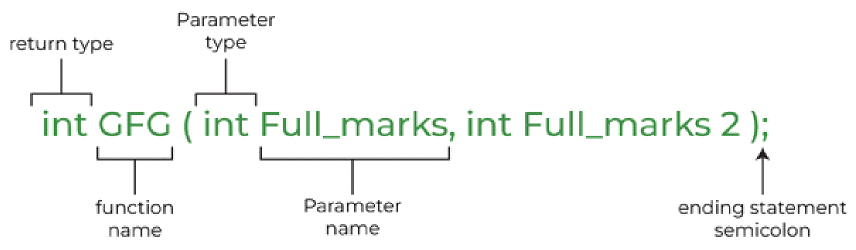
Functions

17 August 2024 20:33

Functions are a block of code which do some certain/specific tasks.

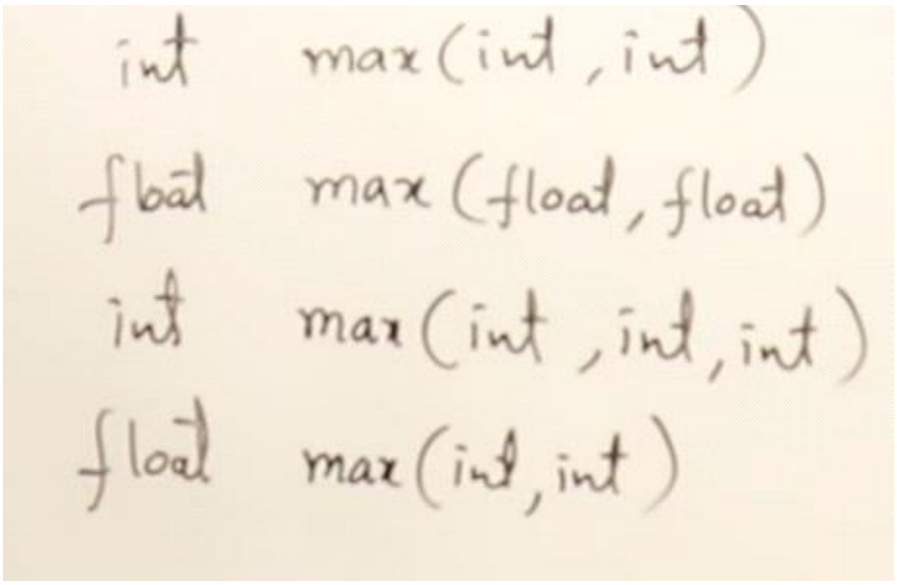

return type
Parameter type
int GFG (int Full_marks, int Full_marks 2);
function name
Parameter name
ending statement semicolon

Function Overloading

We can write functions with same name with different parameters and this is called function Overloading.

```
#include<iostream>
using namespace std;
int add(int x, int y){
    return x + y;
}
int add(double x, double y){
    return x + y;
}
int main() {
    cout<<add(5,4)<<endl; // 9
    cout<<add(5.25,4.37)<<endl; // 9
    return 0;
}
```

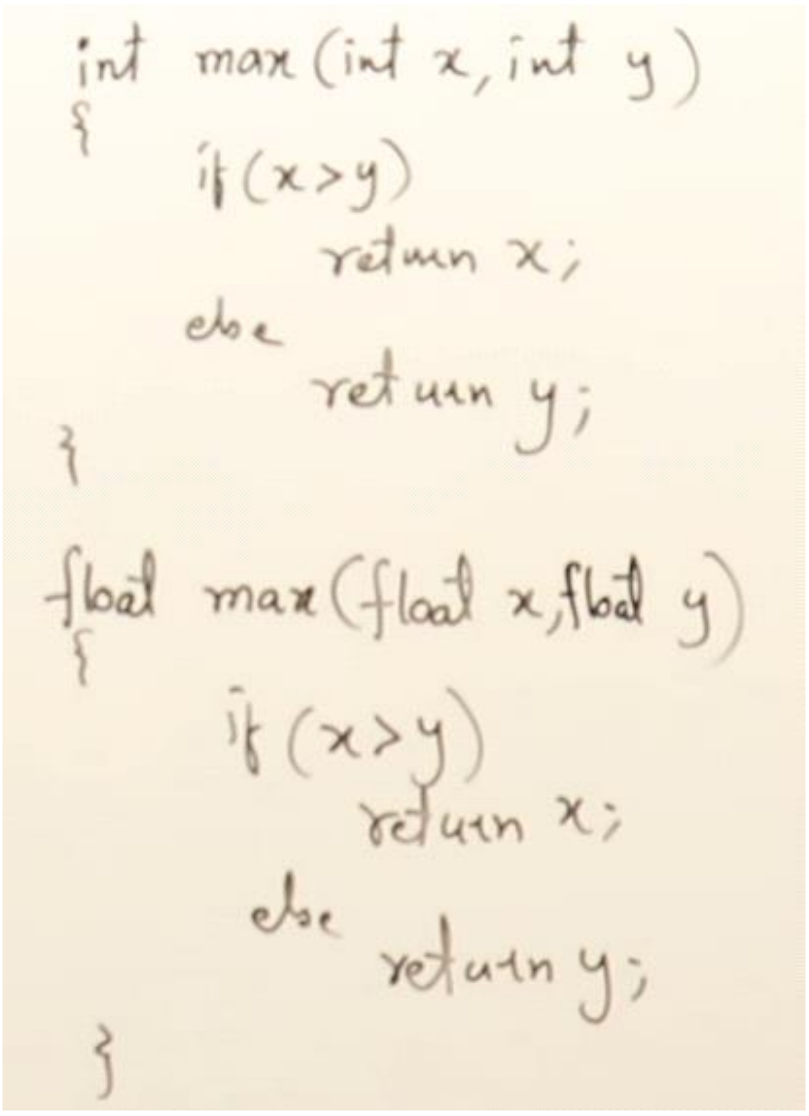
Other Examples:



```
int    max(int, int)
float  max(float, float)
int    max(int, int, int)
float  max(int, int)
```

Function Templates

What are function templates? They are functions that are generic i.e. generalized.
What Problems function templates solve?



```
int max(int x, int y)
{
      if(x>y)
            return x;
      else
            return y;
}

float max(float x, float y)
{
      if (x>y)
            return x;
      else
            return y;
}
```

As you can see that these two functions have same logic but we have to write them again and again. And so, to reduce this redundancy we use function templates.

```
#include<iostream>
using namespace std;
template <class T>
T maximum(T x, T y){
    if (x > y){
        return x;
    } else {
        return y;
    }
}
int main() {
    cout<<maximum(4,5)<<endl;
    cout<<maximum(9.84,5.45)<<endl;
    return 0;
}
```

Default Arguments

```
#include<iostream>
using namespace std;
int add(int x, int y, int z = 0){
    return x + y + z;
}
int main() {
    cout<<add(5,4)<<endl;
    cout<<add(5,4,7)<<endl;
    return 0;
}
```

Parameter Passing

1. Pass by Value

In pass by value, a copy of values are passed and any change inside the funcion will only happen locally and will have no effect on actual arguments.

```
#include<iostream>
using namespace std;
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
    cout<<"Swapped Values of a = "<<a<<" & b = "<<b<<endl;
}
int main() {
    int a,b;
    a = 10;
    b = 20;
    cout<<"Actual Values of a = "<<a<<" & b = "<<b<<" Before 
Swapping"<<endl;
    swap(a,b);
    cout<<"Actual Values of a = "<<a<<" & b = "<<b<<" after 
Swapping"<<endl;
    return 0;
}
```

2. Pass by reference

In pass by reference, original variables are passed instead of their copies hence any change to them will change the value of original Arguments. No additional space is created.

```
#include<iostream>
```

```
using namespace std;
void swap(int& a, int& b){
    int temp = a;
    a = b;
    b = temp;
    cout<<"Swapped Values of a = "<<(a)<<" & b = "<<(b)<<endl;
}
int main() {
    int a,b;
    a = 10;
    b = 20;
    cout<<"Actual Values of a = "<<a<<" & b = "<<b<<" Before
Swapping"<<endl;
    swap(a,b);
    cout<<"Actual Values of a = "<<a<<" & b = "<<b<<" after
Swapping"<<endl;
    return 0;
}
```

3. Pass by Pointer

In pass by pointer, addresses of Actual arguments are passed as arguments and any change inside the function will affect the Actual Arguments.

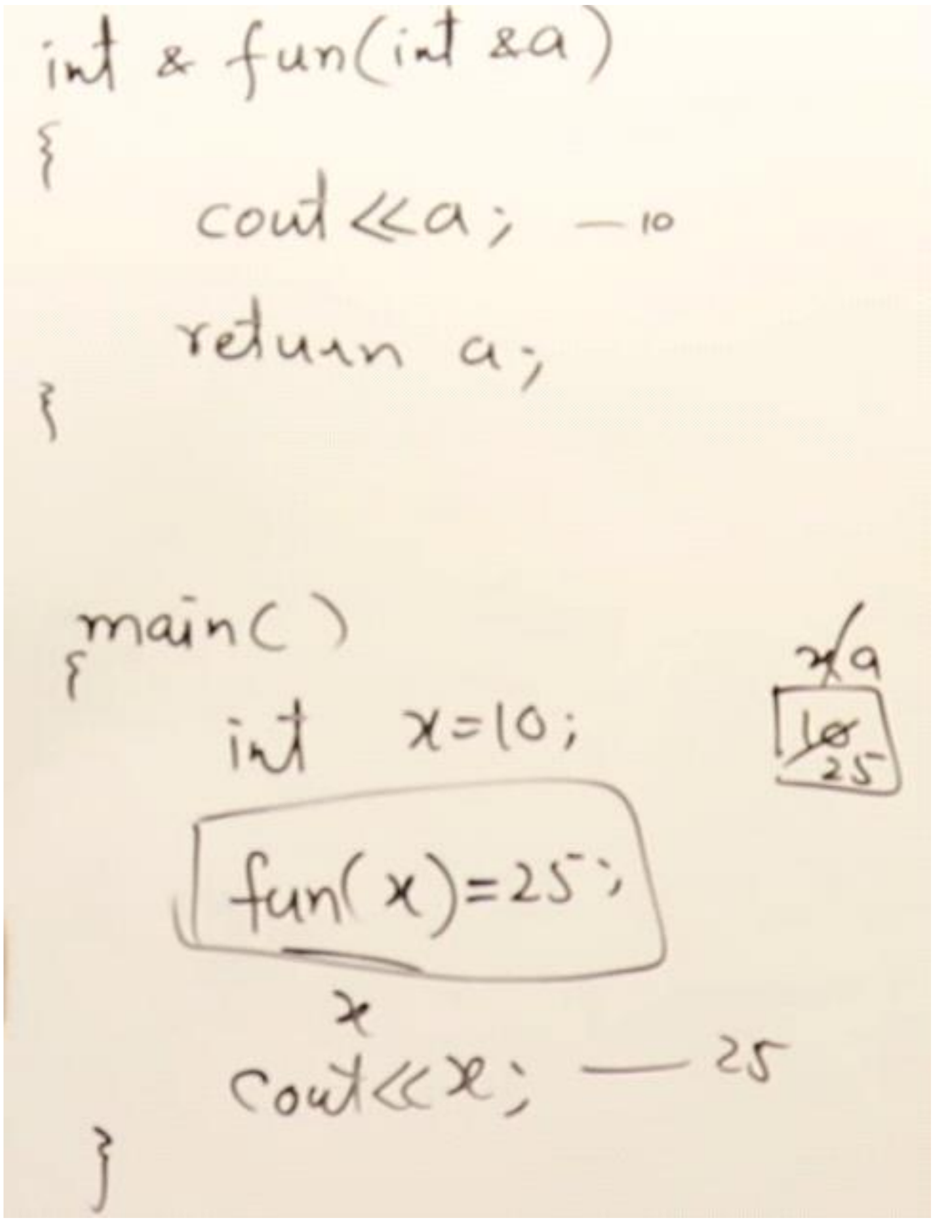
```
#include<iostream>
using namespace std;
void swap(int* a, int* b){
    int temp = *a;
    *a = *b;
    *b = temp;
    cout<<"Swapped Values of a = "<<(*a)<<" & b = "<<(*b)<<endl;
}
int main() {
    int a,b;
    a = 10;
    b = 20;
    cout<<"Actual Values of a = "<<a<<" & b = "<<b<<" Before
Swapping"<<endl;
    swap(&a,&b);
    cout<<"Actual Values of a = "<<a<<" & b = "<<b<<" after
Swapping"<<endl;
    return 0;
}
```

Return by Pointer

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

```
int* array(int size){
    int* p = new int[size];
    for (int i = 0; i < size; i++){
        p[i] = i * 5;
    }
    return p;
}
int main() {
    int* ptr = array(5);
    for (int i = 0; i < 5; i++){
        cout<<*(ptr + i)<<endl;
    }
    return 0;
}
```

Return by Reference



```
int & fun(int &a)
{
    cout<<a;   — 10
    return a;
}

main()
{
    int x=10;
    fun(x)=25;
    cout<<x;   — 25
}
```

Scope means to which extent a variable can be worked with.

1. **Local Scope Variables** are variables that are defined inside some Methods, functions including parameters and can only be used inside that function/ method.
2. **Global Scope Variables** are variables that are defined outside every Methods, function and can be used everywhere.

Static Variables in Functions

When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of the variable in the previous call gets carried through the next function call.

```cpp
#include<iostream>
using namespace std;
int gl = 52; // Global Variable
void fun(){
    int a = 5; // local Variable
    static int x = 5; // Static Variable
    x++;
    cout<<a<<" "<<x<<endl;
}
int main() {
    fun(); // 5 6
    fun(); // 5 7
    fun();  // 5 8
    return 0;
}
```