# Elementary Data Structures

Doan Nhat Quang

doan-nhat.quang@usth.edu.vn
University of Science and Technology of Hanoi
ICTLab

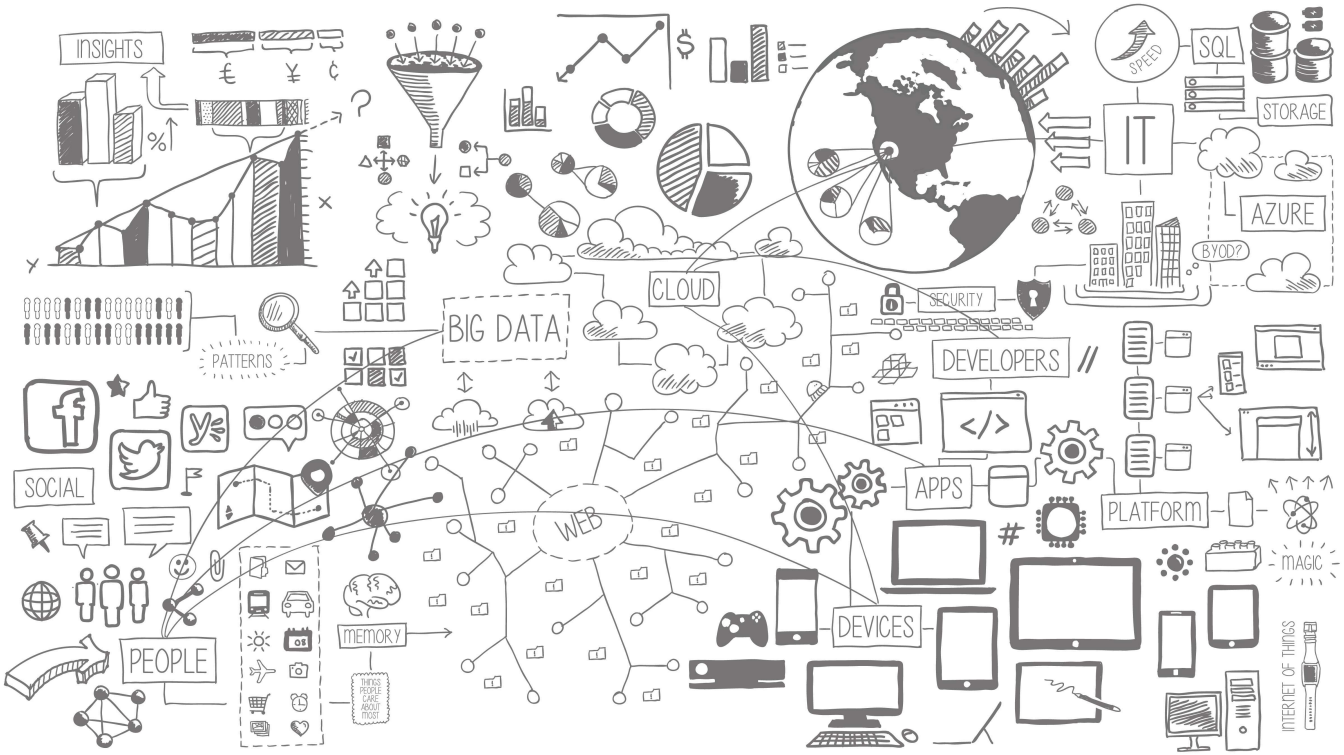
# Today Objectives

- Introduce the fundamental definitions in C/C++.
- Review elementary data types in programming such as array, pointer, structure, enumeration, etc.
- Study the C/C++ examples.

# Definitions

## Data

Data refers to the fact that some existing information or knowledge. Data is a set of values of qualitative or quantitative **variables**.



# Definitions

Everything can be considered as data:

- name, age, adresse of a person
- number, series of number
- pixels or images in RGB color model or in grayscale
- linear functions, polynomial functions, exponential functions
- trees, graphs, maps, documents

# Definitions

## Variables

Being used to store data, variables are simply names used to refer to some location in memory, a location that we can use to write, retrieve, and manipulate throughout the course of program.

# Definitions

## Variables

Being used to store data, variables are simply names used to refer to some location in memory, a location that we can use to write, retrieve, and manipulate throughout the course of program.

## Variable declaration

Variable declaration shows a specific type, which determines the size used in the memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

# Variables

Variable name is an identifier for that variable call-by-name; reference-by-name. The name can be composed of letters, digits, and the underscore character. Upper and lowercase letters are distinct.

```
1 <Type> <Variable>;
2 float F;
3 // declaration of a real number F
4 int id;
5 // declaration of an integer as an id
6 char *address;
7 // declaration of a string of characters
```

# Variables

A variable MUST be initialized with a value before it is used.

**Code C/C++**

```
1  int student_number=1254;
2  double scholarship=1132.50;
3  unsigned char gender=1;
4  string *home_address="Hanoi";
5  char   class_type='A';
```

# Integer Types

There are few ways to declare an integer:

```
1 char                  short int       unsigned short int
2 signed char           int             unsigned int
3 unsigned char         long int        unsigned long int
```

| Type | Size | Description |
|---|---|---|
| char | 8 bits | an integer type $[-127; 127]$ |
| (signed) int | 32 bits | the most natural size of integer for a computer $[-2^{31} - 1; 2^{31} + 1]$ |
| unsigned | 32 bits | non-negative integer number |
| short | 16 bits | a half of normal integer size |
| long | 64 bits | a double of normal integer size |

# Real Types

Represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used.

| Type | Size | Description |
|---|---|---|
| float | 32 bits | a single-precision floating point value |
| double | 64 bits | a double-precision floating point value |
| long double | $\geq$ 64 bits | often more precise than double precision |

# Character Type

Beside the use as an integer, char also can be declared for a character. The value is determined at the character code in the ASCII table.

```
1 char ch = 65;     // an integer
2 char ch = 'A';    // a character
```

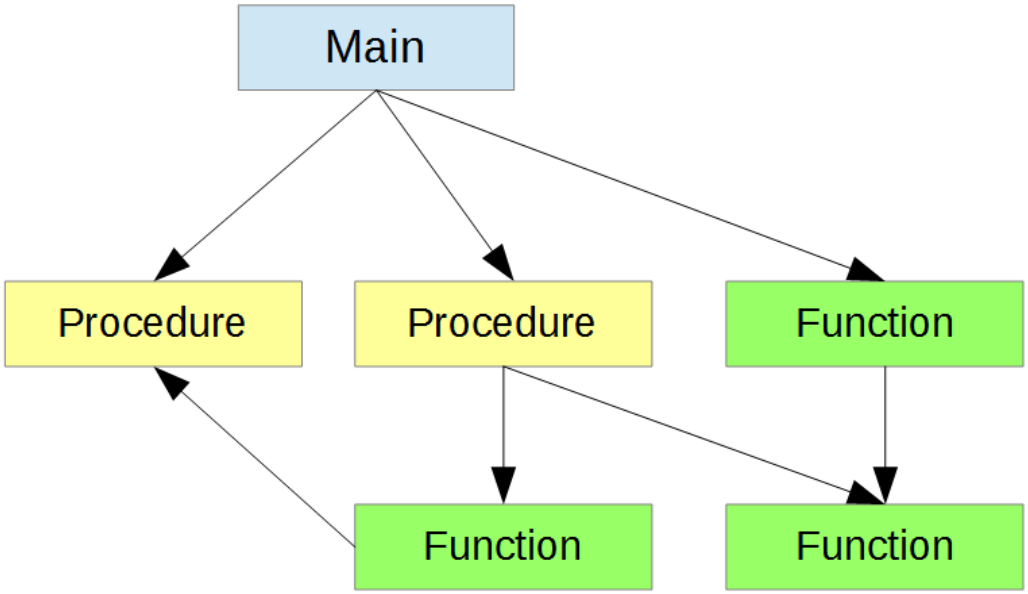
# Examples

```
1  int a, b;  float c;  char d = 'A';
2  b = 1;
3  a = b + 4.5;
4  c = a / 4;
5  d = c + d;
6  printf("%d, %d, %f, %d", a, b, c, d);
```

```
1  int a, b;  float c;  char d = 'A';
2  b = 1;
3  a = b + 4.5;
4  c = a / 4.0;
5  d = c + d;
6  printf("%d, %d, %f, %c", a, b, c, d);
```

## Computer program

**A computer program** is a collection of instructions that performs a specific task when executed by a computer. A program always consists of a main including with many functions and procedures.



# Function

## Function

A named section of a computer program that performs a specific task and returns value.

# Function

## Function

A named section of a computer program that performs a specific task and returns value.

```
1  int sum(int n){
2      int s = 0;
3      for (int i = 1;i<=n;i++)
4          s += i;
5      return s;
6  }
```

# Function

## Function

A named section of a computer program that performs a specific task and returns value.

```
1  int sum(int n){
2      int s = 0;
3      for (int i = 1;i<=n;i++)
4          s += i;
5      return s;
6  }
```

**Attention**

return has to be used to return the value and complete the function.

# Function

## Function

A function can be called or used in other functions.

## Function

A function can be called or used in other functions.

```
1  int doublesum(int n){
2      int sum2 = sum(n) + sum(n);
3      return sum2;
4  }
5  int main(){
6      int n = 10;
7      int sum1 = sum(10);
8      int sum2 = doublesum(10*sum(n));
9      return 0;
10 }
```

# Void

## Void

Void means nothing to be used in C, C++

# Void

## Void

Void means nothing to be used in C, C++

```
1  int myFunction(void) {
2      return 10; // function parameters are absent
3  }
4  void myFunction{}{
5     statement;     // the return value is absent
6  }
```

# Void

## Void

Void means nothing to be used in C, C++

```
1  int myFunction(void) {
2      return 10; // function parameters are absent
3  }
4  void myFunction{}{
5     statement;     // the return value is absent
6  }
```

**Attention**

return in a void function is not necessary; however, it can be used to exit void functions.

# Void

## Void

Void functions can be called and used like normal functions.

# Void

## Void

Void functions can be called and used like normal functions.

```
1  void myPrint() {
2      printf("Hello World!");
3      return;
4  }
5  void myPrint2(int n){
6      printf("Number is %d", n);
7  }
8  int main(){
9     myPrint();
10    myPrint2(100);
11    return 0;
12 }
```

# Global variable vs local variable

## Global variable

A global variable is a variable that it is visible (hence accessible) throughout the program. Its value can be changed anywhere in the code.

# Global variable vs local variable

## Global variable

A global variable is a variable that it is visible (hence accessible) throughout the program. Its value can be changed anywhere in the code.

## Local variable

A local variable is a variable which is either a variable declared within the function or is an argument passed to a function. This type of variable can be only used within a function, after the execution, local variables are removed from the computer memory.

# Global variable vs local variable

## Code C/C++

```
1  int main(){
2      int result = sum(10); // local variable
3  }
4  int sum(int n){
5      int s = 0; // local variable
6      for (int i = 1;i<=n;i++){
7          s += i;
8      }
9      return s;
10 }
```

# Global variable vs local variable

## Code C/C++

```
1  #include <stdio.h>
2  int add_numbers( void );
3  int  value1, value2, value3;
4  int add_numbers( void ){
5      int result = value1 + value2 + value3;
6      return result;
7  }
8  int main(){
9      int result;
10     value1 = 10;  value2 = 20;  value3 = 30;
11     result = add_numbers();
12     printf("The sum of %d + %d + %d is %d\n",
13     value1, value2, value3, result);
14     return 0;
15   }
```

# Arrays

## Definition

- An array is a predefined-size sequential collection of $N$ elements of the same type.
- The objects are called elements of the array and they are indexed by their order in the sequence.
- The element indices are from 0 to $N - 1$.

```
1 <type> <name>[<number of elements>];
2 int age[100]; /* declaration of an array
3 consisted of 100 integer variables */
4 float series[50]; /* declaration of an array
5 consisted of 50 float variables */
```

# Arrays

To access an element in an array, an index is available for use such as $a[0], b[1], a[i], b[i+j]$ with $i, j \in \mathbb{N}$. A basic loop permits to process every elements of the array.

```
1  for (i = 0; i < n; i++){
2     <processing the ith element of the array>;
3   }
```

# Multi-Dimensional Arrays

The simplest form of the multi-dimensional array is the two-dimensional array (a table or a matrix). It can be extended to more general multi-dimensional cases. It's preferable to avoid arrays of dimension more than 3.

```
1 <type> <name> [<nb>][<nb>]...;
2 int a[3][4][5];
3 double b[10][10];
4 char str[17][5];
```

# Multi-Dimensional Arrays

The table indicates the structure of an two dimensional array with an element denoted by $a[i][j]$ where $i$ is the $i^{th}$ row and $j$ is the $j^{th}$ column.

| | Column 0 | Column 1 | Column 2 | Column 3 |
|---|---|---|---|---|
| Row 0 | a[0][0] | a[0][1] | a[0][2] | a[0][3] |
| Row 1 | a[1][0] | a[1][1] | a[1][2] | a[1][3] |
| Row 2 | a[2][0] | a[2][1] | a[2][2] | a[2][3] |

Multi-dimensional arrays may be initialized by specifying bracketed values for each row.

```
1 int a[2][3]={{1,5,8},{2,4,7}};
2 for (int i = 0; i < 2; i++)
3     for (int j = 0; i < 3; j++)
4         statement;
```

# Pointers

## Definition

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, a pointer must be declared before its use to store any variable address.

```
1  int count;
2  int *countPtr = &count;
3  int *undecided = NULL;
4  int &countAlias = count;    // In C++
```

# Pointers

## Definition

Reference of a pointer must be initialized when being declared.
Pointer could be initialized with NULL.

- & or ampersand indicate a reference of a variable.
- * allows to get the value of the variables being pointed by pointers.

# References

## Creating references

- Consider a variable name as a label attached to the variable's location in memory.
- A reference is considered as a second label attached to that memory location. overloading

## Example

We can declare reference variables for i as follows.

```
1  int      i = 17;
2  int&     r = i;
```

Read the & in these declarations as reference: "r is an integer reference initialized to i".

# References

## Example

```
1  #include <iostream>
2   using namespace std;
3   int main () {
4     int     i;
5     double d;
6     // declare reference variables
7     int&    r = i;
8     double& s = d;
9     i = 5;
10    cout << "Value of i : " << i << endl;
11    cout << "Value of i reference : " << r  << endl;
12    d = 11.7;
13    cout << "Value of d : " << d << endl;
14    cout << "Value of d reference : " << s  << endl;
15    return 0;
16  }
```

What is the result?

# References vs Pointers

References are often confused with pointers but

- There is no NULL references. A reference is connected to a legitimate piece of memory.
- Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

# Pass By Value

**Note**

The variables in the formal parameter list are always local variables of a function

- With Pass By Value, function parameters receive copies of the data sent in.
- The original variables passed into a function from another function are not affected by the function call.

# Pass By Value

## Example

```
1  #include <iostream>
2  using namespace std;
3  void twice1(int x){
4    x = x*2; // LOCAL value of x will change
5  }
6  int twice2(int x){
7    return x*2; // return value of x gets changed
8  }
9  int main () {
10   int     i = 10;
11   twice1(i);
12   cout << "value: " << i << endl;
13   i = twice2(i);
14   cout << "value: " << i << endl;
15   return 0;
16 }
```

What is the result?

# Pass By Value

## Example

```
1  #include <iostream>
2  using namespace std;
3  void twice(int x, int y){
4    x = x*2;  // LOCAL value of x will change
5    y = y*2;  // LOCAL value of y will change
6  }
7  int main () {
8      int     a = 10;
9      int     b = 5;
10     twice(a,b);
11     cout << "value 1: " << a << endl;
12     cout << "value 2: " << b << endl;
13     return 0;
14 }
```

What is the result?

# Pass By Reference

- The parameters are still local to the function, but they are reference variables.
- The variables passed into a function DO get changed by the function call.

## Example

```
1 void twice(int &x, int &y){
2   x = x*2;  // these WILL affect the original arguments
3   y = y*2;  // these WILL affect the original arguments
4 }
```

# Pass By Reference

## Example

```
1  #include <iostream>
2  using namespace std;
3  void twice(int &x, int &y){
4    x = x*2;  // these WILL affect the original arguments
5    y = y*2;  // these WILL affect the original arguments
6  }
7  int main () {
8     int     a = 10;
9     int     b = 5;
10    twice(a,b);
11    cout << "value 1: " << a << endl;
12    cout << "value 2: " << b << endl;
13    return 0;
14 }
```

What is the result?

# Pass By Reference

**Note**

When a function expects strict reference types in the parameter list, a value (i.e. a variable, or storage location) must be passed in.

## Example

```
1  void twice(int &x, int &y){
2    x = x*2;  // these WILL affect the original arguments
3    y = y*2;  // these WILL affect the original arguments
4  }
5  int main(){
6    int a = 6, b = 10;
7    twice(a, b);    // it is legal
8    twice(4, b);    // it is NOT legal
9    twice(num + 6, avg - 10.6); // it is NOT legal
10 }
```

# Value vs. Reference

## Pass By Value

- The local parameters are copies of the original arguments passed in.
- Changes made in the function to these variables do not affect originals.

## Pass By Reference

- The local parameters are references to the storage locations of the original arguments passed in.
- Changes to these variables in the function will affect the originals.
- No copy is made, so overhead of copying (time, storage) is saved.

# Arrays vs Pointers

- A variable declared as an array of some type acts as a pointer to that type.
- A pointer can be indexed to access to an array.

```
1  int a[10], *intPtr;
2  intPtr = a; //intPtr pointing to a[0]
3  *(intPtr+5) = 4; //a[5]=4
4  intPtr = &a[7]; //intPtr pointing to the 7th element
5  intPtr++; //intPtr pointing to the 8th element
```

# Arrays vs Pointers

Pointers can also be assigned to reference dynamically allocated memory. The **malloc()** and **calloc()** functions are often used to do this.

```
1  int *intPtr;
2  int size;
3  scanf("%d", &size);
4  intPtr = (int *)malloc(sizeof(int)*(size+10));
5  *(intPtr + 3) = 5;
6  intPtr[3] = 5;
7  free(intPtr);
```

# Arrays vs Pointers

An array of pointers can be used to point to a data array with each element of the pointer array pointing to an element of the data array.

```
1  int *Ptr[5];
2  char *Ptr = "Hello, World";
3  char *Ptr[4]={"Spring","Summer","Autumn","Winter"};
```

Since an array could be replaced by a pointer. An array of pointer is equivalent to pointer to pointer.

```
1  int **Ptr;
2  char **Ptr;
```

## String

String is an one-dimensional array of characters which is terminated by a NULL character '\0'. Built-in functions for C-string is in <string.h>.

```
1  char str1[5]; //maximal 4 characters
2  char str3[]="HANOI";
3  char *str4;
4  char *str4= (char *)calloc(6,sizeof(char));
5  char *str4="HANOI";
```

# Strings

- One possible way to read in a string is by using **scanf()**. This function finishes reading when it reaches a space, or the string would get cut off.
- The function **gets()** can overcome this issue.

```
1 scanf(``%s``,str);
2 // finish when it reaches space or enter
3
4 gets(str)
5 /* finish when it reaches EOL or EOF replace
6  it with 0  they do not do the bound checking
7  of the string! */
```

# Structures

Structure is user defined data type available in C programming, which allows to combine one or more variables, possibly of different types, grouped together under a single name for convenient handling.

```
1 struct [structure tag]{
2     member definition;
3     member definition;
4     ...
5     member definition;
6 } [structure name];
```

# Structures

```
1 typedef struct Student{
2     int age;
3     char name[50];
4     unsigned char gender;
5 } ;
6 struct Student s1, s2;
```

```
1 struct StudentUSTH{
2     int age;
3     char name[50];
4     unsigned char gender;
5 };
6 typedef struct StudentUSTH STH;
7 STH s1, s2;
```

# Structures

To access and process structure fields, dot '.' operator can be used

```
1  s1.age = 20;
2  s2.name = 'Nguyen Van An';
```

Or using this symbol $'->'$ when it involves in pointers

```
1  s1->age = 20;
2  s2->name = 'Nguyen Van An';
```

# Structures and Pointers

Pointer can be used for a single structure variable, but it is mostly used with array of structure variables.

```
1  #include <stdio.h>
2  struct Book{
3    char name[1000];
4    int price;
5  };
6  int main(){
7    struct Book a;        // Single structure variable
8    struct Book* ptr;     // Pointer of Structure type
9    ptr = &a;
10   struct Book b[10];    // Array of structure variables
11   struct Book* p;       // Pointer of Structure type
12   p = &b;
13 }
```

# Enumeration

Enumerated types are types that are defined with finite number of values, known as enumerators, as possible values. The key word for an enumerated type is enum. Here is the syntax:

```
enum <type_name> {
    enum_val1,
    enum_val2,
    enum_val3,
    ...};
```

```
enum Season {Spring, Fall, Summer, Winter};
Season s1, s2;
s1 = Summer;
s2 = Fall;
if (s1 == Summer)
    printf(''Summer is comming'');
```