

C++ Programming

Trainer : Rajiv K

Email: rajiv.kamune@sunbeaminfo.com



Agenda

- Namespace
- scope resolution operator ::
- Using directive
- cin , cout
- escape sequence and manipulators
- Macro
- Inline function
- Default Argument
- Function Overloading



Namespace

- Namespaces are used to **organize code** into **logical groups** and to prevent **name collisions** that can occur specially when your code base includes multiple libraries.
- C language **does not** have a **namespace mechanism**.
- Namespace provide the space where we can define or **declare identifier** i.e. **variable, method, classes**.
- We **cannot create instance** of namespace.
- To understand this in better way just take an example
- Assume we are having 2 file with same name inside same folder OS will not allow this so I have only 2 option either I will change the name of one of the file or copy the other file inside some other folder.

- **Syntax:**

```
namespace namespace_name
{ /* you may have following identifiers in side name space
   variables
   functions
   Classes*/
}
```



Namespace

MyFile.cpp

```
int main()
{
    int num;
    int num;
}
```

//This declaration is Not allowed as
//num is Declared twice in same
//block scope

//This declaration of
//num
//is allowed as num //is
declared
//in different name
//spaces

MyFile.cpp

```
namespace ns1
{
    int num;
}
namespace ns2
{
    int num;
}
namespace ns3
{
    int num;
}
```



Namespace

Example:

namespace ns

```
{    // you may have following identifiers in side name space variables, functions, Classes
    class Time                //Class inside the namespace
    {
        int hr,min,sec;
        void accept()
        { //accept business logic}
        void display()
        { //display business logic}
    }
    int num;                //variable inside namespace
    void test()            // function inside namespace
    { //function body}
}
```



Namespace Example & Scope resolution operator ::

- **Scope Resolution and using key word.**
- If a function having the same name exists inside two namespace we can use the **namespace name** with the scope resolution **operator(::)** to refer that function without any conflicts.
- **Syntax:**
 - namespace_name::identifier_name
- **Example:**
 - n::num;
- To access a global variable when there is a local variable with same name.
- If name of local variable and global variable is same then **preference** is always given to the **local variable**.



Using directive

- **Using directive**

- Bring a specific member from the namespace into the current scope.
- Bring all members from the namespace into the current scope.
- Bring a base class method or variable into the current class's scope.

- **Syntax:**

using namespace <name_of_namespace>

- **Example:**

```
namespace ns
{
    /* you may have following identifiers in side name space
       variables
       functions
       Classes
    */
}

using namespace ns;           //user defined namespace
using namespace std;         //predefined namespace in side the library
```



Program Demo

1. Define one namespace 'na' having one data member num1. Define one global function display_num() access num1 using namespace na. Inside main call display_num() function using scope resolution operator(::).
2. declare one global variable and one variable in main having same name, and try to access local variable and global variable (::)
3. Declare one variable in namespace and access through main.
4. Define two namespace having different variables defined and access thru main.
5. Program to demonstrate global variable and using nested name space define global variable n1, define namespace ns1, having int n1, define namespace ns2 having int n1, int n2, define nested namespace ns3 having int n1



cin and cout

- cin is **an object** of the **input stream** and is used to take input from input streams like files, console, etc.
- cout is an **object** of the **output stream** that is used to show output. Basically, cin is an input statement while cout is an output statement.
- They also use different operators. **cin uses the extraction operator(>>)** while **cout uses the insertion operator (<<)**.

- **Syntax:**

- cin>>variable_name; `//scanf("format specifier",&variablr_name);`
 - cout<<variable_name; `//printf("format specifier",variablr_name);`

- **Example:**

- cin>>num; `//scanf("%d",&num);`
 - cout<<"Num="<<num; `//printf("Num=%d",num);`



Escape Sequence

- In C/C++, **escape sequence** is a **character** which is used to format the output.
- \b , \t , \n , \\, \v,

Escape sequence	Explanation	Name
\a	plays the beep during the program's execution.	Alarm (Beep)
\b	delete the previous character.	Backspace
\f	skip to the start of the next page.	Form Feed
\r	display the output after erasing the number of elements from the start	Carriage Return
\n	jump to the next line.	New line
\t	insert the tab(horizontal tab) in string or text.	Horizontal tab
\v	insert the vertical tab in string or text.	Vertical tab
\'	display the single quote mark in the text.	Single quote mark
\"	display the double quote mark in the text.	Double quote mark
\?	used to display the question mark in the text.	Question mark



Manipulators

- In C++, **manipulator** is a **function** which is used to format the output.
- There are basically 2 types of manipulator parameterized and non parameterized
- Example : endl, setw, fixed, scientific, setprecision, dec, oct, hex etc.
- All the manipulators are declared in **std namespace** but **header files are different**
 - 1. To use endl include <iostream>header file
 - 2. To use remaining manipulators include <iomanip> header file
- #include<iomanip> Header file is used input output manipulation.
 - setbase(16) // hex
 - setbase(8) // oct
 - setbase(10) // dec
 - endl
 - setw (val)
 - setprecision (val)

```
double f =3.14159;  
cout<< std::setprecision(5) << f <<'\n';
```



Macro

- A macro is a piece of code in a program that is **replaced by the value of the macro**.
- Macro definition **does not require the semicolon(;) at the end of the its definition**.
- In program where ever macro name arrives it get's replaced with definition.
- Macro can **turn a long declaration into a short declaration**.
- **Limitation of macro:**
 - In case of macro no **type checking**(checking the type of the variable and report in case of violation) which may lead to undesired result.
 - Macros are generally used for **code reuse** but **not for avoiding the time overhead**.
- **Syntax:**
#define name_of_macro value
- **Example:**
#define NUM 10
#define **sum**(a,b)(a+b) //observe no **type checking** is there as a and b dose not preceded with
//data type



Inline Function

- An inline function is one for which the **compiler copies the code** from the **function definition directly into** the code of the **calling function** rather than creating a separate set of instructions in memory.
- If we implement **member function inside structure/class** then it is **by default treated as inline**.
- The inline specifier **is only a suggestion/request** to the **compiler**, that an **inline expansion** can be performed; the **compiler can ignore** the suggestion.
- We can **use** the inline function **when performance is needed**.
- **Type checking** is done while calling the function.
- We can use the inline function over macros since macros are not type check.
- Inline functions are **used** for **avoiding the time overhead**
- **Syntax:**
`inline return_type functionname(// parameter list){//function body;}`
- **Example:**

```
inline void sum(int num1,int num2)
{
    cout<<"Sum of num1 and num2="<<num1+num2;
}
```



Inline Function

```
inline void sum(int num1,int num2)
```

```
{
```

```
    cout<<"Sum of num1 and num2="<<num1+num2;
```

```
}
```

```
int main()
```

```
{
```

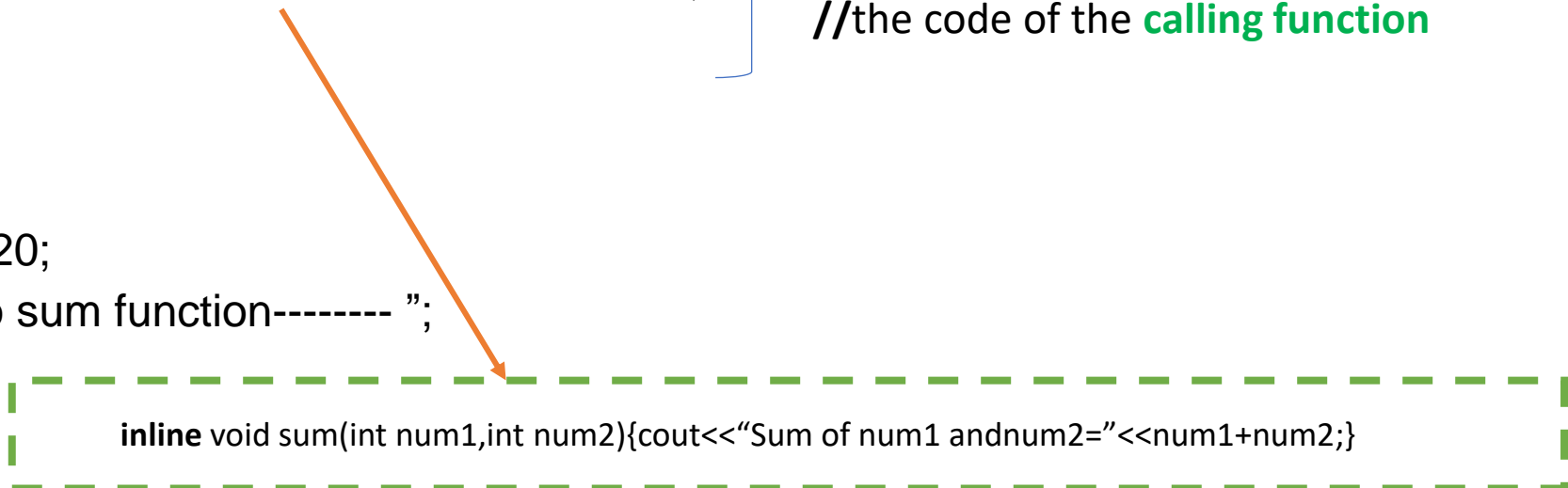
```
    int num1=10,int num2=20;
```

```
    cout<<"-----First call to sum function----- ";
```

```
    sum(num1,num2);
```

```
}
```

//Since **sum()** is inline function compiler copies the
// code from the function definition directly into
//the code of the calling function



```
inline void sum(int num1,int num2){cout<<"Sum of num1 andnum2="<<num1+num2;}
```



Default Argument

- In C++, functions may have **arguments with the default values**. Passing these arguments while calling a function is **optional**.
- A default argument is a **default value provided for a function parameter/argument**.
- If the user does not supply an **explicit argument** for a parameter with a **default argument**, the default value will be used.
- If such **argument is not passed**, then its **default value is considered**. Otherwise arguments are treated as normal arguments.
- Default arguments should be given in **right to left order**.
 - ```
int sum (int a, int b, int c=0, int d=0){
 return a + b + c + d;
}
```
  - Followings are possible function calls for sum()
    - Res=sum(10,20);
    - Res=sum(10,20,40);
    - Res=sum(10,30,40,50);



# Function Overloading

- Compile time / Static polymorphism / Static binding / Early binding / Weak typing / False Polymorphism.
- Functions with same name and different signature are called as overloaded functions.
- **Return type** is **not considered** for function overloading.
- **Function call** is resolved according to **types of arguments** passed.
- Function overloading is **possible due** to **name mangling** done by the C++ **compiler** (Name mangling process , mangled name).

**Name mangling:** Name given by compiler to the overloaded function just for its understanding just like a **version**.





# Function Overloading

- **Conditions for function overloading:**
  - **Case1:** Differ in **number of input arguments**.
  - **Case2:** Differ in **data type of input arguments**
  - **Case3:** Differ at least in the **sequence** of the input **arguments**.

- **Example :**

1. `void sum(int a, int b)`  
    `{ cout<<"a+b="<< a+b; }`
2. `void sum(float a, float b)`  
    `{cout<<"a+b="<< a+b; }`
3. `void sum(int a, int b, int c)`  
    `{cout<<"a+b+c="<< a+b+c; }`
4. `void sum(int a, float b)`  
    `{cout<<"a+b="<< a+b; }`
5. `void sum(float a, int b)`  
    `{cout<<"a+b="<< a+b; }`



# Function Overloading

- **Case1:** Differ in **number of input arguments**.

- **EXAMPLE:**

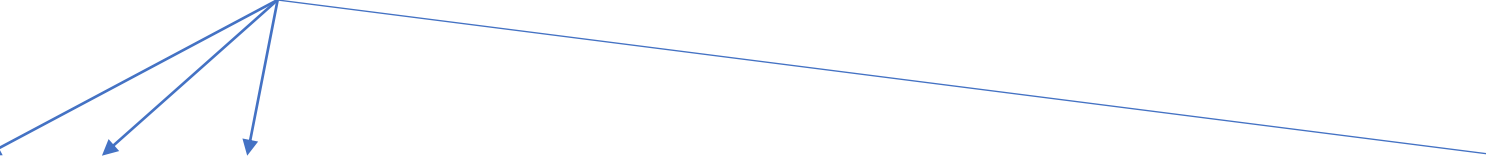
1. `void sum(int a, int b)` //Overload of function sum() for two int arguments

```
{
 cout<<"a+b="<< a+b;
}
```



2. `void sum(int a, int b, int c)` //Overload of function sum() for three int arguments

```
{
 cout<<"a+b+c="<< a+b+c;
}
```



# Function Overloading

- **Case2:** Differ in **data type** of input arguments

- **Example:**

1. void sum(int a, int b)

{

cout<<"a+b="<< a+b;

}

//Overload of function sum() for two int arguments

2. void sum(float a, float b)

{

cout<<"a+b="<< a+b;

}

//Overload of function sum() for two float arguments

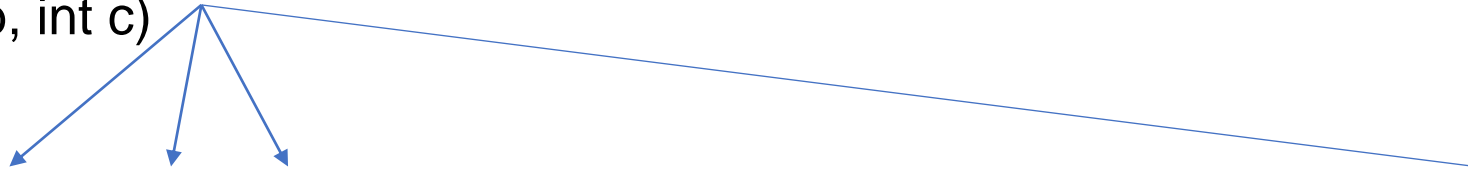


# Function Overloading

- **Case3:** Differ at least in the **sequence** of the input **arguments**.

## Example:

void sum(int a, int b, int c)

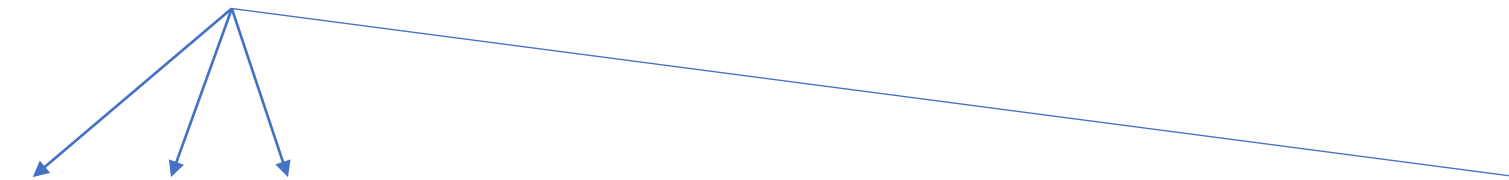


1. void sum(int a, float b, int c) **//Overload of function sum() for int and float argument sequence.**

{

cout<<"a+b+c="<< a+b+c;

}



2. void sum(float a, int b, int c) **//Overload of function sum() for float and int argument sequence.**

{

cout<<"a+b+c="<< a+b+c;

}



---

# Thank You

