Module 08: Programming C++

Default Parameters & Function Overloading

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan

Module Objectives

- Understand default parameters
- Understand function overloading and Resolution

Module Outline

- Default parameter
  - Motivation
  - Call function with default parameter
  - Highlighted Points
  - Restrictions
- Function overloading
  - Meaning & Motivation
  - Necessity of function overloading in Contrast with C
- Static Polymorphism
  - Meaning
  - Overloading function
- Overload Resolution
- Default parameters and Function Overloading

Motivation: Example `CreateWindow` in MSDN

Declaration of CreateWindow

```
HWND WINAPI CreateWindow(
    _In_opt_ LPCTSTR   lpClassName,
    _In_opt_ LPCTSTR   lpWindowName,
    _In_     DWORD     dwStyle,
    _In_     int       x,
    _In_     int       y,
    _In_     int       nWidth,
    _In_     int       nHeight,
    _In_opt_ HWND      hWndParent,
    _In_opt_ HMENU     hMenu,
    _In_opt_ HINSTANCE hInstance,
    _In_opt_ LPVOID    lpParam
);
```

Calling CreateWindow

```
hWnd = CreateWindow(
    ClsName,
    WndName,
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL
);
```

- There are 11 Number of parameters in `CreateWindow()`
- Out of these 11, 7 parameters (4 are `CWUSEDEFAULT` and 3 are `NULL`) usually get fixed values in a call
- Instead of using these 7 fixed valued Parameters at the time of calling, we could have avoided those by assigning those value much earlier in function formal parameter
- C++ allows us to do so through the mechanism called **Default parameters**

Program 08.01: Function with a default parameter


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

int IdentityFunction(int a = 10) { // Default value for the parameter
    return (a);
}

int main() {
    int x = 5, y;

    y = IdentityFunction(x);  // Usual function call
    cout << "y = " << y << endl;

    y = IdentityFunction();   // Uses default parameter
    cout << "y = " << y << endl;
}
----------
y = 5
y = 10
```

Program 08.02: Function with 2 default parameters


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

int Add(int a = 10, int b = 20) {
    return (a + b);
}
int main(){
    int x = 5, y = 6, z;

    z = Add(x, y); // Usual function call -- a = x = 5 & b = y = 6
    cout << "Sum = " << z << endl;

    z = Add(x);    // One parameter defaulted -- a = x = 5 & b = 20
    cout << "Sum = " << z << endl;

    z = Add();     // Both parameter defaulted -- a = 10 & b = 20
    cout << "Sum = " << z << endl;
}
----------
Sum = 11
Sum = 25
Sum = 30
```

Default Parameter: Highlighted Points

- C++ allows programmer to assign default values to the function parameters
- Default values are specified while prototyping the function
- Default parameters are required while calling functions with fewer arguments or without any argument
- Better to use default value for less used parameters
- Default arguments may be expressions also

Restrictions on default parameters

- All parameters to the right of a parameter with default argument must have default arguments (function `f`)
- Default arguments cannot be re-defined (function `g`)
- All non-defaulted parameters needed in a call (call `g()`)

```
#include <iostream>

void f(int, double = 0.0, char *);
// Error C2548: 'f': missing default parameter for parameter 3

void g(int, double = 0, char * = NULL); // OK
void g(int, double = 1, char * = NULL);
// Error C2572: 'g': redefinition of default parameter : parameter 3
// Error C2572: 'g': redefinition of default parameter : parameter 2

int main() {
    int i = 5; double d = 1.2; char c = 'b';

    g(); // Error C2660: 'g': function does not take 0 arguments
    g(i);
    g(i, d);
    g(i, d, &c);
    return 0;
}
```

Restrictions on default parameters

- Default parameters should be supplied only in a header file and not in the definition of a function

```
// Header file: myFunc.h
void g(int, double, char = 'a');
-------------------------------------------------
// Source File: myFunc.cpp
#include <iostream>
using namespace std;
#include "myFunc.h"

void g(int i, double d, char c) {
    cout << i << ' ' << d << ' ' << c << endl;
}
-------------------------------------------------
// Application File: Apps.cpp
#include <iostream>
#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 a
    g(i);         // Prints: 5 0 a
    g(i, d);      // Prints: 5 1.2 a
    g(i, d, c);   // Prints: 5 1.2 b
    return 0;
}
```

Function overloads: Matrix Multiplication in C

- Similar functions with different data types & algorithms

```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10];  } VecRow; // Row Vector
typedef struct { int data[10][1];  } VecCol; // Column Vector

void Multiply_M_M  (Mat a,    Mat b,    Mat* c)    { /* c = a * b */ }
void Multiply_M_VC (Mat a,    VecCol b, VecCol* c) { /* c = a * b */ }
void Multiply_VR_M (VecRow a, Mat b,    VecRow* c) { /* c = a * b */ }
void Multiply_VC_VR(VecCol a, VecRow b, Mat* c)    { /* c = a * b */ }
void Multiply_VR_VC(VecRow a, VecCol b, int* c)    { /* c = a * b */ }

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply_M_M  (m1, m2, &rm);  // rm  <-- m1 * m2
    Multiply_M_VC (m1, cv, &rcv); // rcv <-- m1 * cv
    Multiply_VR_M (rv, m2, &rrv); // rrv <-- rv * m2
    Multiply_VC_VR(cv, rv, &rm);  // rm  <-- cv * rv
    Multiply_VR_VC(rv, cv, &r);   // r   <-- rv * cv
    return 0;
}
```

- 5 multiplication functions share same functionality but different argument types
- C treats them as 5 separate functions
- C++ has an elegant solution

Function overloads: Matrix Multiplication in C++

- Functions having similar functionality but different in details.

```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10];  } VecRow; // Row Vector
typedef struct { int data[10][1];  } VecCol; // Column Vector

void Multiply(const Mat& a,    const Mat& b,    Mat& c)    { /* c = a * b */ };
void Multiply(const Mat& a,    const VecCol& b, VecCol& c) { /* c = a * b */ };
void Multiply(const VecRow& a, const Mat& b,    VecRow& c) { /* c = a * b */ };
void Multiply(const VecCol& a, const VecRow& b, Mat& c)    { /* c = a * b */ };
void Multiply(const VecRow& a, const VecCol& b, int& c)    { /* c = a * b */ };

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply(m1, m2, rm);  // rm  <-- m1 * m2
    Multiply(m1, cv, rcv); // rcv <-- m1 * cv
    Multiply(rv, m2, rrv); // rrv <-- rv * m2
    Multiply(cv, rv, rm);  // rm  <-- cv * rv
    Multiply(rv, cv, r);   // r   <-- rv * cv
    return 0;
}
```

- These 5 functions having different argument types are treated as one function (`Multiply`) in C++
- This is called **Function Overloading or Static Polymorphism**

Program 08.03/04: Function Overloading

- Define multiple functions having the same name
- Binding happens at compile time

Same # of Parameters

```
#include <iostream>
using namespace std;
int Add(int a, int b) { return (a + b); }
double Add(double c, double d) {
    return (c + d);
}
int main() {
    int x = 5, y = 6, z;
    z = Add(x, y);
        // int Add(int, int)
    cout << "int sum = " << z;

    double s = 3.5, t = 4.25, u;
    u = Add(s, t);
        // double Add(double, double)
    cout << "double sum = " << u << endl;

    return 0;
}
```

int sum = 11 double sum = 7.75

- Same Add function
- Same # of parameters but different types

Different # of Parameters

```
#include <iostream>
using namespace std;
int Area(int a, int b) { return (a * b); }
int Area(int c) {
    return (c * c);
}
int main(){
    int x = 10, y = 12, z = 5, t;
    t = Area(x, y);
        // int Add(int, int)
    cout << "Area of Rectangle = " << t;

    int z = 5, u;
    u = Area(z);
        // int Add(int)
    cout << " Area of Square = " << u << endl;

    return 0;
}
```

Area of Rectangle = 12 Area of Square = 25

- Same Area function
- Different # of parameters

Program 08.05: Restrictions in Function Overloading

- Two functions having the same signature but different return types cannot be overloaded

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

int    Area(int a, int b) { return (a * b); }
double Area(int a, int b) { return (a * b); }
// Error C2556: 'double Area(int,int)': overloaded function differs only by return type
//              from 'int Area(int,int)'
// Error C2371: 'Area': redefinition; different basic types

int main() {
    int x = 10, y = 12, z = 5, t;
    double f;

    t = Area(x, y);
    // Error C2568: '=': unable to resolve function overload
    // Error C3861: 'Area': identifier not found

    cout << "Multiplication = " << t << endl;

    f = Area(y, z); // Errors C2568 and C3861 as above
    cout << "Multiplication = " << f << endl;

    return 0;
}
```

Function Overloading – Summary of Rules

- The same function name may be used in several definitions
- Functions with the same name must have different number of formal parameters and/or different types of formal parameters
- Function selection is based on the number and the types of the actual parameters at the places of invocation
- Function selection (Overload Resolution) is performed by the compiler
- Two functions having the same signature but different return types will result in a compilation error due to *attempt to re-declare*
- Overloading allows **Static Polymorphism**

Overload Resolution

- To resolve overloaded functions with one parameter
  - Identify the set of *Candidate Functions*
  - From the set of candidate functions identify the set of *Viable Functions*
  - Select the *Best viable function* through (*Order is important*)
    - Exact Match
    - Promotion
    - Standard type conversion
    - User defined type conversion

Overload Resolution: Exact Match

- lvalue-to-rvalue conversion
  - Most common
- Array-to-pointer conversion

  Definitions:
  ```
  int ar[10];
  void f(int *a);
  ```
  Call:
  ```
  f(ar)
  ```
- Function-to-pointer conversion

  Definitions:
  ```
  typedef int (*fp) (int);
  void f(int, fp);
  int g(int);
  ```
  Call:
  ```
  f(5, g)
  ```
- Qualification conversion
  - Converting pointer (only) to `const` pointer

Overload Resolution: Promotion & Conversion

- Examples of Promotion
  - `char` to `int`; `float` to `double`
  - `enum` to `int` / `short` / `unsigned int` / ...
  - `bool` to `int`
- Examples of Standard Conversion
  - integral conversion
  - floating point conversion
  - floating point to integral conversion
    The above 3 may be dangerous!
  - pointer conversion
  - bool conversion

Example: Overload Resolution with one parameter

- In the context of a list of function prototypes:

```
int g(double);                    // F1
void f();                         // F2
void f(int);                      // F3
double h(void);                   // F4
int g(char, int);                 // F5
void f(double, double = 3.4);     // F6
void h(int, double);              // F7
void f(char, char *);             // F8
```

  The call site to resolve is:

```
f(5.6);
```

- Resolution:
  - Candidate functions (by name): F2, F3, F6, F8
  - Viable functions (by # of parameters): F3, F6
  - Best viable function (by type `double` – Exact Match): F6

Example: Overload Resolution fails

- Consider the overloaded function signatures:

```
int fun(float a) {...}              // Function 1
int fun(float a, int b) {...}       // Function 2
int fun(float x, int y = 5) {...} // Function 3

int main() {
    float p = 4.5, t = 10.5;
    int s = 30;

    fun(p, s); // CALL - 1
    fun(t);    // CALL - 2
    return 0;
}
```

- `CALL - 1`: Matches `Function 2` & `Function 3`
- `CALL - 2`: Matches `Function 1` & `Function 3`
- Results in ambiguity

Program 08.06/07: Default Parameter & Function Overload

- Compilers deal with default parameters as a special case of function overloading

Default Parameters

```
#include <iostream>
using namespace std;
int f(int a = 1, int b = 2);



int main() {
    int x = 5, y = 6;

    f();     // a = 1, b = 2
    f(x);    // a = x = 5, b = 2
    f(x, y); // a = x = 5, b = y = 6

    return 0;
}
```

- Function `f` has 2 parameters overloaded
- `f` can have 3 possible forms of call

Function Overload

```
#include <iostream>
using namespace std;
int f();
int f(int);
int f(int, int);

int main() {
    int x = 5, y = 6;

    f();     // int f();
    f(x);    // int f(int);
    f(x, y); // int f(int, int);

    return 0;
}
```

- Function `f` is overloaded with up to 3 parameters
- `f` can have 3 possible forms of call
- No overload here use default parameters

Program 08.08: Default Parameter & Function Overload

- Function overloading can use default parameter
- However, with default parameters, the overloaded functions should still be resolvable

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

int Area(int a, int b = 10) { return (a * b); }
double Area(double c, double d) { return (c * d); }

int main() {
    int x = 10, y = 12, t;
    double z = 20.5, u = 5.0, f;

    t = Area(x);    // Binds int Area(int, int = 10)
    cout << "Area = " << t << endl; // t = 100

    f = Area(z, y); // Binds double Area(double, double)
    cout << "Area = " << f << endl;  // f = 102.5

    return 0;
}
```

Program 08.09: Default Parameter & Function Overload

- Function overloading with default parameters may fail

```
#include <iostream>
using namespace std;
int f();
int f(int = 0);
int f(int, int);

int main() {
    int x = 5, y = 6;

    f();     // Error C2668: 'f': ambiguous call to overloaded function
             // More than one instance of overloaded function "f"
             // matches the argument list:
             //     function "f()"
             //     function "f(int = 0)"

    f(x);    // int f(int);
    f(x, y); // int f(int, int);

    return 0;
}
```

Module Summary

- Introduced the notion of Default parameters and discussed several examples
- Identified the necessity of function overloading
- Introduced static Polymorphism and discussed examples and restrictions
- Discussed an outline for Overload resolution
- Discussed the mix of default Parameters and function overloading

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, *Instructor*	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, *TA*	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, *TA*	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, *TA*	himadribhuyan@gmail.com	9438911655