Object Oriented Programming with C++

4. C++ Functions

By: Prof. Pandav Patel

Second Semester, 2020-21
Computer Engineering Department
Dharmsinh Desai University

C Vs C++ - main function

	C	C++
Return type	• Support of **int** is mandated by standard • Other return types may be supported by compiler. Not mandated by standard	• Return type must be **int** • Compilers can not support other return types
Formal arguments	• Standard mandates support of **(void)** and **(int, char **)** • Other arguments may be supported by compiler. Not mandated by standard	• Standard mandates support of **(void)** and **(int, char **)** • Other arguments may be supported by compiler. Not mandated by standard

C Vs C++ - function prototype

- Both in **C** and **C++**, if function is called only after function definition, then it is not necessary to have function prototype
  - Function definition works as function prototype
- In **C++**, function definition or prototype is must before function call
  - Function **name, return type and arguments** (number, order and type) in prototype must be same as function definition.
- In C, function definition or prototype is not must before function call
  - In absence of function definition and prototype function call works as implicit declaration (**with warning**)
    - It assumes return type to be **int**
    - And number, order and types of arguments is derived from values passed for function call
    - If definition and implicit declaration mismatch, then it results in **error**.
- If function prototype has empty parantheses
  - in C, it means any number of arguments
  - while in C++ it means **void** as argument (required for function overloading)

C++ Inline functions

- Function like macros (C also supports this)

```
#include<iostream>

#define SQUARE(x) (x) * (x)

int sqr(int x)
{
	return x * x;
}

int main()
{
	std::cout << SQUARE(3) << std::endl;
	std::cout << SQUARE(3.1 + 1) << std::endl << std::endl;

	std::cout << sqr(3) << std::endl;
	std::cout << sqr(3.1 + 1) << std::endl;

	return 0;
}
```

C++ Inline functions

- Benefit of Macros (~Drawback of function)
  - No function call needed, hence no overhead (like required for function call)

- Drawbacks of Macros (~Benefits of function)
  - Sometimes you get result which you might not have expected
  - No type checking

C++ Inline functions

- Concept of **inline** functions have been introduced to benefit from better of both worlds – macros and functions

```
#include<iostream>

inline int cube(int n)
{
	return n * n * n;
}

int main()
{
	float ans1 = cube(3);
	float ans2 = cube(3.1 + 1);
	std::cout << ans1 << std::endl;
	std::cout << ans2 << std::endl;
	return 0;
}
```

C++ Inline functions

- When a function is defined as **inline**, it is a just a **request (NO GUARANTEE)** to the compiler to expand it in line when it is called.
  - It is **as if** whole code of the inline function gets inserted or substituted at the point of inline function call.
  - Compiler is intelligent - Same variable names in calling function and inline function will not clash
  - Making function inline will not change the final output at all (Except performance difference due to inlining)
  - Inline function must be defined before it is called – (so that compiler knows exact code)

- Inline functions are not same as macros – macros are handled by preprocessor and inline functions are handled by compiler

C++ Inline functions

- Only small function (one/two lines) should be defined as inline
- There is a trade-off – performance Vs size of output file (binary file)
  - If you inline function (specifically larger functions) at many locations, then size of output binary file will increase as code will be repeated at multiple locations

- inline function **CAN NOT**
  - be recursive in nature
  - contain static variable
  - contain loop, switch or goto

C++ functions - default arguments

```
#include<iostream>
int sum(int n1, int n2, int n3 = 111, int n4 = 222);
int main()
{
	std::cout << sum(1, 2, 3, 4) << std::endl;   // 10
	std::cout << sum(1, 2, 3) << std::endl;      // 228
	std::cout << sum(1, 2) << std::endl;         // 336
	return 0;
}
int sum(int n1, int n2, int n3, int n4)
{
	return n1 + n2 + n3 + n4;
}
```

- int sum(int n1, int n2, int n3 = 4, int n4); // **Invalid**
- It is also valid to provide default values for all arguments
- int sum(int n1 = 11, int n2 = 33, int n3 = 44, int n4 = 66);

C++ functions - default arguments

```
#include<iostream>
int sum(int n1, int n2, int n3 = 3, int n4 = 4);
void fun()
{
	std::cout << "fun" << std::endl;
	std::cout << sum(1, 2, 3, 4) << std::endl;
	std::cout << sum(1, 2, 3) << std::endl;
	std::cout << sum(1, 2) << std::endl;
}
int main()
{
	int sum(int n1, int n2, int n3 = 111, int n4 = 222);
	std::cout << "main" << std::endl;
	std::cout << sum(1, 2, 3, 4) << std::endl;
	std::cout << sum(1, 2, 3) << std::endl;
	std::cout << sum(1, 2) << std::endl;
	fun();
	return 0;
}
int sum(int n1, int n2, int n3 = 3, int n4 = 4)   // Error – You can not assign default values in definition and declaration
{                                                 // if definition and declaration are in the same socpe
	return n1 + n2 + n3 + n4;
}
```

C++ functions - default arguments

```
#include<iostream>
int sum(int n1, int n2, int n3, int n4);
void fun()
{
	std::cout << "fun" << std::endl;
	std::cout << sum(1, 2, 3, 4) << std::endl;
	std::cout << sum(1, 2, 3) << std::endl;    // Error – No default values until after definition of sum function
	std::cout << sum(1, 2) << std::endl;       // Error – No default values until after definition of sum function
}
int main()
{
	int sum(int n1, int n2, int n3 = 111, int n4 = 222);
	std::cout << "main" << std::endl;
	std::cout << sum(1, 2, 3, 4) << std::endl;
	std::cout << sum(1, 2, 3) << std::endl;
	std::cout << sum(1, 2) << std::endl;
	fun();
	return 0;
}
int sum(int n1, int n2, int n3 = 3, int n4 = 4)
{
	return n1 + n2 + n3 + n4;
}
```

C++ functions - default arguments

```
#include<iostream>
int sum(int n1, int n2, int n3, int n4);
int main()
{
	int sum(int n1, int n2, int n3 = 111, int n4 = 222);
	std::cout << "main" << std::endl;
	std::cout << sum(1, 2, 3, 4) << std::endl;
	std::cout << sum(1, 2, 3) << std::endl;		// Will get default values from local declaration
	std::cout << sum(1, 2) << std::endl;		// Will get default values from local declaration
	void fun(); fun();
	return 0;
}
int sum(int n1, int n2, int n3 = 3, int n4 = 4)
{
	return n1 + n2 + n3 + n4;
}
void fun()
{
	std::cout << "fun" << std::endl;
	std::cout << sum(1, 2, 3, 4) << std::endl;
	std::cout << sum(1, 2, 3) << std::endl;		// Default values from definition
	std::cout << sum(1, 2) << std::endl;		// Default values from definition
}
```

```
main
10
228
336
fun
10
10
10
```

C++ functions - default arguments

```
#include<iostream>
int sum(int n1, int n2, int n3 = 3, int n4 = 4);
int main()
{
	int sum(int n1, int n2, int n3 = 111, int n4 = 222);
	std::cout << "main" << std::endl;
	std::cout << sum(1, 2, 3, 4) << std::endl;
	std::cout << sum(1, 2, 3) << std::endl;		// Will get default values from local declaration
	std::cout << sum(1, 2) << std::endl;		// Will get default values from local declaration
	void fun(); fun();
	return 0;
}
int sum(int n1, int n2, int n3, int n4)
{
	return n1 + n2 + n3 + n4;
}
void fun()
{
	std::cout << "fun" << std::endl;
	std::cout << sum(1, 2, 3, 4) << std::endl;
	std::cout << sum(1, 2, 3) << std::endl;		// Default values from global declaration of sum function still hold
	std::cout << sum(1, 2) << std::endl;		// Even after definition of sum function without default values
}
```

```
main
10
228
336
fun
10
10
10
```

C++ functions - default arguments

- If function prototype and function definition are in global scope
  - Default values be provided with either definition or prototype, not both (even if you provide same values)
    - If default values are in definition then until function definition, function would be treated as if it does not have default values (as prototype does not have default values)
    - If default values are in prototype, then default values from prototype would be applicable through out (even after definition, even though definition does not have default values)
- If prototype is within another function and definition in global scope
  - It is allowed to have default values in both definition and prototype
    - Default values from locally declared prototype will have priority over default values from definition in global scope

- **Avoid default values in definition, have default values in prototype**

Bad(but working) Example:

```
// Header file: myFunc.h
void g(int, double, char = 'a');

-------------------------------------------------
// Source File: myFunc.cpp

#include <iostream>

using namespace std;

void g(int i, double d, char c)
{
  cout << i << ' ' << d << ' ' << c << endl;
}
```

Compile : g++ Apps.cpp myFunc.cpp
Run : ./a.out

```
// Application File: Apps.cpp

#include "myFunc.h“
// void g(int, double, char = 'a');

void g(int i, double f = 0.0, char ch); // OK a new overload
void g(int i = 0, double f, char ch); // OK a new overload

int main() {
  int i = 5; double d = 1.2; char c = 'b';
  g(); // Prints:  0 0.0 a
  g(i); // Prints: 5 0.0 a
  g(i, d); // Prints: 5 1.2 a
  g(i, d, c); // Prints: 5 1.2 b
  return 0;
}
```

Default parameters **”should”**

be supplied only in a header file

and not in the definition of a function or anywhere else

```
// Header file: myFunc.h
void g(int=0, double=0.0, char = 'a');

-----------------------------------------------
// Source File: myFunc.cpp

#include <iostream>

using namespace std;


void g(int i, double d, char c)
{
  cout << i << ' ' << d << ' ' << c << endl;
}
```

```
// Application File: Apps.cpp

#include "myFunc.h“
// void g(int, double, char = 'a');

void g(int i, double f = 0.0, char ch); // OK a new overload
void g(int i = 0, double f, char ch); // OK a new overload

int main() {
  int i = 5; double d = 1.2; char c = 'b';
  g(); // Prints: 0  0.0 a
  g(i); // Prints: 5 0.0 a
  g(i, d); // Prints: 5 1.2 a
  g(i, d, c); // Prints: 5 1.2 b
  return 0;
}
```