23 June 2020

|  |  |  |  |
| --- | --- | --- | --- |
| Date: | 23 June 2020 | Name: | Srinidhi J C |
| Course: | C++ Programming | USN: | 4al16ec078 |
| Topic: | Data types, arrays & pointers, Functions | Semester & Section: | 8th & b |
| Github Repository: | SrinidhiJC078 |  |  |
| FORENOON SESSION DETAILS | | | | | |
| Image of sessionA screenshot of a cell phone  Description automatically generated  A screenshot of a cell phone  Description automatically generated | | | | | |

Report:

# C++ Data Types

All [variables](https://www.geeksforgeeks.org/variables-and-keywords-in-c/) use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data-type with which it is declared. Every data type requires a different amount of memory.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/20191113115600/DatatypesInC.png)

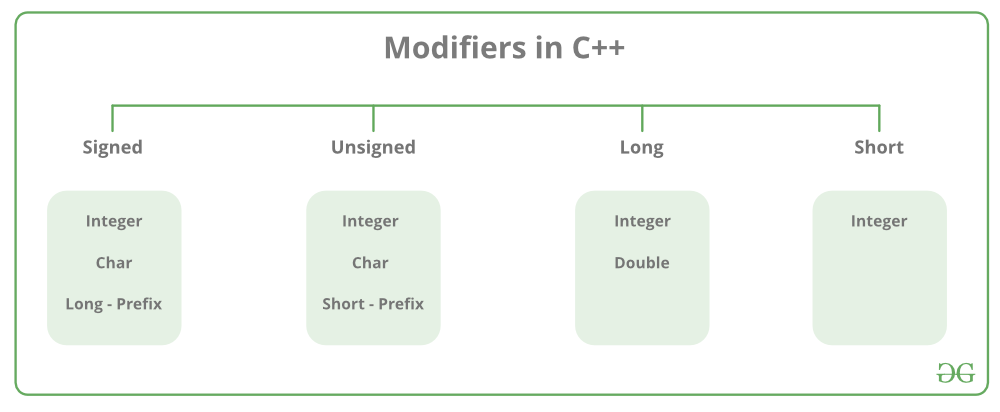
Data types in C++ is mainly divided into three types:

1. **Primitive Data Types**: These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char , float, bool etc. Primitive data types available in C++ are:
   * Integer
   * Character
   * Boolean
   * Floating Point
   * Double Floating Point
   * Valueless or Void
   * Wide Character
2. [**Derived Data Types:**](https://www.geeksforgeeks.org/derived-data-types-in-c/) The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:
   * Function
   * Array
   * Pointer
   * Reference
3. [**Abstract or User-Defined Data Types**](https://www.geeksforgeeks.org/user-defined-derived-data-types-in-c/): These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:
   * Class
   * Structure
   * Union
   * Enumeration
   * Typedef defined DataType

This article discusses **primitive data types** available in C++.

* **Integer**: Keyword used for integer data types is **int**. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.
* **Character**: Character data type is used for storing characters. Keyword used for character data type is **char**. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.
* **Boolean**: Boolean data type is used for storing boolean or logical values. A boolean variable can store either true or false. Keyword used for boolean data type is **bool**.
* **Floating Point**: Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is **float**. Float variables typically requires 4 byte of memory space.
* **Double Floating Point**: Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for double floating point data type is **double**. Double variables typically requires 8 byte of memory space.
* **void**: Void means without any value. void datatype represents a valueless entity. Void data type is used for those function which does not returns a value.
* [**Wide Character**](https://www.geeksforgeeks.org/wide-char-and-library-functions-in-c/): Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by **wchar\_t**. It is generally 2 or 4 bytes long.

**Datatype Modifiers**

As the name implies, datatype modifiers are used with the built-in data types to modify the length of data that a particular data type can hold.  
[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/20191113121347/ModifiersInC.png)  
Data type modifiers available in C++ are:

* **Signed**
* **Unsigned**
* **Short**
* **Long**

Below table summarizes the modified size and range of built-in datatypes when combined with the type modifiers:

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size (in bytes)** | **Range** |
| short int | 2 | -32,768 to 32,767 |
| unsigned short int | 2 | 0 to 65,535 |
| unsigned int | 4 | 0 to 4,294,967,295 |
| int | 4 | -2,147,483,648 to 2,147,483,647 |
| long int | 4 | -2,147,483,648 to 2,147,483,647 |
| unsigned long int | 4 | 0 to 4,294,967,295 |
| long long int | 8 | -(2^63) to (2^63)-1 |
| unsigned long long int | 8 | 0 to 18,446,744,073,709,551,615 |
| signed char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |
| float | 4 |  |
| double | 8 |  |
| long double | 12 |  |
| wchar\_t | 2 or 4 | 1 wide character |

We can display the size of all the data types by using the sizeof() operator and passing the keyword of the datatype as argument to this function as shown below:

|  |
| --- |
| // C++ program to sizes of data types  #include<iostream>  using namespace std;    int main()  {      cout << "Size of char : " << sizeof(char)        << " byte" << endl;      cout << "Size of int : " << sizeof(int)        << " bytes" << endl;      cout << "Size of short int : " << sizeof(short int)        << " bytes" << endl;      cout << "Size of long int : " << sizeof(long int)         << " bytes" << endl;      cout << "Size of signed long int : " << sizeof(signed long int)         << " bytes" << endl;      cout << "Size of unsigned long int : " << sizeof(unsigned long int)         << " bytes" << endl;      cout << "Size of float : " << sizeof(float)         << " bytes" <<endl;      cout << "Size of double : " << sizeof(double)         << " bytes" << endl;      cout << "Size of wchar\_t : " << sizeof(wchar\_t)         << " bytes" <<endl;        return 0;  } |

Output:

Size of char : 1 byte

Size of int : 4 bytes

Size of short int : 2 bytes

Size of long int : 8 bytes

Size of signed long int : 8 bytes

Size of unsigned long int : 8 bytes

Size of float : 4 bytes

Size of double : 8 bytes

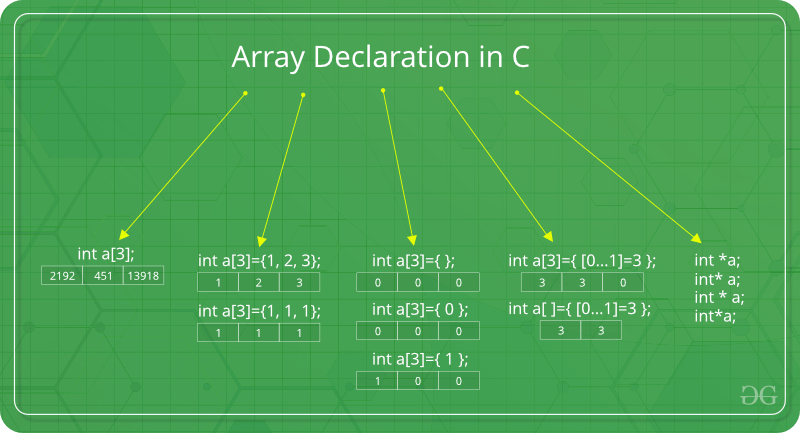
Size of wchar\_t : 4 bytes

# Arrays in C/C++

An array in C or C++ is a collection of items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They are used to store similar type of elements as in the data type must be the same for all elements. They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C or C++ can store derived data types such as the structures, pointers etc. Given below is the picturesque representation of an array.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2015/05/Arrays.png)

**Why do we need arrays?**  
We can use normal variables (v1, v2, v3, ..) when we have a small number of objects, but if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.

**Array declaration in C/C++:**  


There are various ways in which we can declare an array. It can be done by specifying its type and size, by initializing it or both.

1. **Array declaration by specifying size**

|  |
| --- |
| // Array declaration by specifying size  int arr1[10];  // With recent C/C++ versions, we can also  // declare an array of user specified size  int n = 10;  int arr2[n]; |

1. **Array declaration by initializing elements**

|  |
| --- |
| // Array declaration by initializing elements  int arr[] = { 10, 20, 30, 40 }  // Compiler creates an array of size 4.  // above is same as  "int arr[4] = {10, 20, 30, 40}" |

1. **Array declaration by specifying size and initializing elements**

|  |
| --- |
| // Array declaration by specifying size and initializing  // elements  int arr[6] = { 10, 20, 30, 40 }  // Compiler creates an array of size 6, initializes first  // 4 elements as specified by user and rest two elements as 0.  // above is same as  "int arr[] = {10, 20, 30, 40, 0, 0}" |

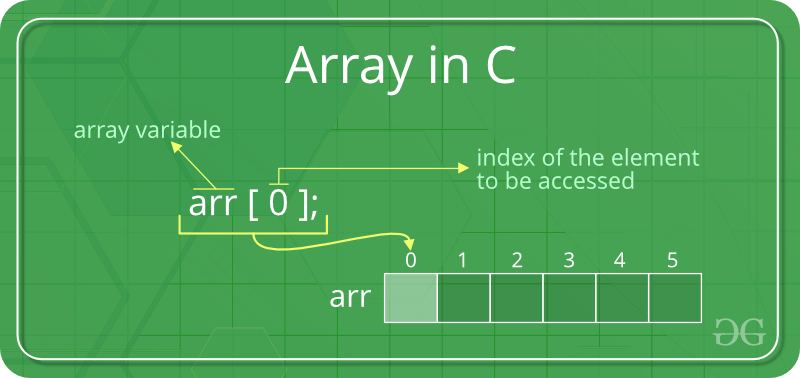
**Advantages of an Array in C/C++:**

1. Random access of elements using array index.
2. Use of less line of code as it creates a single array of multiple elements.
3. Easy access to all the elements.
4. Traversal through the array becomes easy using a single loop.
5. Sorting becomes easy as it can be accomplished by writing less line of code.

**Disadvantages of an Array in C/C++:**

1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.
2. Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

**Facts about Array in C/C++:**

* **Accessing Array Elements:**  
  Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1.  
  

**Example:**

|  |
| --- |
| #include <stdio.h>    int main()  {      int arr[5];      arr[0] = 5;      arr[2] = -10;      arr[3 / 2] = 2; // this is same as arr[1] = 2      arr[3] = arr[0];        printf("%d %d %d %d", arr[0], arr[1], arr[2], arr[3]);        return 0;  } |

**Output:**

5 2 -10 5

 **No Index Out of bound Checking:**  
There is no index out of bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

|  |
| --- |
| // This C program compiles fine  // as index out of bound  // is not checked in C.    #include <stdio.h>    int main()  {      int arr[2];        printf("%d ", arr[3]);      printf("%d ", arr[-2]);        return 0;  } |

**Output:**

2008101287 4195777

 In C, it is not compiler error to initialize an array with more elements than the specified size. For example, the below program compiles fine and shows just Warning.

|  |
| --- |
| #include <stdio.h>  int main()  {        // Array declaration by initializing it with more      // elements than specified size.      int arr[2] = { 10, 20, 30, 40, 50 };        return 0;  } |

**Warnings:**

prog.c: In function 'main':

prog.c:7:25: warning: excess elements in array initializer

int arr[2] = { 10, 20, 30, 40, 50 };

^

prog.c:7:25: note: (near initialization for 'arr')

prog.c:7:29: warning: excess elements in array initializer

int arr[2] = { 10, 20, 30, 40, 50 };

^

prog.c:7:29: note: (near initialization for 'arr')

prog.c:7:33: warning: excess elements in array initializer

int arr[2] = { 10, 20, 30, 40, 50 };

^

prog.c:7:33: note: (near initialization for 'arr')

 **The elements are stored at contiguous memory locations**  
**Example:**

|  |
| --- |
| // C program to demonstrate that array elements are stored  // contiguous locations    #include <stdio.h>  int main()  {      // an array of 10 integers.  If arr[0] is stored at      // address x, then arr[1] is stored at x + sizeof(int)      // arr[2] is stored at x + sizeof(int) + sizeof(int)      // and so on.      int arr[5], i;        printf("Size of integer in this compiler is %lu\n", sizeof(int));        for (i = 0; i < 5; i++)          // The use of '&' before a variable name, yields          // address of variable.          printf("Address arr[%d] is %p\n", i, &arr[i]);        return 0;  } |

**Output:**

Size of integer in this compiler is 4

Address arr[0] is 0x7ffd636b4260

Address arr[1] is 0x7ffd636b4264

Address arr[2] is 0x7ffd636b4268

Address arr[3] is 0x7ffd636b426c

Address arr[4] is 0x7ffd636b4270

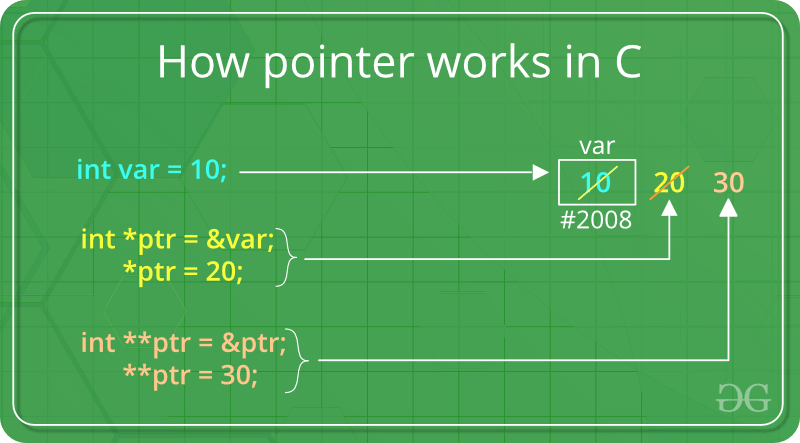
# Pointers in C/C++ with Examples

Pointers are symbolic representation of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. It’s general declaration in C/C++ has the format:

**Syntax:**

datatype \*var\_name;

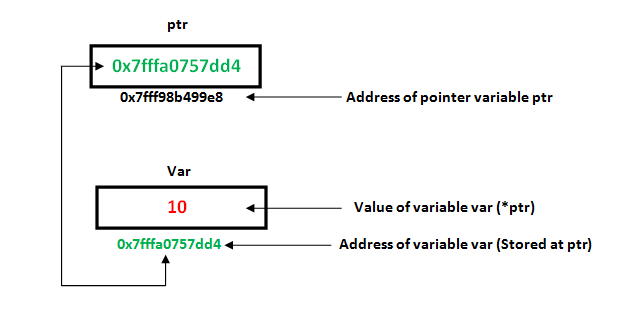
int \*ptr; //ptr can point to an address which holds int data



**How to use a pointer?**

* Define a pointer variable
* Assigning the address of a variable to a pointer using unary operator (&) which returns the address of that variable.
* Accessing the value stored in the address using unary operator (\*) which returns the value of the variable located at the address specified by its operand.

The reason we associate data type to a pointer is **that it knows how many bytes the data is stored in**. When we increment a pointer, we increase the pointer by the size of data type to which it points.

[](https://media.geeksforgeeks.org/wp-content/uploads/pointers-in-c.png)

|  |
| --- |
| // C++ program to illustrate Pointers in C++  #include <stdio.h>  void geeks()  {      int var = 20;        // declare pointer variable      int \*ptr;        // note that data type of ptr and var must be same      ptr = &var;      // assign the address of a variable to a pointer      printf("Value at ptr = %p \n",ptr);      printf("Value at var = %d \n",var);      printf("Value at \*ptr = %d \n", \*ptr);  }    // Driver program  int main()  {      geeks();  } |

**Output:**

Value at ptr = 0x7ffcb9e9ea4c

Value at var = 20

Value at \*ptr = 20

**References and Pointers**

There are 3 ways to pass C++ arguments to a function:

* call-by-value
* call-by-reference with pointer argument

|  |
| --- |
| // C++ program to illustrate call-by-methods in C++  #include <bits/stdc++.h>  using namespace std;  //Pass-by-Value  int square1(int n)  {      //Address of n in square1() is not the same as n1 in main()      cout << "address of n1 in square1(): " << &n << "\n";        // clone modified inside the function      n \*= n;      return n;  }  //Pass-by-Reference with Pointer Arguments  void square2(int \*n)  {      //Address of n in square2() is the same as n2 in main()      cout << "address of n2 in square2(): " << n << "\n";        // Explicit de-referencing to get the value pointed-to      \*n \*= \*n;  }  //Pass-by-Reference with Reference Arguments  void square3(int &n)  {      //Address of n in square3() is the same as n3 in main()      cout << "address of n3 in square3(): " << &n << "\n";        // Implicit de-referencing (without '\*')      n \*= n;  }  void geeks()  {      //Call-by-Value      int n1=8;      cout << "address of n1 in main(): " << &n1 << "\n";      cout << "Square of n1: " << square1(n1) << "\n";      cout << "No change in n1: " << n1 << "\n";      //Call-by-Reference with Pointer Arguments      int n2=8;      cout << "address of n2 in main(): " << &n2 << "\n";      square2(&n2);      cout << "Square of n2: " << n2 << "\n";      cout << "Change reflected in n2: " << n2 << "\n";        //Call-by-Reference with Reference Arguments      int n3=8;      cout << "address of n3 in main(): " << &n3 << "\n";      square3(n3);      cout << "Square of n3: " << n3 << "\n";      cout << "Change reflected in n3: " << n3 << "\n";  }  //Driver program  int main()  {      geeks();  } |

Output:

address of n1 in main(): 0x7ffcdb2b4a44

address of n1 in square1(): 0x7ffcdb2b4a2c

Square of n1: 64

No change in n1: 8

address of n2 in main(): 0x7ffcdb2b4a48

address of n2 in square2(): 0x7ffcdb2b4a48

Square of n2: 64

Change reflected in n2: 64

address of n3 in main(): 0x7ffcdb2b4a4c

address of n3 in square3(): 0x7ffcdb2b4a4c

Square of n3: 64

Change reflected in n3: 64

In C++, by default arguments are passed by value and the changes made in the called function will not reflect in the passed variable. The changes are made into a clone made by the called function.  
If wish to modify the original copy directly (especially in passing huge object or array) and/or avoid the overhead of cloning, we use pass-by-reference. Pass-by-Reference with Reference Arguments does not require any clumsy syntax for referencing and dereferencing.

* [Function pointers in C](https://www.geeksforgeeks.org/function-pointer-in-c/)
* [Pointer to a function](https://www.geeksforgeeks.org/how-to-declare-a-pointer-to-a-function/)

**Array Name as Pointers**

An array name contains the address of first element of the array which acts like constant pointer. It means, the address stored in array name can’t be changed.  
For example, if we have an array named val then **val** and **&val[0]** can be used interchangeably.

|  |
| --- |
| // C++ program to illustrate Array Name as Pointers in C++  #include <bits/stdc++.h>  using namespace std;  void geeks()  {      //Declare an array      int val[3] = { 5, 10, 20 };        //declare pointer variable      int \*ptr;      //Assign the address of val[0] to ptr      // We can use ptr=&val[0];(both are same)      ptr = val ;      cout << "Elements of the array are: ";      cout << ptr[0] << " " << ptr[1] << " " << ptr[2];  }  //Driver program  int main()  {      geeks();  } |

Output:

Elements of the array are: 5 10 20

[](https://media.geeksforgeeks.org/wp-content/uploads/Untitled-presentation-2.png)  
If pointer ptr is sent to a function as an argument, the array val can be accessed in a similar fashion.

[Pointer vs Array](https://www.geeksforgeeks.org/difference-pointer-array-c/)

**Pointer Expressions and Pointer Arithmetic**

A limited set of arithmetic operations can be performed on pointers which are:

* incremented ( ++ )
* decremented ( — )
* an integer may be added to a pointer ( + or += )
* an integer may be subtracted from a pointer ( – or -= )
* difference between two pointers (p1-p2)

|  |
| --- |
| // C++ program to illustrate Pointer Arithmetic in C++  #include <bits/stdc++.h>  using namespace std;  void geeks()  {      //Declare an array      int v[3] = {10, 100, 200};        //declare pointer variable      int \*ptr;      //Assign the address of v[0] to ptr      ptr = v;        for (int i = 0; i < 3; i++)      {              cout << "Value at ptr = " << ptr << "\n";              cout << "Value at \*ptr = " << \*ptr << "\n";                // Increment pointer ptr by 1              ptr++;      }  }  //Driver program  int main()  {      geeks();  } |

Output:

Value at ptr = 0x7fff9a9e7920

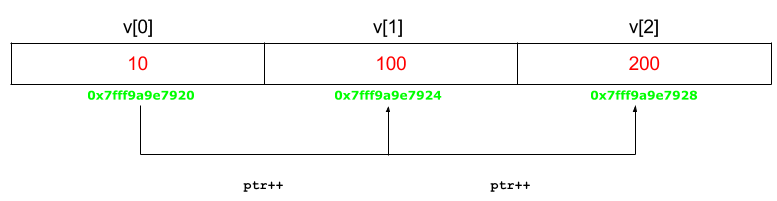
Value at \*ptr = 10

Value at ptr = 0x7fff9a9e7924

Value at \*ptr = 100

Value at ptr = 0x7fff9a9e7928

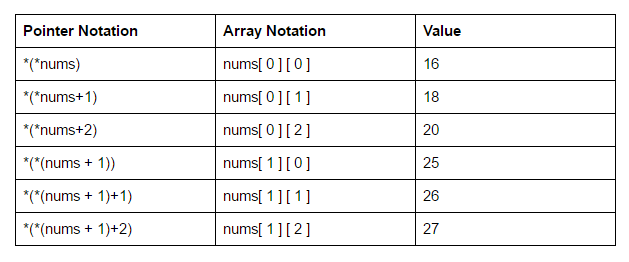
Value at \*ptr = 200

[](https://media.geeksforgeeks.org/wp-content/uploads/Untitled-presentation-31.png)  
**Advanced Pointer Notation**

Consider pointer notation for the two-dimensional numeric arrays. consider the following declaration

int nums[2][3] = { { 16, 18, 20 }, { 25, 26, 27 } };

**In general, nums[ i ][ j ] is equivalent to \*(\*(nums+i)+j)**

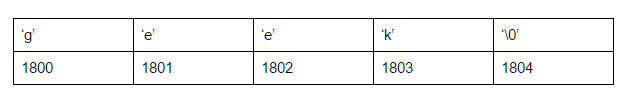
[](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-222.png)

**Pointers and String literals**

String literals are arrays containing null-terminated character sequences. String literals are arrays of type character plus terminating null-character, with each of the elements being of type const char (as characters of string can’t be modified).

const char \* ptr = "geek";

This declares an array with the literal representation for “geek”, and then a pointer to its first element is assigned to ptr. If we imagine that “geek” is stored at the memory locations that start at address 1800, we can represent the previous declaration as:

[](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-23.png)

As pointers and arrays behave in the same way in expressions, ptr can be used to access the characters of string literal. For example:

char x = \*(ptr+3);

char y = ptr[3];

Here, both x and y contain k stored at 1803 (1800+3).

**Pointers to pointers**

In C++, we can create a pointer to a pointer that in turn may point to data or other pointer. The syntax simply requires the unary operator (\*) for each level of indirection while declaring the pointer.

char a;

char \*b;

char \*\* c;

a = ’g’;

b = &a;

c = &b;

Here b points to a char that stores ‘g’ and c points to the pointer b.

[**Void Pointers**](http://quiz.geeksforgeeks.org/void-pointer-c/)

This is a special type of pointer available in C++ which represents absence of type. void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).  
This means that void pointers have great flexibility as it can point to any data type. There is payoff for this flexibility. These pointers cannot be directly dereferenced. They have to be first transformed into some other pointer type that points to a concrete data type before being dereferenced.

|  |
| --- |
| // C++ program to illustrate Void Pointer in C++  #include <bits/stdc++.h>  using namespace std;  void increase(void \*data,int ptrsize)  {      if(ptrsize == sizeof(char))      {          char \*ptrchar;            //Typecast data to a char pointer          ptrchar = (char\*)data;            //Increase the char stored at \*ptrchar by 1          (\*ptrchar)++;          cout << "\*data points to a char"<<"\n";      }      else if(ptrsize == sizeof(int))      {          int \*ptrint;          //Typecast data to a int pointer          ptrint = (int\*)data;            //Increase the int stored at \*ptrchar by 1          (\*ptrint)++;          cout << "\*data points to an int"<<"\n";      }  }  void geek()  {      // Declare a character      char c='x';        // Declare an integer      int i=10;        //Call increase function using a char and int address respectively      increase(&c,sizeof(c));      cout << "The new value of c is: " << c <<"\n";      increase(&i,sizeof(i));      cout << "The new value of i is: " << i <<"\n";    }  //Driver program  int main()  {      geek();  } |

Output:

\*data points to a char

The new value of c is: y

\*data points to an int

The new value of i is: 11

**Invalid pointers**

A pointer should point to a valid address but not necessarily to valid elements (like for arrays). These are called invalid pointers. Uninitialized pointers are also invalid pointers.

int \*ptr1;

int arr[10];

int \*ptr2 = arr+20;

Here, ptr1 is uninitialized so it becomes an invalid pointer and ptr2 is out of bounds of arr so it also becomes an invalid pointer.  
(Note: invalid pointers do not necessarily raise compile errors)

[**NULL Pointers**](http://quiz.geeksforgeeks.org/few-bytes-on-null-pointer-in-c/)

Null pointer is a pointer which point nowhere and not just an invalid address.  
Following are 2 methods to assign a pointer as NULL;

int \*ptr1 = 0;

int \*ptr2 = NULL;

# Functions in C/C++

A function is a set of statements that take inputs, do some specific computation and produces output.

The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can call the function.  
**Example:**   
Below is a simple C/C++ program to demonstrate functions.

|  |
| --- |
| #include <iostream>  using namespace std;  int max(int x, int y)  {      if (x > y)      return x;      else      return y;  }  int main() {      int a = 10, b = 20;      // Calling above function to find max of 'a' and 'b'      int m = max(a, b);      cout << "m is " << m;      return 0;  } |

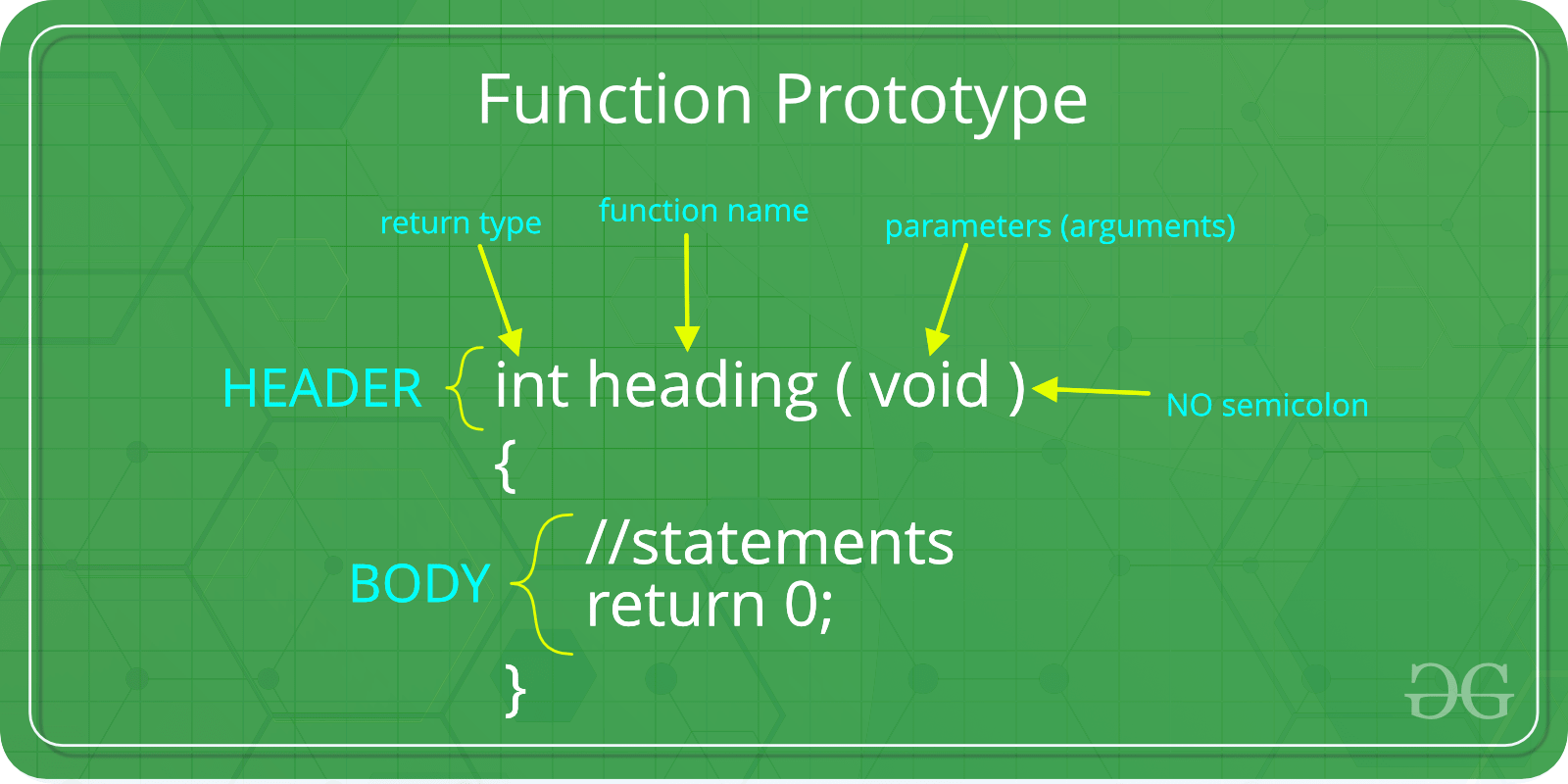
**Output:**

m is 20

**Why do we need functions?**

* Functions help us in reducing code redundancy. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.
* Functions make code modular. Consider a big file having many lines of codes. It becomes really simple to read and use the code if the code is divided into functions.
* Functions provide abstraction. For example, we can use library functions without worrying about their internal working.

**Function Declaration**  
A function declaration tells the compiler about the number of parameters function takes, data-types of parameters and return type of function. Putting parameter names in function declaration is optional in the function declaration, but it is necessary to put them in the definition. Below are an example of function declarations. (parameter names are not there in below declarations)



|  |
| --- |
| // A function that takes two integers as parameters  // and returns an integer  int max(int, int);  // A function that takes a int pointer and an int variable as parameters  // and returns an pointer of type int  int \*swap(int\*,int);  // A function that takes a charas parameters  // and returns an reference variable  char \*call(char b);  // A function that takes a char and an int as parameters  // and returns an integer  int fun(char, int); |

It is always recommended to declare a function before it is used (See [this](https://www.geeksforgeeks.org/what-is-the-purpose-of-a-function-prototype/), [this](https://www.geeksforgeeks.org/g-fact-95/) and [this](https://www.geeksforgeeks.org/importance-of-function-prototype-in-c/) for details)

In C, we can do both declaration and definition at the same place, like done in the above example program.

C also allows to declare and define functions separately, this is especially needed in case of library functions. The library functions are declared in header files and defined in library files. Below is an example declaration.   
**Parameter Passing to functions**  
The parameters passed to function are called ***actual parameters***. For example, in the above program 10 and 20 are actual parameters.  
The parameters received by function are called ***formal parameters***. For example, in the above program x and y are formal parameters.  
There are two most popular ways to pass parameters.

***Pass by Value:*** In this parameter passing method, values of actual parameters are copied to function’s formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

***Pass by Reference*** Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

Parameters are always passed by value in C. For example. in the below code, value of x is not modified using the function fun().

|  |
| --- |
| #include <iostream>  using namespace std;  void fun(int x) {      x = 30;  }  int main() {      int x = 20;      fun(x);      cout << "x = " << x;      return 0;  } |

**Output:**

x = 20

However, in C, we can use pointers to get the effect of pass by reference. For example, consider the below program. The function fun() expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator \* is used to access the value at an address. In the statement ‘\*ptr = 30’, value at address ptr is changed to 30. The address operator & is used to get the address of a variable of any data type. In the function call statement ‘fun(&x)’, the address of x is passed so that x can be modified using its address.

|  |
| --- |
| #include <iostream>  using namespace std;  void fun(int \*ptr)  {      \*ptr = 30;  }  int main() {      int x = 20;      fun(&x);      cout << "x = " << x;      return 0;  } |

Output:

x = 30