**Functions that cannot be overloaded in C++**

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

1) Function declarations that differ only in the return type. For example, the following program fails in compilation.

|  |
| --- |
| #include<iostream>  int foo() {    return 10;  }    char foo() {    return 'a';  }    int main()  {     char x = foo();     getchar();     return 0;  } |

2) Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration. For example, following program fails in compilation.

|  |
| --- |
| #include<iostream>  class Test {     static void fun(int i) {}     void fun(int i) {}  };    int main()  {     Test t;     getchar();     return 0;  } |

3) 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. For example, following two function declarations are equivalent.

|  |
| --- |
| int fun(int \*ptr);  int fun(int ptr[]); // redeclaration of fun(int \*ptr) |

4) 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.

|  |
| --- |
| void h(int ());  void h(int (\*)()); // redeclaration of h(int()) |

5) 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. For example, following program fails in compilation with error *“redefinition of `int f(int)’ “*

Example:

|  |
| --- |
| #include<iostream>  #include<stdio.h>    using namespace std;    int f ( int x) {      return x+10;  }    int f ( const int x) {      return x+10;  }    int main() {    getchar();    return 0;  } |

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.

6) Two parameter declarations that differ only in their default arguments are equivalent. For example, following program fails in compilation with error *“redefinition of `int f(int, int)’ “*

|  |
| --- |
| #include<iostream>  #include<stdio.h>    using namespace std;    int f ( int x, int y) {      return x+10;  }    int f ( int x, int y = 10) {      return x+y;  }    int main() {    getchar();    return 0;  } |

References:  
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf>

**Function overloading and const keyword**

Predict the output of following C++ program.

|  |
| --- |
| #include<iostream>  using namespace std;    class Test  {  protected:      int x;  public:      Test (int i):x(i) { }      void fun() const      {          cout << "fun() const called " << endl;      }      void fun()      {          cout << "fun() called " << endl;      }  };    int main()  {      Test t1 (10);      const Test t2 (20);      t1.fun();      t2.fun();      return 0;  } |

Output: The above program compiles and runs fine, and produces following output.

fun() called

fun() const called

The two methods ‘void fun() const’ and ‘void fun()’ have same signature except that one is const and other is not. Also, if we take a closer look at the output, we observe that, ‘const void fun()’ is called on const object and ‘void fun()’ is called on non-const object.  
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. See [this](http://www.parashift.com/c++-faq-lite/const-overloading.html) for more details.

**What about parameters?**  
Rules related to const parameters are interesting. Let us first take a look at following two examples. 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;  } |

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

|  |
| --- |
| // PROGRAM 2 (Compiles and runs fine)  #include<iostream>  using namespace std;    void fun(char \*a)  {    cout << "non-const fun() " << a;  }    void fun(const char \*a)  {    cout << "const fun() " << a;  }    int main()  {    const char \*ptr = "GeeksforGeeks";    fun(ptr);    return 0;  } |

Output:

const fun() GeeksforGeeks

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. That is why the program 1 failed in compilation, but the program 2 worked fine. This rule actually makes sense. 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.

As an exercise, predict the output of following program.

|  |
| --- |
| #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;  } |

**Does overloading work with Inheritance?**

If we have a function in base class and a function with same name in derived class, can the base class function be called from derived class object? This is an interesting question and as an experiment predict the output of the following **C++** program.

|  |
| --- |
| #include <iostream>  using namespace std;  class Base  {  public:      int f(int i)      {          cout << "f(int): ";          return i+3;      }  };  class Derived : public Base  {  public:      double f(double d)      {          cout << "f(double): ";          return d+3.3;      }  };  int main()  {      Derived\* dp = new Derived;      cout << dp->f(3) << '\n';      cout << dp->f(3.3) << '\n';      delete dp;      return 0;  } |

The output of this program is:

f(double): 6.3

f(double): 6.6

Instead of the supposed output:

f(int): 6

f(double): 6.6

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/g-fact-89/) for more examples)

**Question**

How does C++ compiler differs between overloaded postfix and prefix operators?

|  |  |
| --- | --- |
| A | C++ doesn't allow both operators to be overlaoded in a class |
| B | A postfix ++ has a dummy parameter |
| C | A prefix ++ has a dummy parameter |
| D | By making prefix ++ as a global function and postfix as a member function. |

Explanation:

See the following example:

class Complex

{

private:

int real;

int imag;

public:

Complex(int r, int i) { real = r; imag = i; }

Complex operator ++(int);

Complex & operator ++();

};

Complex &Complex::operator ++()

{

real++; imag++;

return \*this;

}

Complex Complex::operator ++(int i)

{

Complex c1(real, imag);

real++; imag++;

return c1;

}

int main()

{

Complex c1(10, 15);

c1++;

++c1;

return 0;

}

|  |
| --- |
|  |
| Question 6 | |

Predict the output

|  |
| --- |
| #include<iostream>  using namespace std;  class A  {      int i;  public:      A(int ii = 0) : i(ii) {}      void show() {  cout << i << endl;  }  };    class B  {      int x;  public:      B(int xx) : x(xx) {}      operator A() const {  return A(x); }  };    void g(A a)  {      a.show();  }    int main()  {      B b(10);      g(b);      g(20);      return 0;  } |
|  |

|  |  |
| --- | --- |
| A | Compiler Error |
| B | 10  20 |
| C | 20  20 |
| D | 10  10 |

Explanation:

Note that the class B has as conversion operator overloaded, so an object of B can be converted to that of A. Also, class A has a constructor which can be called with single integer argument, so an int can be converted to A.

|  |
| --- |
| Question 8 |

Predict the output?

|  |  |
| --- | --- |
| #include<stdlib.h>  #include<stdio.h>  #include<iostream>    using namespace std;    class Test {      int x;  public:      void\* operator new(size\_t size);      void operator delete(void\*);      Test(int i) {          x = i;          cout << "Constructor called n";      }      ~Test() { cout << "Destructor called n"; }  };      void\* Test::operator new(size\_t size)  {      void \*storage = malloc(size);      cout << "new called n";      return storage;  }    void Test::operator delete(void \*p )  {      cout<<"delete called n";      free(p);  }    int main()  {      Test \*m = new Test(5);      delete m;      return 0;  } | |
| A | new called  Constructor called  delete called  Destructor called |
| B | new called  Constructor called  Destructor called  delete called |
| C | Constructor called  new called  Destructor called  delete called |
| D | Constructor called  new called  delete called  Destructor called |

Explanation:

Consider the following statement

Test \*ptr = new Test;

There are actually two things that happen in the above statement--memory allocation and object construction; the **new keyword** is responsible for both. One step in the process is to call **operator new** in order to allocate memory; the other step is to actually invoke the constructor. **Operator new** only allows us to change the memory allocation method, but does not do anything with the constructor calling method. **Keyword new is responsible for calling the constructor, not operator new.**

|  |
| --- |
| Question 9 |

|  |  |
| --- | --- |
| #include<iostream>  using namespace std;    class Point {  private:    int x, y;  public:    Point() : x(0), y(0) { }    Point& operator()(int dx, int dy);    void show() {cout << "x = " << x << ", y = " << y; }  };    Point& Point::operator()(int dx, int dy)  {      x = dx;      y = dy;      return \*this;  }    int main()  {    Point pt;    pt(3, 2);    pt.show();    return 0;  } | |
| A | x = 3, y = 2 |
| B | Compiler Error |
| C | x = 2, y = 3 |

Explanation:

This a simple example of function call operator overloading. The function call operator, when overloaded, does not modify how functions are called. Rather, it modifies how the operator is to be interpreted when applied to objects of a given type. If you overload a function call operator for a class its declaration will have the following form:

return\_type operator()(parameter\_list)

|  |
| --- |
| Question 10 |

Which of the following operator functions cannot be global, i.e., must be a member function.

|  |  |
| --- | --- |
| A | new |
| B | delete |
| C | Converstion Operator |
| D | All of the above |

Explanation:

new and delete can be global, see following example.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #include<stdlib.h>  #include<stdio.h>  #include<iostream>    using namespace std;    class Myclass {      int x;  public:      friend void\* operator new(size\_t size);      friend void operator delete(void\*);      Myclass(int i) {          x = i;          cout << "Constructor called \n";      }      ~Myclass() { cout << "Destructor called \n"; }  };      void\* operator new(size\_t size)  {      void \*storage = malloc(size);      cout << "new called \n";      return storage;  }    void operator delete(void \*p )  {      cout<<"delete called \n";      free(p);  }    int main()  {      Myclass \*m = new Myclass(5);      delete m;      return 0;  }  Which of the following operator(s) cannot be overloaded?   |  |  | | --- | --- | | A | . (Member Access or Dot operator) | | B | ?: (Ternary or Conditional Operator ) | | C | :: (Scope Resolution Operator) | | D` | All of the above | |

**Example 1 : Overloading ++ operator**

|  |
| --- |
| // CPP program to illustrate  // operators that can be overloaded  #include <iostream>  using namespace std;    class overload {  private:      int count;    public:      overload()          : count(4)      {      }        void operator++()      {          count = count + 1;      }      void Display()      {          cout << "Count: " << count;      }  };    int main()  {      overload i;      // this calls "function void operator ++()" function      ++i;      i.Display();      return 0;  } |

**Output:**

Count: 5

**Example 2: Overloading this [ ] operator**

|  |
| --- |
| #include <iostream>  using namespace std;  class overload {      int a[3];    public:      overload(int i, int j, int k)      {          a[0] = i;          a[1] = j;          a[2] = k;      }      int operator[](int i)      {          return a[i];      }  };  int main()  {      overload ob(1, 2, 3);      cout << ob[1]; // displays 2      return (0);  } |

**Output:**

2

**Example 3 : Overloading -> operator**

|  |
| --- |
| // CPP program to illustrate  // operators that can be overloaded  #include <bits/stdc++.h>  using namespace std;    class GFG {  public:      int num;      GFG(int j)      {          num = j;      }      GFG\* operator->(void)      {          return this;      }  };    // Driver code  int main()  {      GFG T(5);      GFG\* Ptr = &T;        // Accessing num normally      cout << "T.num = " << T.num << endl;        // Accessing num using normal object pointer      cout << "Ptr->num = " << Ptr->num << endl;        // Accessing num using -> operator      cout << "T->num = " << T->num << endl;        return 0;  } |

**Output :**

T.num = 5

Ptr->num = 5

T->num = 5

**List of operators that cannot be overloaded**

1> [Scope Resolution Operator](https://www.geeksforgeeks.org/scope-resolution-operator-in-c/) (::)

2> Pointer-to-member Operator (.\*)

3> Member Access or Dot operator (.)

4> [Ternary or Conditional Operator](https://www.geeksforgeeks.org/cc-ternary-operator-some-interesting-observations/) (?:)

5> Object size Operator ([sizeof](https://www.geeksforgeeks.org/sizeof-operator-c/))

6> Object type Operator (typeid)