TATA ELXSI

OBJECT ORIENTED PROGRAMMING USING C++
Module 4

Learning & Development Team

Functions

Rules on Function Signature for overloading functions

- The number of parameters should be different in the functions.
  - ➢ void fun(int x, int y){…..}
  - ➢ void fun(int x){….}

- The parameter data type should be different in the functions.
  - ➢ void fun(int x, string y){…..}
  - ➢ void fun(string x, int y){…..}

Static Member Functions

- Member functions may also be declared static.
- Static member functions are subject to several restrictions.
- They may only directly refer to other static members of the class.
- A static member function does not have a **this** pointer.

Static Member Functions

➢There cannot be a **static** and a **non-static** version of the same function.

➢A static member function may not be **virtual**.

➢Finally, they cannot be declared as **const** or **volatile**.

➢One good use for them is to “pre-initialize” private static data before any object is actually created.

Static Member Functions

```
class Demo_static
{
   private:
           static int i;
   public:
           static void init ( int x)
           { i = x; }
           void show( )
           {
                      cout << i;
           }
};
int Demo_static::i ;     // define i
```

Static Member Functions

```
int main( )
 {
   // initialize static data before object creation
  Demo_static ::init(100);
  Demo_static x;
  x.show( ); // displays
  return 0;
 }
```

Initialization of const static Data Members

- const static data members of an integral type can now be initialized inside their class.
- In this case, the initialization is also a definition, so no further definitions are required outside the class body.
- For example

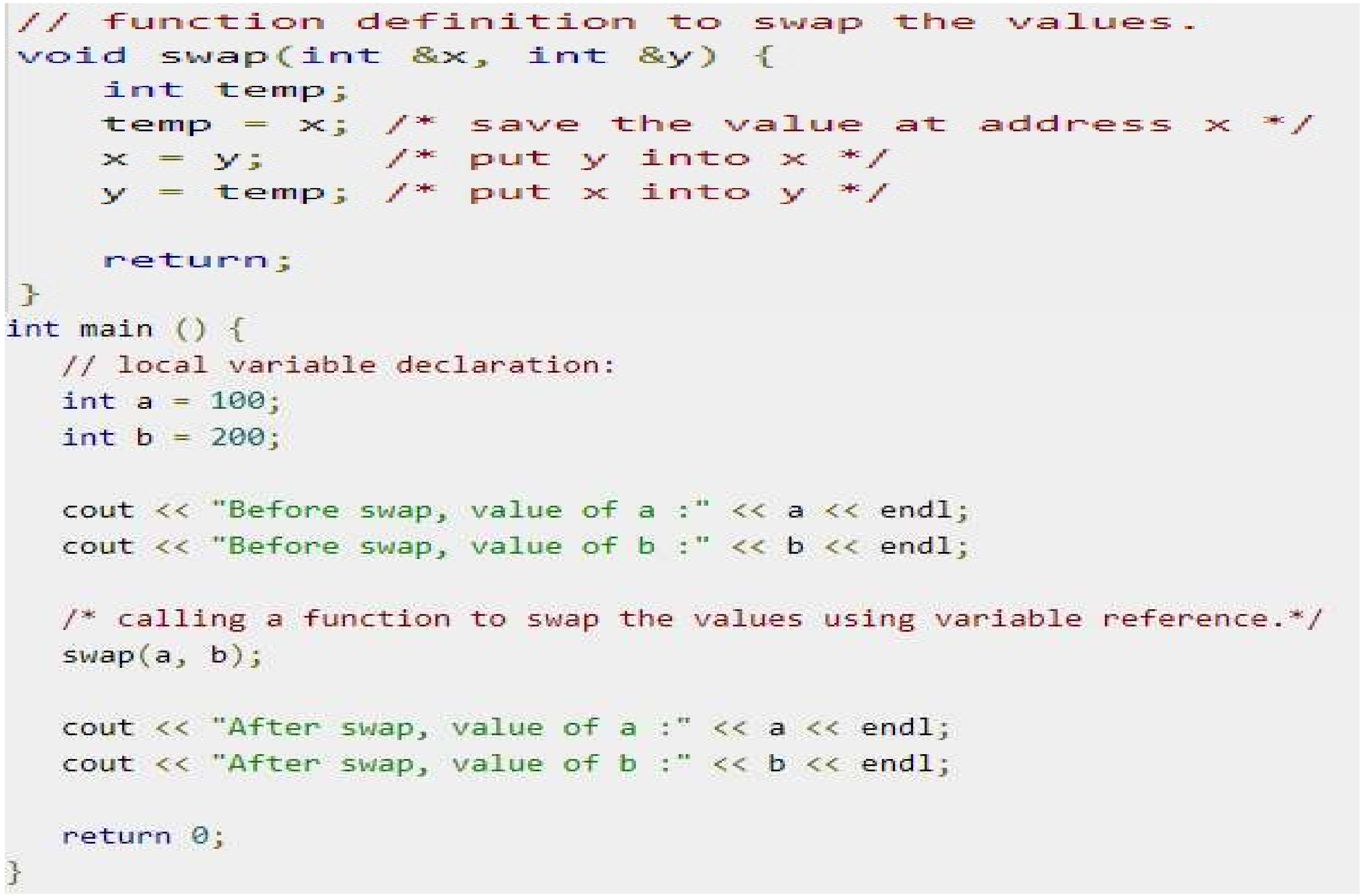
Initialization of const static Data Members

```
#include <string>
class Buff
{
  private:
          static const int MAX = 512; // initialization +definition
          static const char flag = 'a'; // initialization +definition

//non-integral type; must be defined outside the  class body

          static const std::string msg;
//..
};
          const std::string Buff::msg = "hello";
```

Call by Reference & its advantages

- The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter.
- Inside the function, the reference is used to access the actual argument used in the call.
- This means that changes made to the parameter affect the passed argument.
- To pass the value by reference, argument reference is passed to the functions just like any other value.
- So accordingly you need to declare the function parameters as reference types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

Call by Reference: sample example



```
// function definition to swap the values.
void swap(int &x, int &y) {
    int temp;
    temp = x; /* save the value at address x */
    x = y;    /* put y into x */
    y = temp; /* put x into y */

    return;
}
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;

   /* calling a function to swap the values using variable reference.*/
   swap(a, b);

   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;

   return 0;
}
```

Inline functions

- When a function has to be inlined, inline keyword has to be declared.
- However, if a function is declared inside a class declaration, it is automatically made into an inline function. It is not necessary to precede the function declaration with the inline keyword.
- Constructor / Destructor functions can be inlined.

Inline functions - example

```
#include <iostream>
using namespace std;
class mycl
{
   int a, b;
  public:
// automatic inline
   void init(int i, int j)
   {
       a=i;
       b=j;
   }
 void show()
   {
       cout << a;
       cout << b;
   }
};
int main()
{
   myclass x;
   x.init(10, 20);
   x.show();
   return 0;
}
```

Inline Vs. Macros

- To compute the square of a given number, either a macro or an inline function can be used as follows:

```
#define square(x) x*x
                OR
inline int square(int x)
   {     return (x*x);   }

void main( )
 {
  cout << square( 1 + 2) << “\n”;
 }
```

Default Function Arguments

- The default values must be specified only once, and this must be the first time the function is declared within the file.
- All parameters that take default values must appear to the right of those that do not.
- A default argument can also be used as a flag telling the function to reuse a previous argument.

Default Function Arguments

- Default parameters can be given in an object's constructor function. Two advantages of this:
  - No need for an overloaded no argument constructor
  - Defaulting common initial values is more convenient than specifying them each time an object is declared.
- Default arguments sometimes provide an alternative to function overloading.
- Guideline: No default argument should cause a harmful or destructive action.

Specifying Default Arguments

```
void clrscr(int size=25);
int main()
{
   register int i;
   for(i=0; i<30; i++ ) cout << i << endl;
   cin.get();
   clrscr(); // clears 25 lines
   for(i=0; i<30; i++ ) cout << i << endl;
   cin.get();
   clrscr(10); // clears 10 lines
   return 0;
}
void clrscr(int size)
{
   for(; size> 0; size--) cout << endl;
}
```

Constant Arguments Example.

```
void foo(int *const iPtr)
 {
            *iPtr++; // Error
 }
```

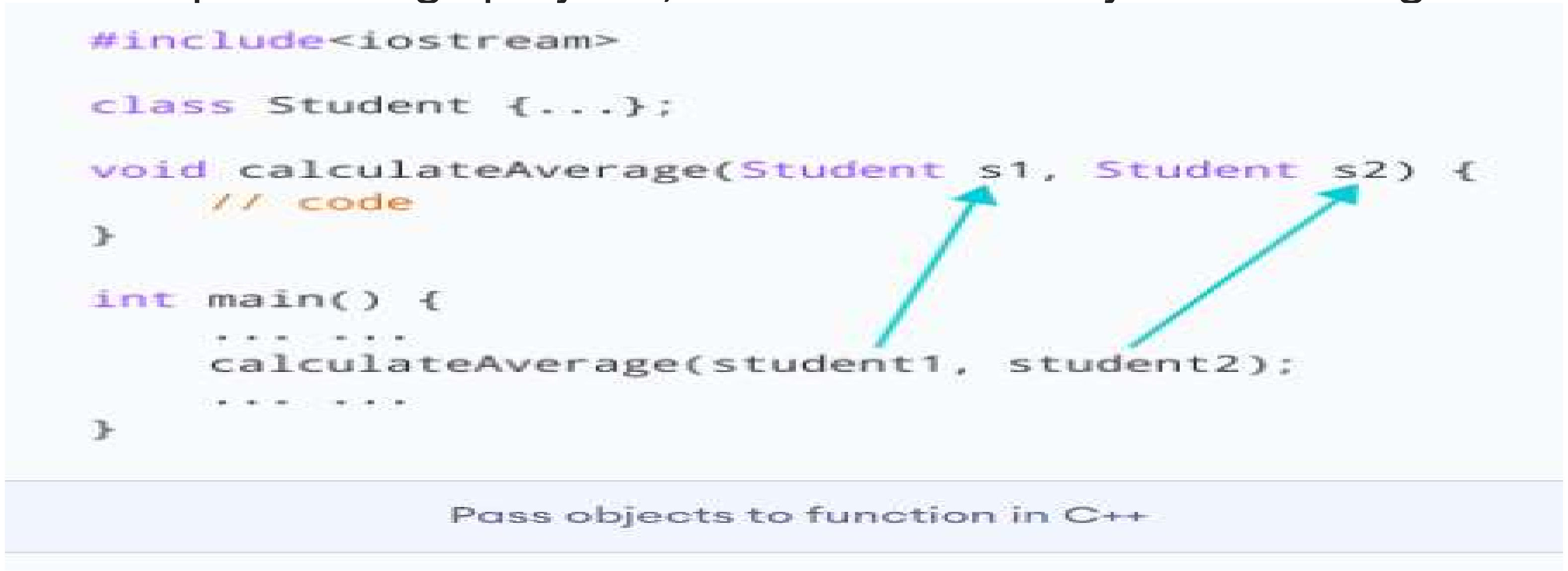
```
main()
{
  int    iNum=100;
  foo(&iNum);

}
```

- In a function if the parameter passed is of type constant, then we cannot change the variable value inside the function.
- Is it possible to change outside the function (Y / N).

Ans : No

Objects as arguments

- In C++ programming language, we can also pass an object as an argument within the member function of class.
- This is useful, when we want to initialize all data members of an object with another object, we can pass objects and assign the values of supplied object to the current object.
- For complex or large projects, we need to use objects as an argument or parameter.

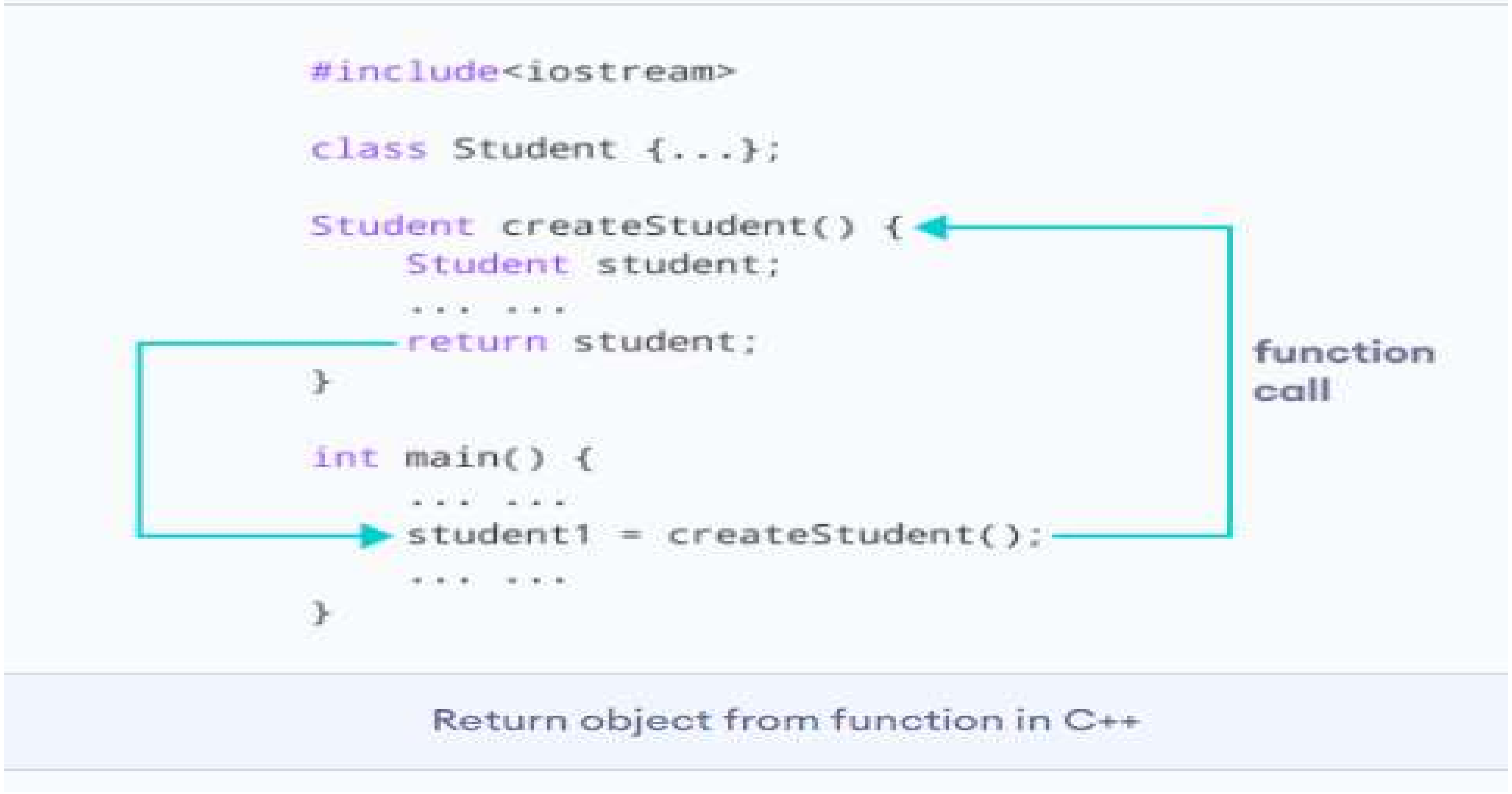


```
#include<iostream>

class Student {...};

void calculateAverage(Student s1, Student s2) {
    // code
}

int main() {
    ... ...
    calculateAverage(student1, student2);
    ... ...
}
```

Pass objects to function in C++

Return Object from a Function

- In C++ programming language, we can also return an object from a function of class.



```
#include<iostream>

class Student {...};

Student createStudent() {
    Student student;
    ... ...
    return student;
}

int main() {
    ... ...
    student1 = createStudent();
    ... ...
}
```

Return object from function in C++

Function Overloading and Ambiguity

- Situations in which the compiler is unable to choose between two (or more) overloaded functions is said to be ambiguous.
- Ambiguous statements are errors, and programs containing ambiguity will not compile.
- The main cause of ambiguity involves C++'s automatic type conversions.

Function Overloading and Ambiguity - Eg.

```
float myfunc(float i){ return i;}
double myfunc(double i){ return -i;}
int main()
{
   cout << myfunc(10.1);//unambiguous, calls myfunc(double)
   cout << myfunc(10); // ambiguous
   return 0;
}
```

Function Overloading and Ambiguity

- Another way ambiguity is caused is by using default arguments in overloaded functions.

```
int myfunc(int i){ return i; }
int myfunc(int i, int j=1) {  return i*j;  }
int main()
{
   cout << myfunc(4, 5); // unambiguous
   cout << myfunc(10); // ambiguous
   return 0;
}
```

Function Overloading and Ambiguity

- Two functions cannot be overloaded when the only difference is that one takes a reference parameter and the other takes a normal, call-by-value parameter.

```
void f(int x){cout << "In f(int)\n";}
void f(int &x){cout << "In f(int &)\n";}
int main()
{
   int a=10;
   f(a); // error, which f()?
   return 0;
}
```

Are you ready to solve…

1. Static member function cannot be declared as **const** or **volatile**.
   a. True
   b. False

   Ans: **a. True**

2. Function Overloading depends on return type of the function.

   a. True b. False

   Ans: **b. False**

End of Module 4

Disclaimer

- Some examples and concepts have been sourced from the below links and are open source material
  - ❖ **http://cppreference.com**
  - ❖ **www.cplusplus.com**

- References:
  - ❖ *C++: The Complete Reference* - 4th Edition by Herbert Schildt, Tata McGraw-Hill publications.
  - ❖ *The C++ Programming Language*- by Bjarne Stroustrup.
  - ❖ *Practical C++ Programming*- by Steve Oualline, O'Reilly publications.

Learning & Development Team

ITPB Road Whitefield

Bangalore 560 048 India
Tel +91 80 2297 9123
Fax +91 80 2841 1474
e-mail info@tataelxsi.com

www.tataelxsi.com

TATA ELXSI