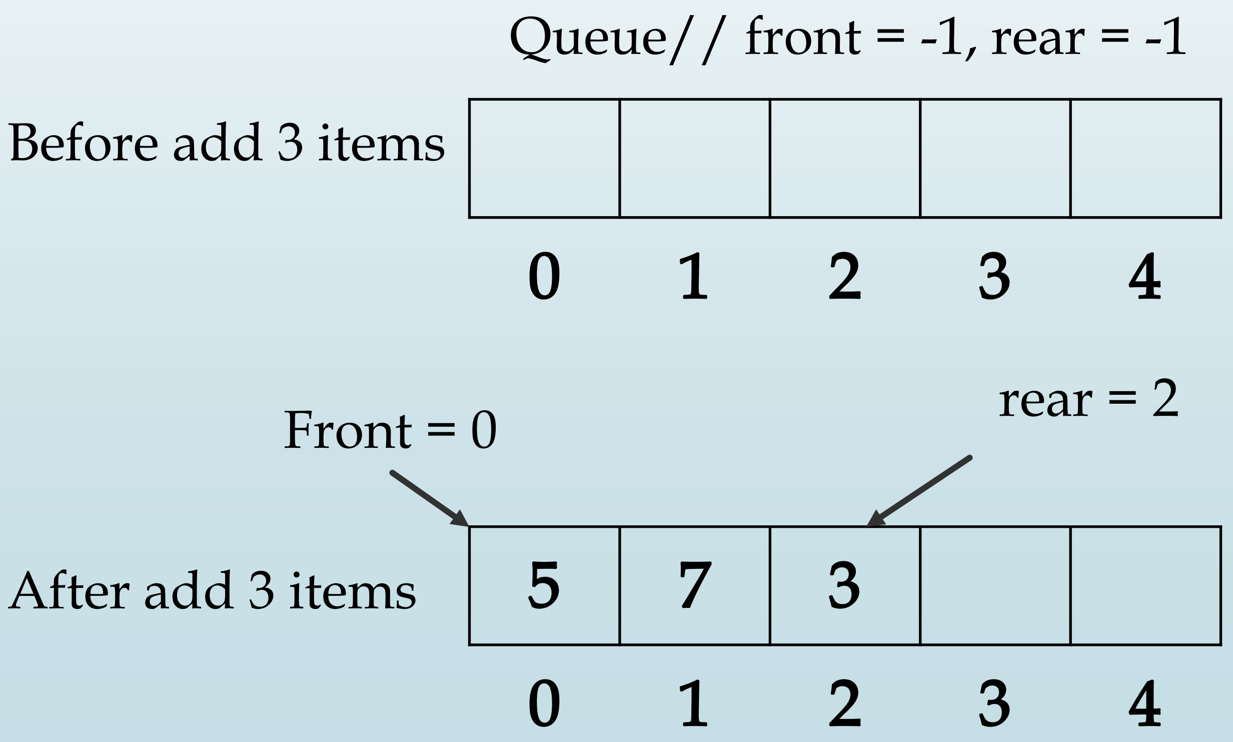
Operations on Queue

Queue is represented in C++ as follow:

```
queueType queueName[size of queue];
int front = -1, rear = -1;  // when queue is empty
```

For examples:

```
const int size = 5;
float queue[size];
int front = -1, rear = -1;
```



1. Insertion algorithm (Insert)

The C++ subroutine to insert an item in queue is as follow:

```
void insert(queueType item){
    if ((rear+1) >= size)
        cout << "The queue is overflow !";
    else{
        rear = rear + 1;
        queue[rear] = item;
        if (front == -1)
            front = 0;
    }
}
```

2. Deletion algorithm (Remove)

The C++ subroutine to remove an item from stack is as follow:

```
queueType remove(){
    if (front == -1){
        cout << “The queue is empty !”;
        return -1;
    }else{
        queueType item = queue[front];
       if (front == rear){
            front = -l;
            rear = -1;
        }else
            front = front + 1;
        return item;
    }
}
```

Ministry of Higher Education & Scientific Research
Al-Furat Al-Awsat Technical University
Karbala Technical Institute
Department of Computer System Techniques.

Data Structures

Second Stage

Pointers

Assist. Prof. Dr. Wathiq Laftah Al-Yaseen

2022-2023

In this lesson we will learn …

- Address Operator.
- Represent Address Operator in C++.
- Pointer Variables and their Representing in C++.
- Initialization of Pointer in C++.
- Comparing Pointers in C++.
- Pointers and Numbers.
- Allocating Memory with new operator.
- Freeing Memory with delete operator.

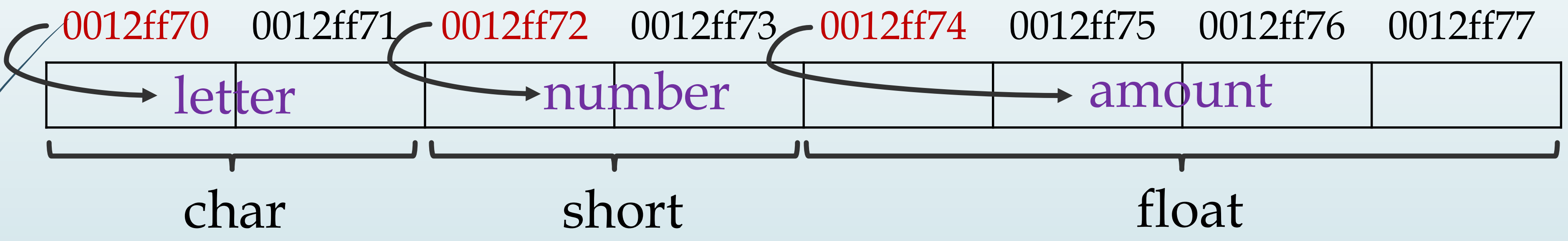
Address operator

- Every variable allocates a location of memory through its define, the memory address of this variable can be retrieved by using the address operator (&).
- The address of a memory location is called a pointer.
- Every variable in an executing program is allocated a section of memory large enough to hold a value of that variable's type.
- A variable's address is the address of the first byte allocated to that variable. Suppose that the following variables are defined in a program.

Address operator

```
char letter;
short number;
float amount;
```

```
cout << &number;
```



Then

The address of letter is (0012ff70),
and the address of number is (0012ff72),
and address of amount is (0012ff74).

Representing address operator in C++

- C++ has an address operator (&) that can be used to retrieve the address of any variable.
- To use it, place it before the variable whose address we want.
- Here is a statement that displays the variable's address to the screen:

```
cout << &amount;
```

- By default, C++ prints addresses in hexadecimal. But we can used a function-style cast to long to make the address print in the usual decimal format:

```
cout << long(&amount);
```

Representing address operator in C++

```
//This program uses the & operator to determine a variable's address.
#include <iostream>
char letter;
short number;
float amount;
double profit;
int main(){
     cout << "Address of letter is: " << long(&letter) << endl;
     cout << "Address of number is: " << long(&number) << endl;
     cout << "Address of amount is: " << long(&amount) << endl;
     cout << "Address of profit is: " << long(&profit) << endl;
     return 0;
}
```

The output of program

```
Address of letter is: 4299190272
Address of number is: 4299190274
Address of amount is: 4299190276
Address of profit is: 4299190280
```

Pointer variables and their representing in C++

- A pointer variable is a variable that holds addresses of memory locations.
- A pointer is a variable whose value is the address of another variable.
- Like other data values, memory addresses, or pointer values, can be stored in variables of the appropriate type.
- The general form for declaring pointer in C++ is:

```
Datatype *Pointer_Name;
```

The following are the valid pointer declaration:

```
int *p;          // pointer to an integer
double *dp;      // pointer to a double
float *fp;       // pointer to a float
char *ch;        // pointer to character
```

Pointer variables and their representing in C++

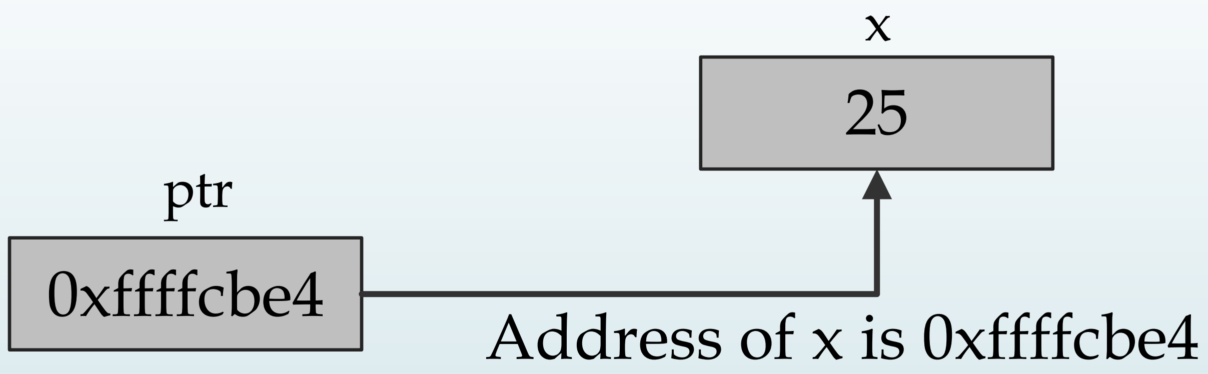
```
//This program stores the address of a variable in a pointer.
#include <iostream>
int main(){
    int x = 25;    // Integer variable
    int *ptr;      // Pointer variable, can point to an int
    ptr = &x;      // Store the address of x in ptr
    cout << "The value in x is " << x << endl;
    cout << "The address of x is " << ptr << endl;
    return 0;
}
```

The output of program

```
The value in x is 25
The address of x is 0xffffcbe4
```

Pointer variables and their representing in C++

The following figure illustrates the relationship between (ptr) and (x) in previous program.



We can use a pointer to indirectly access and modify the variable being pointed to. Look to the next program.

Pointer variables and their representing in C++

```
//This program demonstrates the use of the indirection operator.
#include <iostream>
 int main(){
        int x = 25; int *ptr;
        ptr = &x;
        //Use both x and ptr to display the value in x.
        cout << "Here is the value in x, printed twice:\n";
        cout << x << "  " << *ptr << endl;
        //Assign 100 to the location pointed to by ptr.
        //This will actually assign 100 to x.
       *ptr = 100;
        cout << "Once again, here is the value in x:\n";
        cout << x << "  " << *ptr << endl;
        return 0;
}
```

The output of program

```
Here is the value in x, printed twice:
25  25
Once again, here is the value in x:
100  100
```

Initialization of pointer in C++

- Pointers can be initialized with the address of an existing object.
- When a pointer is initialized with an address of object, it must be the pointer and the object of the same data type. For example:

```
int x;
int *p = &x;            // Type is matching
int y;
float *f = &y;          // Type is mismatching
```

Comparing pointers in C++

- C++'s relational operators (==, !=, <, >, >=, <=) can be used to compare pointer values.
- Comparing two pointers is not the same as comparing the values the two pointers point to. For example:

1. The following if statement compares the addresses stored in the pointer variables ptr1 and ptr2.

```
if (ptr1 < ptr2)
```

2. The following if statement, compares the values that ptr1 and ptr2 point to.

```
if (*ptr1 < *ptr2)
```

3. The following if statement is wrong because we cannot compare address with normal value.

```
if (ptr > *ptr)
```

Allocate memory with *new* Operator

We've initialized pointers to the addresses of variables; the variables are named memory allocated during *compile time*, and each pointer only provides an alias for memory we could access directly by name anyway.

The true worth of pointers comes into play when we allocate unnamed memory *during runtime* to hold values. In this case, pointers become the only access to that memory.

In C++, we can allocate memory by using the ***new*** operator.

The general form for declaring pointer with ***new*** operator:

```
Datatype *Pointer_Name = new Datatype;
```

Allocate memory with *new* Operator

Let's try out this new technique by creating ***unnamed*** runtime storage for a type ***int*** value and accessing the value with a pointer. Here's an example:

```
int *pn = new int;
```

```
int main(){
    int *pt = new int;   // allocate space for an int
    *pt = 1001;          // store a value there
    cout << "int value = " << *pt << ": location = " << pt << endl;
    double *pd = new double;      // allocate space for a double
    *pd = 10000001.0;             // store a double there
```

Allocate memory with *new* Operator

```
    cout << "double value = " << *pd << ": location = " << pd << endl;
    cout << "size of pt = " << sizeof(pt);
    cout << ": size of *pt = " << sizeof(*pt) << endl;
    cout << "size of pd = " << sizeof (pd);
    cout << ": size of *pd = " << sizeof(*pd) << endl;
    return 0;
}
```

The output of program

```
int value = 1001: location = 0x004301a8
double value = 1e+07: location = 0x004301d8
size of pt = 4: size of *pt = 4
size of pd = 4: size of *pd = 8
```

Freeing Memory with *delete* operator

Using new to request memory when you need it is just the more glamorous half of the C++ memory-management package. The other half is the delete operator, which enables you to return memory to the memory pool when you are finished with it. That is an important step toward making the most effective use of memory. Memory that you return, or free, can then be reused by other parts of the program. You use delete by following it with a pointer to a block of memory originally allocated with new:

```
int *ps = new int;          // allocate memory with new
. . . // use the memory
delete ps;                  // free memory with delete when done
```

Ministry of Higher Education & Scientific Research
Al-Furat Al-Awsat Technical University
Karbala Technical Institute
Department of Computer System Techniques.

Data Structures

Second Stage

Linked List

Assist. Prof. Dr. Wathiq Laftah Al-Yaseen

2022-2023

In this lesson we will learn …

- Types of storage allocation.
- Comparison between sequential and dynamic storage allocation.
- Single linked list.
- Operations on single linked list.
- Double linked list.
- Operations on double linked list.
- Circular linked lists.
- Operations on circular linked lists.

Types of storage allocation

There are two types of storage allocation depending on the structure of the data:

- ***Sequential Allocation Storage***

Is the simples way to store lists in memory sequentially, and from the *Base address* which is the first location of the list, we can know the location of any item in the list.

Advantages

1. Simple in representation.
2. Take less memory space.
3. Efficient in random access.

Disadvantages

1. Hard to apply addition and deletion.
2. Number of elements must be predefined.

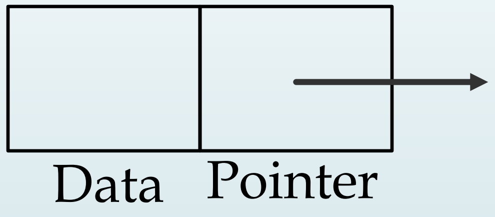
Types of storage allocation

- Dynamic Allocation Storage

The second way to store lists is to use link (or pointer), each element contain the location of the next element, so elements may not stored sequentially in memory.

Each element (node) consist of 2 parts:

1. Data
2. pointer (link) to the next address



Advantages

1. Insertion and deletion is easy to implement (not need shifting).
2. Easy to merge and split by only change the pointers.

Disadvantages

1. Take more memory space.
2. To access any element randomly, we must start from the beginning of the list.

Comparison between sequential and dynamic allocation

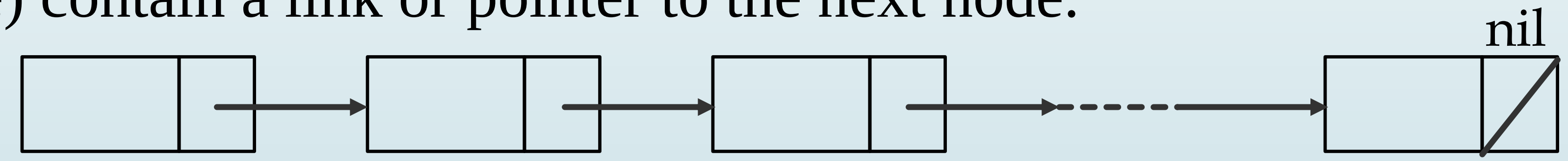
1. ***Amount of storage***
   The dynamic storage need more memory space because of the need to use pointer to next element.
2. ***Insertion and deletion operations***
   These operations simplest to execute in dynamical storage because they don’t need shifting.
3. ***Random access***
   The sequential way is easier in accessing randomly, but the dynamic way require to start searching from the beginning of the list.
4. ***Merge and sort***
   In the dynamic storage these operations are simple to execute by only change the pointer in merging location while the sequential storage need shifting and reorganization.

Linked lists

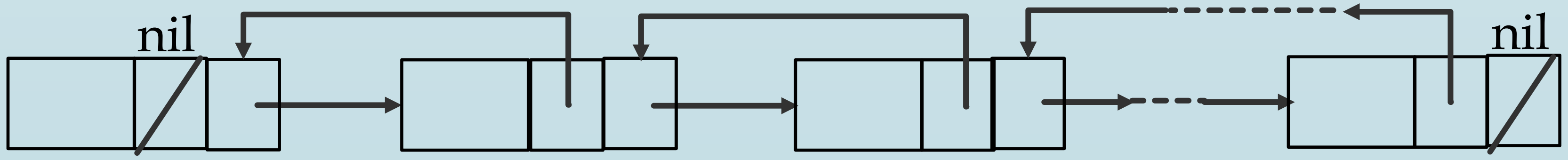
A list that use pointers or link to refer to the elements of data structures, in a way that element which have logically adjacent need not to be physically adjacent in memory.

Types of linked lists

1. **Single Linked List:** is a list contains set of elements, and each element (node) contain a link or pointer to the next node.

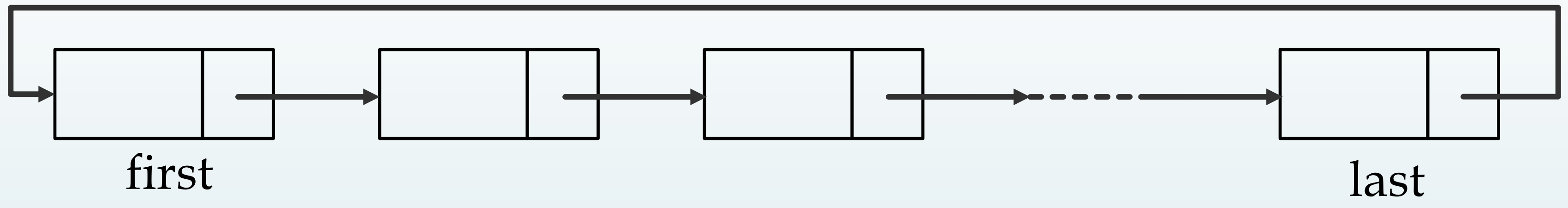


2. **Double Linked List:** a list has more than one pointer, which has two pointers pointing to the previous and next node.



Linked lists

3. **Circuler Linked List:** a list that last node points to the first node.

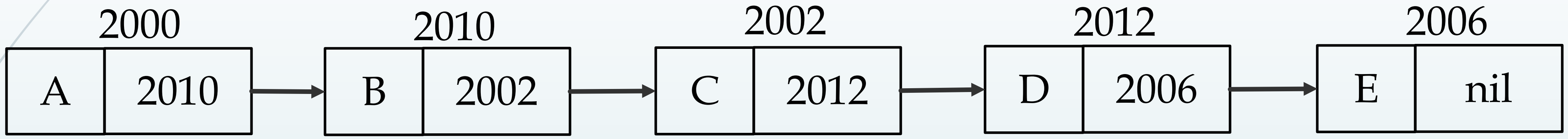


Example: Consider the following Linked List (Ordered):

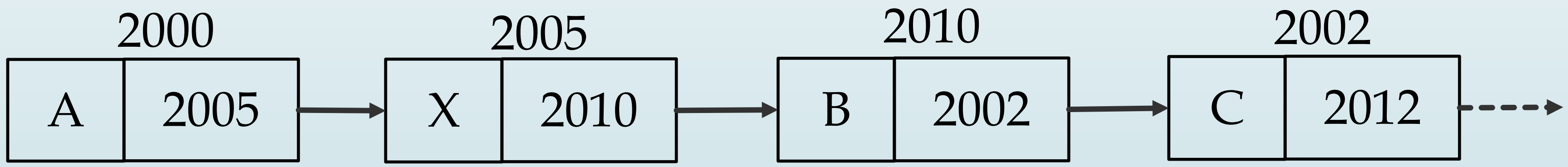
	Address	Data
1.	2000	A
2.	2010	B
3.	2002	C
4.	2012	D
5.	2006	E

Linked lists

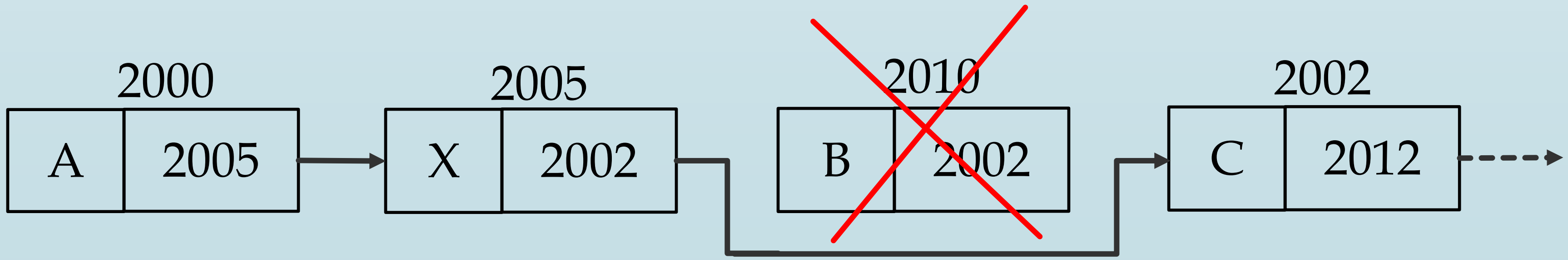
a. Draw the list.



b. Insert node X after A at location 2005.



b. Delete the node B.



Operation on single linked list

- Creating linked list of 2 nodes

```
struct node{
        int data;
        node *next;
}
node *p = new node;
p->data = 100;
p->next  = NULL;
node *start = p;
cout << "The first data: " <<start->data << endl;
p = new node;
p->data = 200;
p->next = NULL;
start->next = p;
cout << "The second data: " << start->next->data;
```

Output of this code:

```
The first data: 100
The second data: 200
```

Operation on single linked list

- Creating linked list of *N* nodes

```
struct node{
        int data;
        node *next;
}
node *p = new node;
cin >> p->data;
p->next  = NULL;
node *start = p;
node *q = p;
// Read N nodes of linked list
for (int i = 2; i <= N; i++){
    p = new node;
    cin >> p->data;
    p->next = NULL;
    q->next = p;
    q = p;
}
```

```
// Output linked list
p = start;
while (p != NULL){
    cout << p->data << endl;
    p = p->next;
}
```

Operation on single linked list

- Delete node depends on element

```
p = start;
While ((p != NULL) && (p->data != value)){
    q = p;
    p = p->next;
}
If (p == NULL)
    cout << "The value is not found"
Else{
    if (p == start){
        start = start->next;
        p->next = NULL;
        delete (p);
    }
```

Operation on single linked list

- Delete node depends on element

```
     else{
          q->next = p->next;
          p->next = NULL;
          delete(p);
          p = q->next;
     }
}
```

Operation on single linked list

- Insert node to begin of linked list

```
node *r = new node;
cout << "Enter new value :";
cin >> r->data;
r->next = start;
start = r;
```

- Insert node to end of linked list

```
node *r = new node;
cout << "Enter new value :";
cin >> r->data;
r->next = NULL;
p = start ;
while (p->next != NULL){
      p = p->next;
p->next = r;
```

Operation on single linked list

- Insert node to position *n* of linked list

```
node *r = new node;
cout << “Enter new value :”;
cin >> r->data;
r->next = NULL;
cout << “Enter position n: ”
cin >> n;
p = start;
int i = 1;
if (n == 1){
    r->next = start;
    start = r;
}
else
    while ((p != NULL) && (i < n)){
        q = p;
        p = p->next;
        i++;
```

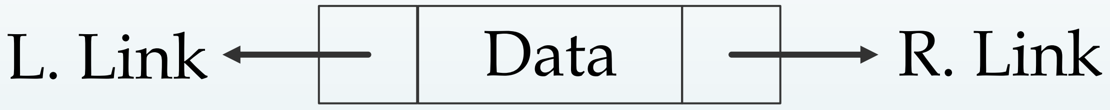
Operation on single linked list

- Insert node to position *n* of linked list

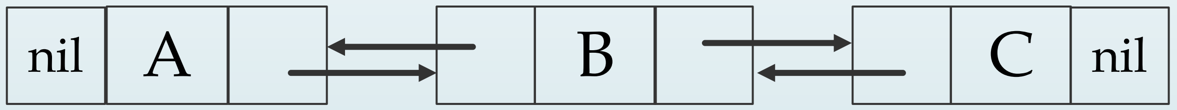
```
    }
    if (p == NULL)
         if (i == n)
              q->next = r;
         else
              cout << "The position is out of range !!";
    else{
       q->next = r;
       r->next = p;
    }
}
```

Double linked list

In these lists there are two pointers the first points to the previous element (left-link) and the second points to the next element (right-link).



For example



- Define double linked list

```
struct node{
            int data;
            node *previous;   //L. Link
            node *next;       //R. Link
}
```

Double linked list

- Create new double linked list with one node

```
node *p = new node;
cout << “Enter new value :”;
cin >> p->data;
p->next = NULL;
p->previous = NULL;
start = p;
```

- Insert element to the beginning of double linked list

```
node *p = new node;
cout << “Enter new value :”;
cin >> p->data;
p->next = start;
p->previous = NULL;
start->previous = p;
start = p;
```

Double linked list

- Insert element to the end of double linked list

```
node *p = new node;
cout << "Enter new value :";
cin >> p->data;
p->next = NULL;
node *q = start;
while (q->next != NULL)
      q = q->next;
p->previous = q;
q->next = p;
```

- Insert element before the *n* node of double linked list

```
node *p = new node;
cout << "Enter new value :";
cin >> p->data;
node *q = start;
```

Double linked list

```
int i = 1;
while (i < n){
      i++;
      q = q->next;
}
p->previous = q->previous;
p->next = q;
q->previous = p;
```

- Insert element after the *n* node of double linked list

…. // same the last search code about *n* node in the double linked list

```
p->next = q->next;
q->next = p;
p->previous = q;
```

Double linked list

- Delete the first node of double linked list

```
p = start;
start = start->next;
p->next = NULL;
start->previous = NULL;
delete (p);
```

- Delete the last node of double linked list

```
p = start;
while (p->next != NULL)
      p = p->next;
p->previous->next = NULL;
p->previous = NULL;
delete (p);
```

Double linked list

- Delete the *n* node of double linked list

```
p = start;
int i = 1;
while (i < n){
      i++;
      p = p->next;
}
q = p->previous;
r = p->next;
q->next = r;
r->previous = q;
p->previous = NULL;
p->next = NULL;
delete (p);
```

Double linked list

- Print double linked list from left to right

```
p = start;
while (p != NULL){
      cout << p->data << endl;
      p = p->next;
}
```

- Print double linked list from right to left

```
p = start;
while (p->next != NULL)
      p = p->next;
while (p != NULL){
      cout << p->data << endl;
      p = p->previous
}
```

Circular linked list

- Create the first node of circular linked list

```
node *p = new node;
cout << "Input the data: ";
cin >> p->data;
p->next = p;
start = p;
```

- Insert element to the beginning of circular linked list

```
node *p = new node;
cout << "Input the data: ";
cin >> p->data;
q = start;
while (q->next != start)
      q = q->next;
p->next = start;
q->next = p;
start  = p;
```

Circular linked list

- Insert element to the end of circular linked list

```
node *p = new node;
cout << "Input the data: ";
cin >> p->data;
q = start;
while (q->next != start)
      q = q->next;
p->next = start;
q->next = p;
```

✓ What are the differences between insert element to the beginning and end of circular linked list ?

✓ How can insert element to the *n* position of circular linked list ?

Circular linked list

- Delete the first node of circular linked list

```
q = start;
while (q->next != start)
        q = q->next;
p = start;
start = start->next;
q->next = start;
p->next = NULL;
delete (p);
```

- Delete the last node of circular linked list

```
q = start;
while (q->next->next != start)
        q = q->next;
p = q->next;
q->next = start;
p->next = NULL;
delete (p);
```