**Function Overloading**

# Polymorphism

Polymorphism means one name many forms. It has two types

1. Static Polymorphism
2. Dynamic Polymorphism

Static polymorphism is achieved using method overloading.

Dynamic polymorphism using method overriding. It is closely related to inheritance. We can write a code that works on the superclass, and it will work with any subclass type as well.

This chapter deals with Static polymorphism.

# Function overloading

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.

Function overloading can be considered as an example of polymorphism feature in C++.

**Why Overloading necessary?**

To reduce extra work on you as you write the program, and on readers as they try to understand it.

In C++, another factor forces the overloading of function names: the constructor. Constructor can have same name as Class, but to initialize an object we might have different way, so, it forces to overload it.

Following is a simple C++ example to demonstrate function overloading.

#include <iostream>

using namespace std;

void print(int i) { cout << "int " << i << '\n'; }

void print(double f) { cout << "float " << f << '\n'; }

void print(char\* c) { cout << "char\* " << c << '\n'; }

int main() {

print(10);

print(10.10);

print("ten");

return 0;

}

Output:

int 10

float 10.1

char\* ten

# Overloading on return values

Overloading on return values is **not allowed.**

void f();

int f();

This works fine when the compiler can unequivocally determine the meaning from the context, as in int x = f( );.

You ignore the return value (that is, you can call the function for its side effects). How can the compiler distinguish which call is meant in this case?

Possibly worse is the difficulty the reader has in knowing which function call is meant. Overloading solely on return value is a bit too subtle, and thus isn’t allowed in C++.

# Type-safe linkage

Consider the following scenario. In one file is the definition for a function

// Function definition

void f(int) {}

In the second file, the function is mis declared and then called

// Function misdeclaration

void f(char);

int main() {

f(1); // Causes a linker error

}

Even though you can see that the function is actually f(int), the compiler doesn’t know this because it was told – through an explicit declaration – that the function is f(char). Thus, the compilation is successful. In C, the linker would also be successful, but not in C++.

Because the compiler decorates the names, the definition becomes something like f\_int, whereas the use of the function is f\_char. When the linker tries to resolve the reference to f\_char, it can only find f\_int, and it gives you an error message. This is type-safe linkage. Although the problem doesn’t occur all that often, when it does it can be incredibly difficult to find, especially in a large project. This is one of the cases where you can easily find a difficult error in a C program simply by running it through the C++ compiler.

# Default arguments

A default argument is a value given in the declaration that the compiler automatically inserts if you don’t provide a value in the function call. In the Stash example, we can replace the two functions:

Stash(int size); // Zero quantity

Stash(int size, int initQuantity);

with the single function

Stash(int size, int initQuantity = 0);

Now, the two object definitions will produce exactly the same results

Stash A(100), B(100, 0);

There are two rules you must be aware of when using default arguments.

1. Only trailing arguments may be defaulted. That is, you can’t have a default argument followed by a non-default argument.
2. Once you start using default arguments in a particular function call, all the subsequent arguments in that function’s argument list must be defaulted (this follows from the first rule).

**Default arguments are only placed in the declaration of a function (typically placed in a header file).** The compiler must see the default value before it can use it. Sometimes people will place the commented values of the default arguments in the function definition, for documentation purposes

void fn(int x /\* = 0 \*/) { } // This is wrong way for default argument declaration

# Placeholder arguments

Arguments in a function declaration can be declared without identifiers. When these are used with default arguments, it can look a bit funny. You can end up with

void f(int x, int = 0, float = 1.1);

In C++ you don’t need identifiers in the function definition, either:

void f(int x, int, float flt) { /\* ... \*/ }

In the function body, x and flt can be referenced, but not the middle argument, because it has no name. Function calls must still provide a value for the placeholder, though: f(1) or f(1,2,3.0).

# Choosing overloading vs. default arguments

You **shouldn’t use a default argument as a flag upon which to conditionally execute code**. You should instead break the function into two or more overloaded functions if you can. A default argument should be a value you would ordinarily put in that position. It’s a value that is more likely to occur than all the rest, so client programmers can generally ignore it or use it only if they want to change it from the default value.

The default argument is included to make function calls easier, especially when those functions have many arguments with typical values. Not only is it much easier to write the calls, it’s easier to read them, especially if the class creator can order the arguments so the least-modified defaults appear latest in the list.

An especially important use of default arguments is when you start out with a function with a set of arguments, and after it’s been used for a while, you discover you need to add arguments. By defaulting all the new arguments, you ensure that all client code using the previous interface is not disturbed.

# Functions that cannot be overloaded in C++

In C++, following function declarations cannot be overloaded.

## Function declarations that differ only in the return type

#include <iostream>

using namespace std;

int foo() { return 10; }

string foo() { return "a"; }

int main() {

int x = foo();

return 0;

}

Compilation Error:

<source>:4:8: error: ambiguating new declaration of 'std::string foo()'

4 | string foo() { return "a"; }

| ^~~

<source>:3:5: note: old declaration 'int foo()'

3 | int foo() { return 10; }

| ^~~

## Function with same name and static and non-static

int foo() { return 10; }

// error: redefinition of 'int foo()'

// static int foo() { return 10; }

class Test {

static void fun(int i) {}

// error: 'void Test::fun(int)' cannot be overloaded with

// 'static void Test::fun(int)'

// void fun(int i) {}

};

int main() { return 0; }

## Declarations that differ only in a pointer \* and an array []

Parameter declarations that differ only in a pointer \* versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration. Only the second and subsequent array dimensions are significant in parameter types.

int foo(int \* ptr) { return 10; }

// error: redefinition of 'int foo(int\*)'

// int foo(int ptr[]) { return 10; }

## Declarations that differ only in a function type and pointer to the same function type

Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.

int foo(int (\*ptr)()) { return 10; }

// error: redefinition of 'int foo(int (\*)())'

// int foo(int ()) { return 10; }

## Declarations that differ only in the presence or absence of const and/or volatile

Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the const and volatile type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called.

int foo(int x) { return 10; }

// error: redefinition of 'int foo(int)'

// int foo(const int x) { return 10; }

// error: redefinition of 'int foo(int)'

// int foo(volatile int x) { return 10; }

But C++ allows functions to be overloaded on the basis of const-ness of parameters only if the const parameter is a reference or a pointer.

Check [What about parameters?](#_What_about_parameters?)

int foo\_ref(int & x) { return 10; }

int foo\_ref(const int & x) { return 10; }

int foo\_pointer(int \* x) { return 10; }

int foo\_pointer(const int \* x) { return 10; }

Only the const and volatile type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; const and volatile type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations. In particular, for any type T, “pointer to T,” “pointer to const T,” and “pointer to volatile T” are considered distinct parameter types, as are “reference to T,” “reference to const T,” and “reference to volatile T.” For example, see the example in [this comment](https://www.geeksforgeeks.org/archives/9707/comment-page-1#comment-3319) posted by Venki.

## Declarations that differ only in their default arguments

Two parameter declarations that differ only in their default arguments are equivalent.

int foo(int x) { return 10; }

// error: redefinition of 'int foo(int)'

// int foo(int x = 5) { return 10; }

# Function overloading and const keyword

C++ allows member methods to be overloaded on the basis of const type. Overloading on the basis of const type can be useful when a function return reference or pointer. We can make one function const, that returns a const reference or const pointer, other non-const function, that returns non-const reference or pointer. Check [const-overloading](https://isocpp.org/wiki/faq/const-correctness#const-overloading) for more details.

#include <iostream>

using namespace std;

class ConstOverload {

public:

void fun() const { cout << "fun() const called \n"; }

void fun() { cout << "fun() called \n"; }

};

int main() {

ConstOverload t1; t1.fun();

const ConstOverload t2; t2.fun();

return 0;

}

Output:

fun() called

fun() const called

## What about parameters?

C++ allows functions to be overloaded on the basis of const-ness of parameters only if the const parameter is a reference or a pointer.

In program 1, the parameter ‘i’ is passed by value, so ‘i’ in fun() is a copy of ‘i’ in main(). Hence fun() cannot modify ‘i’ of main(). Therefore, it doesn’t matter whether ‘i’ is received as a const parameter or normal parameter. When we pass by reference or pointer, we can modify the value referred or pointed, so we can have two versions of a function, one which can modify the referred or pointed value, other which cannot.

Rules related to const parameters are interesting. The program 1 fails in compilation, but program 2 compiles and runs fine.

// PROGRAM 1 (Fails in compilation)

#include<iostream>

using namespace std;

void fun(const int i) { cout << "fun(const int) called "; }

void fun(int i) { cout << "fun(int ) called " ; }

int main() {

const int i = 10;

fun(i);

return 0;

}

// Compiler Error: redefinition of 'void fun(int)'

// PROGRAM 2 (Compiles and runs fine)

#include <iostream>

using namespace std;

void fun\_ptr(int\* a) {

\*a = 3;

cout << "non-const pointer parameter " << \*a << '\n';

}

void fun\_ptr(const int\* a) {

//\*a = 3; // error: assignment of read-only location '\* a'

cout << "const pointer parameter " << \*a << '\n';

}

void fun\_ref(int& a) {

a = 3;

cout << "non-const ref parameter " << a << '\n';

}

void fun\_ref(const int& a) {

//a = 3; // error: assignment of read-only reference 'a'

cout << "const ref parameter " << a << '\n';

}

int main() {

int val = 10; fun\_ptr(&val); fun\_ref(val);

const int cval = 10; fun\_ptr(&cval); fun\_ref(cval);

return 0;

}

Output:

non-const pointer parameter 3

non-const ref parameter 3

const pointer parameter 10

const ref parameter 10

# Does overloading work with Inheritance?

Overloading doesn’t work for derived class in C++ programming language. There is **no overload resolution between Base and Derived**. The compiler looks into the scope of Derived, finds the single function “double f(double)” and calls it. It never disturbs with the (enclosing) scope of Base. In C++, there is no overloading across scopes – derived class scopes are not an exception to this general rule. (See [this](https://www.geeksforgeeks.org/hiding-of-all-overloaded-methods-with-same-name-in-base-class-in-cpp/)for more examples)

Like C++, there is no overload resolution between class Base and class Derived. In C#, there is no overloading across scopes derived class scopes are not an exception to this general rule. This is same as C++ because C# is designed to be much closer to C++, according to the [Anders hejlsberg](http://en.wikipedia.org/wiki/Anders_Hejlsberg) the creator of C# language.

**In Java overloading works across scopes contrary to C++.** Java compiler determines correct version of the overloaded method to be executed at compile time based upon the type of argument used to call the method and parameters of the overloaded methods of both these classes receive the values of arguments used in call and executes the overloaded method.

#include <iostream>

using namespace std;

class Base {

public:

int fun() { cout<<"Base::fun() called"; }

int fun(int i) { cout<<"Base::fun(int i) called"; }

};

class Derived: public Base {

public:

int fun() { cout<<"Derived::fun() called"; }

};

int main() {

Derived d;

d.fun(5); // CE error: no matching function for call to 'Derived::fun(int)'

return 0;

}

// CE error: no matching function for call to 'Derived::fun(int)'

#include <iostream>

using namespace std;

class Base {

public:

int fun(int i) { cout<<"Base::fun(int i) called" << endl; }

};

class Derived: public Base {

public:

int fun(double i) { cout<<"Derived::fun(double i) called" << endl; }

};

int main() {

Derived d;

d.fun(5);

d.fun(53.7);

d.fun('A');

Base b;

b.fun(5);

b.fun(53.7);

b.fun('A');

return 0;

}

Output:

Derived::fun(double i) called

Derived::fun(double i) called

Derived::fun(double i) called

Base::fun(int i) called

Base::fun(int i) called

Base::fun(int i) called

# Resolve function call ambiguity with function pointer

Function pointer can be used to eliminate issue of ambiguous function call using object.

#include <iostream>

using namespace std;

class AmbiguousOverload {

public:

int fun(double i) { cout << "AmbiguousOverload::fun(double i) called\n"; }

int fun(char ch) { cout << "AmbiguousOverload::char(char ch) called\n"; }

};

int main() {

AmbiguousOverload b;

//b.fun(5); // error: call of overloaded 'fun(int)' is ambiguous

int (AmbiguousOverload::\*bfptr)(double) = &AmbiguousOverload::fun;

(b.\*bfptr)(5);

b.fun(53.7);

b.fun('A');

return 0;

}

Output:

AmbiguousOverload::fun(double i) called

AmbiguousOverload::fun(double i) called

AmbiguousOverload::char(char ch) called

# Can main() be overloaded in C++?

Predict the output of following C++ program.

#include <iostream>

using namespace std;

int main(int a) { cout << a << "\n"; return 0; }

int main(char \*a) { cout << a << endl; return 0; }

int main(int a, int b) { cout << a << " " << b; return 0; }

int main() {

main(3);

main("C++");

main(9, 6);

return 0;

}

// CE: conflicting declaration of C function 'int main(int, int)'

To overload main() function in C++, it is necessary to use class and declare the main as member function.

Note that main is not reserved word in programming languages like C, C++, Java and C#.

For example, we can declare a variable whose name is main, try below example:

#include <iostream>

int main() {

int main = 10;

std::cout << main;

return 0;

}

Output:

10

The following program shows overloading of main() function in a class.

#include <iostream>

using namespace std;

class Test {

public:

int main(int s) { cout << s << "\n"; return 0; }

int main(char \*s) { cout << s << endl; return 0; }

int main(int s ,int m) { cout << s << " " << m; return 0; }

};

int main() {

Test obj;

obj.main(3);

obj.main("I love C++");

obj.main(9, 6);

return 0;

}

Ouput:

3

I love C++

9 6

# Ambiguous Function Overloading

Although polymorphism is a widely useful phenomena in C++ yet it can be quite complicated at times. For instance, consider the following code snippet:

#include <iostream>

using namespace std;

void test(float s,float t) { cout << "Function with float called "; }

void test(int s, int t) { cout << "Function with int called "; }

int main() {

test(3.5, 5.6);

return 0;

}

// CE: call of overloaded 'test(double, double)' is ambiguous

It may appear that the call to the function test in main() will result in output “Function with float called” but the code gives above error.

The reason behind the ambiguity in above code is that the floating literals 3.5 and 5.6 are actually treated as double by the compiler. As per C++ standard, floating point literals (compile time constants) are treated as double unless explicitly specified by a suffix (See 2.14.4 of C++ standard [here](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf)). Since compiler could not find a function with double argument and got confused if the value should be converted from double to int or float.

Rectifying the error: We can simply tell the compiler that the literal is a float and not double by providing suffix f.

test(3.5f, 5.6f);

Output:

Function with float called

# References

Thinking in C++, Volume 1, 2nd Edition

Chapter 07: Function Overloading & Default Arguments

Chapter 12: Operator Overloading

<https://www.geeksforgeeks.org/c-plus-plus/#FunctionOverloading>

Chapter 8 Type Conversion and Function Overloading | <https://www.learncpp.com/>