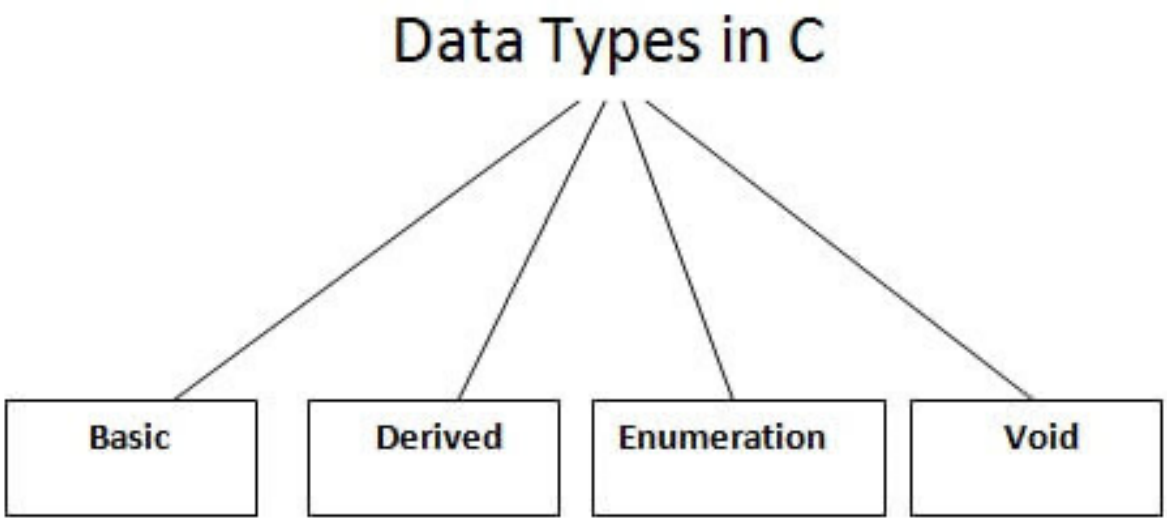
Data Types in C

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.



There are the following data types in C language.

Types	Data Types
Basic Data Type	int, char, float, double
Derived Data Type	array, pointer, structure, union
Enumeration Data Type	enum
Void Data Type	void

Basic Data Types

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals.

The memory size of the basic data types may change according to 32 or 64-bit operating system.

Let's see the basic data types. Its size is given **according to 32-bit architecture**.

Data Types	Memory Size	Range
char	1 byte	−128 to 127
signed char	1 byte	−128 to 127
unsigned char	1 byte	0 to 255

short	2 byte	−32,768 to 32,767
signed short	2 byte	−32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	2 byte	−32,768 to 32,767
signed int	2 byte	−32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	−32,768 to 32,767
signed short int	2 byte	−32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
float	4 byte	
double	8 byte	
long double	10 byte	

Int:

Integers are entire numbers without any fractional or decimal parts, and the ***int data type*** is used to represent them.

It is frequently applied to variables that include ***values***, such as ***counts, indices***, or other numerical numbers. The ***int data type*** may represent both ***positive*** and ***negative numbers*** because it is signed by default.

An ***int*** takes up ***4 bytes*** of memory on most devices, allowing it to store values between around -2 billion and +2 billion.

Char:

Individual characters are represented by the ***char data type***. Typically used to hold ***ASCII*** or ***UTF-8 encoding scheme characters***, such as ***letters, numbers, symbols***, or ***commas***. There are ***256 characters*** that can be represented by a single char, which takes up one byte of memory. Characters such as ***'A', 'b', '5',*** or ***'$'*** are enclosed in single quotes.

Float:

To represent integers, use the ***floating data type***. Floating numbers can be used to represent fractional units or numbers with decimal places.
The ***float type*** is usually used for variables that require very good precision but may not be very precise. It can store values with an accuracy of about ***6 decimal places*** and a range of about ***3.4 x 1038*** in ***4 bytes*** of memory.

Double:

Use two data types to represent ***two floating integers***. When additional precision is needed, such as in scientific calculations or financial applications, it provides greater accuracy compared to float.
Double type, which uses ***8 bytes*** of memory and has an accuracy of about ***15 decimal places, yields larger values***. C treats floating point numbers as doubles by default if no explicit type is supplied.

```
1. int age = 25;
2. char grade = 'A';
3. float temperature = 98.6;
4. double pi = 3.14159265359;
```

In the example above, we declare four variables: an ***int variable*** for the person's age, a ***char variable*** for the student's grade, a ***float variable*** for the temperature reading, and two variables for the ***number pi.***

Derived Data Type

Beyond the fundamental data types, C also supports ***derived data types,*** including ***arrays, pointers, structures,*** and ***unions***. These data types give programmers the ability to handle heterogeneous data, directly modify memory, and build complicated data structures.

Array:

An ***array, a derived data type***, lets you store a sequence of ***fixed-size elements*** of the same type. It provides a mechanism for joining multiple targets of the same data under the same name.
The index is used to access the elements of the array, with a ***0 index*** for the first entry. The size of the array is fixed at declaration time and cannot be changed during program execution. The array components are placed in adjacent memory regions.
Here is an example of declaring and utilizing an array:

```
#include <stdio.h>

int main() {
    int numbers[5]; // Declares an integer array with a size of 5 elements

    // Assign values to the array elements
    numbers[0] = 10;
    numbers[1] = 20;
```

```
    numbers[2] = 30;
    numbers[3] = 40;
    numbers[4] = 50;

    // Display the values stored in the array
    printf("Values in the array: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Values in the array: 10 20 30 40 50
```

Pointer:

A ***pointer*** is a derived data type that keeps track of another data type's memory address. When a ***pointer*** is declared, the ***data type*** it refers to is ***stated first***, and then the ***variable name*** is preceded by ***an asterisk (*)***.

You can have incorrect access and change the value of variable using pointers by specifying the memory address of the variable. ***Pointers*** are commonly used in ***tasks*** such as ***function pointers, data structures***, and ***dynamic memory allocation***.

Here is an example of declaring and employing a pointer:

```
#include <stdio.h>

int main() {
    int num = 42;      // An integer variable
    int *ptr;          // Declares a pointer to an integer

    ptr = #        // Assigns the address of 'num' to the pointer

    // Accessing the value of 'num' using the pointer
    printf("Value of num: %d\n", *ptr);

    return 0;
}
```

Output:

```
Value of num: 42
```

Structure:

A structure is a derived data type that enables the creation of composite data types by allowing the grouping of many data types under a single name. It gives you the ability to create your own unique data structures by fusing together variables of various sorts.

1. A structure's members or fields are used to refer to each variable within it.
2. Any data type, including different structures, can be a member of a structure.
3. A structure's members can be accessed by using the dot (.) operator.

A declaration and use of a structure is demonstrated here:

```c
#include <stdio.h>
#include <string.h>
// Define a structure representing a person
struct Person {
    char name[50];
    int age;
    float height;
};

int main() {
    // Declare a variable of type struct Person
    struct Person person1;

    // Assign values to the structure members
    strcpy(person1.name, "John Doe");
    person1.age = 30;
    person1.height = 1.8;

    // Accessing the structure members
    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    printf("Height: %.2f\n", person1.height);

    return 0;
}
```

Output:

```
Name: John Doe
Age: 30
Height: 1.80
```

Union:

A derived data type called a ***union*** enables you to store various data types in the same memory address. In contrast to structures, where each member has a separate memory space, members of a union all share a single memory space. A value can only be held by one member of a union at any given moment. When you need to represent many data types interchangeably, unions come in handy. Like structures, you can access the members of a union by using the ***dot (.)*** operator.
Here is an example of a union being declared and used:

```c
#include <stdio.h>
// Define a union representing a numeric value
union NumericValue {
    int intValue;
    float floatValue;
    char stringValue[20];
};
int main() {
    // Declare a variable of type union NumericValue
    union NumericValue value;
    // Assign a value to the union
    value.intValue = 42;
    // Accessing the union members
    printf("Integer Value: %d\n", value.intValue);
    // Assigning a different value to the union
    value.floatValue = 3.14;
    // Accessing the union members
    printf("Float Value: %.2f\n", value.floatValue);

    return 0;
}
```

Output:

```
Integer Value: 42
Float Value: 3.14
```

Enumeration Data Type

A set of named constants or ***enumerators*** that represent a collection of connected values can be defined in C using the ***enumeration data type (enum). Enumerations*** give you the means to give names that make sense to a group of integral values, which makes your code easier to read and maintain.
Here is an example of how to define and use an enumeration in C:

```c
#include <stdio.h>
```

```c
// Define an enumeration for days of the week
enum DaysOfWeek {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
};

int main() {
    // Declare a variable of type enum DaysOfWeek
    enum DaysOfWeek today;

    // Assign a value from the enumeration
    today = Wednesday;

    // Accessing the enumeration value
    printf("Today is %d\n", today);

    return 0;
}
```

Output:

```
Today is 2
```

Void Data Type

The ***void data type*** in the C language is used to denote the lack of a particular type. ***Function return types, function parameters***, and ***pointers*** are three situations where it is frequently utilized.

Function Return Type:

A ***void return type*** function does not produce a value. A ***void function*** executes a task or action and ends rather than returning a value.

Example:

```c
void printHello() { printf("Hello, world!\n"); }
```

Function Parameters:

The ***parameter void*** can be used to indicate that a function accepts no arguments.

Example:

```c
void processInput(void) { /* Function logic */ }
```

Pointers:

Any address can be stored in a pointer of type ***void****, making it a universal pointer. It offers a method for working with pointers to ambiguous or atypical types.

Example:

```
1. void* dataPtr;
```

The ***void data type*** is helpful for defining functions that don't accept any arguments when working with generic pointers or when you wish to signal that a function doesn't return a value. It is significant to note that while ***void**** can be used to build generic pointers, void itself cannot be declared as a variable type.

Here is a sample of code that shows how to utilize void in various situations:

```
#include <stdio.h>
// Function with void return type
void printHello() {
    printf("Hello, world!\n");
}
// Function with void parameter
void processInput(void) {
    printf("Processing input...\n");
}

int main() {
    // Calling a void function
    printHello();

    // Calling a function with void parameter
    processInput();

    // Using a void pointer
    int number = 10;
    void* dataPtr = &number;
    printf("Value of number: %d\n", *(int*)dataPtr);

    return 0;
}
```

Output:

```
Hello, world!
Processing input...
Value of number: 10
```

Conclusion:

As a result, ***data types*** are essential in the C programming language because they define the kinds of information that variables can hold. They provide the data's size and format, enabling the compiler to allot memory and carry out the necessary actions. Data types supported by C include ***void, enumeration, derived***, and ***basic types***. In addition to floating-point types like ***float*** and ***double***, basic data types in C also include integer-based kinds like ***int, char***, and ***short***. These forms can be ***signed*** or ***unsigned***, and they fluctuate in size and range. To create dependable and efficient code, it is crucial to comprehend the memory size and scope of these types.

A few examples of ***derived data types*** are ***unions, pointers, structures***, and ***arrays***. Multiple elements of the same kind can be stored together in contiguous memory due to arrays. ***Pointers*** keep track of memory addresses, allowing for fast data structure operations and dynamic memory allocation. While ***unions*** allow numerous variables to share the same memory space, structures group relevant variables together.

Code becomes more legible and maintainable when named constants are defined using enumeration data types. ***Enumerations*** give named constants integer values to enable the meaningful representation of related data. The void data type indicates the lack of a particular type. It is used as a return type for both ***functions*** and ***function parameters*** that don't take any arguments and don't return a value. The ***void* pointer*** also functions as a general pointer that can ***store addresses*** of various types.

C programming requires a solid understanding of ***data types***. Programmers can ensure adequate memory allocation, avoid ***data overflow*** or ***truncation***, and enhance the readability and maintainability of their code by selecting the right ***data type***. C programmers may create ***effective, dependable***, and well-structured code that satisfies the requirements of their applications by having a firm understanding of data types.