CHAPTER-2 part-3

- **Variables:** Scope & lifetime of variables, variable declaration at the point of use, Ordinary Variable Vs. Pointer Variable Vs. Reference Variable (variable aliases)

Function: Parameter passing by value Vs. by address Vs. by reference, inline function, function overloading, default arguments.

Scope of Variables in C++

In general, scope is defined as the extent upto which something can be worked with. In programming also scope of a variable is defined as the extent of the program code within which the variable can be accessed or declared or worked with. There are mainly two types of variable scopes as discussed below:

Local Variables

Variables defined within a function or block are said to be local to those functions.

Anything between ‘{‘ and ‘}’ is said to inside a block.

Local variables do not exist outside the block in which they are declared, i.e. they can not be accessed or used outside that block.

Declaring local variables: Local variables are declared inside a block.

```
// CPP program to illustrate
// usage of local variables
#include<iostream>
using namespace std;

void func()
{
    // this variable is local to the function func() and cannot be accessed outside
//this function
    int age=18;
}
 int main()
{
    cout<<"Age is: "<<age;

    return 0;
}
```

Output:

Error: age was not declared in this scope

The above program displays an error saying “age was not declared in this scope”. The variable age was declared within the function func() so it is local to that function and not visible to portion of program outside this function.

Rectified Program : To correct the above error we have to display the value of variable age from the function func() only. This is shown in the below program:

```
// CPP program to illustrate usage of local variables
#include<iostream>
using namespace std;
 void func()
{
    // this variable is local to the function func() and cannot be
    // accessed outside this function
    int age=18;
    cout<<age;
}
int main()
{    cout<<"Age is: ";
    func();
    return 0; }
```

output:

Age is: 18

Global Variables

As the name suggests, Global Variables can be accessed from any part of the program.

They are available through out the life time of a program.

They are declared at the top of the program outside all of the functions or blocks.

global variables: Declaring Global variables are usually declared outside of all of the functions and blocks, at the top of the program. They can be accessed from any portion of the program.

```
// CPP program to illustrate usage of global variables
#include<iostream>
using namespace std;
 // global variable
int global = 5;
 // global variable accessed from within a function
void display()
{
    cout<<global<<endl;
}
```

```
 // main function

int main()
{
    display();
    // changing value of global
    // variable from main function
    global = 10;
    display();
}

Output:
5
10
```

In the program, the variable “global” is declared at the top of the program outside all of the functions so it is a global variable and can be accessed or updated from anywhere in the program.

What if there exists a local variable with the same name as that of global variable inside a function?

```
// CPP program to illustrate scope of local variables
// and global variables together
#include<iostream>
using namespace std;
 // global variable
int global = 5;
 // main function
int main()
{
    // local variable with same
    // name as that of global variable

    int global = 2;
    cout << global << endl;
}
```

Look at the above program. The variable “global” declared at the top is global and stores the value 5 where as that declared within main function is local and stores a value 2. So, the question is when the value stored in the variable named “global” is printed from the main function then what will be the output? 2 or 5?

Usually when two variable with same name are defined then the compiler produces a compile time error. But if the variables are defined in different scopes then the compiler allows it.

Whenever there is a local variable defined with same name as that of a global variable then the compiler will give precedence to the local variable

Here in the above program also, the local variable named “global” is given precedence. So the output is 2.

How to access a global variable when there is a local variable with same name?

What if we want to do the opposite of above task. What if we want to access global variable when there is a local variable with same name?

To solve this problem we will need to use the scope resolution operator. Below program explains how to do this with the help of scope resolution operator.

/

```
/ C++ program to show that we can access a global
// variable using scope resolution operator :: when
// there is a local variable with same name
#include<iostream>
using namespace std;
  // Global x
int x = 0;
 int main()
{
  // Local x
  int x = 10;
  cout << "Value of global x is " << ::x;
  cout<< "\nValue of local x is " << x;
 return 0;
 }
```

Output:

```
Value of global x is 0
Value of local x is 10
```

Reference Variables

A reference is an alias, or an alternate name to an existing variable.

For example, suppose you make peter a reference (alias) to paul, you can refer to the person as either peter or paul.

The main use of references is acting as function formal parameters to support pass-by-reference. If an reference variable is passed into a function, the function works on the original copy (instead of a clone copy in pass-by-value). Changes inside the function are reflected outside the function.

A reference is similar to a pointer. In many cases, a reference can be used as an alternative to pointer, in particular, for the function parameter.

References (or Aliases) (&)

Recall that C/C++ use & to denote the address-of operator in an expression. C++ assigns an additional meaning to & in declaration to declare a reference variable.

The meaning of symbol & is different in an expression and in a declaration. When it is used in an expression, & denotes the address-of operator, which returns the address of a variable, e.g., if number is an int variable, &number returns the address of the variable number.

However, when & is used in a declaration (including function formal parameters), it is part of the type identifier and is used to declare a reference variable (or reference or alias or alternate name). It is used to provide another name, or another reference, or alias to an existing variable.

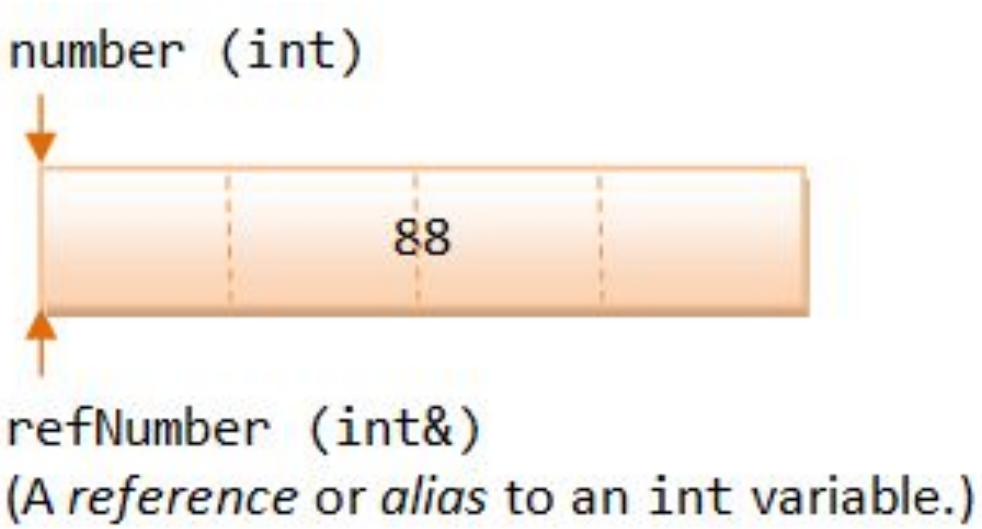
syntax:

```
type &newName = existingName;
// or
type& newName = existingName;
// or
type & newName = existingName;  // I shall adopt this convention
```

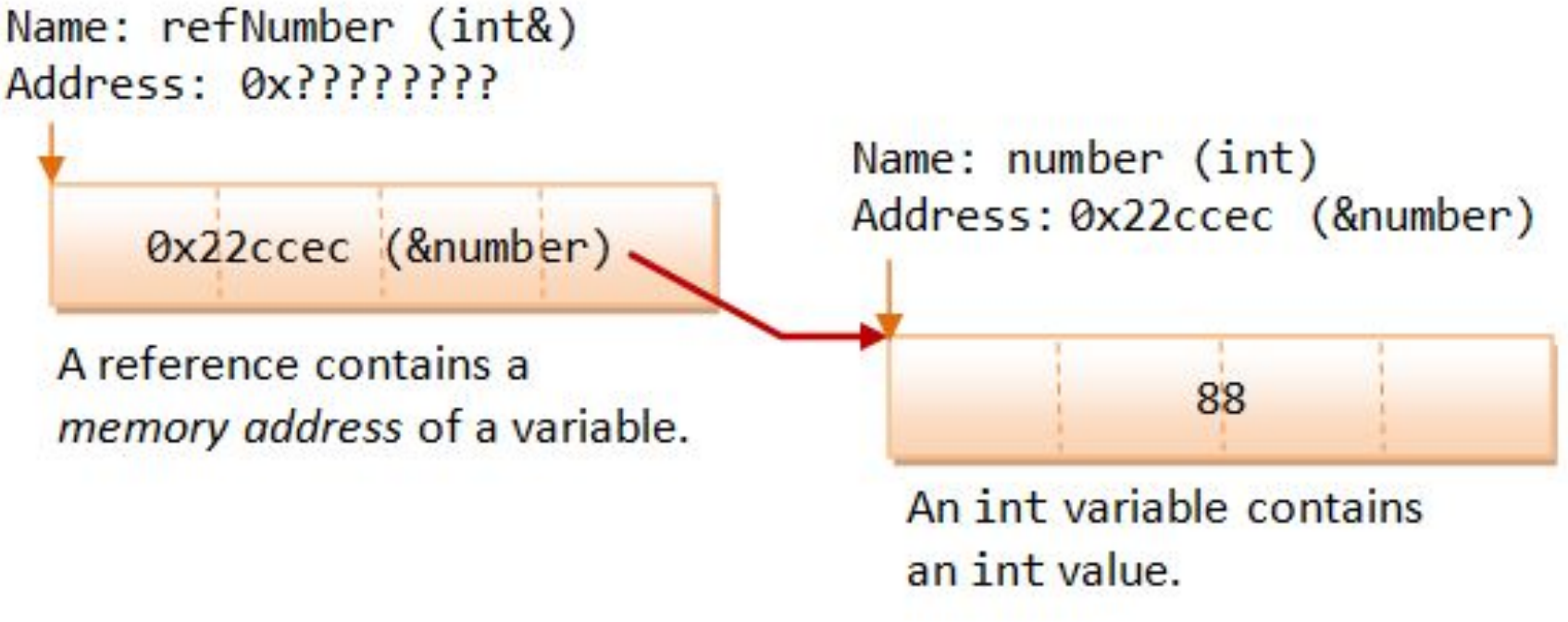
It shall be read as "newName is a reference to exisitngName", or "newName is an alias of existingName". You can now refer to the variable as newName or existingName.

Example:

```
#include <iostream>
using namespace std;

int main() {
   int number = 88;          // Declare an int variable called number
   int  &refNumber = number; // Declare a reference (alias) to the variable number
                             // Both refNumber and number refer to the same value

   cout << number << endl;    // Print value of variable number (88)
   cout << refNumber << endl; // Print value of reference (88)

   refNumber = 99;            // Re-assign a new value to refNumber
   cout << refNumber << endl;
   cout << number << endl;    // Value of number also changes (99)

   number = 55;               // Re-assign a new value to number
   cout << number << endl;
   cout << refNumber << endl; // Value of refNumber also changes (55)
}
```



A reference works as a pointer. A reference is declared as an alias of a variable. It stores the address of the variable, as illustrated:



What are the differences between a pointer variable and a reference variable in C++?

1. A pointer can be re-assigned:

```
int x = 5;
int y = 6;
int *p;
p =  &x;
p = &y;
*p = 10;
assert(x == 5);
assert(y == 10);
```

2. A reference cannot, and must be assigned at initialization:

```
int x = 5;
int y = 6;
int &r = x;
```

3. You can have pointers to pointers to pointers offering extra levels of indirection. Whereas references only offer one level of indirection.

```
int x = 0;
int y = 0;
int *p = &x;
int *q = &y;
int **pp = &p;
pp = &q;//*pp = q
**pp = 4;
assert(y == 4);
assert(x == 0);
```

4. Pointer can be assigned nullptr directly, whereas reference cannot. If you try hard enough, and you know how, you can make the address of a reference nullptr. Likewise, if you try hard enough you can have a reference to a pointer, and then that reference can contain nullptr.

```
int *p = nullptr;
int &r = nullptr; <--- compiling error
```

5. Pointers can iterate over an array, you can use ++ to go to the next item that a pointer is pointing to, and + 4 to go to the 5th element. This is no matter what size the object is that the pointer points to.

6. A pointer needs to be dereferenced with * to access the memory location it points to, whereas a reference can be used directly.

7. A pointer is a variable that holds a memory address. Regardless of how a reference is implemented, a reference has the same memory address as the item it references.

8. References cannot be stuffed into an array, whereas pointers can be.

Pass-By-Reference into Functions with Reference Arguments vs. Pointer Arguments vs Pass-by-Value

In C/C++, by default, arguments are passed into functions by value (except arrays which is treated as pointers). That is, a clone copy of the argument is made and passed into the function. Changes to the clone copy inside the function has no effect to the original argument in the caller. In other words, the called function has no access to the variables in the caller.

```
/* Pass-by-value into function (TestPassByValue.cpp) */
#include <iostream>
using namespace std;
 int square(int);
 int main() {
   int number = 8;
   cout <<  "In main(): " << &number << endl;  // 0x22ff1c
   cout << number << endl;         // 8
   cout << square(number) << endl; // 64
   cout << number << endl;         // 8 - no change
}
 int square(int n) {  // non-const
   cout <<  "In square(): " << &n << endl;  // 0x22ff00
   n *= n;           // clone modified inside the function
   return n;
}
```

The output clearly shows that there are two different addresses.

```
/* Pass-by-reference using pointer (TestPassByPointer.cpp) */
#include <iostream>
using namespace std;
 void square(int *);
 int main() {
   int number = 8;
   cout <<  "In main(): " << &number << endl;  // 0x22ff1c
   cout << number << endl;   // 8
   square(&number);          // Explicit referencing to pass an address
   cout << number << endl;   // 64
}

void square(int * pNumber) {  // Function takes an int pointer (non-const)
   cout <<  "In square(): " << pNumber << endl;  // 0x22ff1c
   *pNumber *= *pNumber;      // Explicit de-referencing to get the value pointed-to
}
```

The called function operates on the same address, and can thus modify the variable in the caller.

Pass-by-Reference with Reference Arguments

Instead of passing pointers into function, you could also pass references into function, to avoid the clumsy syntax of referencing and dereferencing.

```
/* Pass-by-reference using reference */
#include <iostream>
using namespace std;
void square(int &);
int main() {
   int number = 8;
   cout <<  "In main(): " << &number << endl;  // 0x22ff1c
   cout << number << endl;  // 8
   square(number);          // Implicit referencing (without '&')
   cout << number << endl;  // 64
}
void square(int & rNumber) {  // Function takes an int reference (non-const)
   cout <<  "In square(): " << &rNumber << endl;  // 0x22ff1c
   rNumber *= rNumber;        // Implicit de-referencing (without '*')
}
```

Again, the output shows that the called function operates on the same address, and can thus modify the caller's variable.

Take note referencing (in the caller) and dereferencing (in the function) are done implicitly. The only coding difference with pass-by-value is in the function's parameter declaration.

Recall that references are to be initialized during declaration. In the case of function formal parameter, the references are initialized when the function is invoked, to the caller's arguments.

References are primarily used in passing reference in/out of functions to allow the called function accesses variables in the caller directly.

Inline Functions in C++

Inline function is one of the important feature of C++. So, let's first understand why inline functions are used and what is the purpose of inline function?

1. When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee).
2. For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
    // function code
}
```

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

1) If a function contains a loop. (for, while, do-while)

2) If a function contains static variables.

3) If a function is recursive.

4) If a function return type is other than void, and the return statement doesn't exist in function body.

5) If a function contains switch or goto statement.

Inline functions provide following advantages:

1) Function call overhead doesn’t occur.

2) It also saves the overhead of push/pop variables on the stack when function is called.

3) It also saves overhead of a return call from a function.

4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.

5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Inline function disadvantages:

1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization.

This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

The following program demonstrates the use of use of inline function.

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
    return s*s*s;
}
int main()
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
} //Output: The cube of 3 is: 27
```

Inline function and classes:

It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.

For example:

```
class S
{
public:
    inline int square(int s) // redundant use of inline
    {
        // this function is automatically inline
        // function body
    }
};
```

The above style is considered as a bad programming style. The best programming style is to just write the prototype of function inside the class and specify it as an inline in the function definition.

For example:

```
class S{
public:
    int square(int s); // declare the function
};

inline int S::square(int s) // use inline prefix
{

}
```

The following program demonstrates this concept:

```
#include <iostream>
using namespace std;
class operation
{
    int a,b,add,sub,mul;
    float div;
public:
    void get();
    void sum();
    void difference();
    void product();
    void division();
};
```

```
inline void operation :: get()
{
    cout << "Enter first value:";
    cin >> a;
    cout << "Enter second value:";
    cin >> b;
}

inline void operation :: sum()
{
    add = a+b;
    cout << "Addition of two numbers: " << a+b << "\n";
}
 inline void operation :: difference()
{
    sub = a-b;
    cout << "Difference of two numbers: " << a-b << "\n";
}
```

```
inline void operation :: product()
{
    mul = a*b;
    cout << "Product of two numbers: " << a*b << "\n";
}

inline void operation ::division()
{
    div=a/b;
    cout<<"Division of two numbers: "<<a/b<<"\n" ;
}
```

```
int main()
{
    cout << "Program using inline function\n";
    operation s;
    s.get();
    s.sum();
    s.difference();
    s.product();
    s.division();
    return 0;
}
```

Output:

```
Enter first value: 45
Enter second value: 15
Addition of two numbers: 60
Difference of two numbers: 30
Product of two numbers: 675
Division of two numbers: 3
```

Function Overloading in C++

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.

Function overloading can be considered as an example of polymorphism feature in C++.

Following is a simple C++ example to demonstrate function overloading.

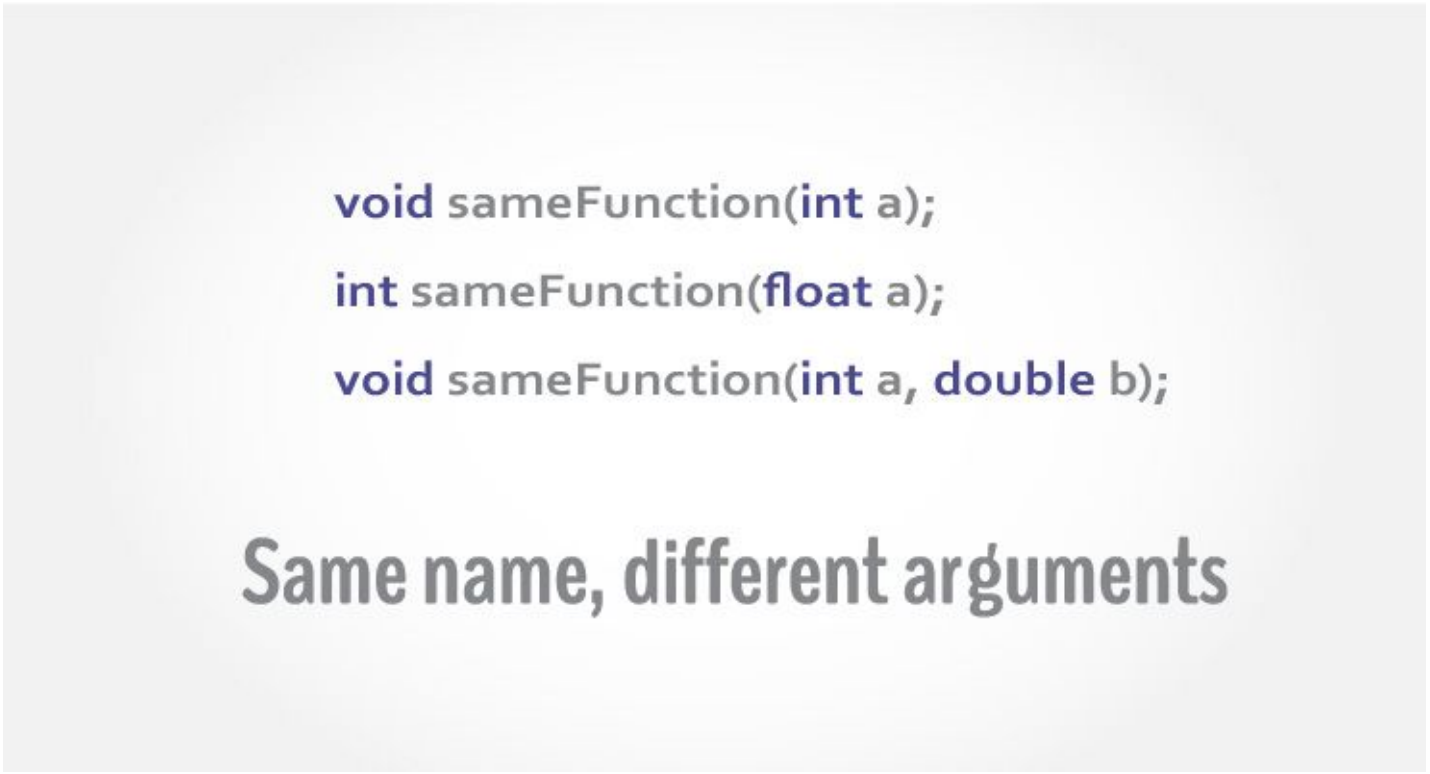
```
#include <iostream>
using namespace std;

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
```

```
void print(char* c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Output:

```
Here is int 10
Here is float 10.1
Here is char* ten
```

Two or more functions having same name but different argument(s) are known as overloaded functions



Example 2: Function Overloading

```
#include <iostream>
using namespace std;

void display(int);
void display(float);
void display(int, float);

int main() {

    int a = 5;
    float b = 5.5;

    display(a);
    display(b);
    display(a, b);
    return 0;
}
```

```
void display(int var) {
    cout << "Integer number: " << var << endl;
}

void display(float var) {
    cout << "Float number: " << var << endl;
}
void display(int var1, float var2) {
    cout << "Integer number: " << var1;
    cout << " and float number:" << var2;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Here, the display() function is called three times with different type or number of arguments.

The return type of all these functions are same but it's not necessary.

Default Arguments in C++

A default argument is a value provided in function declaration that is automatically assigned by the compiler if caller of the function doesn't provide a value for the argument with default value.

Following is a simple C++ example to demonstrate use of default arguments. We don't have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.

```
#include<iostream>
using namespace std;

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
   return (x + y + z + w);
}
```

```
int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```

Output:

25

50

80

Once default value is used for an argument, all subsequent arguments must have default value.

```
// Invalid because z has default value, but w after it
// doesn't have default value
int sum(int x, int y, int z=0, int w)
```

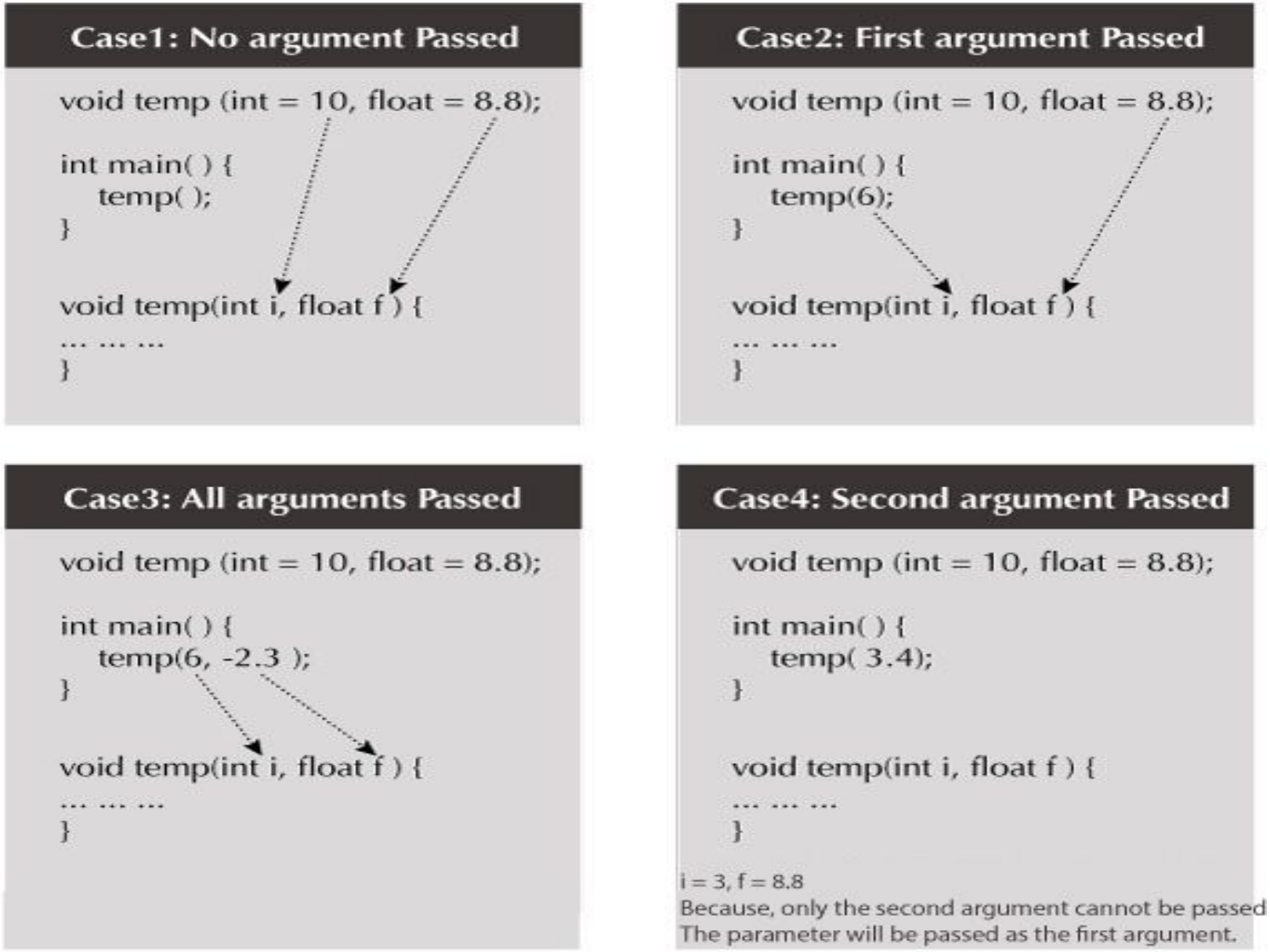


Figure: Working of Default Argument in C++

```
#include <iostream>
using namespace std;
int sum(int a, int b=10, int c=20);

int main(){
   /* In this case a value is passed as 1 and b and c values are taken from
    default arguments. */
   cout<<sum(1)<<endl;

   /* In this case a value is passed as 1 and b value as 2, value of c values is
     taken from default arguments.*/
cout<<sum(1, 2)<<endl;
   /* In this case all the three values are passed during function call, hence no
     default arguments have been used. */
   cout<<sum(1, 2, 3)<<endl;
   return 0;
}
```

```
int sum(int a, int b, int c){
   int z;
   z = a+b+c;
   return z;
}
```

Output:

31

23

6

Rules of default arguments

As you have seen in the above example that I have assigned the default values for only two arguments b and c during function declaration. It is up to you to assign default values to all arguments or only selected arguments but remember the following rule while assigning default values to only some of the arguments:

If you assign default value to an argument, the subsequent arguments must have default values assigned to them, else you will get compilation error.

For example: Lets see some valid and invalid cases.

Valid: Following function declarations are valid –

```
int sum(int a=10, int b=20, int c=30);
int sum(int a, int b=20, int c=30);
int sum(int a, int b, int c=30);
```

```
// C++ Program to demonstrate working of default argument

#include <iostream>
using namespace std;
void display(char = '*', int = 1);

int main()
{
    cout << "No argument passed:\n";
    display();

    cout << "\nFirst argument passed:\n";
    display('#');

    cout << "\nBoth argument passed:\n";
    display('$', 5);
    return 0;
}
```

```
void display(char c, int n)
{
    for(int i = 1; i <= n; ++i)
    {
        cout << c;
    }
    cout << endl;
}
```

Output

No argument passed:

*

First argument passed:

#

Both argument passed:

$$$$

Invalid: Following function declarations are invalid –

/* Since a has default value assigned, all the arguments after a (in this case b and c) must have default values assigned */

```
int sum(int a=10, int b, int c=30);
```

/* Since b has default value assigned, all the arguments after b (in this case c) must have default values assigned */

```
int sum(int a, int b=20, int c);
```

/* Since a has default value assigned, all the arguments after a (in this case b and c) must have default values assigned, b has default value but c doesn't have, thats why this is also invalid */

```
int sum(int a=10, int b=20, int c);
```