Objectives & Outline

Default Parameters
- Examples
- Highlights
- Restrictions

Function Overloading
- Examples
- Restrictions
- Rules

Overload Resolution
- Exact Match
- Promotion & Conversion
- Examples
- Ambiguity

Defa. Parames in Overloading

Module Summary

Programming in Modern C++

Module M08: Default Parameters & Function Overloading

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Introduced reference in C++
- Studied the difference between call-by-value and call-by-reference
- Studied the difference between return-by-value and return-by-reference
- Discussed the difference between References and Pointers

Module Objectives

- Understand Default Parameters
- Understand Function Overloading and Resolution

Module Outline

1. Default Parameters
   - Examples
   - Highlights
   - Restrictions on default parameters
2. Function Overloading
   - Examples
   - Restrictions
   - Rules
3. Overload Resolution
   - Exact Match
   - Promotion & Conversion
   - Examples
   - Ambiguity
4. How to overload Default Parameter?
5. Module Summary

Default Parameters

Default Parameters

Motivation: Example `CreateWindow` in MSVC++

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 parameters in `CreateWindow()`
- Of these 11, 8 parameters (4 are `CWUSEDEFAULT`, 3 are `NULL`, and 1 is `hInstance`) usually get same values in most calls
- Instead of using these 8 fixed valued Parameters at call, we may assign the *values in 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 parameter a
    return (a);
}

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

    y = IdentityFunction(x);  // Usual function call. Actual parameter taken as x = 5
    cout << "y = " << y << endl;

    y = IdentityFunction();   // Uses default parameter. Actual parameter taken as 10
    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 violates)
- *Default arguments* **cannot be** *re-defined* (second signature of function g violates)
- *All non-defaulted parameters* **needed** in a *call* (first call of g() violates)

```
#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);
}
```

Restrictions on default parameters

- Default parameters to 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’); // Defaults ch
void g(int i, double f = 0.0, char ch); // A new overload. Defaults f & ch
void g(int i = 0, double f, char ch);   // A new overload. Defaults i, f & ch
// void g(int i = 0, double f = 0.0, char ch = ’a’); // Alternate signature. Defaults all in one go
----------------------------------------------------
// Source File
#include <iostream>
using namespace std;
#include "myFunc.h" // Defaults taken from header
void g(int i, double d, char c) { cout << i << ’ ’ << d << ’ ’ << c << endl; } // No defaults here
----------------------------------------------------
// Application File
#include <iostream>
#include "myFunc.h"
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
}
```

Function Overloading

Module M08

Partha Pratim Das

Objectives & Outline

Default Parameters

Examples

Highlights

Restrictions

Function Overloading

Examples

Restrictions

Rules

Overload Resolution

Exact Match

Promotion & Conversion

Examples

Ambiguity

Defa. Parames in Overloading

Module Summary

Function Overloading

Function overloads: Matrix Multiplication in C

- *Similar functions* with *different* **data types** and **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 *similar functionality* but *different argument types*
- C treats them by 5 different function names. Makes it difficult for the user to remember and use
- **C++ has an elegant solution**

Function overloads: Matrix Multiplication in C++

- Functions *having the same* **name**, *similar functionality* but *different* **algorithms**, and identified by *different interfaces* **data types**

```
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 represented as *one function name* (`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;
}
```

int sum = 11 double sum = 7.75

- Same Add function to add two ints or two doubles
- 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 Area(int, int)
    cout << "Area of Rectangle = " << t;

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

Area of Rectangle = 12 Area of Square = 25

- Same Area function for *rectangle*s and for squares
- *Different number 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;
}
```

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

Module M08

Partha Pratim Das

Objectives & Outline

Default Parameters

Examples

Highlights

Restrictions

Function Overloading

Examples

Restrictions

Rules

Overload Resolution

Exact Match

Promotion & Conversion

Examples

Ambiguity

Defa. Parames in Overloading

Module Summary

Overload Resolution

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*: Read the value from an object
  - 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

- **Promotion**
  - Objects of an integral type can be converted to another wider integral type, that is, a type that can represent a larger set of values. This widening type of conversion is called *integral promotion*
  - C++ promotions are *value-preserving*, as the value after the promotion is guaranteed to be the same as the value before the promotion
  - Examples
    - ▷ `char` to `int`; `float` to `double`
    - ▷ `enum` to `int` / `short` / `unsigned int` / ...
    - ▷ `bool` to `int`

Overload Resolution: Promotion & Conversion

- **Standard Conversions**
  - *Integral conversions* between *integral types* – `char`, `short`, `int`, and `long` with or without qualifiers `signed` or `unsigned`
  - *Floating point Conversions* from *less precise floating type* to a *more precise floating type* like `float` to `double` or `double` to `long double`. Conversion can happen to a *less precise* type, if it is in a range representable by that type
  - *Conversions between integral and floating point types*: Certain expressions can cause objects of floating type to be converted to integral types, or vice versa. **May be dangerous!**
    - ▷ When an object of *integral type* is converted to a *floating type*, and the original value is not representable exactly, the result is either the next higher or the next lower representable value
    - ▷ When an object of *floating type* is converted to an *integral type*, the fractional part is truncated, or rounded toward zero. A number like 1.3 is converted to 1, and -1.3 is converted to -1
  - *Pointer Conversions*: Pointers can be converted during assignment, initialization, comparison, and other expressions
  - *Bool Conversion*: `int` to `bool` or vice versa based on the context

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: Ambiguity in Overload Resolution

- 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 for both calls

Default Parameters in Overloading

Default Parameters in Overloading

Program 08.06/07: Default Parameter & Function Overload

- Compilers deal with *default parameters* as a special case of *function overloading*
- These need to be mixed carefully

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
}
```

- Function f has 2 parameters defaulted
- 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);
}
```

- Function f is overloaded with up to 2 parameters
- f can have 3 possible forms of call
- *No overload* here use *default parameters*. Can it?

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;
// Overloaded Area functions
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; // Area = 100

    t = Area(x, y);     // Binds int Area(int, int = 10)
    cout << "Area = " << t << endl; // Area = 120

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

    f = Area(z); // Binds int Area(int, int = 10)
    cout << "Area = " << f << endl; // Area = 200

    // Un-resolvable between int Area(int a, int b = 10) and  double Area(double c, double d)
    f = Area(z, y); // Error: call of overloaded Area(double&, int&) is ambiguous
}
```

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