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 can not.

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;  } |

Output:

**fun(const int &) called**