SRM UNIVERSITY AP
Amaravati

Object Oriented Programming using C++

CSE-III

CSE206-OOP

Dr. Priyanka Singh

Default argument function

- If a function with default arguments is called without passing arguments, then the default parameters are used.
- However, if arguments are passed while calling the function, the default arguments are ignored.
- Once we provide a default value for a parameter, all subsequent parameters must also have default values.

Invalid

```
void add(int a, int b = 3, int c, int d);
```

Invalid

```
void add(int a, int b = 3, int c, int d = 4);
```

Valid

```
void add(int a, int c, int b = 3, int d = 4);
```

Case 1 : No argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp();
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 2 : First argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(6);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 3 : All arguments are passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(6, -2.3);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 4 : Second argument is passed

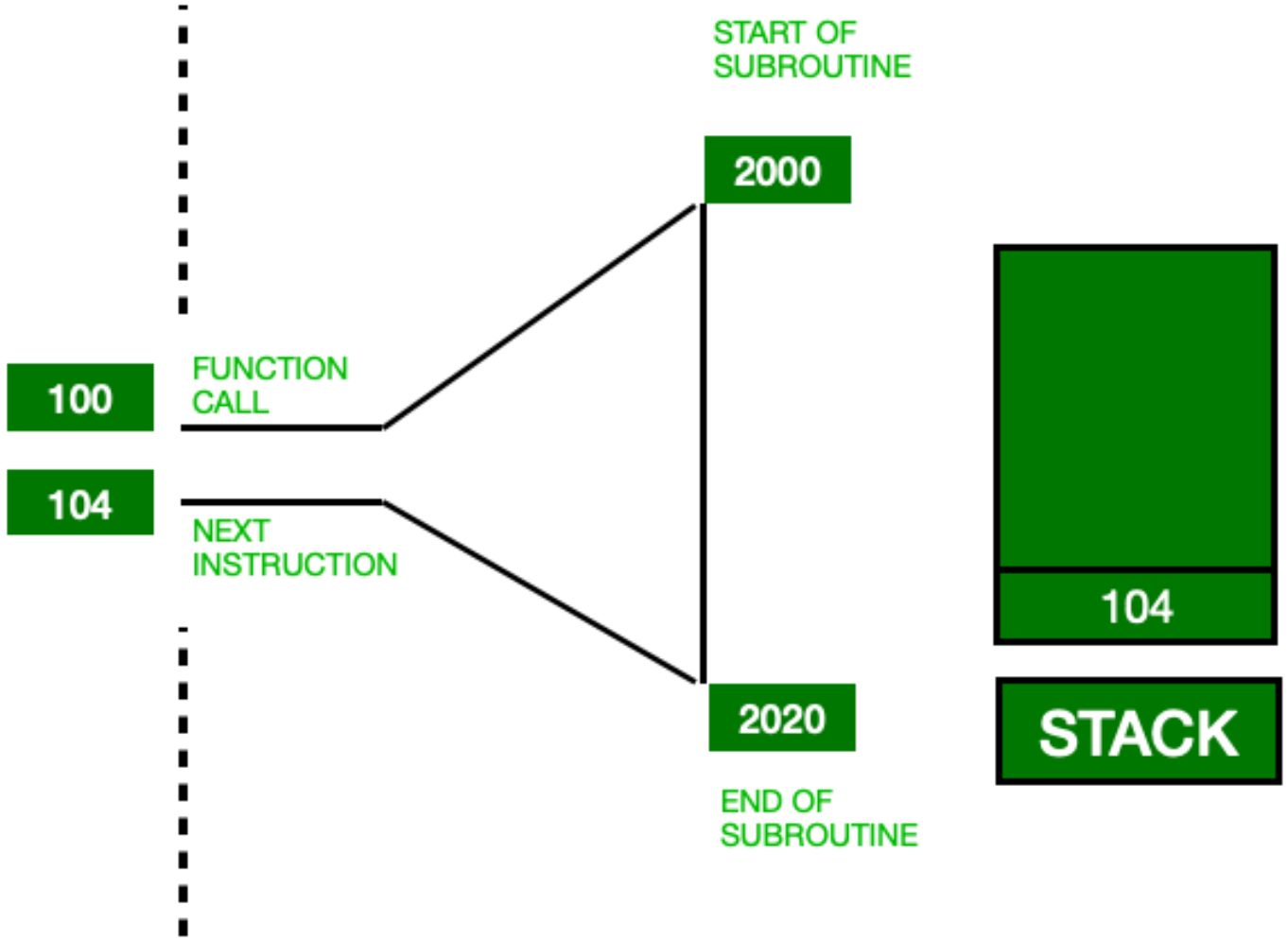
```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(3.4);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

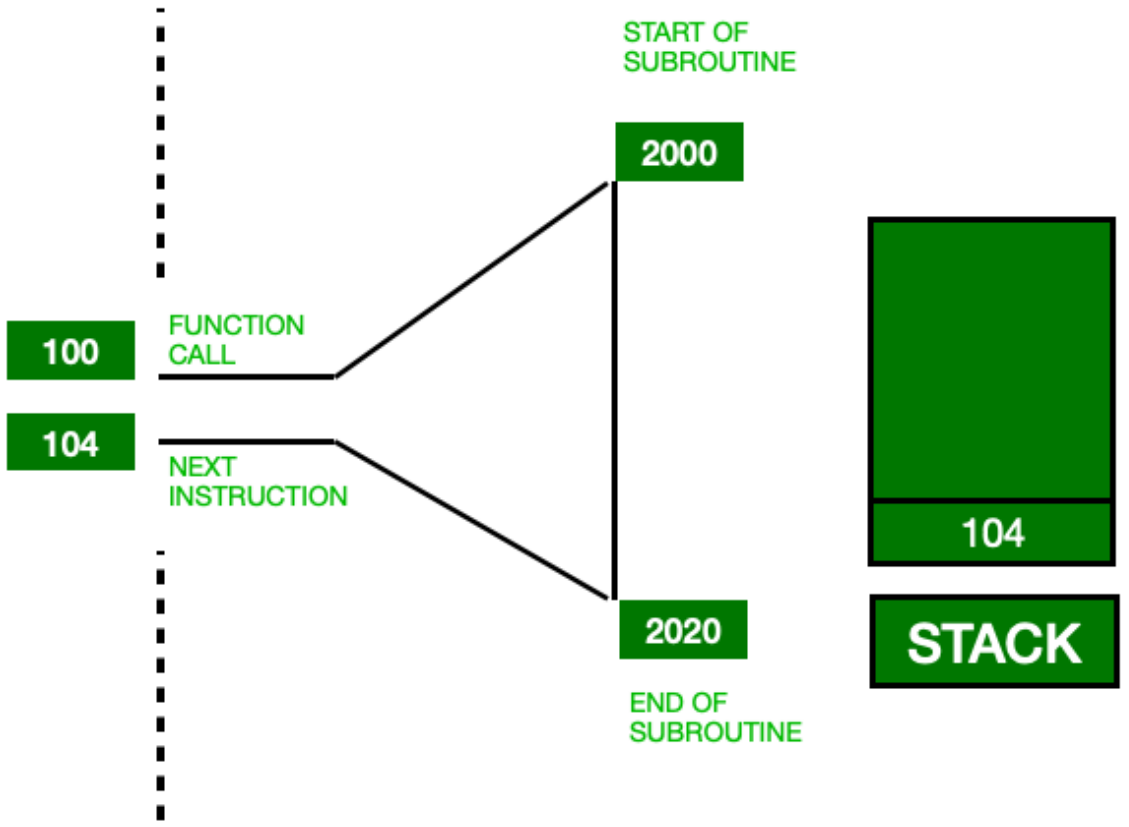
Error

Function execution



Series of operations when we call a function:

1. Stack Frame is pushed into stack.
2. Sub-routine instructions are executed.
3. Stack Frame is popped from the stack.
4. Now Program Counter is holding the return address.



Inline Functions in C++

- Inline Function is a function that is expanded inline by the compiler when it is invoked. During function call, a lot of overhead tasks are performed like saving registers, pushing arguments to the stack, and returning to the calling function. These overheads are time-consuming and inefficient for small-size functions.

- In C++, the inline function is used to solve these overhead costs. It is expanded in line by the compiler when it is invoked, thus overhead cost is avoided. A keyword known as ‘inline‘ is used before the function declaration.

Inline Functions in C++

- 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.
- 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.

Inline Functions in C++

- 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.
- One of the major objectives of using functions in a program is to save memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing tasks such as jumping to the calling function.

Inline Functions in C++

- 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:
  - If a function contains a loop. (for, while, do-while)
  - If a function contains static variables.
  - If a function is recursive.
  - If a function return type is other than void, and the return statement doesn’t exist in function body.
  - If a function contains switch or goto statement.

Inline Functions

- 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.
- Syntax:

```
inline return_type function_name(parameters)
{
        // Insert your Function code here
}
```

Program to demonstrate inline function

```
#include <iostream>
using namespace std;
// Inline function to multiply two integer numbers
inline int multiplication(int x, int y)
{
                return x * y; // Returning the product of two numbers
}

int main()
{
        // Print and display the multiplication resultant number with usage of above created
        cout << "Multiplication ( 20 , 30 ):“ << multiplication(20, 30) << endl;
        return 0;
}
```

Inline example

```
#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;
}
```

Inline Function	Normal Function
It is expanded inline when it is invoked.	It is a function that provides modularity to the program.
It is generally used to increase the execution time of the program.	It is generally used to improve the reusability of code and make it more maintainable.
It is basically a function that is used when functions are small and called very often.	It is basically a group of statements that performs a particular task. It is used when functions are big.
It requires ‘inline‘ keyword in its declaration.	It does not require any keyword in its declaration.
It generally executes much faster as compared to normal functions.	It generally executes slower than inline function for small size function.
In this, the body of functions is copied to every context where it is used that in turn reduces the time of searching the body in the storage device or hard disk.	In this, the body of the function is stored in the storage device and when that particular function is called every time, then CPU has to load the body from hard disk to RAM for execution.
The compiler always places a copy of the code of that function at each point where the function is called at compile time.	It does not provide such a type of functionality.
It generally includes only 2–3-line codes.	When the number of line codes is real massive I.e., normal functions contain much code as per need.
It is a little harder to understand and test as compared to normal function.	It is much easier to understand and test as compared to the inline function.
Functions that are present inside a class are implicitly inline.	Functions that are present outside class are considered normal functions.
Too many inline functions affect file size after compilation as it duplicates the same code.	Too many normal functions do not affect file size after compilation.

Advantage of inline function

- Function call overhead doesn’t occur.
- It also saves the overhead of push/pop variables on the stack when function is called.
- It also saves overhead of a return call from a function.
- 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.
- 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.

Disadvantage of inline function

- 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.
- 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.
- 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.
- 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.
- Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.
- Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

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.

Implicit

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

Explicit

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

Storage Classes

- Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.
- Syntax:

  storage_class var_data_type var_name;

Scope, lifetime and visibility

- The scope of a variable is the range of program statements that can access that variable. The lifetime of a variable is the interval of time in which storage is bound to the variable.
- A variable is visible within its scope and invisible or hidden outside it. The scope of an identifier is that portion of the program code in which it is visible, that is, it can be used.
- An identifier's "visibility" determines the portions of the program in which it can be referenced — its "scope." An identifier is visible (i.e., can be used) only in portions of a program encompassed by its "scope," which may be limited (in order of increasing restrictiveness) to the file, function, block, or function prototype in which it appears.

Scope types

- **File scope:** If the declaration of an identifier is outside any block or list of parameters, it has file scope.
- **Block scope:** If the declaration of an identifier is inside a block, it has block scope and the scope ends with the end of the block. A block of statements starts with the left brace ({) and ends with the closing right brace (}).
- **Function scope:** If the declaration of an identifier is inside a list of parameters in function definition, it has function scope, which ends with the end of function definition.
- **Function prototype scope:** If the declaration of identifier appears within the list of parameters of a function prototype, which is not part of function definition, the scope of identifier is limited to the function prototype. The scope ends with the end of declaration.

Introduction

- A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify.
- There are following storage classes, which can be used in a C++ Program:
  - auto
  - register
  - static
  - extern
  - mutable

C++ Storage Classes

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extem	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage

auto

- The auto keyword provides type inference capabilities, using which automatic deduction of the data type of an expression in a programming language can be done. This consumes less time having to write out things the compiler already knows.
- As all the types are deduced in compiler phase only, the time for compilation increases slightly but it does not affect the run time of the program.
- This feature also extends to functions and non-type template parameters. Since C++14 for functions, the return type will be deduced from its return statements.
- Since C++17, for non-type template parameters, the type will be deduced from the argument.

auto

- The auto storage class is the default storage class for all local variables:

```
{   int mount;
    auto int month;
}
```

- The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.
- Automatic variables are allocated memory automatically at runtime.
- The scope and visibility of the automatic variables is limited to the block in which they are defined.
- The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.

Program to illustrate auto

```
#include <iostream>
using namespace std;
void autoStorageClass()
{
    cout << "Demonstrating auto class\n";
     // Declaring an auto variable. No data-type declaration needed
    auto a = 32;
    auto b = 3.2;
    auto c = "CSE 206";
    auto d = 'G';
    // printing the auto variables
    cout << a << " \n";
    cout << b << " \n";
    cout << c << " \n";
    cout << d << " \n";
}
```

```
int main()
{
    // To demonstrate auto Storage Class
    autoStorageClass();
    return 0;
}
```

register

- The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
    register
    int  miles;
}
```

- The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

Program to illustrate register

```
#include <iostream>
using namespace std;
void registerStorageClass()
{
    cout << "Demonstrating register class\n";
    // declaring a register variable
    register char b = 'G';
     // printing the register variable 'b'
    cout << "Value of the variable 'b'"
         << " declared as register: " << b;
}
int main()
{     // To demonstrate register Storage Class
    registerStorageClass();
    return 0;
}
```

static

- The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.
- The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.
- In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

Program to illustrate static

```
#include <iostream>
void func(void);
static int count = 10; /* Global variable */
main() {
   while(count--) {
      func();
   }
   return 0;
} void func( void ) {
   static int i = 5; // local static variable
   i++;
   std::cout << "i is " << i ;
   std::cout << " and count is " << count << std::endl;
}
```

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

Program to illustrate static

```
#include <iostream>
using namespace std;
// Function containing static variables
// memory is retained during execution
int staticFun()
{
    cout << "For static variables: ";
    static int count = 0;
    count++;
    return count;
}
// Function containing non-static variables
// memory is destroyed
int nonStaticFun()
{
    cout << "For Non-Static variables: ";
    int count = 0;
    count++;
    return count;
}
```

```
int main()
{
    // Calling the static parts
    cout << staticFun() << "\n";
    cout << staticFun() << "\n";

    // Calling the non-static parts

    cout << nonStaticFun() << "\n";
    cout << nonStaticFun() << "\n";

    return 0;
}
```

For static variables: 1
For static variables: 2
For Non-Static variables: 1
For Non-Static variables: 1

extern

- The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.
- When you have multiple files and you define a global variable or function, which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding extern is used to declare a global variable or function in another file.
- The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

Program to illustrate extern

```
#include <iostream>
using namespace std;
 // declaring the variable which is to
// be made extern an initial value can
// also be initialized to x
int x;
void externStorageClass()
{
    cout << "Demonstrating extern class\n";
    // telling the compiler that the variable
    // x is an extern variable and has been
    // defined elsewhere (above the main
    // function)
    extern int x;
     // printing the extern variables 'x'
    cout << "Value of the variable 'x'"
         << "declared, as extern: " << x <<
"\n";
    // value of extern variable x modified
    x = 2;
    // printing the modified values of
    // extern variables 'x'
    cout
        << "Modified value of the
variable 'x'"
        << " declared as extern: \n"
        << x;
}
int main()
{
    // To demonstrate extern Storage
Class
    externStorageClass();

    return 0;
}
```

mutable

- The mutable specifier applies only to class objects.
- It allows a member of an object to override const member function. That is, a mutable member can be modified by a const member function.
- Sometimes there is a requirement to modify one or more data members of class/struct through const function even though you don't want the function to update other members of class/struct. This task can be easily performed by using the mutable keyword.
- The keyword mutable is mainly used to allow a particular data member of const object to be modified.
- When we declare a function as const, this pointer passed to function becomes const. Adding mutable to a variable allows a const pointer to change members.

Program to illustrate mutable

```
#include <iostream>
using std::cout;
class Test
{
public:
    int x;
    // defining mutable variable y
    // now this can be modified
    mutable int y;
    Test()
    {
        x = 4;
        y = 10;
    }
};
```

```
int main()
{
    // t1 is set to constant
    const Test t1;

    // trying to change the value
    t1.y = 20;
    cout << t1.y;

    // Uncommenting below lines will throw
error
    // t1.x = 8;
    // cout << t1.x;
    return 0;
}
```