DATA Structures

Dr. Asmaa Gad El-Kareem

Lecture #2

Lecture 2: Introduction (cont.)

Lecture Contents

- Arrays
- Function
- Recursive Functions
- Elementary Data Structures

Arrays

One-Dimensional Arrays

- An array is a group of related data values that are all of the same type.

int X[5]

X[0]	345
X[1]	210
X[2]	4
X[3]	125
X[4]	90

float y[6]

y[0]	45.6
Y[1]	20.4
Y[2]	1.9
Y[3]	33.5
Y[4]	0.59
Y[5]	7.8

char z[7]

Z[0]	F
Z[1]	G
Z[2]	H
Z[3]	Q
Z[4]	R
Z[5]	I
Z[6]	K

Arrays

- There are a couple of reasons why a programmer might want to use an array in a program.
- One motivation for using an array is the reduction of identifiers.

Arrays

- For example, suppose you have a problem where you need to store the values of fifty test scores.
- Without arrays, you would have to declare fifty variables in order to keep track of all of their individual scores:

E.g. :

in case of 5 variables:

```
int test1, test2,test3, test4, test5;
```

in case of 50 variables, we need to define

```
int test1, test2, test3, ..., test50
```

Arrays

- With arrays, you could simply declare an array that would hold fifty elements, such as:

  int test[50];

- The individual test scores are not given different names in the array.
- Instead, they are named test as a group of related values.
- When you want to access a particular text score, you have to use a subscript (or technically speaking, an index value), such as
  - test[0] for the first test,
  - test[1] for the second test, and so on.

Arrays

- The first element in an array is also given an index value of 0 (e,g, test[0], and then the index is incremented for each new element, e.g. test[1], test[2],….

- Another reason for using an array is if you are faced with a problem where you need to use/access the data involved more than once.

Arrays

- The form of declaring an array is as follows:

```
data_type arrayName[MAX];
```

where **MAX** is the maximum number of elements. In the above example,

```
int test[50];
```

would be an appropriate array declaration.

Arrays

```
#include <iostream.h>
void main()                    // average of n numbers
{
int x[100], n; float sum=0;
cin>> n;
for(int i =0; i<n ; i++)
  {     cin >> x[i];
        sum += x[i];    }
cout << "average = " << sum/n;   }
```

Arrays

- You can also initialize arrays by using curly braces { } as follows:

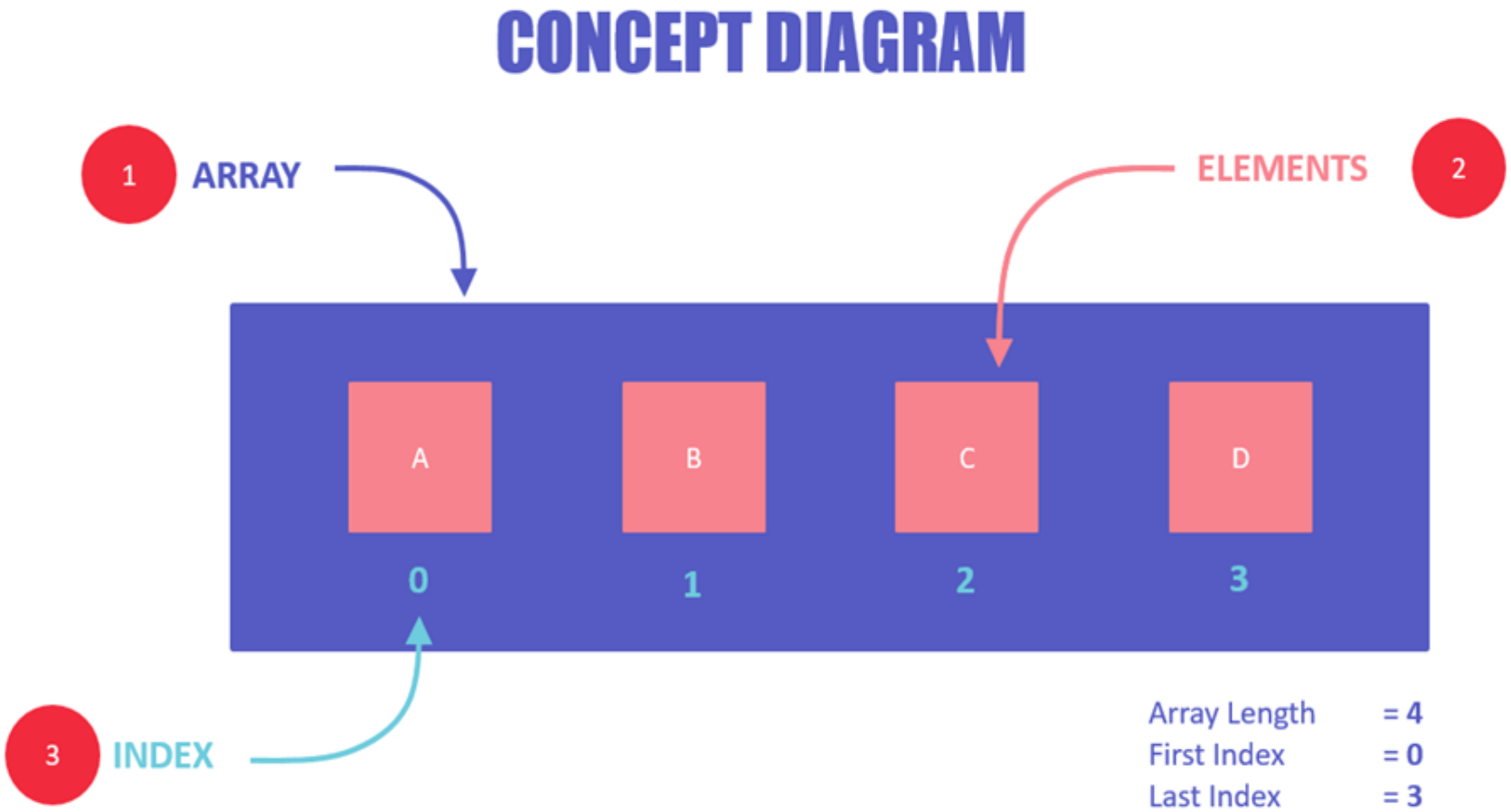
```
int test[5] = { 1, 2, 3, 4, 5 };
```

This initializes test[0] to 1, test[1] to 2, test[2] = 3, test[3] = 4, and test[4] = 5.

You can initialize all of the elements in an array to 0 as follows:

```
int test[50] = { 0 };
```

Summary of One-Dimensional



1. An array is a container of elements.
2. Elements have a specific value and data type, like "ABC", TRUE or FALSE, etc.
3. Each element also has its own index, which is used to access the element.

What are Arrays?

Array is a container which can hold a fixed number of items and these items should be of the same type.
Most of the data structures make use of arrays to implement their algorithms.

•Following are the important terms to understand the concept of Array.

Element − Each item stored in an array is called an element.

Index − Each location of an element in an array has a numerical index, which is used to identify the element.

UNDERSTAND SYNTAX

PYTHON

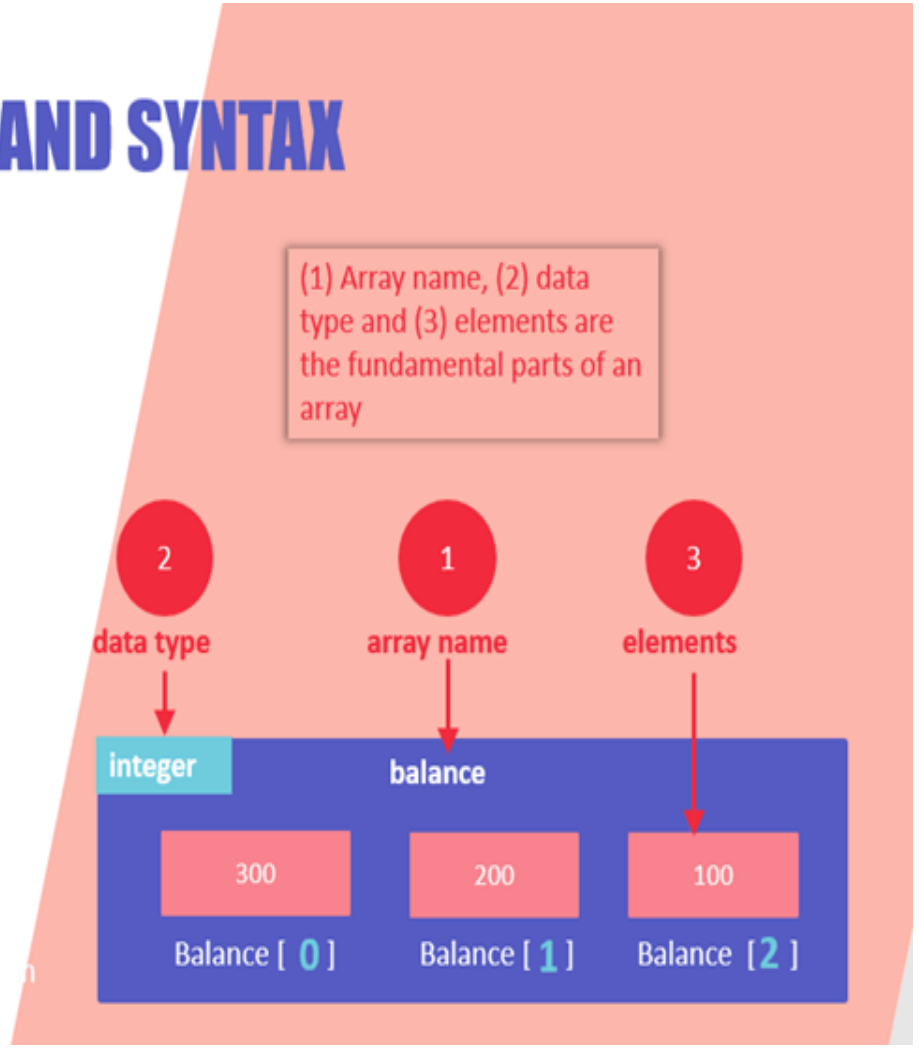
```
balance = array.array( 'i', [300,200,100] )
```

array name — data type — elements

C++

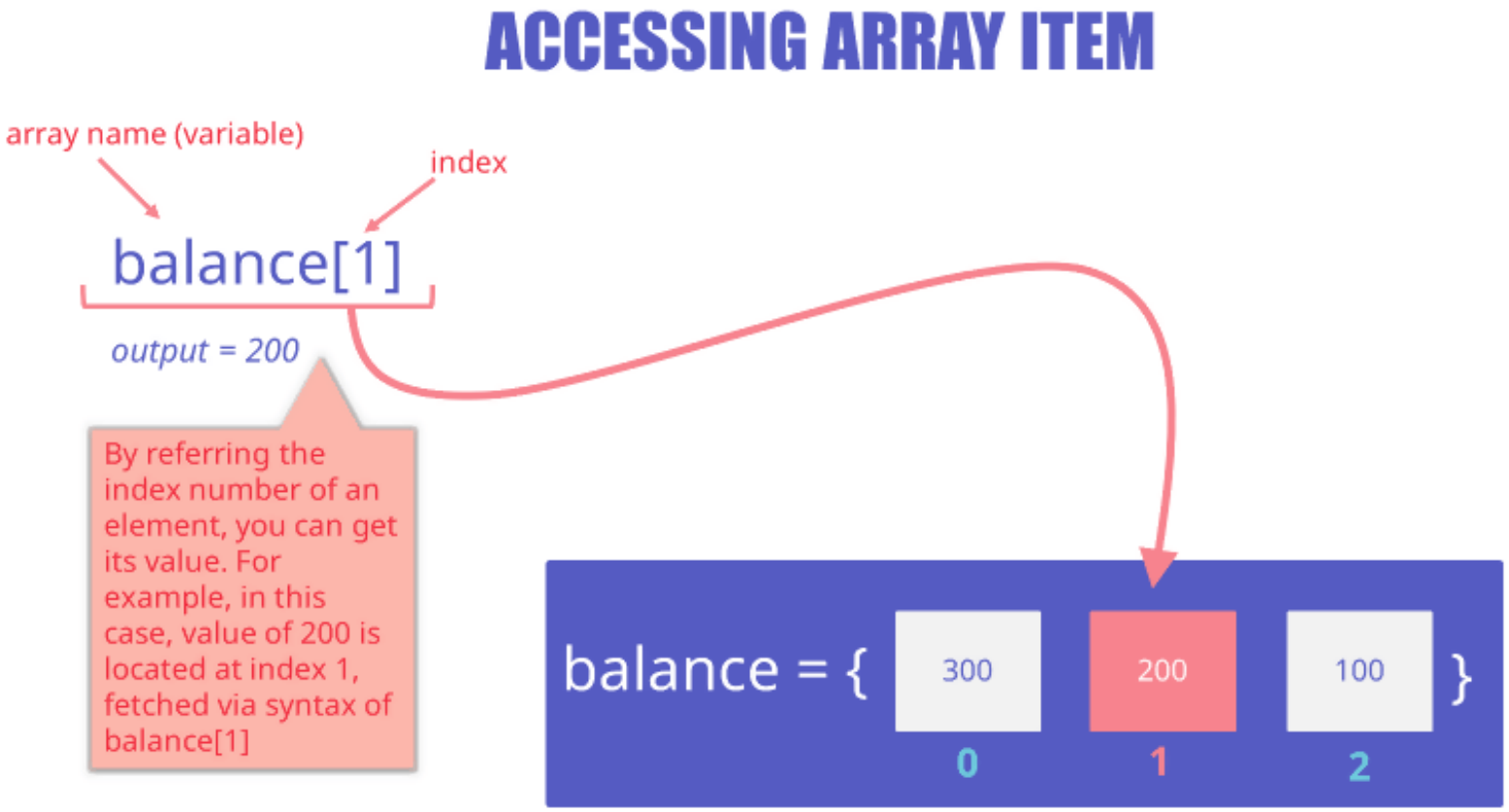
```
int balance[3] = { 300, 200, 100 }
```

data type — array name — elements



- **Array name:** necessary for easy reference to the collection of elements
- **Data Type:** necessary for type checking and data integrity
- **Elements:** these are the data values present in an array

- **Elements are stored at contiguous memory locations.**
- **An index is always less than the total number of array items.**
- **In terms of syntax, any variable that is declared as an array can store multiple values.**
- **Almost all languages have the same comprehension of arrays but have different ways of declaring and initializing them.**
- **However, three parts will always remain common in all the initializations, i.e., array name, elements, and the data type of elements.**



Here, we have accessed the second value of the array using its index, which is 1. The output of this will be 200, which is basically the second value of the balance array.

How to access a specific array value?

You can access any array item by using its index

Syntax

arrayName[indexNum]

Example

balance[1]

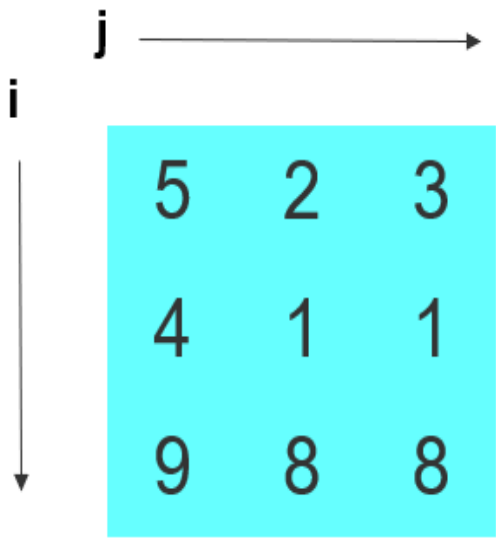
2-D Arrays

Useful in representing a variety of data,

E.g. Tables, Matrices, Graphs and Images

Day / City	Fri	Sat	Sun
Cairo	35	32	33
Tanta	34	31	31
Alex	29	28	28

A Table of Temp.



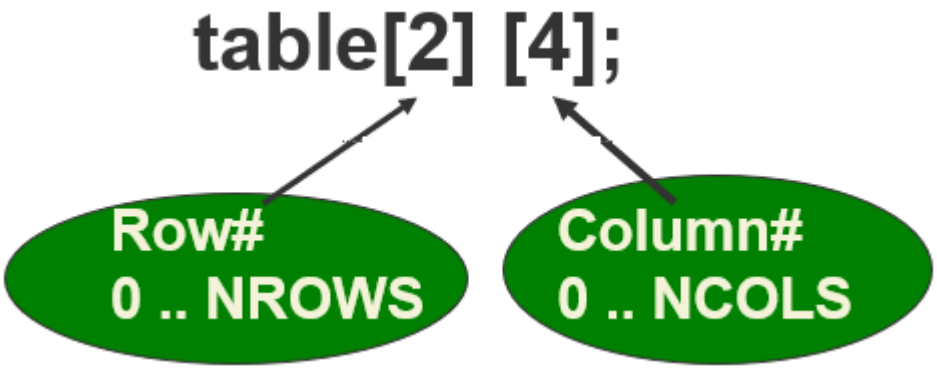
A Matrix of Integers

Declaration & Indexing

Declaration:

<element-type> <arrayName> [size 1][size2];

- **e.g. double table[NROWS] [NCOLS];**
- **Indexing:**



Functions

- A function is a collection of statements that performs a specific task.
- In this way, a program can be designed that makes sense when read.
- A program has a structure that is easier to understand quickly.
- The worst programs usually only have the required function, main, and fill it with pages of jumbled code.

Functions

- Also a function that is defined for use in one program may be of value in another program that needs to perform the same task.
- Functions that a programmer writes will generally require a prototype (declaration).
- The prototype tells the compiler what the function will return, what the function will be called (function name), as well as what arguments the function can be passed.

Functions

- **The general format for a prototype is simple:**

```
<return type>    FunctionName    ( arg_type
                 arg1, ..., arg_type argN );
```

- arg_type just means the type for each argument – for instance, an int, a float, or a char.
- It's exactly the same thing as what you would put if you were declaring a variable.

Functions

- There can be more than one argument passed to a function or none at all (where the parentheses are empty), and it does not have to return a value.
- Functions that do not return values have a return type of void.

Functions

- **Lets look at a function prototype:**

  int multi (int x, int y);

- **This prototype specifies that the function *multi* will accept two arguments, both integers, and that it will return an integer.**

Functions

- When the programmer actually defines the function, it will begin with the prototype.

```
int multi ( int x, int y );
int main(){
  int x,y;
  …
   multi(x,y)
  ;…
}
```

- Then there should always be a block with the code that the function is to execute.

Functions

- Any of the arguments passed to the function can be used as if they were declared in the block.
  - e.g. in the above example x, y have the same type (integer) in the block and the prototype.
- The function itself can be written after the *main* () function as in the following example.

Functions

```
<return type>    FunctionName   ( arg_type
        arg1, ..., arg_type argN )
                    {
                Statement;
                Statement;
                Statement;
                    }
```

Functions

Let's look at an example program:

```
#include <iostream>
int multi ( int x, int y );
int main()
{
  int x;   int y;
  cout<<"Please input two numbers to be multiplied: ";
  cin>> x >> y;
  cout<<"The product of your two numbers is "<< multi ( x, y )
    <<"\n";
}
int multi ( int x, int y )
{
 return x * y;
}
```

Functions

Write C++ Program that find factorial using function.

The factorial of a positive integer is the product of all positive integers less than or equal to the number

5 factorial is written 5!

5! = 5 * 4 * 3 * 2 * 1 = 120

3! = 3 * 2 * 1 = 6

0! is defined to be 1

Functions

We can easily calculate a factorial from the previous one:

n	n!		
1	**1**	1	1
2	2 × **1**	= 2 × **1!**	= 2
3	3 × **2 × 1**	= 3 × **2!**	= 6
4	4 × **3 × 2 × 1**	= 4 × **3!**	= 24
5	5 × **4 × 3 × 2 × 1**	= 5 × **4!**	= 120
6	etc	etc	

```
#include <iostream.h>
int factorial(int n) {
 if (n == 0 || n == 1 ) return 1;
int fact=1;    int i=n;
while (i>1){    fact*=i;
   i--;}
return fact;}
int main() {   int n;
cout << "Enter a non-negative integer: ";
cin >> n;
cout << "Factorial of " << n << " is " << factorial(n) <<
   endl;
 return 0;}
```

Function Overloading

- Two or more functions may have the same name.
- Function (Method) overloading Rules:
  - The number of parameters may differ.
    - int sum (int a, int b);
    - int sum (int a, int b, int c);
  - The data types of parameters may differ.
    - int sum (int a, int b);
    - int sum (int a, float b);
  - The order of parameters may differ.
    - int sum (int a, float b);
    - int sum (float b, int a);

Recursive Functions

- Recursive function is a function that makes one or more calls to itself during execution
- Data structures, especially linked implementations of binary trees, sometimes use recursive functions
- Write C++ Program that find factorial using recursive function .

```
#include <iostream.h>
int factorial(int n) {
 if (n == 0 || n== 1 ) return 1;
return n * factorial(n-1);}
int main() {
 int n;
cout << "Enter a non-negative integer: ";
cin >> n;
cout << "Factorial of " << n << " is " <<
factorial(n) << endl;
 return 0;}
```

Content of a Recursive function

- **Base case(s)**

– Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).

– Every possible chain of recursive calls must eventually reach a base case.

- **Recursive Case**

  – Calls to the current function.

Base Case

```
1    int factorial( int num )
2    {
3        if ( num == 0  || num == 1 )
4                return 1;
5      return num * factorial( num – 1 );
6    }
```

Notice that these lines stopped the recursion – without these lines, the function will call itself over and over again (infinite recursion)

Base Case (cont.)

```
1   int factorial( int num )
2   {
3       if ( num == 0  || num == 1 )
4               return 1;
5       return num * factorial( num – 1 );
6   }
```

These lines are called the *base case* – the case that stops the recursion

Recursive Case

```
1   int factorial( int num )
2   {
3       if ( num == 0  || num == 1 )
4               return 1;
5       return num * factorial( num – 1 );
6   }
```

This line that produces a recursive function call is called the *recursive case*.

All recursive functions have a base case and a recursive case (and sometimes more than one of each).

What If?

```
1   int factorial( int n )
2   {
3       if ( n == 0  || n == 1 )
4               return 1;
5       return n* factorial( n – 1 );
6   }
```

If one makes a mistake and inputs a negative number into this function:

factorial(-2);

what will happen?

Infinite Recursion

```
1   int factorial( int n )
2   {
3       if ( n == 0  || n == 1 )
4               return 1;
5       return n* factorial( n – 1 );
6   }
```

If one makes a mistake and inputs a negative number into this function:

factorial(-2);

what will happen? Infinite recursion.

Elementary Data Structures

- Static and Dynamic Data Structures
- Static Arrays
- Pointers
- Run-Time Arrays
- Structure

Static & Dynamic Data Structures

- Static data are allocated memory at Compile Time, i.e. before the program is executed.
- Static data are allocated their memory space in a place called the Data Segment
- Static data cannot change size during Run Time, i.e. while the program is running.

Static & Dynamic Data Structures

- The Heap (free memory)
- A Dynamic Data Structure is allocated memory at run-time. Consists of nodes to store data and pointers to these nodes to access the data.
- Nodes are created (allocated) and destroyed (de-allocated) at run-time.
- Using dynamic allocation allows your programs to create data structures with sizes that can be defined while the program is running and to expand the sizes when needed.

Static Arrays

The array is the simplest type of data structures.

- **Abstraction:**
- **A homogeneous sequence of elements with a fixed size that allows direct access to its elements.**
- **Elements (members):**
- **The data structure consists of a set of nodes that are sometimes called items or elements of any type, but all elements must be of the same type.**
- **Relationship:**
- **Linear (One-To-one). Ordered storage with direct access.**

Static Arrays

Fundamental Operations:

- **Create array (by declaring it) with a size fixed at compile time**
- **Store an element in the array at a given position (direct access)**
- **Retrieve an element from a given position (direct access)**

Operations on Arrays

Create Array:

- **Size is fixed (constant):**

  e.g. int x[20], string name[50];

- **Retrieve an element: e.g. z = x[i] ;**
- **Store an element: e.g. name[i] = "Anna" ;**

Passing to and from Functions

- Arrays are always passed by reference
- e.g. int findmax (int x [ ] , int size);
- Can use const if array elements are not to be modified, e.g.
- int findmax (const int x [ ] , int size);
-

Computer Memory

- **Each variable is assigned a memory slot (the size depends on the data type) and the variable's data is stored there**

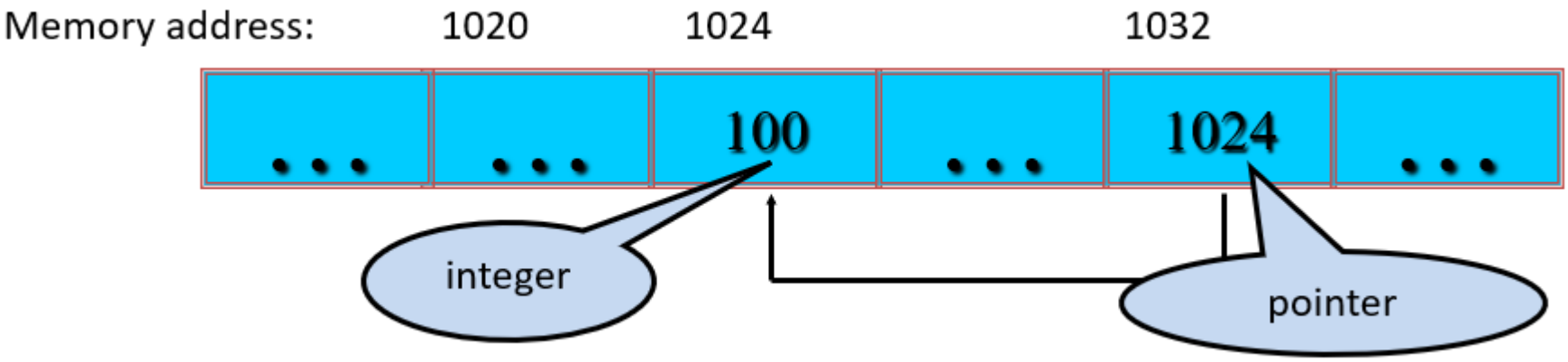
Memory address:

Variable a's value, i.e., 100, is stored at memory location 1024

```
int a = 100;
```

Pointers: The Address of a Variable

- A pointer is a variable used to store the address of a memory cell.
- We can use the pointer to reference this memory cell



Pointer Types

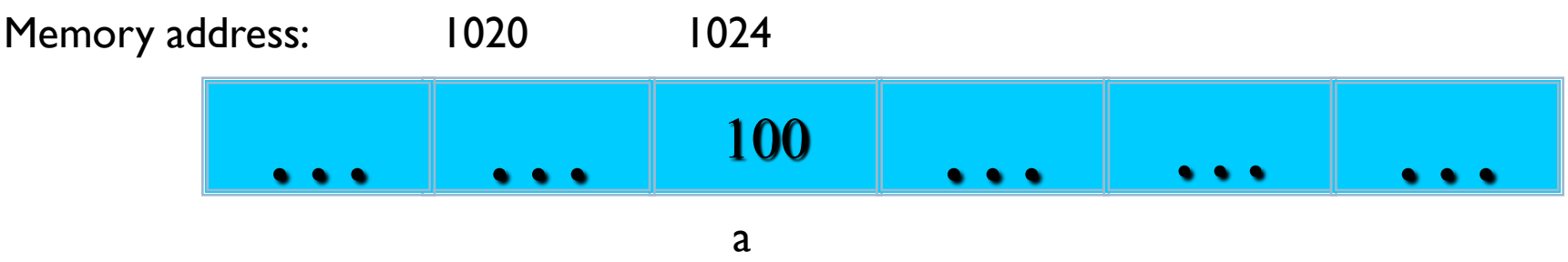
- Pointer
- C++ has pointer types for each type of object
- Pointers to int objects
- Pointers to char objects
- Pointers to user-defined objects
- (e.g., RationalNumber)
- Even pointers to pointers
- Pointers to pointers to int objects

Pointers: The Address of a Variable

- The ampersand symbol & is called the address operator
- The purpose of & is to return the address of a variable in memory.
- For example, if x is a variable then &x is its address in memory.
- We can store the address of a variable in a special variable called a pointer
- A Pointer is a variable whose value is a memory address of an item, not its value
- A pointer knows about the type of the item it points to
- All pointers have fixed size (typically 4 bytes)

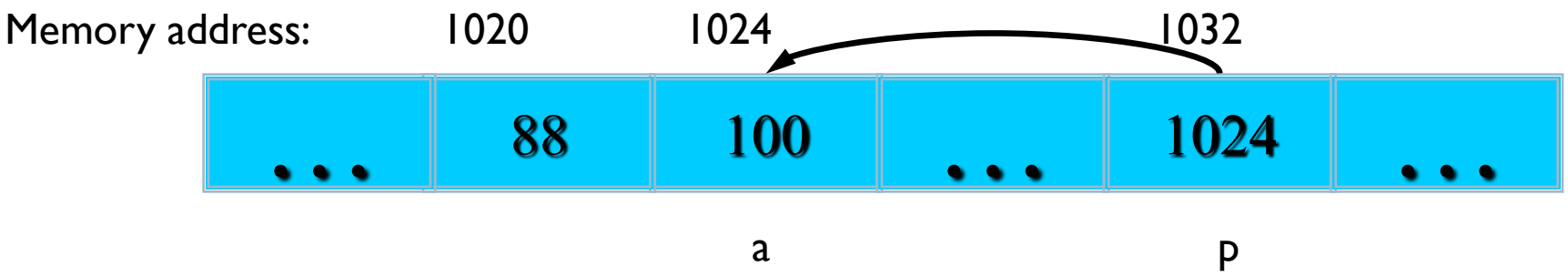
Address Operator &

- *The "address of" operator* (&) **gives the memory address of the variable**
  - **Usage:** `&variable_name`




```
int a = 100;
//get the value,
cout << a;  //prints 100
//get the memory address
cout << &a;   //prints 1024
```

Pointer Variables




```
int a = 100;
int *p = &a;
cout << a << " " << &a <<endl;
cout << p << " " << &p <<endl;
```

Result is:
100 1024
1024 1032

- The value of pointer `p` is the address of variable `a`
- A pointer is also a variable, so it has its own memory address

Pointer Variable

- **Declaration of Pointer variables**

```
type* pointer_name;
 //or
type *pointer_name;
```

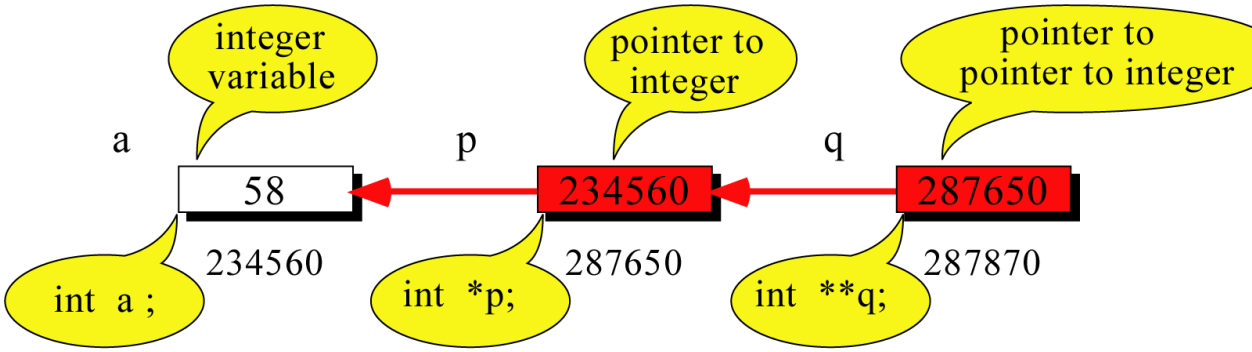
where *type* is the type of data pointed to (e.g. int, char, double)

Examples:

```
int *n;

int* p;
```

Pointer to Pointer

```
// Local Declarations
int         a ;
int        *p ;
int       **q ;
```

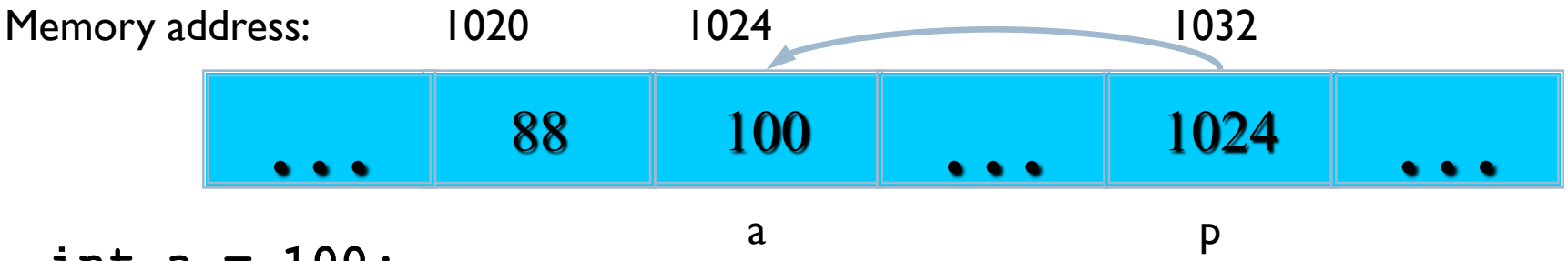


What is the output?

```
// Statements
a = 58 ;
p = &a ;
q = &p ;
cout <<       a << " ";
cout <<      *p << " ";
cout <<     **q << " ";
```

Dereferencing Operator *

- **We can access to the value stored in the variable pointed to by using the dereferencing operator (*),**




```
int a = 100;
int *p = &a;
cout << a << endl;
cout << &a << endl;
cout << p << " " << *p << endl;
cout << &p << endl;
```

Result is:
100
1024
1024 100
1032

Don't get confused

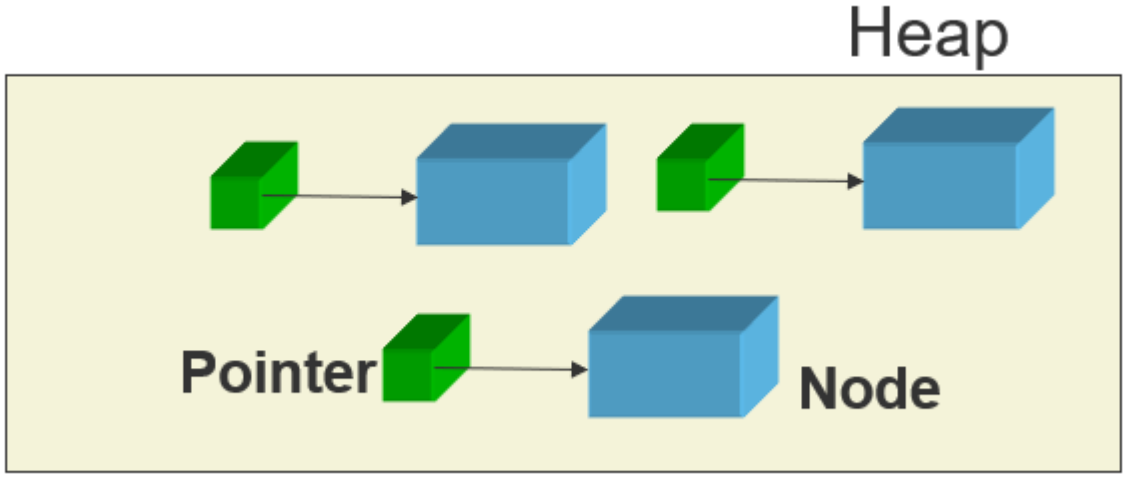
- Declaring a pointer means only that it is a pointer: `int *p;`
- Don't be confused with the dereferencing operator, which is also written with an asterisk (*). They are simply two different tasks represented with the same sign

```
int a = 100, b = 88, c = 8;
int *p1 = &a, *p2, *p3 = &c;
p2 = &b;  // p2 points to b
p2 = p1;  // p2 points to a
b = *p3;  //assign c to b
*p2 = *p3;//assign c to a
cout << a << b << c;
```

Result is:
888

Nodes & Pointers

- A node is an anonymous variable (has no name)
- No name is needed because there is always a pointer pointing to the node.



Creating Nodes: the "new" Operator

- The new operator allocates memory from the heap to a node of specified type at Run Time. It returns the address of that node.
- The statements:

```
int *p  ;
.....
p = new int;
```

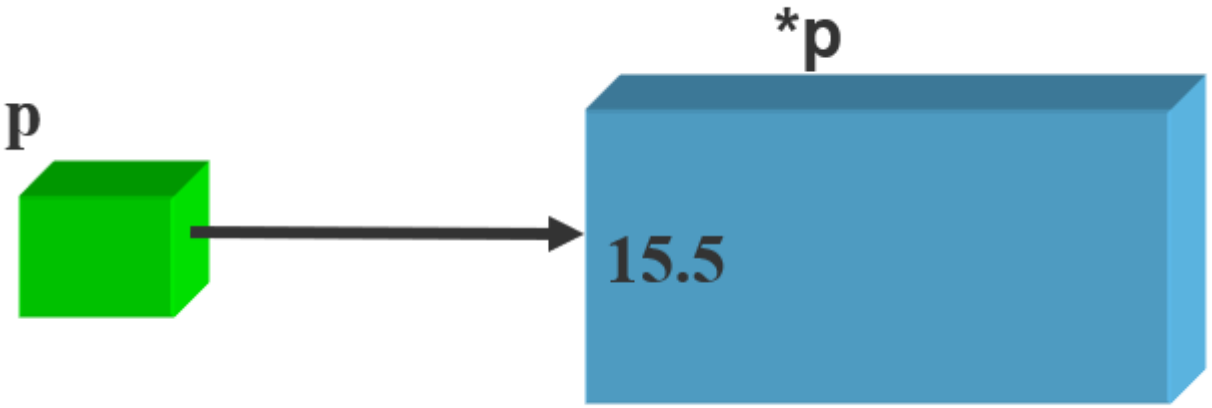
- create a new node of type int and let a pointer p point to it. No data is put into the node
- The node created has no name, it is called an Anonymous Variable. It can only be accessed via its pointer using the indirection operator, i.e. by using (*p)

Accessing Data with Pointers

- * - indirection operator
- *p = 15.5;

// *p reads as: contents of node pointed to by p

Stores floating value 15.5 in the node pointed to by p



Returning Nodes to the Heap

- **Operation:**

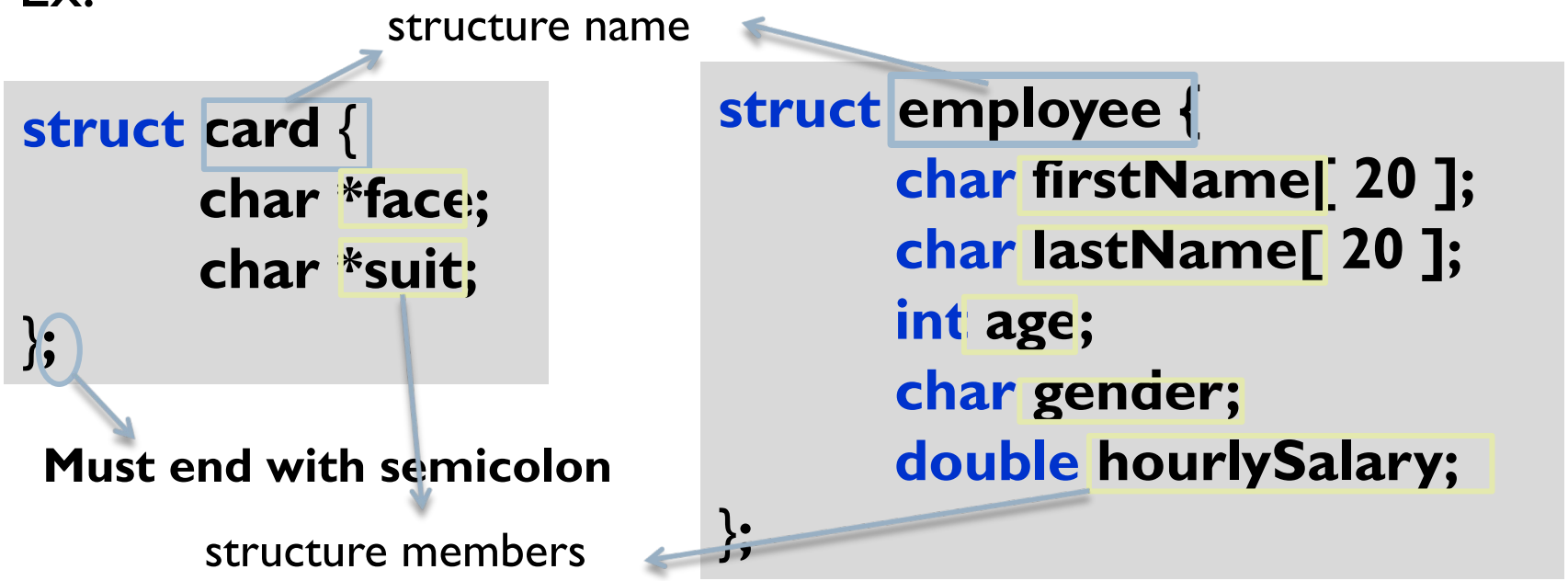
```
delete <pointer variable>;
```

- **Returns space of node pointed to by pointer back to heap for re-use**
- **When finished with a node delete it**
- **Pointer is not destroyed but undefined:**
- **Example:**

```
delete        p;
```

Structure Definitions

- Structures are collections of related variables under one name. They are derived data types
- Ex:



Structures may contain variables of many different data types

Defining Variables of Structure Types

- Structure definitions *do not reserve* any space in memory until you define structure variable.
- To define variables:

```
struct card {
        char *face;
        char *suit;
};
..
struct card aCard;
struct card deck[ 52 ];
struct card *cardPtr;
```

or

Can be omitted like this

```
struct card {
        char *face;
        char *suit;
} aCard, deck[ 52 ], *cardPtr;
```

// define a variable of structure

// define array of structure

// define a pointer to structure

Typedef

- **typedef provides a mechanism for creating synonyms (or aliases) for previously defined data types.**

```
struct card {
        char *face;
        char *suit;
};

typedef struct card Card;
```

or

```
typedef struct card {
        char *face;
        char *suit;
} Card;
```

- **Card** can now be used to declare variables of type **struct card**.

```
Card deck[ 52 ];
```

Self-referential structures

```
struct employee {
        char firstName[ 20 ];
        char lastName[ 20 ];
        int age;
        char gender;
        double hourlySalary;
        struct employee person;
        struct employee *ePtr;

};
```

Can't contain
an instance of itself

/* ERROR */

/* Pointer */

Can include a pointer to
The same structure type
"**self-referential structure**"

Initialize structures& accessing members

```
struct card {
        char *face;
        char *suit;
} aCard;
```

face suit

```
struct card aCard = { "Three", "Hearts" };
```

Or

```
aCard.face = "Three";
aCard.suit = "Hearts";
```

```
cout<<aCard.suit ); /* displays Hearts */
```

Accessing members

Initialize structures& accessing members

```
struct card {
        char *face;
        char *suit;
} *cardPtr;   /* Pointer */
```

```
(*cardPtr).face = “Three";
(*cardPtr).suit = “Hearts";
```

Or

```
cardPtr->face = “Three";
cardPtr->suit = “Hearts";
```

```
Cout<<cardPtr->suit; // displays Hearts
Cout<<(*cardPtr).suit ); // displays Hearts
```

Initialize structures& accessing members

```
struct card {
        char *face;
        char *suit;
} deck[52];
```

```
deck[2].face = "Three";
deck[2].suit = "Hearts";

                ....

for (int i=0; i<52;i++)
        cout<< deck[i].face, deck[i].suit;
```

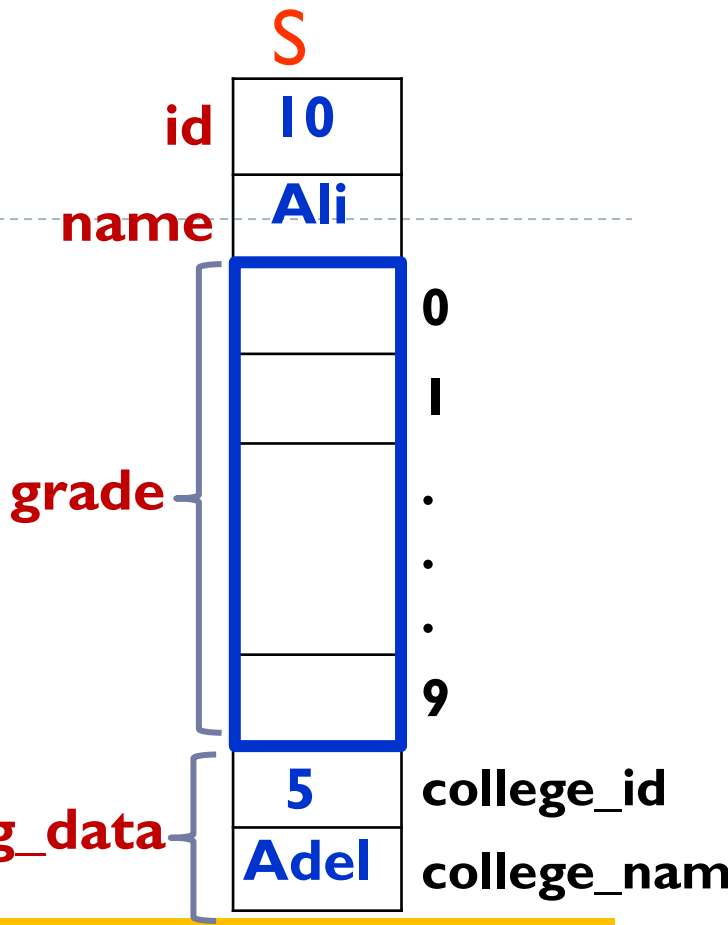
One element of an array of structes, which is a struct

Structures of structures

```
typedef struct student_college
{
    int college_id;
    char college_name[50];
}Stud_col;

struct student
{
    int     id;
    char name[20];
    float grades[10];
    Stud_col      clg_data;
}stu_data,
```

Structures of structures

```
main() {
    struct stu_data s;
    s.id = 10;              s.name = "Ali";

    for( int i=0; i<10; i++)
            cin>>s.grades[i]);

    s.clg_data.college_id = 5;
    s.clg_data.college_name = "Adel";
    }
```



A struct member which is an array

A struct member which is a struct

End of Lecture