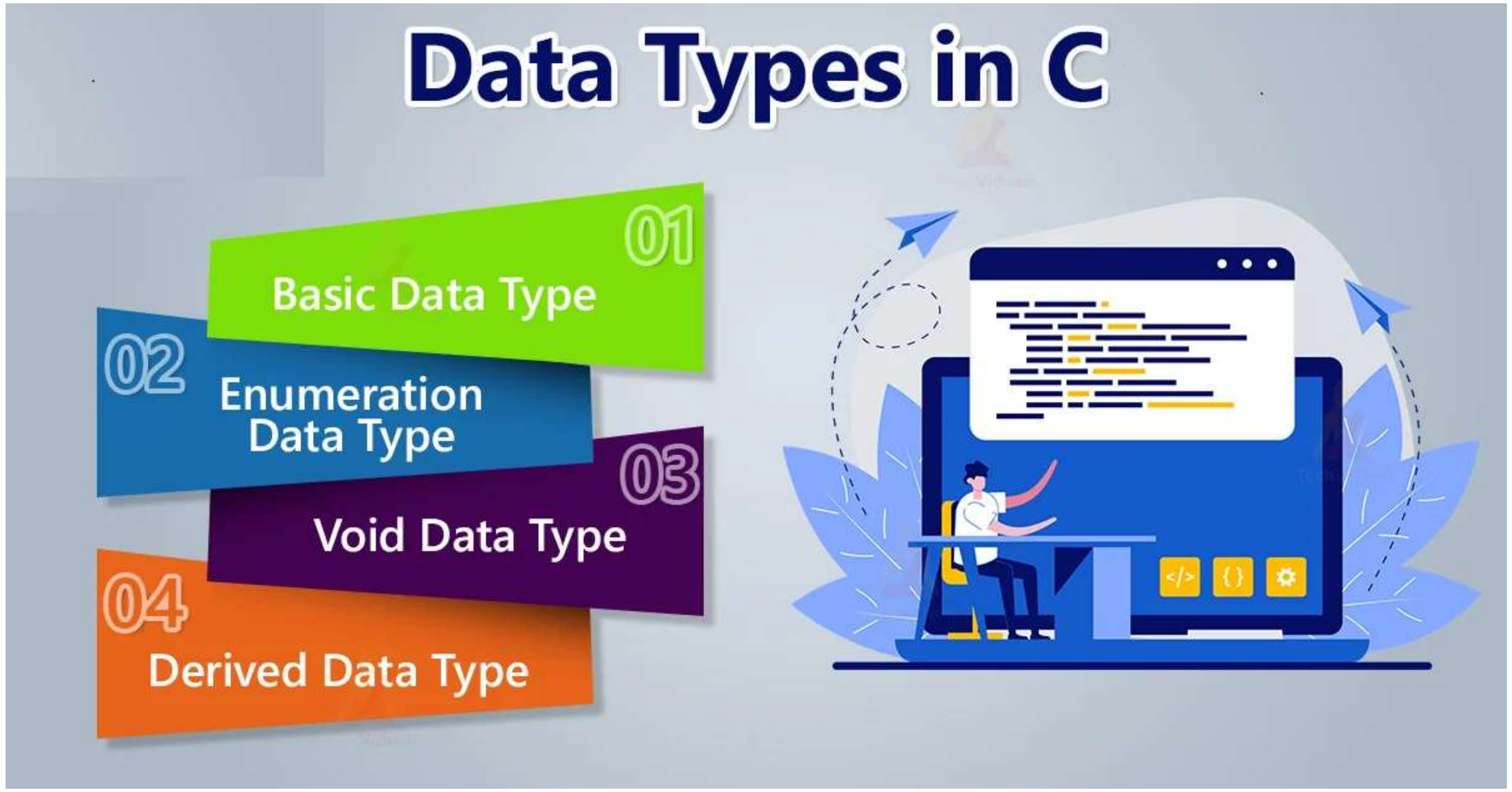
Fundamentals of Language C

Data Types
Array and Pointers
Structure and Union Dynamic Memory Allocation

Learning Objectives

By the end of this chapter, you will be able to:

- Discuss the importance of Data Types available in C
- Explain the importance of Array and Pointers in C
- Discuss the Structure and Union in C
- Explain how to use Dynamic Memory Allocation in C


Data Types in C
01
Basic Data Type
02
Enumeration Data Type
03
Void Data Type
04
Derived Data Type

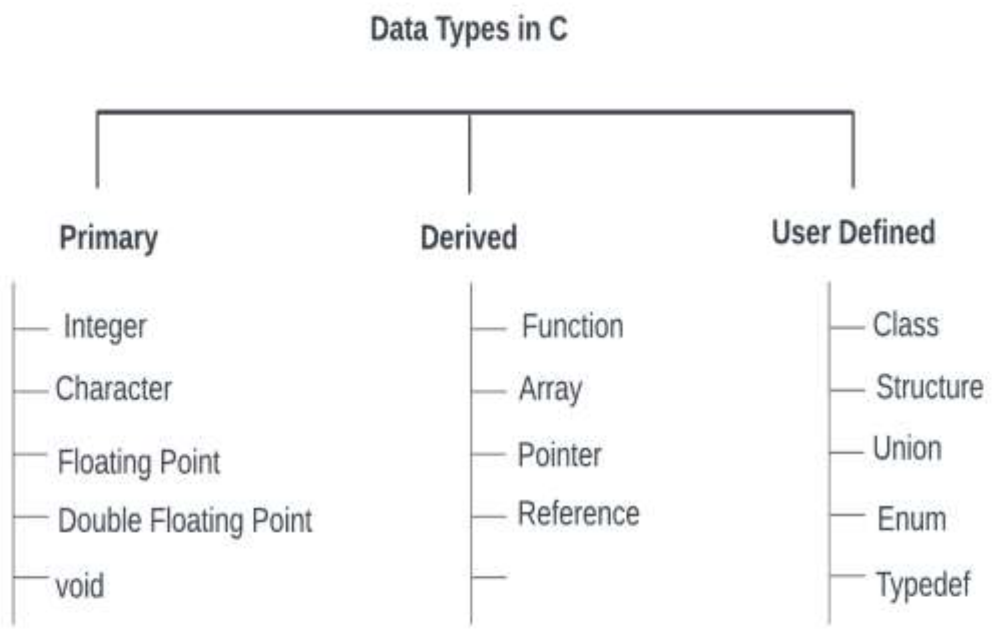
Data Types Available in C

In C, every variable is linked to a specific data type, indicating the kind of data it can hold, such as integer, character, floating-point, double, and more. Each data type requires varying amounts of memory and comes with distinct operations that can be performed on it. A data type represents a collection of data with predefined values and characteristics.

The data types in C can be classified as follows:

Types	Description
Primitive Data Types	Primitive data types are the most basic data types that are used for representing simple values such as integers, float, characters, etc.
User Defined Data Types	The user-defined data types are defined by the user himself.
Derived Types	The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.

Different data types also have different ranges up to which they can store numbers. These ranges may vary from compiler to compiler. Below is a list of ranges along with the memory requirement and format specifiers on the 32-bit GCC compiler.



Data Type	Size (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld

Data Type	Size (bytes)	Range	Format Specifier
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4	1.2E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
long double	16	3.4E-4932 to 1.1E+4932	%Lf

Arrays

- An array is a collection of elements of the same type that are referenced by a common name.
- Compared to the basic data type (int, float & char) it is an aggregate or derived data type.
- All the elements of an array occupy a set of contiguous memory locations.
- Why need to use array type?
- Consider the following issue:

"We have a list of 1000 students' marks of an integer type. If using the basic data type (int), we will declare something like the following…"

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

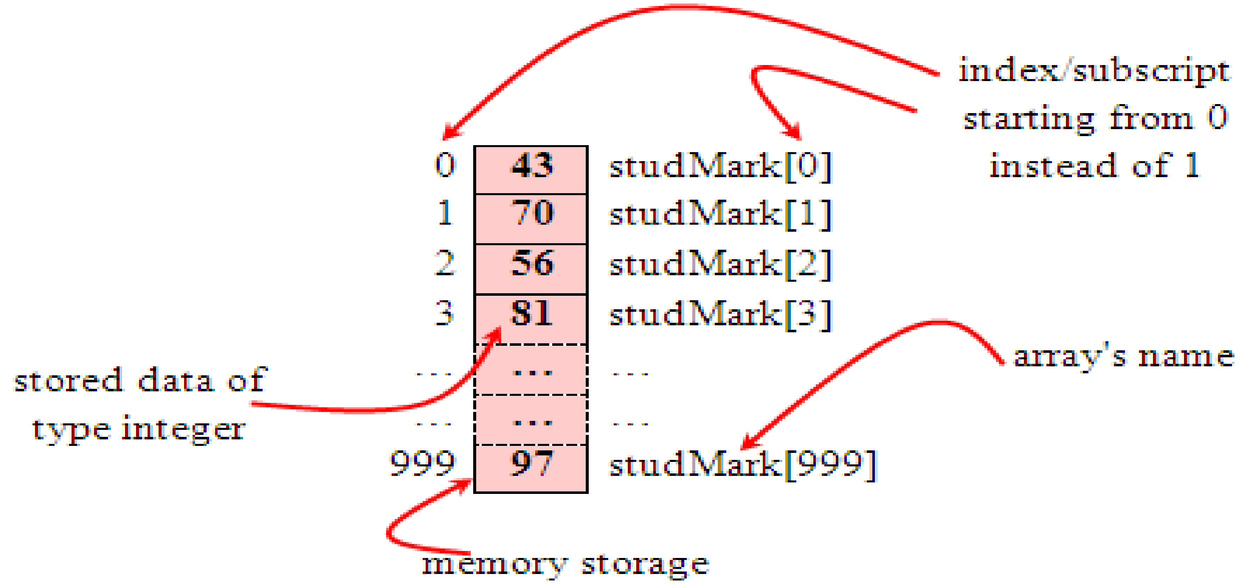
- Can you imagine how long we have to write the declaration part by using normal variable declaration?

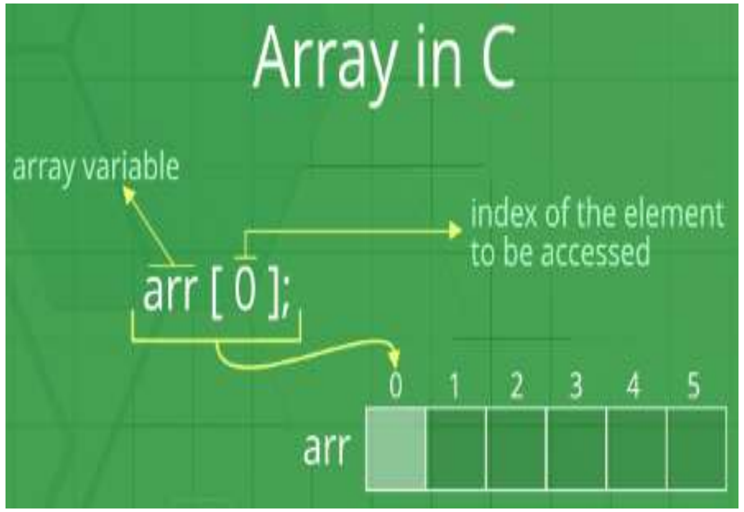
```
void main()
{
        int studMark1, studMark2, studMark3, studMark4, …, …, studMark998, stuMark999,
        studMark1000;
        …
        …
        getch();
}
```

- By using an array, we just declare like this,

  - `int  studMark[1000];`

- This will reserve 1000 contiguous memory locations for storing the students' marks.
- Graphically, this can be depicted as in the following figure.





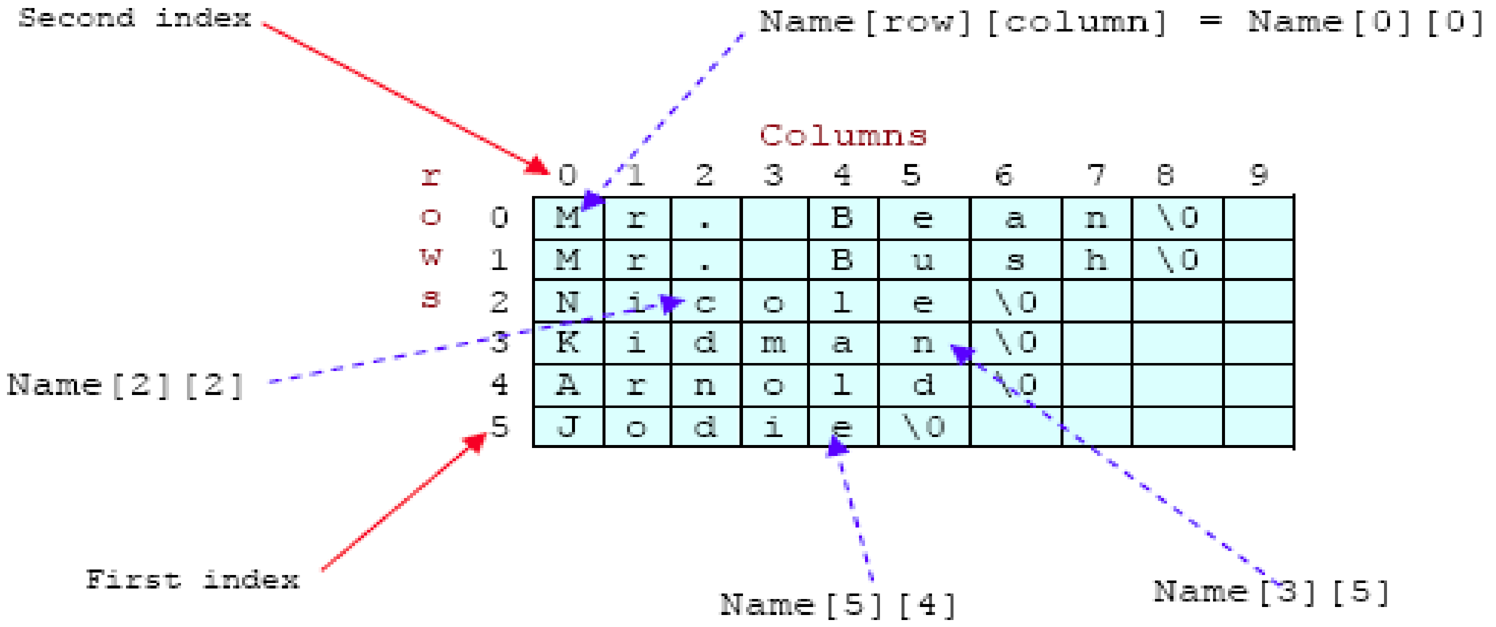
- Dimension refers to the array's size, which is how big the array is.
- A single or one dimensional array declaration has the following form,
  - data_type array_name[array_size];
- Here, **data_type** define the base type of the array, which is the type of each element in the array.
- **array_name** is any valid C / C++ identifier name that obeys the same rule for the identifier naming.
- **array_size** defines how many elements the array will hold.

Array Initilization

- An array may be initialized at the time of declaration.
- Giving initial values to an array.
- Initialization of an array may take the following form,
  - type   array_name[size] = {a_list_of_value};
- For example:
  - int   idNum[7] = {1, 2, 3, 4, 5, 6, 7};
  - float fFloatNum[5] = {5.6, 5.7, 5.8, 5.9, 6.1};
  - char   chVowel[6] = {'a', 'e', 'i', 'o', 'u', '\0'};
- The first line declares an integer array idNum and it immediately assigns the values 1, 2, 3, ..., 7 to idNum[0], idNum[1], idNum[2],..., idNum[6] respectively.
- The second line assigns the values 5.6 to fFloatNum[0], 5.7 to fFloatNum[1], and so on.
- Similarly the third line assigns the characters 'a' to chVowel[0], 'e' to chVowel[1], and so on. Note again, for characters we must use the single apostrophe/quote (') to enclose them.
- Also, the last character in chVowel is NULL character ('\0').

2D Array

- If we assign initial string values for the 2D array it will look something like the following,
  - `char Name[6][10] = {"Mr. Bean", "Mr. Bush", "Nicole", "Kidman", "Arnold", "Jodie"};`
- Here, we can initialize the array with 6 strings, each with maximum 9 characters long.
- If depicted in rows and columns it will look something like the following and can be considered as contiguous arrangement in the memory.



	0	1	2	3	4	5	6	7	8	9
0	M	r	.		B	e	a	n	\0	
1	M	r	.		B	u	s	h	\0	
2	N	i	c	o	l	e	\0			
3	K	i	d	m	a	n	\0			
4	A	r	n	o	l	d	\0			
5	J	o	d	i	e	\0				

Pointers

- When declaring a variable, the compiler sets aside memory storage with a unique address to store that variable.
- The compiler associates that address with the variable's name.
- When the program uses the variable name, it automatically accesses a proper memory location.
- No need to concern which address.
- But we can manipulate the memory address by using pointers.
- Let say we declare a variable named Rate of type integer and assign an initial value as follows,
  - `int   Rate = 10;`

```
#include <stdio.h>
void main()
{
      int Rate = 10;
      printf("variable name is Rate.\n");
      // or sizeof(int)
      printf("size of Rate (int) is %d bytes.\n", sizeof(Rate));//2
bytes
      printf("initial value stored at Rate is %d.\n", Rate); //10
      printf("memory address  of Rate is %u.\n", &Rate); //65522

}
```

- So, the pointer variable declaration becomes something like this,
  - `int   *pToRate;`
- In other word, the pToRate variable holds the memory address of Rate and is therefore a pointer to Rate.
- The asterisk (*) is used to show that is it the pointer variable instead of normal variable.

Definition: A pointer is a variable that holds memory address of another variable, where, the actual data is stored.

- A pointer is a numeric variable and like other variables, must be declared and initialized before it can be used.
- The following is a general form for declaring a pointer variable,
  - type_of_stored_data   * pointer_variable_name;
- For example,
  - char*   chName;
  - int   *   nTypeOfCar;
  - float   *fValue;
- type_of_stored_data is any valid pointer base type such as char, int, float or other valid C derived types such as array and struct.
- It indicates the type of the variable's data to which the pointer is pointed to.
- The pointer_variable_name follows the same rules as other C variable naming convention and must be unique.

- Once a pointer is declared, we must initialize it.
- This makes the pointer pointing to something.
- Don’t make it point to nothing; it is dangerous.
- Uninitialized pointers will not cause a compiler error, but using an uninitialized pointer could result in unpredictable and potentially disastrous outcomes.
- Until pointer holds an address of a variable, it isn’t useful.
- C uses two pointer operators,

1. Indirection operator (*) – asterisk symbol, has been explained previously.
2. Address-of-operator (&) – ampersand symbol, means return the address of…

- When & operator is placed before the name of a variable, it will returns the memory address of the variable instead of stored value.

Structures in C

A structure is

- a convenient way of grouping several pieces of related information together
- a collection of variables under a single name

The variables in structures can be of different types, and each has a name which is used to select it from the structure

Example :

ID (integer) && Name (char array)

→ A Student record

Definition: A structure is a new named type that may involve several different typed variables.

Compound data:

A Student Record is

- an `int rollNo` and
- an `char name[10]day` and
- an `float marks.`

```
struct Student{
   int  rollNo;
   char name[10];
   float marks;
};

struct Student st1;
st1.rollNo=100;
st1.name="Raju";
st1.marks=87.23;
```

Union in C

In C, a union is a unique data type that enables the storage of various data types in the same memory location. While you can define a union with multiple members, only one member can hold a value at any given moment. This feature allows unions to efficiently utilize the same memory location for multiple purposes.

The format of the union statement is as follows –

```
union [union tag]
{
        member definition;
        member definition;
         ...
        member definition;
} [one or more union variables];
```

Here is the way you would define a union type named Data having three members i, f, and str –

```
union Data
{
        int i;
        float f;
        char str [20];
} data;
```

Note: The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string.

Dynamic memory allocation in C

Dynamic memory allocation in the C language empowers C programmers to allocate memory during runtime.

Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

- malloc()
- calloc()
- realloc()
- free()

malloc()	allocates single block of requested memory.
calloc ()	allocates multiple block of requested memory.
realloc ()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() and calloc() function in C

malloc():

The malloc() function allocates single block of requested memory. It doesn't initialize memory at execution time, so it has garbage value initially. It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

- **ptr=(cast-type*)malloc(byte-size)**

calloc():

The calloc() function allocates multiple block of requested memory. It initially initialize all bytes to zero. It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

- **ptr=(cast-type*)calloc(number, byte-size)**

realloc() and free() function in C

realloc() :

f memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

- ptr=realloc(ptr, **new**-size)

free():

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

- free(ptr)

THANK YOU