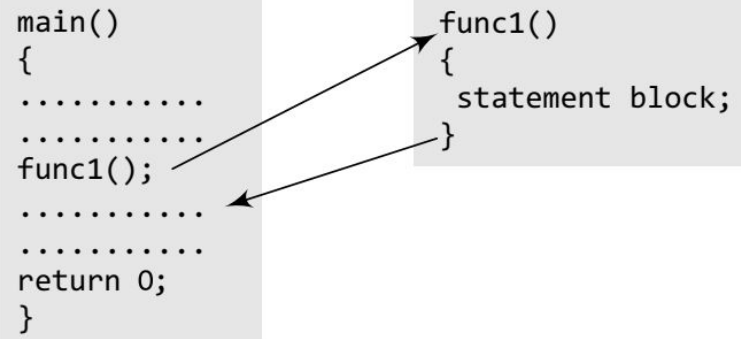


C++

Functions-Pointers-Reference-Array

3. Functions



- A function f that uses another function g is known as the *calling function*, and g is known as the *called function*.
- The inputs that a function takes are known as *arguments*.
- When a called function returns some result back to the calling function, it is said to *return* that result.
- The calling function may or may not pass *parameters* to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- *Function declaration* is a declaration statement that identifies a function's name, a list of arguments that it accepts, and the type of data it returns.
- *Function definition* consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

3. Functions-Declaration

```
main()
{
    .....
    func1();
    .....
    return 0;
}

func1()
{
    statement block;
}
```

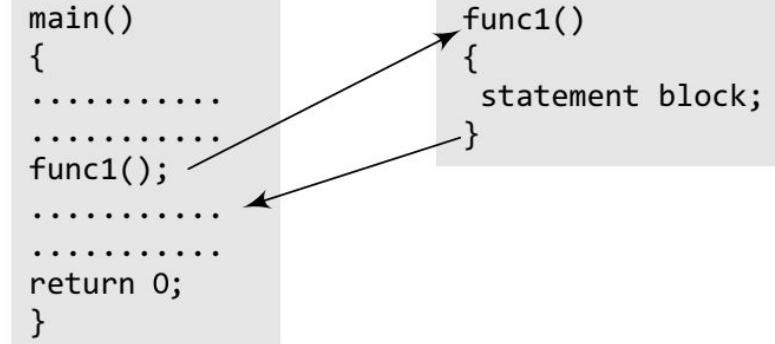
`return_data_type function_name(data_type variable1, data_type variable2,..);`

Here, **function_name** is a valid name for the function. Naming a function follows the same rules that are followed while naming variables. A function should have a meaningful name that must specify the task that the function will perform.

return_data_type specifies the data type of the value that will be returned to the calling function as a result of the processing performed by the called function.

(data_type variable1, data_type variable2, ...) is a list of variables of specified data types. These variables are passed from the calling function to the called function. They are also known as arguments or parameters that the called function accepts to perform its task.

3. Function-Definition



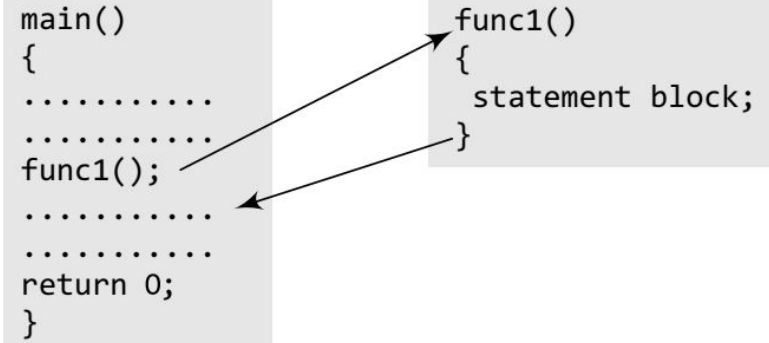
The diagram illustrates the relationship between a function call and its definition. On the left, a code block for `main()` contains a call to `func1();`. On the right, a code block for `func1()` shows its definition with a `{` opening brace, a `statement block;`, and a `}` closing brace. Two arrows originate from the `func1();` line in the `main()` block: one points to the opening brace of the `func1()` definition, and the other points to the `statement block;` line within the definition.

```
main()
{
    .....
    .....
    func1();
    .....
    .....
    return 0;
}
```

```
func1()
{
    statement block;
}
```

```
return_data_type function_name(data_type variable1, data_type variable2,..)
{
    .....
    statements
    .....
    return(variable);
}
```

3. Function-Calling



```
main()
{
    .....
    .....
    func1();
    .....
    .....
    return 0;
}
```

```
func1()
{
    statement block;
}
```

The diagram illustrates the flow of control during a function call. An arrow points from the `func1();` line in the `main()` function to the `func1()` function definition. A second arrow points from the closing curly brace of the `func1()` definition back to the line immediately following `func1();` in the `main()` function, indicating the return of control.

```
variable_name = function_name(variable1, variable2, ...);
```

3. Function-Problem

- **Write a program to add two integers using function**
- **Write a C++ program to swap two numbers using function**

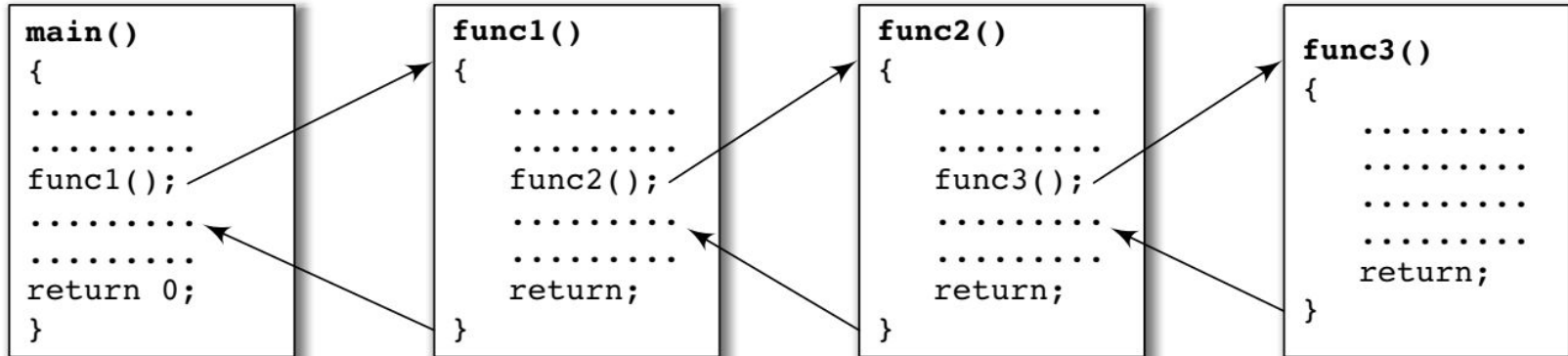
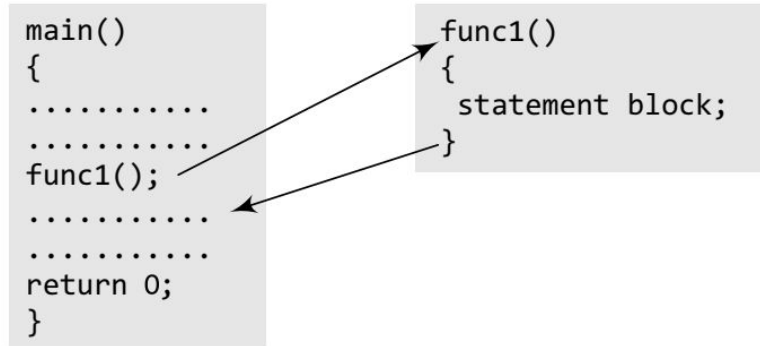
3. Function-Problem

```
#include<iostream.h>
int sum(int a, int b);    // FUNCTION DECLARATION
int main()
{   int num1, num2, total = 0;
    cout<<"\n Enter two numbers : ";
    cin>>num1>>num2;
    total = sum(num1, num2);    // FUNCTION CALL
    cout<<"\n Total = "<<total;
}
// FUNCTION DEFINITION
int sum ( int a, int b)    // FUNCTION HEADER
{   // FUNCTION BODY
    int result;
    result = a + b;
    return result;
}
```

OUTPUT

```
Enter the two numbers : 20    30
Total = 50
```





3. Functions - Recursive Call



3. Function-Problem

- **Write a recursive function to obtain the running sum of first 25 natural numbers.**

3. Function-Declaration

Function declaration	Use of the function
<pre>char convert_to_uppercase (char ch);</pre> <p>Return Data Type</p>	Converts a character to upper case. The function receives a character as an argument, converts it into upper case, and returns the converted character back to the calling program.
<pre>float avg (int a, int b);</pre> <p>Function Name</p>	Calculates average of two numbers a and b received as arguments. The function returns a floating point value.
<pre>int find_largest (int a, int b, int c);</pre> <p>Data Type of Variable</p>	Finds the largest of three numbers a, b, and c received as arguments. An integer value which is the largest number of the three numbers is returned to the calling function.
<pre>double multiply(float a, float b);</pre> <p>Variable I</p>	Multiplies two floating point numbers a and b that are received as arguments and returns a double value.
<pre>void swap (int a, int b);</pre>	Swaps or interchanges the value of integer variables a and b received as arguments. The function returns no value; therefore, the data type is void.
<pre>void print(void);</pre>	The function is used to print information on screen. The function neither accepts any value as argument nor returns any value. Therefore, the return type is void and the argument list contains void.

3. Function-Problem

The diagram illustrates the relationship between a main function and a sub-function. On the left, the `main()` function is shown with a call to `func1();`. On the right, the `func1()` function is shown with its internal `statement block;`. Two arrows indicate the flow of control: one arrow points from the `func1();` line in `main()` to the `func1()` definition, and another arrow points from the `statement block;` in `func1()` back to the line following `func1();` in `main()`.

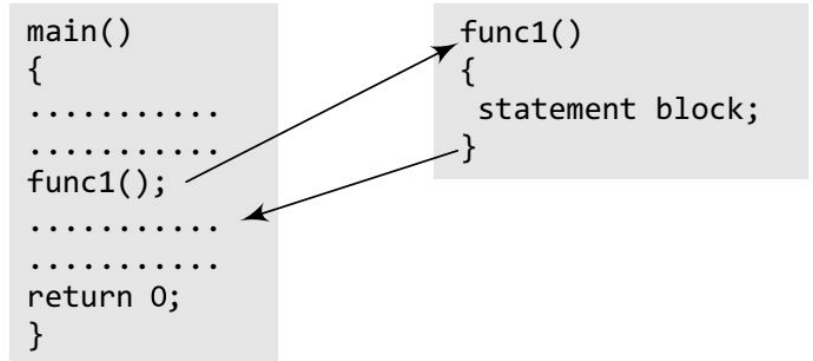
```
main()
{
    .....
    .....
    func1();
    .....
    .....
    return 0;
}

func1()
{
    statement block;
}
```

Write a program to find whether a number is even or odd using functions.

3. Function-Passing Parameters

- **Call by Value**
- **Call by References**
- **Call by Address**



3. Function-Passing Parameters

- **Call by Value**
- **Call by References**
- **Call by Address**

When a function is called, the calling function may have to pass some values to the called function. There are three different ways in which arguments or parameters can be passed to the called function.

They include the following:

- **Call-by-value** : in which values of the variables are passed by the calling function to the called function. The programs that we have written so far call the function using call-by-value method of passing parameters.
- **Call-by-address** : in which the address of the variables is passed by the calling function to the called function.
- **Call-by-reference** : in which a reference of the variable is passed by the calling function to the called function.

3. Function-Passing Parameters

- **Call by Value**
- **Call by References**
- **Call by Address**

In this method, the called function creates new variables to store the value of the arguments passed to it.

Therefore, the called function uses a copy of the actual arguments to perform its intended task.

If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function.

In the calling function, no change will be made to the value of the variables.

3. Function-Problem

```
main()
{
    .....
    .....
    func1();
    .....
    .....
    return 0;
}

func1()
{
    statement block;
}
```

Write a function which can swap two variable using call by value.

3. Function-Passing Parameters

```
#include<iostream.h>
void add(int n); // FUNCTION DECLARATION
int main()
{   int num = 2;
    cout<<"\n The value of num before calling the function = "<<num;
    add(num); // FUNCTION CALL
    cout<<"\n The value of num after calling the function = "<<num;
}
void add(int n) // FUNCTION DEFINITION
{   n = n + 10;
    cout<<"\n The value of num in the called function = "<<n;
}
```

OUTPUT

The value of num before calling the function = 2

The value of num in the called function = 20

The value of num after calling the function = 2

3. Function-Passing Parameters

- Call by Value
- Call by References
- **Call by Address**

The call-by-address method of passing arguments to a function copies the address of an argument into the formal parameter.

The function then uses the address to access the actual argument. This means that any changes made to the value stored at a particular address will be reflected in the calling function also.

The called function uses pointers to access the data. In other words, the dereferencing operator is used to access the variables in the called function.

3. Function-Problem

```
main()
{
    .....
    .....
    func1();
    .....
    .....
    return 0;
}

func1()
{
    statement block;
}
```

Write a function which can swap two variable using call by address.

3. Function-Passing Parameters

```
#include<iostream.h>
void add( int *n);
int main()
{   int num = 2;
    cout<<"\n The value of num before calling the function = "<<num;
    add(&num);
    cout<<"\n The value of num after calling the function = "<<num;
}
void add( int *n)
{   *n = *n + 10;
    cout<<"\n The value of num in the called function = "<<n;
}
```

OUTPUT

```
The value of num before calling the function = 2
The value of num in the called function = 12
The value of num after calling the function = 12
```

3. Function-Passing Parameters

- Call by Value
- **Call by References**
- Call by Address

When the calling function passes arguments to the called function using call-by-value method, the only way to return the modified value of the argument to the caller is by using the return statement explicitly.

A better option when a function can modify the value of the argument is to pass arguments using call-by-reference technique.

In call-by-reference, we declare the function parameters as references rather than normal variables.

When this is done, any changes made by the function to the arguments it received are visible by the calling program.

3. Function-Problem

```
main()
{
    .....
    .....
    func1();
    .....
    .....
    return 0;
}

func1()
{
    statement block;
}
```

Write a function which can swap two variable using call by reference.

3. Function-Passing Parameters

```
#include<iostream.h>
void add( int &n);
int main()
{   int num = 2;
    cout<<"\n The value of num before calling the function = "<<num;
    add(&num);
    cout<<"\n The value of num after calling the function = "<<num;
}
void add(int &n)
{   n = n + 10;
    cout<<"\n The value of num in the called function = "<<n;
}
```

OUTPUT

```
The value of num before calling the function = 2
The value of num in the called function = 12
The value of num after calling the function = 12
```

4. Pointers in C++

- **Every computer has a primary memory. All our data and programs need to be placed in the primary memory for execution.**
- **The primary memory or RAM (Random Access Memory which is a part of the primary memory) is a collection of memory locations (often known as cells) and each location has a specific address. Each memory location is capable of storing 1 byte of data.**
- **Generally, the computer has three areas of memory each of which is used for a specific task. These areas of memory include- stack, heap and global memory.**

4. Pointers in C++

- **Stack-** A fixed size of stack is allocated by the system and is filled as needed from the bottom to the top, one element at a time. These elements can be removed from the top to the bottom by removing one element at a time. That is, the last element added to the stack is removed first.
- **Heap-** Heap is a contiguous block of memory that is available for use by the program when need arise. A fixed size heap is allocated by the system and is used by the system in a random fashion.
- When the program requests a block of memory, the dynamic allocation technique carves out a block from the heap and assigns it to the program.
- When the program has finished using that block, it returns that memory block to the heap and the location of the memory locations in that block is added to the free list.

4. Pointers in C++

- **Global Memory-** The block of code that is the `main()` program (along with other functions in the program) is stored in the global memory. The memory in the global area is allocated randomly to store the code of different functions in the program in such a way that one function is not contiguous to another function. Besides, the function code, all global variables declared in the program are stored in the global memory area.
- **Other Memory Layouts-** C provides some more memory areas like- text segment, BSS and shared library segment.
- **The text segment** is used to store the machine instructions corresponding to the compiled program. This is generally a read-only memory segment.
- **BSS** is used to store un-initialized global variables shared libraries segment contains the executable image of shared libraries that are being used by the program.

4. Pointers in C++

```
int x = 10;  
int *ptr;  
ptr = &x;
```

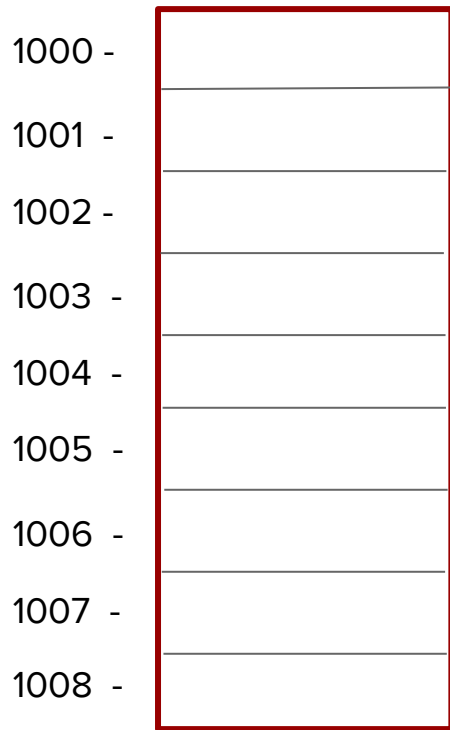


4. Pointers in C++

```
int x=10;
```

```
int *x;
```

```
int x = 10;  
int *ptr;  
ptr = &x;
```



4. Pointers in C++

- **Declaration of Pointers**
- **Pointer Mathematics**
- **NULL Pointers**
- **Generic Pointers**
- **Pointer to Pointer**
- **Drawbacks of Pointers**

4. Pointers in C++ - Declaration of Pointers

```
data_type *ptr_name;
```

```
int *pnum;
```

```
char *pch;
```

```
float *pfnum;
```

Prob: Write a program which can display the size of the pointer variables.

```
1  # include<iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      int      *pnum;
8      char     *pch;
9      float    *pfnum;
10     double   *pdnum;
11     long     *plnum;
12
13     cout<< "\n The size of the integer pointer is : " <<sizeof(pnum);
14     cout<< "\n The size of the characte pointer is : " <<sizeof(pch);
15     cout<< "\n The size of the float pointer is : " <<sizeof(pfnum);
16     cout<< "\n The size of the double pointer is : " <<sizeof(pdnum);
17     cout<< "\n The size of the long pointer is : " <<sizeof(plnum);
18
19     return 0;
20 }
```

4. Pointers in C++ - Declaration of Pointers

```
1  # include<iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      int num, *pnum;
8      pnum = &num;
9      cout<<"\n Enter the number : ";
10     cin>>num;
11     cout<<"\n The entered number is : "<<num;
12     return 0;
13 }
```

4. Pointers in C++ - Pointer Mathematics

```
1  #include<iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      int num1 = 1, num2 = 3, sum = 0, mul = 0, div = 1;
8      int *ptr1, *ptr2;
9      ptr1 = &num1;
10     ptr2 = &num2;
11     sum = *ptr1 + *ptr2;
12     mul = sum * (*ptr1);
13     *ptr2 += 1;
14     div = 9+ (*ptr1)/(*ptr2) - 30;
15
16     return 0;
17 }
```


4. Pointers-Problem

- **Write a program which can display the size of the pointer variables.**
- **Write a program which can display the values and address of a pointer Variable.**
- **Write a program to add two floating point numbers using pointer.**
- **Write a program to find out the biggest of three numbers using pointer.**

4. Pointers in C++ - NULL Pointers

- A null pointer which is a special pointer value that is known not to point anywhere. This means that a NULL pointer does not point to any valid memory address.

To declare a null pointer you may use the predefined constant NULL,

```
int *ptr = NULL;
```

- We can always check whether a given pointer variable stores address of some variable or contains a null by writing,

```
if ( ptr == NULL)
```

```
{
```

```
    Statement block;
```

```
}
```

- Null pointers are used in situations if one of the pointers in the program points somewhere some of the time but not all of the time. In such situations it is always better to set it to a null pointer when it doesn't point anywhere valid, and to test to see if it's a null pointer before using it.

4. Pointers in C++ - NULL Pointers

```
int *ptr = NULL;
```

```
int *ptr,  
ptr = 0;
```

```
if (ptr == NULL)  
{  
    Statement block;  
}
```

4. Pointers in C++ - Generic Pointers

- A generic pointer is pointer variable that has void as its data type. The generic pointer can be pointed at variables of any data type.

It is declared by writing

```
void *ptr;
```

- We need to cast a void pointer to another kind of pointer before using.
- Generic pointers are used when a pointer has to point to data of different types at different times.

4. Pointers in C++ - Generic Pointers

```
void *ptr;
```

4. Pointers in C++ - Generic Pointers

```
1  # include<iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      int x=10;
8      char ch = 'A';
9      void *gp;
10     gp = &x;
11     cout<<"\n Generic pointer points to the integer value : "<< *(int*)gp;
12     gp = &ch;
13     cout<<"\n Generic pointer points to the integer value : "<< *(char*)gp;
14     return 0;
15 }
```

Problem

- **Write a program to display an array of given numbers from 1-9 using pointer.**
- **Write a program to add two integers using function.**

4. Pointers in C++ - Pointers & Array

```
#include<iostream.h>
main()
{
    int arr[]={1,2,3,4,5,6,7,8,9};
    int *ptr1, *ptr2;
    ptr1 = arr; // ptr1 pointing to the first element of the array
    ptr2 = &arr[8]; // ptr2 pointing to the last element of the array
    while(ptr1<=ptr2)
    {
        cout<<" "<<*ptr1;
        ptr1++;
    }
}
```

OUTPUT

1 2 3 4 5 6 7 8 9

Problem

- **Write a program to display an array to read and display using pointer.**
- **Write a program to find out the mean using array & pointer.**

4. Pointers in C++ - Array of Pointers

- An array of pointers can be declared as `int *ptr[10]`
- The above statement declares an array of 10 pointers where each of the pointer points to an integer variable. For example, look at the code given below.

```
int *ptr[10];  
int p=1, q=2, r=3, s=4, t=5;  
ptr[0]=&p;  
ptr[1]=&q;  
ptr[2]=&r;  
ptr[3]=&s;  
ptr[4]=&t
```

What will be the output of the following statement?

```
cout<<"\n"<< *ptr[3];
```

4. Pointers in C++ - Array of Pointers

- An array of pointers can be declared as `int *ptr[10]`
- The above statement declares an array of 10 pointers where each of the pointer points to an integer variable. For example, look at the code given below.

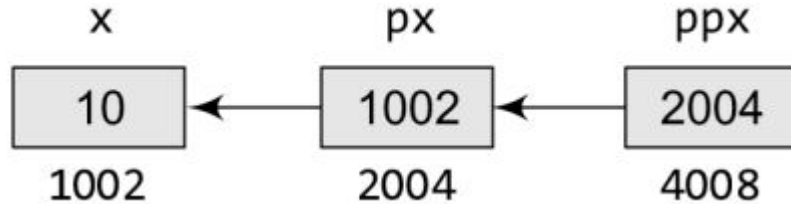
```
int *ptr[10];  
int p=1, q=2, r=3, s=4, t=5;  
ptr[0]=&p;  
ptr[1]=&q;  
ptr[2]=&r;  
ptr[3]=&s;  
ptr[4]=&t
```

What will be the output of the following statement?

```
cout<<"\n"<< *ptr[3];
```

Yes, the output will be 4 because `ptr[3]` stores the address of integer variable `s` and `*ptr[3]` will therefore print the value of `s` that is 4.

4. Pointers in C++ - Pointer to Pointer



```
int x=10;  
int *px, **ppx;  
px = &x;  
ppx = &px;
```

```
cout<< **ppx;
```

4. Pointers in C++ - Drawbacks of Pointers

Problem

- **Write a program to display the word “Your Name” using pointer and also count the number of characters.**
- **Write a program to add two integers using function.**

4. Reference Variable in C++

Reference variable is an alternate name of already existing variable. It cannot be changed to refer another variable and should be initialized at the time of declaration and cannot be NULL. The operator ‘&’ is used to declare reference variable.

Syntax :

```
datatype variable_name; // variable declaration
```

```
datatype& refer_var = variable_name; // reference variable
```

- **datatype** - The datatype of variable like int, char, float etc.
- **variable_name** - This is the name of variable given by user.
- **refer_var** - The name of reference variable.

4. Reference Variable in C++

```
#include <iostream>
using namespace std;
int main() {
    int a = 8;
    int& b = a;
    cout << " The variable a : " << a;
    cout << "\n The reference variable r : " << b;
    return 0;
}
```