C-Plus-Plus Archive

Write a C program that won't compile in C++

Although C++ is designed to have backward compatibility with C there can be many C programs that would produce compiler error when compiled with a C++ compiler. Following are some of them.

1) In C++, it is a compiler error to call a function before it is declared. But in C, it may compile (See http://www.geeksforgeeks.org/g-fact-95/)

```
#include<stdio.h>
int main()
{
  foo(); // foo() is called before its declaration/definition
}

int foo()
{
  printf("Hello");
  return 0;
}
```

2) In C++, it is compiler error to make a normal pointer to point a const variable, but it is allowed in C. (See Const Qualifier in C)

```
#include <stdio.h>

int main(void)
{
   int const j = 20;

   /* The below assignment is invalid in C++, results in error
      In C, the compiler *may* throw a warning, but casting is
      implicitly allowed */
   int *ptr = &j;  // A normal pointer points to const

   printf("*ptr: %d\n", *ptr);

   return 0;
}
```

3) In C, a void pointer can directly be assigned to some other pointer like int *, char *. But in C++, a void pointer must be explicitly typcasted.

```
#include <stdio.h>
int main()
{
  void *vptr;
  int *iptr = vptr; //In C++, it must be replaced with int *iptr=(int *)vptr;
  return 0;
}
```

This is something we notice when we use malloc(). Return type of malloc() is void *. In C++, we must explicitly typecast return value of malloc() to appropriate type, e.g., "int *p = (void *)malloc(sizeof(int))". In C, typecasting is not necessary.

4) Following program compiles & runs fine in C, but fails in compilation in C++. const variable in C++ must be initialized but in c it isn't necessary. Thanks to Pravasi Meet for suggesting this point.

```
#include <stdio.h>
int main()
{
   const int a;   // LINE 4
   return 0;
}
```

```
Line 4 [Error] uninitialized const 'a' [-fpermissive]
```

5) This is the worst answer among all, but still a valid answer. We can use one of the C++ specific keywords as variable names. The program won't compile in C++, but would compiler in C.

```
#include <stdio.h>
int main(void)
{
   int new = 5;  // new is a keyword in C++, but not in C
   printf("%d", new);
}
```

Similarly, we can use other keywords like delete, explicit, class, .. etc.

6) C++ does more strict type checking than C. For example the following program compiles in C, but not in C++. In C++, we get compiler error "invalid conversion from 'int' to 'char*'". Thanks to Pravasi Meet for adding this point.

```
#include <stdio.h>
int main()
{
   char *c = 333;
   printf("c = %u", c);
   return 0;
}
```

7) A C/C++ Function Call Puzzle

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-puzzle

Name Mangling and extern "C" in C++

C++ supports function overloading, i.e., there can be more than one functions with same name and differences in parameters. How does C++ compiler distinguishes between different functions when it generates object code – it changes names by adding information about arguments. This technique of adding additional information to function names is called Name Mangling. C++ standard doesn't specify any particular technique for name mangling, so different compilers may append different information to function names.

Consider following declarations of function *f()*

```
int  f (void) { return 1; }
int  f (int)  { return 0; }
void g (void) { int i = f(), j = f(0); }
```

A C++ compiler may mangle above names to following (Source: Wiki)

```
int  __f_v (void) { return 1; }
int  __f_i (int)  { return 0; }
void __g_v (void) { int i = __f_v(), j = __f_i(0); }
```

How to handle C symbols when linking from C++?

In C, names may not be mangled as C doesn't support function overloading. So how to make sure that name of a symbol is not changed when we link a C code in C++. For example, see the following C++ program that uses printf() function of C.

```
// Save file as .cpp and use C++ compiler to compile it
int printf(const char *format,...);

int main()
{
   printf("GeeksforGeeks");
   return 0;
```

```
}
```

Output:

```
undefined reference to `printf(char const*, ...)'
      ld returned 1 exit status
```

The reason for compiler error is simple, name of *printf* is changed by C++ compiler and it doesn't find definition of the function with new name.

The solution of problem is extern "C" in C++. When some code is put in extern "C" block, the C++ compiler ensures that the function names are unmangled – that the compiler emits a binary file with their names unchanged, as a C compiler would do.

If we change the above program to following, the program works fine and prints "GeeksforGeeks" on console.

```
// Save file as .cpp and use C++ compiler to compile it
extern "C"
{
   int printf(const char *format,...);
}

int main()
{
   printf("GeeksforGeeks");
   return 0;
}
```

Output:

```
GeeksforGeeks
```

Therefore, all C style header files (stdio.h, string.h, .. etc) have their declarations in extern "C" block.

```
#ifdef __cplusplus
extern "C" {
#endif
   /* Declarations of this file */
#ifdef __cplusplus
}
#endif
```

Following are main points discussed above

1. Since C++ supports function overloading, additional information has to be added to function names (called name mangling) to avoid conflicts in binary code.

2. Function names may not be changed in C as C doesn't support function overloading. To avoid linking problems, C++ supports extern "C" block. C++ compiler makes sure that names inside extern "C" block are not changed.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
C++
Extern "C"

How does "void *" differ in C and C++?

C allows a void* pointer to be assigned to any pointer type without a cast, whereas C++ does not; this idiom appears often in C code using malloc memory allocation. For example, the following is valid in C but not C++:

```
void* ptr;
int *i = ptr; /* Implicit conversion from void* to int* */
```

or similarly:

```
int *j = malloc(sizeof(int) * 5); /* Implicit conversion from void* to int* */
```

In order to make the code compile in both C and C++, one must use an explicit cast:

```
void* ptr;
int *i = (int *) ptr;
int *j = (int *) malloc(sizeof(int) * 5);
```

Source:
http://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
GFacts

Write a program that produces different results in C and C++

Write a program that compiles and runs both in C and C++, but produces different results when compiled by C and C++ compilers.

There can be many such programs, following are some of them.

1) Character literals are treated differently in C and C++. In C character literals like 'a', 'b', ..etc are treated as integers, while as characters in C++. (See this for details)

For example, the following program produces sizeof(int) as output in C, but sizeof(char) in C++.

```
#include<stdio.h>
int main()
{
  printf("%d", sizeof('a'));
  return 0;
}
```

2) In C, we need to use struct tag whenever we declare a struct variable. In C++, the struct tag is not necessary. For example, let there be a structure for *Student*. In C, we must use '*struct Student*' for *Student* variables. In C++, we can omit struct and use '*Student*' only. Following is a program that is based on the fact and produces different outputs in C and C++. It prints sizeof(int) in C and sizeof(struct T) in C++.

```
#include <stdio.h>
int T;

int main()
{
   struct T { double x; };  // In C++, this T hides the global variable T,
                            // but not in C
   printf("%d", sizeof(T));
   return 0;
}
```

3) Types of boolean results are different in C and C++. Thanks to Gaurav Jain for suggesting this point.

```
// output = 4 in C (which is size of int)
printf("%d", sizeof(1==1));

// output = 1 in c++ (which is the size of boolean datatype)
cout << sizeof(1==1);
```

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-puzzle

Type difference of character literals in C and C++

Every literal (constant) in C/C++ will have a type information associated with it.

In both C and C++, numeric literals (e.g. 10) will have *int* as their type. It means *sizeof(10)* and *sizeof(int)* will return same value.

However, character literals (e.g. 'V') will have different types, *sizeof('V')* returns different values in C and C++. In C, a character literal is treated as *int* type where as in C++, a character literal is treated as *char* type (*sizeof('V')* and *sizeof(char)* are same in C++ but not in C).

```
int main()
{
  printf("sizeof('V') = %d sizeof(char) = %d", sizeof('V'), sizeof(char));
  return 0;
}
```

Result of above program:

C result – *sizeof('V') = 4 sizeof(char) = 1*

C++ result – *sizeof('V') = 1 sizeof(char) = 1*

Such behaviour is required in C++ to support function overloading. An example will make it more clear. Predict the output of the following C++ program.

```
void foo(char c)
{
  printf("From foo: char");
}
void foo(int i)
{
  printf("From foo: int");
}

int main()
{
  foo('V');
  return 0;
}
```

The compiler must call

```
void foo(char);
```

since 'V' type is *char*.

Article contribution by Venki and Geeksforgeeks team.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

References in C++

When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting '&' in the declaration.

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

int main()
{
  int x = 10;

  // ref is a reference to x.
  int& ref = x;
```

```
  // Value of x is now changed to 20
  ref = 20;
  cout << "x = " << x << endl ;

  // Value of x is now changed to 30
  x = 30;
  cout << "ref = " << ref << endl ;

  return 0;
}
```

Output:

```
 x = 20
ref = 30
```

Following is one more example that uses references to swap two variables.

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

void swap (int& first, int& second)
{
   int temp = first;
   first = second;
   second = temp;
}

int main()
{
   int a = 2, b = 3;
   swap( a, b );
   cout << a << " " << b;
   return 0;
}
```

Output:

```
 3 2
```

References vs Pointers

Both references and pointers can be used to change local variables of one function inside another function. Both of them can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain.
Despite above similarities, there are following differences between references and pointers.

A pointer can be declared as void but a reference can never be void
For example

```
int a = 10;
void* aa = &a;. //it is valid
void &ar = a; // it is not valid
```

Thanks to Shweta Bansal for adding this point.

References are less powerful than pointers

1) Once a reference is created, it cannot be later made to reference another object; it cannot be reseated. This is often done with pointers.
2) References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.
3) A reference must be initialized when declared. There is no such restriction with pointers

Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc. In Java, references don't have above restrictions, and can be used to implement all data structures. References being more powerful in Java, is the main reason Java doesn't need pointers.

References are safer and easier to use:

1) Safer: Since references must be initialized, wild references like wild pointers are unlikely to exist. It is still possible to have references that don't refer to a valid location (See questions 5 and 6 in the below exercise)

2) Easier to use: References don't need dereferencing operator to access the value. They can be used like normal variables. '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator ('.'), unlike pointers where arrow operator (->) is needed to access members.

Together with the above reasons, there are few places like copy constructor argument where pointer cannot be used. Reference must be used pass the argument in copy constructor. Similarly references must be used for overloading some operators like ++.

Exercise:
Predict the output of following programs. If there are compilation errors, then fix them.

Question 1

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

int &fun()
{
   static int x = 10;
   return x;
}
int main()
{
   fun() = 30;
   cout << fun();
   return 0;
}
```

Question 2

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

int fun(int &x)
{
   return x;
}
int main()
{
   cout << fun(10);
   return 0;
}
```

Question 3

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

void swap(char * &str1, char * &str2)
{
  char *temp = str1;
  str1 = str2;
  str2 = temp;
}

int main()
{
  char *str1 = "GEEKS";
  char *str2 = "FOR GEEKS";
  swap(str1, str2);
  cout<<"str1 is "<<str1<<endl;
  cout<<"str2 is "<<str2<<endl;
  return 0;
}
```

Question 4

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

int main()
{
  int x = 10;
  int *ptr = &x;
  int &*ptr1 = ptr;
}
```

Question 5

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

int main()
{
  int *ptr = NULL;
  int &ref = *ptr;
  cout << ref;
}
```

Question 6

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

int &fun()
{
   int x = 10;
   return x;
}
int main()
{
   fun() = 30;
   cout << fun();
   return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-basics

Can references refer to invalid location in C++?

In C++, Reference variables are safer than pointers because reference variables must be initialized and they cannot be changed to refer to something else once they are initialized. But there are exceptions where we can have invalid references.

1) Reference to value at uninitialized pointer.

```
int *ptr;
int &ref = *ptr;  // Reference to value at some random memory location
```

2) Reference to a local variable is returned.

```
int& fun()
{
  int a = 10;
  return a;
}
```

Once fun() returns, the space allocated to it on stack frame will be taken back. So the reference to a local variable will not be valid.

Please write comments if you find anything incorrect in the above GFact or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

When do we pass arguments by reference or pointer?

In C++, variables are passed by reference due to following reasons:

1) *To modify local variables of the caller function:* A reference (or pointer) allows called function to modify a local variable of the caller function. For example, consider the following example program where *fun()* is able to modify local variable *x* of *main()*.

```
void fun(int &x) {
   x = 20;
}

int main() {
   int x = 10;
   fun(x);
   cout<<"New value of x is "<<x;
   return 0;
}
```

Output:
New value of x is 20

2) *For passing large sized arguments:* If an argument is large, passing by reference (or pointer) is more efficient because only an address is really passed, not the entire object. For example, let us consider the following *Employee* class and a function *printEmpDetails()* that prints Employee details.

```
class Employee {
private:
   string name;
   string desig;

   // More attributes and operations
};

void printEmpDetails(Employee emp) {
    cout<<emp.getName();
    cout<<emp.getDesig();

   // Print more attributes
}
```

The problem with above code is: every time *printEmpDetails()* is called, a new Employee abject is constructed that involves creating a copy of all data members. So a better implementation would be to pass Employee as a reference.

```
void printEmpDetails(const Employee &emp) {
    cout<<emp.getName();
    cout<<emp.getDesig();

   // Print more attributes
}
```

This point is valid only for struct and class variables as we don't get any efficiency advantage for basic types like int, char.. etc.

3) *To avoid Object Slicing:* If we pass an object of subclass to a function that expects an object of superclass then the passed object is sliced if it is pass by value. For example, consider the following program, it prints "This is Pet Class".

```
#include <iostream>
#include<string>

using namespace std;

class Pet {
public:
   virtual string getDescription() const {
      return "This is Pet class";
   }
};

class Dog : public Pet {
public:
   virtual string getDescription() const {
      return "This is Dog class";
   }
};

void describe(Pet p) { // Slices the derived class object
   cout<<p.getDescription()<<endl;
}

int main() {
   Dog d;
   describe(d);
   return 0;
}
```

Output:
This is Pet Class

If we use pass by reference in the above program then it correctly prints “This is Dog Class”. See the following modified program.

```
#include <iostream>
#include<string>

using namespace std;

class Pet {
public:
   virtual string getDescription() const {
      return "This is Pet class";
   }
};

class Dog : public Pet {
public:
   virtual string getDescription() const {
      return "This is Dog class";
   }
};

void describe(const Pet &p) { // Doesn't slice the derived class object.
   cout<<p.getDescription()<<endl;
}

int main() {
   Dog d;
   describe(d);
   return 0;
}
```

Output:
This is Dog Class

This point is also not valid for basic data types like int, char, .. etc.

4) *To achieve Run Time Polymorphism in a function*

We can make a function polymorphic by passing objects as reference (or pointer) to it. For example, in the following program, print() receives a reference to the base class object. print() calls the base class function show() if base class object is passed, and derived class function show() if derived class object is passed.

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

class base {
public:
   virtual void show() {  // Note the virtual keyword here
      cout<<"In base \n";
   }
};


class derived: public base {
public:
   void show() {
      cout<<"In derived \n";
   }
};

// Since we pass b as reference, we achieve run time polymorphism here.
void print(base &b) {
   b.show();
}

int main(void) {
   base b;
   derived d;
   print(b);
   print(d);
   return 0;
}
```

Output:
In base
In derived

Thanks to Venki for adding this point.

As a side note, it is a recommended practice to make reference arguments const if they are being passed by reference only due to reason no. 2 or 3 mentioned above. This is recommended to avoid unexpected modifications to the objects.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
pointer

Function Overloading in C++

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++.

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

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char* c) {
```

```
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Output:

```
Here is int 10
Here is float 10.1
Here is char* ten
```

See this for function overloading related rules in C++.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

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 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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

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 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;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
C++

Function overloading and return type

In C++ and Java, functions can not be overloaded if they differ only in the return type.

For example, the following program C++ and Java programs fail in compilation.

C++ Program

```
#include<iostream>
int foo() {
   return 10;
}

char foo() {  // compiler error; new declaration of foo()
   return 'a';
}

int main()
{
   char x = foo();
   getchar();
   return 0;
}
```

Java Program

```
// filename Main.java
public class Main {
   public int foo() {
      return 10;
   }
   public char foo() { // compiler error: foo() is already defined
      return 'a';
   }
   public static void main(String args[])
   {
   }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
Java
Java

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 for more examples)

Reference: technical FAQs on www.stroustrup.com

Now consider **Java** version of this program:

```
class Base
{
    public int f(int i)
    {
        System.out.print("f (int): ");
        return i+3;
    }
}
class Derived extends Base
{
    public double f(double i)
    {
        System.out.print("f (double) : ");
        return i + 3.3;
    }
}
class myprogram3
{
    public static void main(String args[])
    {
        Derived obj = new Derived();
        System.out.println(obj.f(3));
        System.out.println(obj.f(3.3));
    }
}
```

The output of the above program is:

```
f (int): 6
f (double): 6.6
```

So 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.

Finally, let us try the output of following **C#** program:

```
using System;
class Base
{
   public int f(int i)
   {
      Console.Write("f (int): ");
      return i + 3;
   }
}
class Derived : Base
{
   public double f(double i)
   {
      Console.Write("f (double) : ");
      return i+3.3;
   }
}
class MyProgram
{
   static void Main(string[] args)
   {
      Derived obj = new Derived();
      Console.WriteLine(obj.f(3));
      Console.WriteLine(obj.f(3.3));
      Console.ReadKey(); // write this line if you use visual studio
   }
}
```

Note: Console.ReadKey() is used to halt the console. It is similar to getch as in C/C++.
The output of the above program is:

```
f(double) : 6.3
f(double):  6.6
```

instead of the assumed output

```
f(int) : 6
f(double) : 6.6
```

The reason is same as explained in C++ program. 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 the creator of C# language.

This article is contributed by **Pravasi Meet**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-functions
cpp-inheritance

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;
}
```

The above program fails in compilation and produces warnings and errors (See this for produced warnings and errors). You may get different errors on different compilers.

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;
}
```

Ouput:

```
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;
}
```

The outcome of program is:

```
3
I love C++
```

```
9 6
```

This article is contributed by **Pravasi Meet**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-functions
cpp-main

Default Arguments in C++

A default argument is a value provided in function declaration that is automatically assigned by the compiler if caller of the function doesn't provide a value for the argument with default value.

Following is a simple C++ example to demonstrate use of default arguments. We don't have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.

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

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
   return (x + y + z + w);
}

/* Drier program to test above function*/
int main()
{
   cout << sum(10, 15) << endl;
   cout << sum(10, 15, 25) << endl;
   cout << sum(10, 15, 25, 30) << endl;
   return 0;
}
```

Output:

```
25
50
80
```

Once default value is used for an argument, all subsequent arguments must have default value.

```
// Invalid because z has default value, but w after it
// doesn't have default value
int sum(int x, int y, int z=0, int w)
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Inline Functions in C++

Inline function is one of the important feature of C++. So, let's first understand why inline functions are used and what is the purpose of

inline function?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.
The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
   // function code
}
```

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:
1) If a function contains a loop. (for, while, do-while)
2) If a function contains static variables.
3) If a function is recursive.
4) If a function return type is other than void, and the return statement doesn't exist in function body.
5) If a function contains switch or goto statement.

Inline functions provide following advantages:
1) Function call overhead doesn't occur.
2) It also saves the overhead of push/pop variables on the stack when function is called.
3) It also saves overhead of a return call from a function.
4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.
5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Inline function disadvantages:
1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

The following program demonstrates the use of use of inline function.

```
#include <iostream>
using namespace std;
inline int cube(int s)
```

```
{
   return s*s*s;
}
int main()
{
   cout << "The cube of 3 is: " << cube(3) << "\n";
   return 0;
} //Output: The cube of 3 is: 27
```

Inline function and classes:

It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.

For example:

```
class S
{
public:
   inline int square(int s) // redundant use of inline
   {
      // this function is automatically inline
      // function body
   }
};
```

The above style is considered as a bad programming style. The best programming style is to just write the prototype of function inside the class and specify it as an inline in the function definition.

For example:

```
class S
{
public:
   int square(int s); // declare the function
};

inline int S::square(int s) // use inline prefix
{

}
```

The following program demonstrates this concept:

```
#include <iostream>
using namespace std;
class operation
{
   int a,b,add,sub,mul;
   float div;
public:
   void get();
   void sum();
   void difference();
   void product();
   void division();
};
inline void operation :: get()
{
   cout << "Enter first value:";
   cin >> a;
   cout << "Enter second value:";
   cin >> b;
}

inline void operation :: sum()
{
   add = a+b;
   cout << "Addition of two numbers: " << a+b << "\n";
}
```

```
inline void operation :: difference()
{
   sub = a-b;
   cout << "Difference of two numbers: " << a-b << "\n";
}

inline void operation :: product()
{
   mul = a*b;
   cout << "Product of two numbers: " << a*b << "\n";
}

inline void operation ::division()
{
   div=a/b;
   cout<<"Division of two numbers: "<<a/b<<"\n" ;
}

int main()
{
   cout << "Program using inline function\n";
   operation s;
   s.get();
   s.sum();
   s.difference();
   s.product();
   s.division();
   return 0;
}
```

Output:

```
Enter first value: 45
Enter second value: 15
Addition of two numbers: 60
Difference of two numbers: 30
Product of two numbers: 675
Division of two numbers: 3
```

What is wrong with macro?

Readers familiar with the C language knows that C language uses macro. The preprocessor replace all macro calls directly within the macro code. It is recommended to always use inline function instead of macro. According to Dr. Bjarne Stroustrup the creator of C++ that macros are almost never necessary in C++ and they are error prone. There are some problems with the use of macros in C++. Macro cannot access private members of class. Macros looks like function call but they are actually not.

Example:

```
#include <iostream>
using namespace std;
class S
{
   int m;
public:
#define MAC(S::m)    // error
};
```

C++ compiler checks the argument types of inline functions and necessary conversions are performed correctly. Preprocessor macro is not capable for doing this. One other thing is that the macros are managed by preprocessor and inline functions are managed by C++ compiler.

Remember: It is true that all the functions defined inside the class are implicitly inline and C++ compiler will perform inline call of these functions, but C++ compiler cannot perform inlining if the function is virtual. The reason is call to a virtual function is resolved at runtime instead of compile time. Virtual means wait until runtime and inline means during compilation, if the compiler doesn't know which function will be called, how it can perform inlining?

One other thing to remember is that it is only useful to make the function inline if the time spent during a function call is more compared to the function body execution time. An example where inline function has no effect at all:

```
inline void show()
{
   cout << "value of S = " << S << endl;
```

```
}
```

The above function relatively takes a long time to execute. In general function which performs input output (I/O) operation shouldn't be defined as inline because it spends a considerable amount of time. Technically inlining of show() function is of limited value because the amount of time the I/O statement will take far exceeds the overhead of a function call.

Depending upon the compiler you are using the compiler may show you warning if the function is not expanded inline. Programming languages like Java & C# doesn't support inline functions.
But in Java, the compiler can perform inlining when the small final method is called, because final methods can't be overridden by sub classes and call to a final method is resolved at compile time. In C# JIT compiler can also optimize code by inlining small function calls (like replacing body of a small function when it is called in a loop).

Last thing to keep in mind that inline functions are the valuable feature of C++. An appropriate use of inline function can provide performance enhancement but if inline functions are used arbitrarily then they can't provide better result. In other words don't expect better performance of program. Don't make every function inline. It is better to keep inline functions as small as possible.

References:

1) Effective C++ , Scott Meyers
2) http://www.parashift.com/c++-faq/inline-and-perf.html
3) http://www.cplusplus.com/forum/articles/20600/
4) Thinking in C++, Volume 1, Bruce Eckel.
5) C++ the complete reference, Herbert Schildt

This article is contributed by **Meet Pravasi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Category: C++

---

malloc() vs new

Following are the differences between malloc() and operator new.

1) new calls constructors, while malloc() does not. In fact primitive data types (char, int, float.. etc) can also be initialized with new. For example, below program prints 10.

```
#include<iostream>

using namespace std;

int main()
{
  int *n = new int(10); // initialization with new()
  cout<<*n;
  getchar();
  return 0;
}
```

2) new is an operator, while malloc() is a fucntion.

3) new returns exact data type, while malloc() returns void *.

Please write comments if you find anything incorrect in the above post, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

---

delete and free() in C++

In C++, delete operator should only be used either for the pointers pointing to the memory allocated using new operator or for a NULL pointer, and free() should only be used either for the pointers pointing to the memory allocated using malloc() or for a NULL pointer.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int x;
   int *ptr1 = &x;
   int *ptr2 = (int *)malloc(sizeof(int));
   int *ptr3 = new int;
   int *ptr4 = NULL;

   /* delete Should NOT be used like below because x is allocated
      on stack frame */
   delete ptr1;

   /* delete Should NOT be used like below because x is allocated
      using malloc() */
   delete ptr2;

   /* Correct uses of delete */
   delete ptr3;
   delete ptr4;

   getchar();
   return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
pointer

Basic Concepts of Object Oriented Programming using C++

Object: Objects are basic run-time entities in an object oriented system, objects are instances of a class these are defined user defined data types.
ex:

```
class person
{
   char name[20];
   int id;
public:
   void getdetails(){}
};

int main()
{
  person p1; //p1 is a object
}
```

Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each others data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

Class: Class is a blueprint of data and functions or methods. Class does not take any space.

syntax for class:

```
class class_name
{
  private:
    //data members and member functions declarations
  public:
    //data members and member functions declarations
  protected:
    //data members and member functions declarations
};
```

Class is a user defined data type like structures and unions in C.

By default class variables are private but in case of structure it is public. in above example person is a class.

Encapsulation and Data abstraction: Wrapping up(combing) of data and functions into a single unit is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapping in the class can access it. This insulation of the data from direct access by the program is called data hiding or information hiding.

Data abstraction refers to, providing only needed information to the outside world and hiding implementation details. For example, consider a class Complex with public functions as getReal() and getImag(). We may implement the class as an array of size 2 or as two variables. The advantage of abstractions is, we can change implementation at any point, users of Complex class wont't be affected as out method interface remains same. Had our implementation be public, we would not have been able to change it.

Inheritance: inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. Inheritance provides re usability. This means that we can add additional features to an existing class without modifying it.

Polymorphism: polymorphism means ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.
C++ supports operator overloading and function overloading.
Operator overloading is the process of making an operator to exhibit different behaviors in different instances is known as operator overloading.
Function overloading is using a single function name to perform different types of tasks.
polymorphism is extensively used in implementing inheritance.

Dynamic Binding: In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

This article is contributed by **Vankayala Karunakar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Structure vs class in C++

In C++, a structure is same as class except the following differences:

1) Members of a class are private by default and members of struct are public by default.
For example program 1 fails in compilation and program 2 works fine.

```
// Program 1
#include <stdio.h>

class Test {
   int x; // x is private
};
int main()
{
  Test t;
  t.x = 20; // compiler error because x is private
  getchar();
  return 0;
}
```

```
// Program 2
#include <stdio.h>

struct Test {
   int x; // x is public
};
int main()
{
  Test t;
  t.x = 20; // works fine because x is public
  getchar();
  return 0;
}
```

2) When deriving a struct from a class/struct, default access-specifier for a base class/struct is public. And when deriving a class, default access specifier is private.
For example program 3 fails in compilation and program 4 works fine.

```
// Program 3
#include <stdio.h>

class Base {
public:
   int x;
};

class Derived : Base { }; // is equilalent to class Derived : private Base {}

int main()
{
  Derived d;
  d.x = 20; // compiler error becuase inheritance is private
  getchar();
  return 0;
}
```

```
// Program 4
#include <stdio.h>

class Base {
public:
   int x;
};

struct Derived : Base { }; // is equilalent to struct Derived : public Base {}

int main()
{
```

```
  Derived d;
  d.x = 20; // works fine becuase inheritance is public
  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Can a C++ class have an object of self type?

A class declaration can contain static object of self type, it can also have pointer to self type, but it cannot have a non-static object of self type.

For example, following program works fine.

```
// A class can have a static member of self type
#include<iostream>

using namespace std;

class Test {
  static Test self;  // works fine

  /* other stuff in class*/

};

int main()
{
  Test t;
  getchar();
  return 0;
}
```

And following program also works fine.

```
// A class can have a pointer to self type
#include<iostream>

using namespace std;

class Test {
  Test * self; //works fine

  /* other stuff in class*/

};

int main()
{
  Test t;
  getchar();
  return 0;
}
```

But following program generates compilation error "*field `self' has incomplete type*"

```
// A class cannot have non-static object(s) of self type.
#include<iostream>

using namespace std;

class Test {
  Test self; // Error
```

```
  /* other stuff in class*/

};

int main()
{
  Test t;
  getchar();
  return 0;
}
```

If a non-static object is member then declaration of class is incomplete and compiler has no way to find out size of the objects of the class. Static variables do not contribute to the size of objects. So no problem in calculating size with static variables of self type.
For a compiler, all pointers have a fixed size irrespective of the data type they are pointing to, so no problem with this also.

Thanks to Manish Jain and Venki for their contribution to this post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Why is the size of an empty class not zero in C++?

Predict the output of following program?

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

class Empty {};

int main()
{
  cout << sizeof(Empty);
  return 0;
}
```

Output:

```
1
```

Size of an empty class is not zero. It is 1 byte generally. It is nonzero to ensure that the two different objects will have different addresses. See the following example.

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

class Empty { };

int main()
{
   Empty a, b;

   if (&a == &b)
     cout << "impossible " << endl;
   else
     cout << "Fine " << endl;

  return 0;
}
```

Output:

```
Fine
```

For the same reason (different objects should have different addresses), “new” always returns pointers to distinct objects. See the following example.

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

class Empty { };

int main()
{
   Empty* p1 = new Empty;
   Empty* p2 = new Empty;

   if (p1 == p2)
      cout << "impossible " << endl;
   else
      cout << "Fine " << endl;

   return 0;
}
```

Output:

```
Fine
```

Now guess the output of following program (This is tricky)

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

class Empty { };

class Derived: Empty { int a; };

int main()
{
   cout << sizeof(Derived);
   return 0;
}
```

Output (with GCC compiler. See this):

```
4
```

Note that the output is not greater than 4. There is an interesting rule that says that an empty base class need not be represented by a separate byte. So compilers are free to make optimization in case of empty base classes. As an excercise, try the following program on your compiler.

```
// Thanks to Venki for suggesting this code.
#include <iostream>
using namespace std;

class Empty
{};

class Derived1 : public Empty
{};

class Derived2 : virtual public Empty
{};

class Derived3 : public Empty
{
   char c;
};

class Derived4 : virtual public Empty
{
   char c;
```

```
};

class Dummy
{
  char c;
};

int main()
{
  cout << "sizeof(Empty) " << sizeof(Empty) << endl;
  cout << "sizeof(Derived1) " << sizeof(Derived1) << endl;
  cout << "sizeof(Derived2) " << sizeof(Derived2) << endl;
  cout << "sizeof(Derived3) " << sizeof(Derived3) << endl;
  cout << "sizeof(Derived4) " << sizeof(Derived4) << endl;
  cout << "sizeof(Dummy) " << sizeof(Dummy) << endl;

  return 0;
}
```

Source:

http://www2.research.att.com/~bs/bs_faq2.html

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
CPP

Some interesting facts about static member functions in C++

1) static member functions do not have this pointer.

For example following program fails in compilation with error *"`this' is unavailable for static member functions "*

```
#include<iostream>
class Test {
  static Test * fun() {
    return this; // compiler error
  }
};

int main()
{
  getchar();
  return 0;
}
```

2) A static member function cannot be virtual (See this G-Fact)

3) 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 with error "*'void Test::fun()' and `static void Test::fun()' cannot be overloaded*"

```
#include<iostream>
class Test {
  static void fun() {}
  void fun() {} // compiler error
};

int main()
{
  getchar();
  return 0;
}
```

4) A static member function can not be declared *const*, *volatile*, or *const volatile*.

For example, following program fails in compilation with error *"static member function `static void Test::fun()' cannot have `const' method*

qualifier ”

```
#include<iostream>
class Test {
  static void fun() const { // compiler error
    return;
  }
};

int main()
{
  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

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

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Static data members in C++

Predict the output of following C++ program:

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

class A
{
public:
   A() { cout << "A's Constructor Called " << endl; }
};

class B
{
   static A a;
public:
   B() { cout << "B's Constructor Called " << endl; }
};

int main()
{
   B b;
   return 0;
}
```

Output:

```
B's Constructor Called
```

The above program calls only B’s constructor, it doesn’t call A’s constructor. The reason for this is simple, *static members are only declared in class declaration, not defined. They must be explicitly defined outside the class using scope resolution operator.*
If we try to access static member ‘a’ without explicit definition of it, we will get compilation error. For example, following program fails in compilation.

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

class A
{
   int x;
```

```
public:
   A() { cout << "A's constructor called " << endl; }
};

class B
{
   static A a;
public:
   B() { cout << "B's constructor called " << endl; }
   static A getA() { return a; }
};

int main()
{
   B b;
   A a = b.getA();
   return 0;
}
```

Output:

```
Compiler Error: undefined reference to `B::a'
```

If we add definition of a, the program will works fine and will call A's constructor. See the following program.

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

class A
{
   int x;
public:
   A() { cout << "A's constructor called " << endl; }
};

class B
{
   static A a;
public:
   B() { cout << "B's constructor called " << endl; }
   static A getA() { return a; }
};

A B::a; // definition of a

int main()
{
   B b1, b2, b3;
   A a = b1.getA();

   return 0;
}
```

Output:

```
A's constructor called
B's constructor called
B's constructor called
B's constructor called
```

Note that the above program calls B's constructor 3 times for 3 objects (b1, b2 and b3), but calls A's constructor only once. The reason is, *static members are shared among all objects. That is why they are also known as class members or class fields*. Also, *static members can be accessed without any object*, see the below program where static member 'a' is accessed without any object.

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

class A
{
   int x;
```

```
public:
   A() { cout << "A's constructor called " << endl; }
};

class B
{
   static A a;
public:
   B() { cout << "B's constructor called " << endl; }
   static A getA() { return a; }
};

A B::a;  // definition of a

int main()
{
   // static member 'a' is accessed without any object of B
   A a = B::getA();

   return 0;
}
```

Output:

```
A's constructor called
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

‘this’ pointer in C++

The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. ‘this’ pointer is a constant pointer that holds the memory address of the current object. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).
For a class X, the type of this pointer is ‘X* const’. Also, if a member function of X is declared as const, then the type of this pointer is ‘const X *const’ (see this GFact)

Following are the situations where ‘this’ pointer is used:

1) When local variable’s name is same as member’s name

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

/* local variable is same as a member's name */
class Test
{
private:
  int x;
public:
  void setX (int x)
  {
     // The 'this' pointer is used to retrieve the object's x
     // hidden by the local variable 'x'
     this->x = x;
  }
  void print() { cout << "x = " << x << endl; }
};

int main()
{
  Test obj;
  int x = 20;
  obj.setX(x);
  obj.print();
```

```
  return 0;
}
```

Output:

```
x = 20
```

For constructors, initializer list can also be used when parameter name is same as member's name.

2) To return reference to the calling object

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
  // Some processing
  return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

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

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test &setX(int a) { x = a; return *this; }
  Test &setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj1(5, 5);

  // Chained function calls.  All calls modify the same object
  // as the same object is returned by reference
  obj1.setX(10).setY(20);

  obj1.print();
  return 0;
}
```

Output:

```
x = 10 y = 20
```

Exercise:

Predict the output of following programs. If there are compilation errors, then fix them.

Question 1

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

class Test
{
private:
  int x;
public:
  Test(int x = 0) { this->x = x; }
  void change(Test *t) { this = t; }
  void print() { cout << "x = " << x << endl; }
```

```
};

int main()
{
  Test obj(5);
  Test *ptr = new Test (10);
  obj.change(ptr);
  obj.print();
  return 0;
}
```

Question 2

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

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  static void fun1() { cout << "Inside fun1()"; }
  static void fun2() { cout << "Inside fun2()"; this->fun1(); }
};

int main()
{
  Test obj;
  obj.fun2();
  return 0;
}
```

Question 3

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

class Test
{
private:
  int x;
  int y;
public:
  Test (int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test setX(int a) { x = a; return *this; }
  Test setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj1;
  obj1.setX(10).setY(20);
  obj1.print();
  return 0;
}
```

Question 4

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

class Test
{
private:
```

```
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  void setX(int a) { x = a; }
  void setY(int b) { y = b; }
  void destroy()  { delete this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj;
  obj.destroy();
  obj.print();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-basics
cpp-pointer

Type of ‘this’ pointer in C++

In C++, *this* pointer is passed as a hidden argument to all non-static member function calls. The type of *this* depends upon function declaration. If the member function of a class *X* is declared *const*, the type of *this* is *const X* (see code 1 below),* if the member function is declared *volatile*, the type of this is *volatile X** (see code 2 below), and if the member function is declared *const volatile*, the type of *this* is *const volatile X** (see code 3 below).

Code 1

```
#include<iostream>
class X {
  void fun() const {
   // this is passed as hidden argument to fun().
   // Type of this is const X*
   }
};
```

Code 2

```
#include<iostream>
class X {
  void fun() volatile {
    // this is passed as hidden argument to fun().
    // Type of this is volatile X*
   }
};
```

Code 3

```
#include<iostream>
class X {
  void fun() const volatile {
    // this is passed as hidden argument to fun().
    // Type of this is const volatile X*
   }
};
```

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

“delete this” in C++

Ideally *delete* operator should not be used for *this* pointer. However, if used, then following points must be considered.

1) *delete* operator works only for objects allocated using operator *new* (See http://geeksforgeeks.org/?p=8539). If the object is created using new, then we can do *delete this*, otherwise behavior is undefined.

```
class A
{
  public:
    void fun()
    {
        delete this;
    }
};

int main()
{
  /* Following is Valid */
  A *ptr = new A;
  ptr->fun();
  ptr = NULL // make ptr NULL to make sure that things are not accessed using ptr.


  /* And following is Invalid: Undefined Behavior */
  A a;
  a.fun();

  getchar();
  return 0;
}
```

2) Once *delete this* is done, any member of the deleted object should not be accessed after deletion.

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

class A
{
  int x;
  public:
    A() { x = 0;}
    void fun() {
      delete this;

      /* Invalid: Undefined Behavior */
      cout<<x;
    }
};
```

The best thing is to not do *delete this* at all.

Thanks to Shekhu for providing above details.

References:
https://www.securecoding.cert.org/confluence/display/cplusplus/OOP05-CPP.+Avoid+deleting+this
http://en.wikipedia.org/wiki/This_%28computer_science%29

This article is contributed by **Rahul Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Constructors in C++

What is constructor?

A constructor is a member function of a class which initializes objects of a class.

Following is a simple example to demonstrate constructors in C++.

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

class Point
{
private:
   int x, y;
public:
   /***Constructor****/
   Point(int x1, int y1) { x = x1; y = y1; }

   int getX()          {  return x; }
   int getY()          {  return y; }
};

int main()
{
   Point p1(10, 15); // constructor is called here

   // Let us access values assigned by constructor
   cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

   return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
```

How constructors are different from a normal member function?

A constructor is different from normal functions in following ways:

(i) Constructor has same name as the class itself

(ii) Constructors don't have return type

(iii) A constructor is automatically called when an object is created.

(iv) If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Can we have more than one constructors in a class?

We can have more than one constructor in a class, as long as each has a different list of arguments.

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

class Point
{
private:
   int x, y;
public:
   // Two constructors
   Point(int x1, int y1) { x = x1; y = y1; }
   Point()             {x = 0; y = 0; }

   int getX()          {  return x; }
   int getY()          {  return y; }
};
```

```
int main()
{
   Point p1(10, 15); // first constructor is called here
   Point p2; // Second constructor is called here

   // Let us access values assigned by constructors
   cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
   cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

   return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
p2.x = 0, p2.y = 0
```

You may like to take a quiz on constructors in C++.

We will soon be covering more on constructors in C++.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Copy Constructor in C++

We have discussed introduction to Constructors in C++. In this post, copy constructor is discussed.

What is a copy constructor?

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
ClassName (const ClassName &old_obj);
```

Following is a simple example of copy constructor.

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

class Point
{
private:
   int x, y;
public:
   Point(int x1, int y1) { x = x1; y = y1; }

   // Copy constructor
   Point(const Point &p2) {x = p2.x; y = p2.y; }

   int getX()          {  return x; }
   int getY()          {  return y; }
};

int main()
{
   Point p1(10, 15); // Normal constructor is called here
   Point p2 = p1; // Copy constructor is called here
```

```
    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15
```

When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:

1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When compiler generates a temporary object.

It is however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example being the return value optimization (sometimes referred to as RVO).
Source: http://www.geeksforgeeks.org/g-fact-13/

When is user defined copy constructor needed?

If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member wise copy between objects. The compiler created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any run time allocation of resource like file handle, a network connection..etc.

Default constructor does only shallow copy.

shallow

Image Source : http://docs.roguewave.com/sourcepro/11.1/html/toolsug/6-4.html

Deep copy is possible only with user defined copy constructor. In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.

deep-300x179

Image Source : http://docs.roguewave.com/sourcepro/11.1/html/toolsug/6-4.html

Copy constructor vs Assignment Operator

Which of the following two statements call copy constructor and which one calls assignment operator?

```
MyClass t1, t2;
MyClass t3 = t1;  // ----> (1)
t2 = t1;          // -----> (2)
```

Copy constructor is called when a new object is created from an existing object, as a copy of the existing object. Assignment operator is called when an already initialized object is assigned a new value from another existing object. In the above example (1) calls copy constrictor and (2) calls assignment operator. See this for more details.

Write an example class where copy constructor is needed?

Following is a complete C++ program to demonstrate use of Copy constructor. In the following String class, we must write copy constructor.

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

class String
{
private:
   char *s;
   int size;
public:
   String(const char *str = NULL); // constructor
   ~String() { delete [] s;  }// destructor
   String(const String&); // copy constructor
   void print() { cout << s << endl; } // Function to print string
   void change(const char *);  // Function to change
};

String::String(const char *str)
{
   size = strlen(str);
   s = new char[size+1];
   strcpy(s, str);
}

void String::change(const char *str)
{
   delete [] s;
   size = strlen(str);
   s = new char[size+1];
   strcpy(s, str);
}

String::String(const String& old_str)
{
   size = old_str.size;
   s = new char[size+1];
   strcpy(s, old_str.s);
}

int main()
{
   String str1("GeeksQuiz");
   String str2 = str1;

   str1.print(); // what is printed ?
   str2.print();

   str2.change("GeeksforGeeks");

   str1.print(); // what is printed now ?
   str2.print();
   return 0;
}
```

Output:

```
GeeksQuiz
GeeksQuiz
GeeksQuiz
GeeksforGeeks
```

What would be the problem if we remove copy constructor from above code?

If we remove copy constructor from above program, we don't get the expected output. The changes made to str2 reflect in str1 as well which is never expected.

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

class String
{
private:
   char *s;
   int size;
public:
   String(const char *str = NULL); // constructor
   ~String() { delete [] s;  }// destructor
   void print() { cout << s << endl; }
   void change(const char *);  // Function to change
};

String::String(const char *str)
{
   size = strlen(str);
   s = new char[size+1];
   strcpy(s, str);
}

void String::change(const char *str)
{
   delete [] s;
   size = strlen(str);
   s = new char[size+1];
   strcpy(s, str);
}

int main()
{
   String str1("GeeksQuiz");
   String str2 = str1;

   str1.print(); // what is printed ?
   str2.print();

   str2.change("GeeksforGeeks");

   str1.print(); // what is printed now ?
   str2.print();
   return 0;
}
```

Output:

```
GeeksQuiz
GeeksQuiz
GeeksforGeeks
GeeksforGeeks
```

Can we make copy constructor private?

Yes, a copy constructor can be made private. When we make a copy constructor private in a class, objects of that class become non-copyable. This is particularly useful when our class has pointers or dynamically allocated resources. In such situations, we can either write our own copy constructor like above String example, or make a private copy constructor so that users get compiler errors rather than surprises at run time.

Why argument to a copy constructor must be passed as a reference?

A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn't allow parameters to be pass by value.

Why argument to a copy constructor should be const?

See http://www.geeksforgeeks.org/copy-constructor-argument-const/

This article is contributed by **Shubham Agrawal**. If you like GeeksforGeeks and would like to contribute, you can also mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Destructors in C++

What is destructor?

Destructor is a member function which destructs or deletes an object.

When is destructor called?

A destructor function is called automatically when the object goes out of scope:

(1) the function ends

(2) the program ends

(3) a block containing local variables ends

(4) a delete operator is called

How destructors are different from a normal member function?

Destructors have same name as the class preceded by a tilde (~)

Destructors don't take any argument and don't return anything

```
class String
{
private:
   char *s;
   int size;
public:
   String(char *); // constructor
   ~String();     // destructor
};

String::String(char *c)
{
   size = strlen(c);
   s = new char[size+1];
   strcpy(s,c);
}

String::~String()
{
   delete []s;
}
```

Can there be more than one destructor in a class?

No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

When do we need to write a user-defined destructor?

If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

Can a destructor be virtual?

Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function. See virtual destructor for more details.

You may like to take a quiz on destructors.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Does compiler create default constructor when we write our own?

In C++, compiler by default creates default constructor for every class. But, if we define our own constructor, compiler doesn't create the default constructor.

For example, program 1 compiles without any error, but compilation of program 2 fails with error "no matching function for call to `myInteger::myInteger()' "

Program 1

```
#include<iostream>

using namespace std;

class myInteger
{
  private:
    int value;

    //...other things in class
};

int main()
{
 myInteger I1;
 getchar();
 return 0;
}
```

Program 2

```
#include<iostream>

using namespace std;

class myInteger
{
  private:
    int value;
  public:
    myInteger(int v)  // parametrized constructor
    {  value = v; }

    //...other things in class
};

int main()
{
 myInteger I1;
 getchar();
 return 0;
}
```

Please write comments if you find anything incorrect in the above GFact or you want to share more information about the topic discussed above.

References:
http://en.wikipedia.org/wiki/Default_constructor
http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=/com.ibm.xlcpp8l.doc/language/ref/cplr375.htm

C/C++ Puzzles

When should we write our own copy constructor?

Don't write a copy constructor if shallow copies are ok: In C++, If an object has no pointers or any run time allocation of resource like file handle, a network connection..etc, a shallow copy is probably sufficient. Therefore the default copy constructor, default assignment operator, and default destructor are ok and you don't need to write your own.

Source: http://www.fredosaurus.com/notes-cpp/oop-condestructors/copyconstructors.html

C/C++ Puzzles

When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:
1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When compiler generates a temporary object.

It is however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example being the return value optimization (sometimes referred to as RVO).

References:
http://www.fredosaurus.com/notes-cpp/oop-condestructors/copyconstructors.html
http://en.wikipedia.org/wiki/Copy_constructor

C/C++ Puzzles
GFacts

Initialization of data members

In C++, class variables are initialized in the same order as they appear in the class declaration.

Consider the below code.

```
#include<iostream>

using namespace std;

class Test {
  private:
    int y;
    int x;
  public:
    Test() : x(10), y(x + 10) {}
    void print();
};
```

```
void Test::print()
{
  cout<<"x = "<<x<<" y = "<<y;
}

int main()
{
   Test t;
   t.print();
   getchar();
   return 0;
}
```

The program prints correct value of x, but some garbage value for y, because y is initialized before x as it appears before in the class declaration.

So one of the following two versions can be used to avoid the problem in above code.

```
// First: Change the order of declaration.
class Test {
 private:
   int x;
   int y;
 public:
   Test() : x(10), y(x + 10) {}
   void print();
};
```

```
// Second: Change the order of initialization.
class Test {
 private:
   int y;
   int x;
 public:
   Test() : x(y-10), y(20) {}
   void print();
};
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

When do we use Initializer List in C++?

Initializer List is used to initialize data members of a class. The list of members to be initialized is indicated with constructor as a comma separated list followed by a colon. Following is an example that uses initializer list to initialize x and y of Point class.

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

class Point {
private:
   int x;
   int y;
public:
   Point(int i = 0, int j = 0):x(i), y(j) {}
   /*  The above use of Initializer list is optional as the
       constructor can also be written as:
       Point(int i = 0, int j = 0) {
          x = i;
          y = j;
       }
   */

   int getX() const {return x;}
```

```
    int getY() const {return y;}
};

int main() {
  Point t1(10, 15);
  cout<<"x = "<<t1.getX()<<", ";
  cout<<"y = "<<t1.getY();
  return 0;
}

/* OUTPUT:
   x = 10, y = 15
*/
```

The above code is just an example for syntax of Initializer list. In the above code, x and y can also be easily initialed inside the constructor. But there are situations where initialization of data members inside constructor doesn't work and Initializer List must be used. Following are such cases:

1) For initialization of non-static const data members:
const data members must be initialized using Initializer List. In the following example, "t" is a const data member of Test class and is initialized using Initializer List.

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

class Test {
    const int t;
public:
    Test(int t):t(t) {}  //Initializer list must be used
    int getT() { return t; }
};

int main() {
    Test t1(10);
    cout<<t1.getT();
    return 0;
}

/* OUTPUT:
   10
*/
```

2) For initialization of reference members:
Reference members must be initialized using Initializer List. In the following example, "t" is a reference member of Test class and is initialized using Initializer List.

```
// Initialization of reference data members
#include<iostream>
using namespace std;

class Test {
    int &t;
public:
    Test(int &t):t(t) {}  //Initializer list must be used
    int getT() { return t; }
};

int main() {
    int x = 20;
    Test t1(x);
    cout<<t1.getT()<<endl;
    x = 30;
    cout<<t1.getT()<<endl;
    return 0;
}
/* OUTPUT:
    20
    30
 */
```

3) For initialization of member objects which do not have default constructor:
In the following example, an object “a” of class “A” is data member of class “B”, and “A” doesn’t have default constructor. Initializer List must be used to initialize “a”.

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

class A {
   int i;
public:
   A(int );
};

A::A(int arg) {
   i = arg;
   cout << "A's Constructor called: Value of i: " << i << endl;
}

// Class B contains object of A
class B {
   A a;
public:
   B(int );
};

B::B(int x):a(x) {  //Initializer list must be used
   cout << "B's Constructor called";
}

int main() {
   B obj(10);
   return 0;
}
/* OUTPUT:
   A's Constructor called: Value of i: 10
   B's Constructor called
*/
```

If class A had both default and parameterized constructors, then Initializer List is not must if we want to initialize “a” using default constructor, but it is must to initialize “a” using parameterized constructor.

4) For initialization of base class members : Like point 3, parameterized constructor of base class can only be called using Initializer List.

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

class A {
   int i;
public:
   A(int );
};

A::A(int arg) {
   i = arg;
   cout << "A's Constructor called: Value of i: " << i << endl;
}

// Class B is derived from A
class B: A {
public:
   B(int );
};

B::B(int x):A(x) { //Initializer list must be used
   cout << "B's Constructor called";
}

int main() {
   B obj(10);
   return 0;
```

```
}
```

5) When constructor's parameter name is same as data member

If constructor's parameter name is same as data member name then the data member must be initialized either using this pointer or Initializer List. In the following example, both member name and parameter name for A() is "i".

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

class A {
   int i;
public:
   A(int );
   int getI() const { return i; }
};

A::A(int i):i(i) { }  // Either Initializer list or this pointer must be used
/* The above constructor can also be written as
A::A(int i) {
   this->i = i;
}
*/

int main() {
   A a(10);
   cout<<a.getI();
   return 0;
}
/* OUTPUT:
   10
*/
```

6) For Performance reasons:

It is better to initialize all class variables in Initializer List instead of assigning values inside body. Consider the following example:

```
// Without Initializer List
class MyClass {
 Type variable;
public:
 MyClass(Type a) {  // Assume that Type is an already
                    // declared class and it has appropriate
                    // constructors and operators
  variable = a;
 }
};
```

Here compiler follows following steps to create an object of type MyClass

1. Type's constructor is called first for "a".
2. The assignment operator of "Type" is called inside body of MyClass() constructor to assign

```
 variable = a;
```

3. And then finally destructor of "Type" is called for "a" since it goes out of scope.

Now consider the same code with MyClass() constructor with Initializer List

```
// With Initializer List
class MyClass {
   Type variable;
public:
   MyClass(Type a):variable(a) {   // Assume that Type is an already
                    // declared class and it has appropriate
                    // constructors and operators
   }
};
```

With the Initializer List, following steps are followed by compiler:

1. Copy constructor of "Type" class is called to initialize : variable(a). The arguments in initializer list are used to copy construct "variable"

directly.
2. Destructor of “Type” is called for “a” since it goes out of scope.

As we can see from this example if we use assignment inside constructor body there are three function calls: constructor + destructor + one addition assignment operator call. And if we use Initializer List there are only two function calls: copy constructor + destructor call. See this post for a running example on this point.

This assignment penalty will be much more in “real” applications where there will be many such variables. Thanks to *ptr* for adding this point.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

C++ Internals | Default Constructors | Set 1

A constructor without any arguments or with default value for every argument, is said to be ***default constructor***. What is the significance of default constructor? Will the code be generated for every default constructor? Will there be any code inserted by compiler to the user implemented default constructor behind the scenes?

The compiler will implicitly *declare* default constructor if not provided by programmer, will *define* it when in need. Compiler defined default constructor is required to do certain initialization of class internals. It will not touch the data members or plain old data types (aggregates like array, structures, etc…). However, the compiler generates code for default constructor based on situation.

Consider a class derived from another class with default constructor, or a class containing another class object with default constructor. The compiler needs to insert code to call the default constructors of base class/embedded object.

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

class Base
{
public:
   // compiler "declares" constructor
};

class A
{
public:
   // User defined constructor
   A()
   {
      cout << "A Constructor" << endl;
   }

   // uninitialized
   int size;
};

class B : public A
{
   // compiler defines default constructor of B, and
   // inserts stub to call A constructor

   // compiler won't initialize any data of A
};

class C : public A
{
public:
   C()
   {
      // User defined default constructor of C
      // Compiler inserts stub to call A's construtor
      cout << "B Constructor" << endl;
```

```
        // compiler won't initialize any data of A
    }
};

class D
{
public:
    D()
    {
        // User defined default constructor of D
        // a - constructor to be called, compiler inserts
        // stub to call A constructor
        cout << "D Constructor" << endl;

        // compiler won't initialize any data of 'a'
    }

private:
    A a;
};

int main()
{
    Base base;

    B b;
    C c;
    D d;

    return 0;
}
```

There are different scenarios in which compiler needs to insert code to ensure some necessary initialization as per language requirement. We will have them in upcoming posts. Our objective is to be aware of C++ internals, not to use them incorrectly.

— by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

Private Destructor

Predict the output of following programs.

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

class Test
{
private:
   ~Test() {}
};
int main()
{ }
```

The above program compiles and runs fine. It is **not** compiler error to create private destructors. What do you say about below program.

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

class Test
{
private:
   ~Test() {}
};
```

```
int main()
{
  Test t;
}
```

The above program fails in compilation. The compiler notices that the local variable ‘t’ cannot be destructed because the destructor is private. What about the below program?

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

class Test
{
private:
   ~Test() {}
};
int main()
{
   Test *t;
}
```

The above program works fine. There is no object being constructed, the program just creates a pointer of type “Test *”, so nothing is destructed. What about the below program?

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

class Test
{
private:
   ~Test() {}
};
int main()
{
   Test *t = new Test;
}
```

The above program also works fine. When something is created using dynamic memory allocation, it is programmer’s responsibility to delete it. So compiler doesn’t bother.

The below program fails in compilation. When we call delete, desturctor is called.

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

class Test
{
private:
   ~Test() {}
};
int main()
{
   Test *t = new Test;
   delete t;
}
```

We noticed in the above programs, when a class has private destructur, only dynamic objects of that class can be created. Following is a way to create classes with private destructors and have a function as friend of the class. The function can only delete the objects.

```
#include <iostream>

// A class with private destuctor
class Test
{
private:
    ~Test() {}
friend void destructTest(Test* );
};
```

```
// Only this function can destruct objects of Test
void destructTest(Test* ptr)
{
   delete ptr;
}

int main()
{
   // create an object
   Test *ptr = new Test;

   // destruct the object
   destructTest (ptr);

   return 0;
}
```

What is the use of private destructor?

Whenever we want to control destruction of objects of a class, we make the destructor private. For dynamically created objects, it may happen that you pass a pointer to the object to a function and the function deletes the object. If the object is referred after the function call, the reference will become dangling. See this for more details.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-destructor

Playing with Destructors in C++

Predict the output of the below code snippet.

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

int i;

class A
{
public:
   ~A()
   {
      i=10;
   }
};

int foo()
{
   i=3;
   A ob;
   return i;
}

int main()
{
   cout << "i = " << foo() << endl;
   return 0;
}
```

Output of the above program is “i = 3”.

Why the output is i= 3 and not 10?

While returning from a function, destructor is the last method to be executed. The destructor for the object “ob” is called after the value of i is copied to the return value of the function. So, before destructor could change the value of i to 10, the current value of i gets copied & hence the output is i = 3.

How to make the program to output “i = 10” ?

Following are two ways of returning updated value:

1) Return by Reference:
Since reference gives the l-value of the variable,by using return by reference the program will output “i = 10”.

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

int i;

class A
{
public:
 ~A()
 {
  i = 10;
 }
};

int& foo()
{
 i = 3;
 A ob;
 return i;
}

int main()
{
 cout << "i = " << foo() << endl;
 return 0;
}
```

The function foo() returns the l-value of the variable i. So, the address of i will be copied in the return value. Since, the references are automatically dereferened. It will output “i = 10”.

2. Create the object ob in a block scope

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

int i;

class A
{
public:
   ~A()
   {
      i = 10;
   }
};

int foo()
{
   i = 3;
   {
      A ob;
   }
   return i;
}

int main()
{
   cout << "i = " << foo() << endl;
   return 0;
}
```

Since the object ob is created in the block scope, the destructor of the object will be called after the block ends, thereby changing the value of i to 10. Finally 10 will copied to the return value.

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect,

or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-destructor

Copy elision in C++

Copy elision (or Copy omission) is a compiler optimization technique that avoids unnecessary copying of objects. Now a days, almost every compiler uses it. Let us understand it with the help of an example.

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

class B
{
public:
   B(const char* str = "\0") //default constructor
   {
      cout << "Constructor called" << endl;
   }

   B(const B &b)  //copy constructor
   {
      cout << "Copy constructor called" << endl;
   }
};

int main()
{
   B ob = "copy me";
   return 0;
}
```

The output of above program is:

```
Constructor called
```

Why copy constructor is not called?

According to theory, when the object “ob” is being constructed, one argument constructor is used to convert “copy me” to a temporary object & that temporary object is copied to the object “ob”. So the statement

```
   B ob = "copy me";
```

should be broken down by the compiler as

```
   B ob = B("copy me");
```

However, most of the C++ compilers avoid such overheads of creating a temporary object & then copying it.

```
The modern compilers break down the statement
   B ob = "copy me"; //copy initialization
as
   B ob("copy me"); //direct initialization
and thus eliding call to copy constructor.
```

However, if we still want to ensure that the compiler doesn’t elide the call to copy constructor [disable the copy elision], we can compile the program using “-fno-elide-constructors” option with g++ and see the output as following:

```
aashish@aashish-ThinkPad-SL400:~$ g++ copy_elision.cpp -fno-elide-constructors
aashish@aashish-ThinkPad-SL400:~$ ./a.out
Constructor called
Copy constructor called
```

If “-fno-elide-constructors” option is used, first default constructor is called to create a temporary object, then copy constructor is called to copy the temporary object to ob.

Reference:

http://en.wikipedia.org/wiki/Copy_elision

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

C++ default constructor | Built-in types

Predict the output of following program?

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

int main() {

  cout << int() << endl;
  return 0;
}
```

A constructor without any arguments or with default values for every argument, is treated as *default constructor*. It will be called by the compiler when in need (precisely code will be generated for default constructor based on need).

C++ allows even built-in type (primitive types) to have default constructors. The function style cast *int()* is analogous to casting 0 to required type. The program prints 0 on console.

The initial content of the article triggered many discussions, given below is consolidation.

It is worth to be cognizant of reference vs. value semantics in C++ and the concept of Plain Old Data types. From Wiki, primitive types and POD types have no user-defined copy assignment operator, no user-defined destructor, and no non-static data members that are not themselves PODs. Moreover, a POD class must be an aggregate, meaning it has no user-declared constructors, no private nor protected non-static data, no base classes and no virtual functions.

An excerpt (from a mail note) from the creator of C++, “I think you mix up ‘actual constructor calls’ with conceptually having a constructor. Built-in types are considered to have constructors”.

The code snippet above mentioned *int()* is considered to be conceptually having constructor. However, there will not be any code generated to make *an explicit constructor* call. But when we observe assembly output, code will be generated to initialize the identifier using value semantics. For more details refer section 8.5 of this document.

Thanks to Prasoon Saurav for initiating the discussion, providing various references and correcting lacuna in my understanding.

Contributed by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

1. The C++ Programming Language, 3e.
2. Latest C++ standard – working draft section 8.5.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

When does compiler create default and copy constructors in C++?

In C++, compiler creates a default constructor if we don't define our own constructor (See this). Compiler created default constructor has empty body, i.e., it doesn't assign default values to data members (In java, default constructors assign default values).

Compiler also creates a copy constructor if we don't write our own copy constructor. Unlike default constructor, body of compiler created copy constructor is not empty, it copies all data members of passed object to the object which is being created.

What happens when we write only a copy constructor – does compiler create default constructor?
Compiler doesn't create a default constructor if we write any constructor even if it is copy constructor. For example, the following program doesn't compile.

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

class Point
{
   int x, y;
public:
  Point(const Point &p) { x = p.x; y = p.y; }
};

int main()
{
   Point p1;  // COMPILER ERROR
   Point p2 = p1;
   return 0;
}
```

Output:

```
COMPILER ERROR: no matching function for call to 'Point::Point()
```

What about reverse – what happens when we write a normal constructor and don't write a copy constructor?
Reverse is not true. Compiler creates a copy constructor if we don't write our own. Compiler creates it even if we have written other constructors in class. For example, the below program works fine.

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

class Point
{
   int x, y;
public:
  Point(int i, int j) { x = 10; y = 20; }
  int getX() { return x; }
  int getY() { return y; }
};

int main()
{
   Point p1(10, 20);
   Point p2 = p1; // This compiles fine
   cout << "x = " << p2.getX() << " y = " << p2.getY();
   return 0;
}
```

Output:

```
x = 10 y = 20
```

So we need to write copy constructor only when we have pointers or run time allocation of resource like file handle, a network connection, etc (See this)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-constrcutor

Why copy constructor argument should be const in C++?

When we create our own copy constructor, we pass an object by reference and we generally pass it as a const reference.

One reason for passing const reference is, we should use const in C++ wherever possible so that objects are not accidentally modified. This is one good reason for passing reference as const, but there is more to it. For example, predict the output of following C++ program. Assume that copy elision is not done by compiler.

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

class Test
{
  /* Class data members */
public:
  Test(Test &t) { /* Copy data members from t*/}
  Test()       { /* Initialize data members */ }
};

Test fun()
{
   cout << "fun() Called\n";
   Test t;
   return t;
}

int main()
{
   Test t1;
   Test t2 = fun();
   return 0;
}
```

Output:

```
Compiler Error in line "Test t2 = fun();"
```

The program looks fine at first look, but it has compiler error. If we add const in copy constructor, the program works fine, i.e., we change copy constructor to following.

```
Test(const Test &t) { cout << "Copy Constructor Called\n"; }
```

Or if we change the line “Test t2 = fun();” to following two lines, then also the program works fine.

```
Test t2;
t2 = fun();
```

The function fun() returns by value. So the compiler creates a temporary object which is copied to t2 using copy constructor in the original program (The temporary object is passed as an argument to copy constructor). The reason for compiler error is, compiler created temporary objects cannot be bound to non-const references and the original program tries to do that. It doesn’t make sense to modify compiler created temporary objects as they can die any moment.

This article is compiled by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-constrcutor

Advanced C++ | Virtual Constructor

Can we make a class constructor *virtual* in C++ to create polymorphic objects? No. C++ being static typed (the purpose of RTTI is different) language, it is meaningless to the C++ compiler to create an object polymorphically. The compiler must be aware of the class type to create the object. In other words, what type of object to be created is a compile time decision from C++ compiler perspective. If we

make constructor virtual, compiler flags an error. In fact except *inline*, no other keyword is allowed in the declaration of constructor.

In practical scenarios we would need to create a derived class object in a class hierarchy based on some input. Putting in other words, *object creation and object type are tightly coupled which forces modifications to extended. The objective of virtual constructor is to decouple object creation from it's type*.

How can we create required type of object at runtime? For example, see the following sample program.

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

//// LIBRARY START
class Base
{
public:

   Base() { }

   virtual // Ensures to invoke actual object destructor
   ~Base() { }

   // An interface
   virtual void DisplayAction() = 0;
};

class Derived1 : public Base
{
public:
   Derived1()
   {
      cout << "Derived1 created" << endl;
   }

   ~Derived1()
   {
      cout << "Derived1 destroyed" << endl;
   }

   void DisplayAction()
   {
      cout << "Action from Derived1" << endl;
   }
};

class Derived2 : public Base
{
public:
   Derived2()
   {
      cout << "Derived2 created" << endl;
   }

   ~Derived2()
   {
      cout << "Derived2 destroyed" << endl;
   }

   void DisplayAction()
   {
      cout << "Action from Derived2" << endl;
   }
};

//// LIBRARY END

class User
{
public:

   // Creates Drived1
   User() : pBase(0)
   {
```

```
        // What if Derived2 is required? - Add an if-else ladder (see next sample)
        pBase = new Derived1();
    }

    ~User()
    {
        if( pBase )
        {
            delete pBase;
            pBase = 0;
        }
    }

    // Delegates to actual object
    void Action()
    {
        pBase->DisplayAction();
    }

private:
    Base *pBase;
};

int main()
{
    User *user = new User();

    // Need Derived1 functionality only
    user->Action();

    delete user;
}
```

In the above sample, assume that the hierarchy *Base*, *Derived1* and *Derived2* are part of library code. The class *User* is utility class trying to make use of the hierarchy. The *main* function is consuming *Base* hierarchy functionality via *User* class.

The *User* class constructor is creating *Derived1* object, always. If the *User*'s consumer (the *main* in our case) needs *Derived2* functionality, *User* needs to create "***new Derived2()***" and it forces recompilation. Recompiling is bad way of design, so we can opt for the following approach.

Before going into details, let us answer, who will dictate to create either of *Derived1* or *Derived2* object? Clearly, it is the consumer of *User* class. The *User* class can make use of if-else ladder to create either *Derived1* or *Derived2*, as shown in the following sample,

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

//// LIBRARY START
class Base
{
public:
    Base() { }

    virtual // Ensures to invoke actual object destructor
    ~Base() { }

    // An interface
    virtual void DisplayAction() = 0;
};

class Derived1 : public Base
{
public:
    Derived1()
    {
        cout << "Derived1 created" << endl;
    }

    ~Derived1()
    {
        cout << "Derived1 destroyed" << endl;
    }
```

```
    void DisplayAction()
    {
        cout << "Action from Derived1" << endl;
    }
};

class Derived2 : public Base
{
public:
    Derived2()
    {
        cout << "Derived2 created" << endl;
    }

    ~Derived2()
    {
        cout << "Derived2 destroyed" << endl;
    }

    void DisplayAction()
    {
        cout << "Action from Derived2" << endl;
    }
};

//// LIBRARY END

class User
{
public:

    // Creates Derived1 or Derived2 based on input
    User() : pBase(0)
    {
        int input; // ID to distinguish between
                   // Derived1 and Derived2

        cout << "Enter ID (1 or 2): ";
        cin  >> input;

        while( (input !=  1) && (input !=  2) )
        {
            cout << "Enter ID (1 or 2 only): ";
            cin  >> input;
        }

        if( input == 1 )
        {
            pBase = new Derived1;
        }
        else
        {
            pBase = new Derived2;
        }

        // What if Derived3 being added to the class hierarchy?
    }

    ~User()
    {
        if( pBase )
        {
            delete pBase;
            pBase = 0;
        }
    }

    // Delegates to actual object
    void Action()
    {
        pBase->DisplayAction();
```

```
    }

private:
    Base *pBase;
};

int main()
{
    User *user = new User();

    // Need either Derived1 or Derived2 functionality
    user->Action();

    delete user;
}
```

The above code is *not* open for extension, an inflexible design. In simple words, if the library updates the *Base* class hierarchy with new class *Derived3*. How can the *User* class creates *Derived3* object? One way is to update the if-else ladder that creates *Derived3* object based on new input ID 3 as shown below,

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

class User
{
public:
    User() : pBase(0)
    {
        // Creates Drived1 or Derived2 based on need

        int input; // ID to distinguish between
                   // Derived1 and Derived2

        cout << "Enter ID (1 or 2): ";
        cin  >> input;

        while( (input !=  1) && (input !=  2) )
        {
            cout << "Enter ID (1 or 2 only): ";
            cin  >> input;
        }

        if( input == 1 )
        {
            pBase = new Derived1;
        }
        else if( input == 2 )
        {
            pBase = new Derived2;
        }
        else
        {
            pBase = new Derived3;
        }
    }

    ~User()
    {
        if( pBase )
        {
            delete pBase;
            pBase = 0;
        }
    }

    // Delegates to actual object
    void Action()
    {
        pBase->DisplayAction();
    }
```

```
private:
   Base *pBase;
};
```

The above modification forces the users of *User* class to recompile, bad (inflexible) design! And won't close *User* class from further modifications due to *Base* extension.

The problem is with the creation of objects. Addition of new class to the hierarchy forcing dependents of *User* class to recompile. Can't we delegate the action of creating objects to class hierarchy itself or to a function that behaves virtually? By delegating the object creation to class hierarchy (or to a static function) we can avoid the tight coupling between *User* and *Base* hierarchy. Enough theory, see the following code,

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

//// LIBRARY START
class Base
{
public:

   // The "Virtual Constructor"
   static Base *Create(int id);

   Base() { }

   virtual // Ensures to invoke actual object destructor
   ~Base() { }

   // An interface
   virtual void DisplayAction() = 0;
};

class Derived1 : public Base
{
public:
   Derived1()
   {
      cout << "Derived1 created" << endl;
   }

   ~Derived1()
   {
      cout << "Derived1 destroyed" << endl;
   }

   void DisplayAction()
   {
      cout << "Action from Derived1" << endl;
   }
};

class Derived2 : public Base
{
public:
   Derived2()
   {
      cout << "Derived2 created" << endl;
   }

   ~Derived2()
   {
      cout << "Derived2 destroyed" << endl;
   }

   void DisplayAction()
   {
      cout << "Action from Derived2" << endl;
   }
};

class Derived3 : public Base
```

```
{
public:
   Derived3()
   {
      cout << "Derived3 created" << endl;
   }

   ~Derived3()
   {
      cout << "Derived3 destroyed" << endl;
   }

   void DisplayAction()
   {
      cout << "Action from Derived3" << endl;
   }
};

// We can also declare "Create" outside Base
// But it is more relevant to limit it's scope to Base
Base *Base::Create(int id)
{
   // Just expand the if-else ladder, if new Derived class is created
   // User code need not be recompiled to create newly added class objects

   if( id == 1 )
   {
      return new Derived1;
   }
   else if( id == 2 )
   {
      return new Derived2;
   }
   else
   {
      return new Derived3;
   }
}
//// LIBRARY END

//// UTILITY START
class User
{
public:
   User() : pBase(0)
   {
      // Receives an object of Base heirarchy at runtime

      int input;

      cout << "Enter ID (1, 2 or 3): ";
      cin >> input;

      while( (input != 1) && (input != 2) && (input != 3) )
      {
         cout << "Enter ID (1, 2 or 3 only): ";
         cin >> input;
      }

      // Get object from the "Virtual Constructor"
      pBase = Base::Create(input);
   }

   ~User()
   {
      if( pBase )
      {
         delete pBase;
         pBase = 0;
      }
   }
```

```
    // Delegates to actual object
    void Action()
    {
        pBase->DisplayAction();
    }

private:
    Base *pBase;
};

//// UTILITY END

//// Consumer of User (UTILITY) class
int main()
{
    User *user = new User();

    // Action required on any of Derived objects
    user->Action();

    delete user;
}
```

The *User* class is independent of object creation. It delegates that responsibility to *Base*, and provides an input in the form of ID. If the library adds new class *Derived4*, the library modifier will extend the if-else ladder inside *Create* to return proper object. Consumers of *User* need not recompile their code due to extension of *Base*.

Note that the function *Create* used to return different types of *Base* class objects at runtime. It acts like virtual constructor, also referred as *Factory Method* in pattern terminology.

Pattern world demonstrate different ways to implement the above concept. Also there are some potential design issues with the above code. Our objective is to provide some insights into virtual construction, creating objects dynamically based on some input. We have excellent books devoted to the subject, interested reader can refer them for more information.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

Advanced C++ | Virtual Copy Constructor

In the virtual constructor idiom we have seen the way to construct an object whose type is not determined until runtime. Is it possible to create an object without knowing it's exact class type? The *virtual copy constructor* address this question.

Sometimes we may need to construct an object from another existing object. Precisely the copy constructor does the same. The initial state of new object will be based on another existing object state. The compiler places call to copy constructor when an object being instantiated from another object. However, the compiler needs concrete type information to invoke appropriate copy constructor.

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

class Base
{
public:
    //
};

class Derived : public Base
{
public:
    Derived()
    {
        cout << "Derived created" << endl;
    }
```

```
    Derived(const Derived &rhs)
    {
        cout << "Derived created by deep copy" << endl;
    }

    ~Derived()
    {
        cout << "Derived destroyed" << endl;
    }
};

int main()
{
    Derived s1;

    Derived s2 = s1;  // Compiler invokes "copy constructor"
                      // Type of s1 and s2 are concrete to compiler

    // How can we create Derived1 or Derived2 object
    // from pointer (reference) to Base class pointing Derived object?

    return 0;
}
```

What if we can't decide from which type of object, the copy construction to be made? For example, virtual constructor creates an object of class hierarchy at runtime based on some input. When we want to copy construct an object from another object created by virtual constructor, we can't use usual copy constructor. We need a special cloning function that can duplicate the object at runtime.

As an example, consider a drawing application. You can select an object already drawn on the canvas and paste one more instance of the same object. From the programmer perspective, we can't decide which object will be copy-pasted as it is runtime decision. We need virtual copy constructor to help.

Similarly, consider clip board application. A clip board can hold different type of objects, and copy objects from existing objects, pastes them on application canvas. Again, what type of object to be copied is a runtime decision. Virtual copy constructor fills the gap here. See the example below,

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

//// LIBRARY SRART
class Base
{
public:
    Base() { }

    virtual // Ensures to invoke actual object destructor
        ~Base() { }

    virtual void ChangeAttributes() = 0;

    // The "Virtual Constructor"
    static Base *Create(int id);

    // The "Virtual Copy Constructor"
    virtual Base *Clone() = 0;
};

class Derived1 : public Base
{
public:
    Derived1()
    {
        cout << "Derived1 created" << endl;
    }

    Derived1(const Derived1& rhs)
    {
        cout << "Derived1 created by deep copy" << endl;
    }

    ~Derived1()
```

```
    {
        cout << "~Derived1 destroyed" << endl;
    }

    void ChangeAttributes()
    {
        cout << "Derived1 Attributes Changed" << endl;
    }

    Base *Clone()
    {
        return new Derived1(*this);
    }
};

class Derived2 : public Base
{
public:
    Derived2()
    {
        cout << "Derived2 created" << endl;
    }

    Derived2(const Derived2& rhs)
    {
        cout << "Derived2 created by deep copy" << endl;
    }

    ~Derived2()
    {
        cout << "~Derived2 destroyed" << endl;
    }

    void ChangeAttributes()
    {
        cout << "Derived2 Attributes Changed" << endl;
    }

    Base *Clone()
    {
        return new Derived2(*this);
    }
};

class Derived3 : public Base
{
public:
    Derived3()
    {
        cout << "Derived3 created" << endl;
    }

    Derived3(const Derived3& rhs)
    {
        cout << "Derived3 created by deep copy" << endl;
    }

    ~Derived3()
    {
        cout << "~Derived3 destroyed" << endl;
    }

    void ChangeAttributes()
    {
        cout << "Derived3 Attributes Changed" << endl;
    }

    Base *Clone()
    {
        return new Derived3(*this);
    }
};
```

```
// We can also declare "Create" outside Base.
// But is more relevant to limit it's scope to Base
Base *Base::Create(int id)
{
   // Just expand the if-else ladder, if new Derived class is created
   // User need not be recompiled to create newly added class objects

   if( id == 1 )
   {
      return new Derived1;
   }
   else if( id == 2 )
   {
      return new Derived2;
   }
   else
   {
      return new Derived3;
   }
}
//// LIBRARY END

//// UTILITY SRART
class User
{
public:
   User() : pBase(0)
   {
      // Creates any object of Base heirarchey at runtime

      int input;

      cout << "Enter ID (1, 2 or 3): ";
      cin >> input;

      while( (input != 1) && (input != 2) && (input != 3) )
      {
         cout << "Enter ID (1, 2 or 3 only): ";
         cin >> input;
      }

      // Create objects via the "Virtual Constructor"
      pBase = Base::Create(input);
   }

   ~User()
   {
      if( pBase )
      {
         delete pBase;
         pBase = 0;
      }
   }

   void Action()
   {
      // Duplicate current object
      Base *pNewBase = pBase->Clone();

      // Change its attributes
      pNewBase->ChangeAttributes();

      // Dispose the created object
      delete pNewBase;
   }

private:
   Base *pBase;
};

//// UTILITY END
```

```
//// Consumer of User (UTILITY) class
int main()
{
   User *user = new User();

   user->Action();

   delete user;
}
```

User class creating an object with the help of virtual constructor. The object to be created is based on user input. *Action()* is making duplicate of object being created and modifying it's attributes. The duplicate object being created with the help of *Clone()* virtual function which is also considered as *virtual copy constructor*. The concept behind *Clone()* method is building block of *prototype pattern*.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
AdvanceCPP
CPP
About Venki
Software Engineer
View all posts by Venki →

C++ Internals | Default Constructors | Set 1

A constructor without any arguments or with default value for every argument, is said to be ***default constructor***. What is the significance of default constructor? Will the code be generated for every default constructor? Will there be any code inserted by compiler to the user implemented default constructor behind the scenes?

The compiler will implicitly *declare* default constructor if not provided by programmer, will *define* it when in need. Compiler defined default constructor is required to do certain initialization of class internals. It will not touch the data members or plain old data types (aggregates like array, structures, etc…). However, the compiler generates code for default constructor based on situation.

Consider a class derived from another class with default constructor, or a class containing another class object with default constructor. The compiler needs to insert code to call the default constructors of base class/embedded object.

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

class Base
{
public:
   // compiler "declares" constructor
};

class A
{
public:
   // User defined constructor
   A()
   {
      cout << "A Constructor" << endl;
   }

   // uninitialized
   int size;
};

class B : public A
{
   // compiler defines default constructor of B, and
   // inserts stub to call A constructor

   // compiler won't initialize any data of A
};

class C : public A
```

```
{
public:
   C()
   {
      // User defined default constructor of C
      // Compiler inserts stub to call A's construtor
      cout << "B Constructor" << endl;

      // compiler won't initialize any data of A
   }
};

class D
{
public:
   D()
   {
      // User defined default constructor of D
      // a - constructor to be called, compiler inserts
      // stub to call A constructor
      cout << "D Constructor" << endl;

      // compiler won't initialize any data of 'a'
   }

private:
   A a;
};

int main()
{
   Base base;

   B b;
   C c;
   D d;

   return 0;
}
```

There are different scenarios in which compiler needs to insert code to ensure some necessary initialization as per language requirement. We will have them in upcoming posts. Our objective is to be aware of C++ internals, not to use them incorrectly.

— by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

When are static objects destroyed?

Remain Careful from these two persons
new friends and old enemies — Kabir

What is static keyword in C++?

static keyword can be applied to local variables, functions, class' data members and objects in C++. static local variable retain their values between function call and initialized only once. static function can be directly called using the scope resolution operator preceded by class name (See this, this and this for more details). C++ also supports static objects.

What are static objects in C++?

An object become static when static keyword is used in its declaration. See the following two statements for example in C++.

```
Test t;          // Stack based object
static Test t1;  // Static object
```

First statement when executes creates object on stack means storage is allocated on stack. Stack based objects are also called

automatic objects or local objects. static object are initialized only once and live until the program terminates. Local object is created each time its declaration is encountered in the execution of program.

static objects are allocated storage in static storage area. static object is destroyed at the termination of program. C++ supports both local static object and global static object
Following is example that shows use of local static object.

```
#include <iostream>
class Test
{
public:
   Test()
   {
      std::cout << "Constructor is executed\n";
   }
   ~Test()
   {
      std::cout << "Destructor is executed\n";
   }
};
void myfunc()
{
   static Test obj;
} // Object obj is still not destroyed because it is static

int main()
{
   std::cout << "main() starts\n";
   myfunc();    // Destructor will not be called here
   std::cout << "main() terminates\n";
   return 0;
}
```

Output:

```
main() starts
Constructor is executed
main() terminates
Destructor is executed
```

If we observe the output of this program closely, we can see that the destructor for the local object named obj is not called after it's constructor is executed because the local object is static so it has scope till the lifetime of program so it's destructor will be called when main() terminates.

What happens when we remove static in above program?
As an experiment if we remove the static keyword from the global function myfunc(), we get the output as below:

```
main() starts
Constructor is called
Destructor is called
main() terminates
```

This is because the object is now stack based object and it is destroyed when it is goes out of scope and its destructor will be called.

How about global static objects?
The following program demonstrates use of global static object.

```
#include <iostream>
class Test
{
public:
   int a;
   Test()
   {
      a = 10;
      std::cout << "Constructor is executed\n";
   }
   ~Test()
   {
```

```
    std::cout << "Destructor is executed\n";
  }
};
static Test obj;
int main()
{
  std::cout << "main() starts\n";
  std::cout << obj.a;
  std::cout << "\nmain() terminates\n";
  return 0;
}
```

Output:

```
Constructor is executed
main() starts
10
main() terminates
Destructor is executed
```

This article is contributed by **Meet Pravasi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-class
cpp-constuctor
cpp-storage-classes

Is it possible to call constructor and destructor explicitly?

```
All the days of the afflicted are evil but he
that is of a merry heart hath a continual feast.
Proverbs 15:15 (Bible)
```

Constructor is a special member function that is automatically called by compiler when object is created and destructor is also special member function that is also implicitly called by compiler when object goes out of scope. They are also called when dynamically allocated object is allocated and destroyed, new operator allocates storage and calls constructor, delete operator calls destructor and free the memory allocated by new.

Is it possible to call constructor and destructor explicitly?
Yes, it is possible to call special member functions explicitly by programmer. Following program calls constructor and destructor explicitly.

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

class Test
{
public:
  Test()  { cout << "Constructor is executed\n"; }
  ~Test() { cout << "Destructor is executed\n";  }
};

int main()
{
  Test();  // Explicit call to constructor
  Test t; // local object
  t.~Test(); // Explicit call to destructor
  return 0;
}
```

Output:

```
Constructor is executed
Destructor is executed
```

```
Constructor is executed
Destructor is executed
Destructor is executed
```

When the constructor is called explicitly the compiler creates a nameless temporary object and it is immediately destroyed. That's why 2nd line in the output is call to destructor.
Here is a conversation between me and Dr. Bjarne Stroustrup via mail about this topic:

Me: Why C++ allows to call constructor explicitly? Don't you think that this shouldn't be?
Dr. Bjarne: No. temporary objects of the class types are useful.

Section 12.4/14 of C++ standard says that
Once a destructor is invoked for an object, the object no longer exists; the behavior is undefined if the destructor is invoked for an object whose lifetime has ended [Example: if the destructor for an automatic object is explicitly invoked, and the block is subsequently left in a manner that would ordinarily invoke implicit destruction of the object, the behavior is undefined. —end example].

As mentioned here, we should never call destructor explicitly on local (automatic) object, because really bad results can be acquired by doing that.

Local objects are automatically destroyed by compiler when they go out of scope and this is the guarantee of C++ language. In general, special member functions shouldn't be called explicitly.
Constructor and destructor can also be called from the member function of class. See following program:

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

class Test
{
public:
   Test()  { cout << "Constructor is executed\n"; }
   ~Test() { cout << "Destructor is executed\n";  }
   void show()  {  Test();  this->Test::~Test();  }
};

int main()
{
   Test t;
   t.show();
   return 0;
}
```

Output:

```
Constructor is executed
Constructor is executed
Destructor is executed
Destructor is executed
Destructor is executed
```

Explicit call to destructor is only necessary when object is placed at particular location in memory by using placement new. Destructor should not be called explicitly when the object is dynamically allocated because delete operator automatically calls destructor.

As an exercise predict the output of following program:

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

class Test
{
public:
   Test()  { cout << "Constructor is executed\n"; }
   ~Test() { cout << "Destructor is executed\n";  }
   friend void fun(Test t);
};
void fun(Test t)
{
   Test();
   t.~Test();
}
```

```
int main()
{
   Test();
   Test t;
   fun(t);
   return 0;
}
```

Sources:

http://www.parashift.com/c++-faq/dont-call-dtor-on-local.html

http://www.parashift.com/c++-faq/placement-new.html

http://msdn.microsoft.com/en-us/library/35xa3368.aspx

This article is contributed **Meet Pravasi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-constructor

What all is inherited from parent class in C++?

Following are the things which a derived class inherits from its parent.

1) Every data member defined in the parent class (although such members may not always be accessible in the derived class!)

2) Every ordinary member function of the parent class (although such members may not always be accessible in the derived class!)

3) The same initial data layout as the base class.

Following are the things which a derived class doesn't inherits from its parent :

1) The base class's constructors and destructor.

2) The base class's friends

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
GFacts

Virtual Functions and Runtime Polymorphism in C++ | Set 1 (Introduction)

Consider the following simple program which is an example of runtime polymorphism.

The main thing to note about the program is, derived class function is called using a base class pointer. The idea is, virtual functions are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.

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

class Base
{
public:
   virtual void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
public:
   void show() { cout<<"In Derived \n"; }
};

int main(void)
```

```
{
   Base *bp = new Derived;
   bp->show(); // RUN-TIME POLYMORPHISM
   return 0;
}
```

Output:

```
In Derived
```

What is the use?

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class *Employee* , the class contains virtual functions like *raiseSalary()*, *transfer()*, *promote()*,.. etc. Different types of employees like *Manager*, *Engineer*, ..etc may have their own implementations of the virtual functions present in base class *Employee*. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise salary of all employees by iterating through list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if *raiseSalary()* is present for a specific employee type, only that function would be called.

```
class Employee
{
public:
   virtual void raiseSalary()
   {  /* common raise salary code */ }

   virtual void promote()
   { /* common promote code */ }
};

class Manager: public Employee {
   virtual void raiseSalary()
   {  /* Manager specific raise salary code, may contain
         increment of manager specific incentives*/  }

   virtual void promote()
   { /* Manager specific promote */ }
};

// Similarly, there may be other types of employees

// We need a very simple function to increment salary of all employees
// Note that emp[] is an array of pointers and actual pointed objects can
// be any type of employees. This function should ideally be in a class
// like Organization, we have made it global to keep things simple
void globalRaiseSalary(Employee *emp[], int n)
{
   for (int i = 0; i < n; i++)
      emp[i]->raiseSalary(); // Polymorphic Call: Calls raiseSalary()
                             // according to the actual object, not
                             // according to the type of pointer
}
```

like *globalRaiseSalary()*, there can be many other operations that can be appropriately done on a list of employees without even knowing the type of actual object.

Virtual functions are so useful that later languages like Java keep all methods as virtual by default.

How does compiler do this magic of late resolution?

Compiler maintains two things to this magic:

virtualFuns

vtable: A table of function pointers. It is maintained per class.
vptr: A pointer to vtable. It is maintained per object (See this for an example).

Compiler adds additional code at two places to maintain and use *vptr*.
1) Code in every constructor. This code sets vptr of the object being created. This code sets *vptr* to point to *vtable* of the class.
2) Code with polymorphic function call (e.g. *bp->show()* in above code). Wherever a polymorphic call is made, compiler inserts code to first look for *vptr* using base class pointer or reference (In the above example, since pointed or referred object is of derived type, vptr of derived class is accessed). Once *vptr* is fetched, *vtable* of derived class can be accessed. Using *vtable*, address of derived derived class function *show()* is accessed and called.

Is this a standard way for implementation of run-time polymorphism in C++?
The C++ standards do not mandate exactly how runtime polymophism must be implemented, but compilers generally use minor variations on the same basic model.

Quiz on Virtual Functions.

References:
http://en.wikipedia.org/wiki/Virtual_method_table
http://en.wikipedia.org/wiki/Virtual_function
http://www.drbio.cornell.edu/pl47/programming/TICPP-2nd-ed-Vol-one-html/Frames.html

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-functions
cpp-inheritance
cpp-virtual

Multiple Inheritance in C++

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.

The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, B's constructor is called before A's constructor.

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

class A
{
public:
  A()  { cout << "A's constructor called" << endl; }
};

class B
{
public:
  B()  { cout << "B's constructor called" << endl; }
};

class C: public B, public A  // Note the order
{
public:
```

```
  C()  { cout << "C's constructor called" << endl; }
};

int main()
{
   C c;
   return 0;
}
```

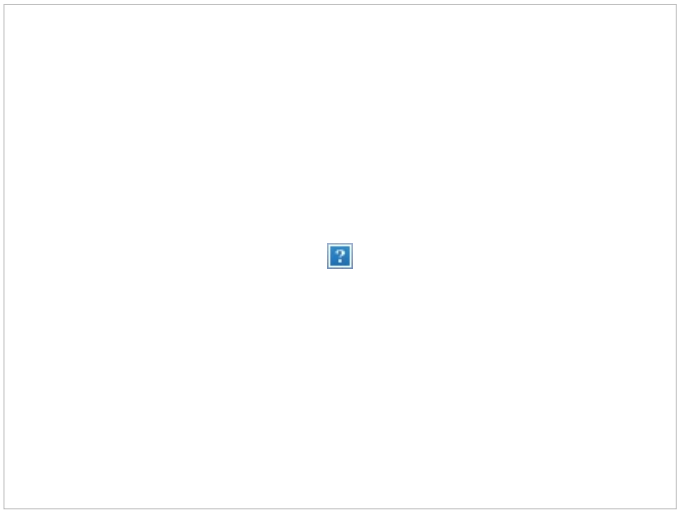
Output:

```
B's constructor called
A's constructor called
C's constructor called
```

The destructors are called in reverse order of constructors.

The diamond problem

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
  // Data members of person
public:
   Person(int x)  { cout << "Person::Person(int ) called" << endl;  }
};

class Faculty : public Person {
  // data members of Faculty
public:
   Faculty(int x):Person(x)  {
     cout<<"Faculty::Faculty(int ) called"<< endl;
   }
};

class Student : public Person {
  // data members of Student
public:
   Student(int x):Person(x) {
      cout<<"Student::Student(int ) called"<< endl;
   }
};

class TA : public Faculty, public Student  {
public:
   TA(int x):Student(x), Faculty(x)  {
      cout<<"TA::TA(int ) called"<< endl;
   }
};
```

```
int main()  {
   TA ta1(30);
}
```

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

In the above program, constructor of 'Person' is called two times. Destructor of 'Person' will also be called two times when object 'ta1' is destructed. So object 'ta1' has two copies of all members of 'Person', this causes ambiguities. *The solution to this problem is 'virtual' keyword*. We make the classes 'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class. For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
   Person(int x)  { cout << "Person::Person(int ) called" << endl;  }
   Person()    { cout << "Person::Person() called" << endl;  }
};

class Faculty : virtual public Person {
public:
   Faculty(int x):Person(x)  {
     cout<<"Faculty::Faculty(int ) called"<< endl;
   }
};

class Student : virtual public Person {
public:
   Student(int x):Person(x) {
      cout<<"Student::Student(int ) called"<< endl;
   }
};

class TA : public Faculty, public Student  {
public:
   TA(int x):Student(x), Faculty(x)  {
      cout<<"TA::TA(int ) called"<< endl;
   }
};

int main()  {
   TA ta1(30);
}
```

Output:

```
Person::Person() called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

In the above program, constructor of 'Person' is called once. One important thing to note in the above output is, *the default constructor of 'Person' is called*. When we use 'virtual' keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

How to call the parameterized constructor of the 'Person' class? The constructor has to be called in 'TA' class. For example, see the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
   Person(int x)  { cout << "Person::Person(int ) called" << endl;  }
   Person()    { cout << "Person::Person() called" << endl;  }
```

```
};

class Faculty : virtual public Person {
public:
   Faculty(int x):Person(x)   {
     cout<<"Faculty::Faculty(int ) called"<< endl;
   }
};

class Student : virtual public Person {
public:
   Student(int x):Person(x) {
      cout<<"Student::Student(int ) called"<< endl;
   }
};

class TA : public Faculty, public Student  {
public:
   TA(int x):Student(x), Faculty(x), Person(x)   {
      cout<<"TA::TA(int ) called"<< endl;
   }
};

int main()  {
   TA ta1(30);
}
```

Output:

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

In general, it is not allowed to call the grandparent's constructor directly, it has to be called through parent class. It is allowed only when 'virtual' keyword is used.

As an exercise, predict the output of following programs.

Question 1

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

class A
{
  int x;
public:
  void setX(int i) {x = i;}
  void print() { cout << x; }
};

class B:  public A
{
public:
  B()  { setX(10); }
};

class C:  public A
{
public:
  C()  { setX(20); }
};

class D: public B, public C {
};

int main()
{
   D d;
   d.print();
```

```
    return 0;
}
```

Question 2

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

class A
{
  int x;
public:
  A(int i) { x = i; }
  void print() { cout << x; }
};

class B: virtual public A
{
public:
  B():A(10) { }
};

class C:  virtual public A
{
public:
  C():A(10) { }
};

class D: public B, public C {
};

int main()
{
    D d;
    d.print();
    return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
C++

What happens when more restrictive access is given to a derived class method in C++?

We have discussed a similar topic in Java here. Unlike Java, C++ allows to give more restrictive access to derived class methods. For example the following program compiles fine.

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

class Base {
public:
    virtual int fun(int i) { }
};

class Derived: public Base {
private:
    int fun(int x)   { }
};

int main()
{ }
```

In the above program, if we change main() to following, will get compiler error becuase fun() is private in derived class.

```
int main()
{
   Derived d;
   d.fun(1);
   return 0;
}
```

What about the below program?

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

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

class Derived: public Base {
private:
   int fun(int x)   { cout << "Derived::fun(int x) called"; }
};

int main()
{
   Base *ptr = new Derived;
   ptr->fun(10);
   return 0;
}
```

Output:

```
 Derived::fun(int x) called
```

In the above program, private function “Derived::fun(int)” is being called through a base class pointer, the program works fine because fun() is public in base class. Access specifiers are checked at compile time and fun() is public in base class. At run time, only the function corresponding to the pointed object is called and access specifier is not checked. So a private function of derived class is being called through a pointer of base class.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
C++

Object Slicing in C++

In C++, a derived class object can be assigned to a base class object, but the other way is not possible.

```
class Base { int x, y; };

class Derived : public Base { int z, w; };

int main()
{
   Derived d;
   Base b = d; // Object Slicing,  z and w of d are sliced off
}
```

Object slicing happens when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off to form the base class object.

```
#include <iostream>
```

```
using namespace std;

class Base
{
protected:
   int i;
public:
   Base(int a)    { i = a; }
   virtual void display()
   { cout << "I am Base class object, i = " << i << endl; }
};

class Derived : public Base
{
   int j;
public:
   Derived(int a, int b) : Base(a) { j = b; }
   virtual void display()
   { cout << "I am Derived class object, i = "
         << i << ", j = " << j << endl; }
};

// Global method, Base class object is passed by value
void somefunc (Base obj)
{
   obj.display();
}

int main()
{
   Base b(33);
   Derived d(45, 54);
   somefunc(b);
   somefunc(d);  // Object Slicing, the member j of d is sliced off
   return 0;
}
```

Output:

```
I am Base class object, i = 33
I am Base class object, i = 45
```

We can avoid above unexpected behavior with the use of pointers or references. Object slicing doesn't occur when pointers or references to objects are passed as function arguments since a pointer or reference of any type takes same amount of memory. For example, if we change the global method myfunc() in the above program to following, object slicing doesn't happen.

```
// rest of code is similar to above
void somefunc (Base &obj)
{
   obj.display();
}
// rest of code is similar to above
```

Output:

```
I am Base class object, i = 33
I am Derived class object, i = 45, j = 54
```

We get the same output if we use pointers and change the program to following.

```
// rest of code is similar to above
void somefunc (Base *objp)
{
   objp->display();
}

int main()
{
   Base *bp = new Base(33) ;
   Derived *dp = new Derived(45, 54);
```

```
    somefunc(bp);
    somefunc(dp);  // No Object Slicing
    return 0;
}
```

Output:

```
I am Base class object, i = 33
I am Derived class object, i = 45, j = 54
```

Object slicing can be prevented by making the base class function pure virtual there by disallowing object creation. It is not possible to create the object of a class which contains a pure virtual method.

This article is contributed by **Pravasi Meet**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-class

Hiding of all overloaded methods with same name in base class

In C++, if a derived class redefines base class member method then all the base class methods with same name become hidden in derived class.
For example, the following program doesn't compile. In the following program, Derived redefines Base's method fun() and this makes fun(int i) hidden.

```
#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);  // Compiler Error
   return 0;
}
```

Even if the signature of the derived class method is different, all the overloaded methods in base class become hidden. For example, in the following program, Derived::fun(char) makes both Base::fun() and Base::fun(int) hidden.

```
#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(char c)  // Makes Base::fun() and Base::fun(int ) hidden
   {
      cout<<"Derived::fun(char c) called";
   }
};

int main()
{
   Derived d;
   d.fun();  // Compiler Error
   return 0;
}
```

Note that the above facts are true for both static and nonstatic methods.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Friend class and function in C++

Friend Class A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

```
class Node
{
private:
  int key;
  Node *next;
  /* Other members of Node Class */

  friend class LinkedList; // Now class  LinkedList can
                           // access private members of Node
};
```

Friend Function Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:

a) A method of another class

b) A global function

```
class Node
{
private:
  int key;
  Node *next;

  /* Other members of Node Class */
```

```
  friend int LinkedList::search(); // Only search() of linkedList
                                   // can access internal members
};
```

Following are some important points about friend functions and classes:
1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

2) Friendship is not mutual. If a class A is friend of B, then B doesn't become friend of A automatically.

3) Friendship is not inherited (See this for more details)

4) The concept of friends is not there in Java.

A simple and complete C++ program to demonstrate friend Class

```
#include <iostream>
class A {
private:
   int a;
public:
   A() { a=0; }
   friend class B;    // Friend Class
};

class B {
private:
   int b;
public:
   void showA(A& x) {
      // Since B is friend of A, it can access
      // private members of A
      std::cout << "A::a=" << x.a;
   }
};

int main() {
  A a;
  B b;
  b.showA(a);
  return 0;
}
```

Output:

```
A::a=0
```

A simple and complete C++ program to demonstrate friend function of another class

```
#include <iostream>

class B;

class A
{
public:
   void showB(B& );
};

class B
{
private:
   int b;
public:
   B()  {  b = 0; }
   friend void A::showB(B& x); // Friend function
};

void A::showB(B &x)
```

```
{
    // Since show() is friend of B, it can
    // access private members of B
    std::cout << "B::b = " << x.b;
}

int main()
{
    A a;
    B x;
    a.showB(x);
    return 0;
}
```

Output:

```
B::b = 0
```

A simple and complete C++ program to demonstrate global friend

```
#include <iostream>

class A
{
    int a;
public:
    A() {a = 0;}
    friend void showA(A&); // global friend function
};

void showA(A& x) {
    // Since showA() is a friend, it can access
    // private members of A
    std::cout << "A::a=" << x.a;
}

int main()
{
    A a;
    showA(a);
    return 0;
}
```

Output:

```
A::a = 0
```

References:

http://en.wikipedia.org/wiki/Friend_class

http://en.wikipedia.org/wiki/Friend_function

http://www.cprogramming.com/tutorial/friends.html

http://www.parashift.com/c++-faq/friends-and-encap.html

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Inheritance and friendship

In C++, friendship is not inherited. If a base class has a friend function, then the function doesn't become a friend of the derived class(es).

For example, following program prints error because *show()* which is a friend of base class *A* tries to access private data of derived class *B*.

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

class A
{
  protected:
    int x;
  public:
    A() { x = 0;}
  friend void show();
};

class B: public A
{
  public:
    B() : y (0) {}
  private:
    int y;
};

void show()
{
  B b;
  cout << "The default value of A::x = " << b.x;

  // Can't access private member declared in class 'B'
  cout << "The default value of B::y = " << b.y;
}

int main()
{
  show();
  getchar();
  return 0;
}
```

Thanks to Venki for the above code and explanation.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Virtual Functions and Runtime Polymorphism in C++ | Set 1 (Introduction)

Consider the following simple program which is an example of runtime polymorphism.

The main thing to note about the program is, derived class function is called using a base class pointer. The idea is, virtual functions are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.

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

class Base
{
public:
    virtual void show() { cout<<" In Base \n"; }
};
```

```
class Derived: public Base
{
public:
   void show() { cout<<"In Derived \n"; }
};

int main(void)
{
   Base *bp = new Derived;
   bp->show();  // RUN-TIME POLYMORPHISM
   return 0;
}
```

Output:

```
In Derived
```

What is the use?

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class *Employee* , the class contains virtual functions like *raiseSalary()*, *transfer()*, *promote()*,.. etc. Different types of employees like *Manager*, *Engineer*, ..etc may have their own implementations of the virtual functions present in base class *Employee*. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise salary of all employees by iterating through list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if *raiseSalary()* is present for a specific employee type, only that function would be called.

```
class Employee
{
public:
   virtual void raiseSalary()
   {  /* common raise salary code */ }

   virtual void promote()
   { /* common promote code */ }
};

class Manager: public Employee {
   virtual void raiseSalary()
   {  /* Manager specific raise salary code, may contain
         increment of manager specific incentives*/ }

   virtual void promote()
   { /* Manager specific promote */ }
};

// Similarly, there may be other types of employees

// We need a very simple function to increment salary of all employees
// Note that emp[] is an array of pointers and actual pointed objects can
// be any type of employees. This function should ideally be in a class
// like Organization, we have made it global to keep things simple
void globalRaiseSalary(Employee *emp[], int n)
{
   for (int i = 0; i < n; i++)
      emp[i]->raiseSalary(); // Polymorphic Call: Calls raiseSalary()
                             // according to the actual object, not
                             // according to the type of pointer
}
```

like *globalRaiseSalary()*, there can be many other operations that can be appropriately done on a list of employees without even knowing the type of actual object.

Virtual functions are so useful that later languages like Java keep all methods as virtual by default.

How does compiler do this magic of late resolution?

Compiler maintains two things to this magic:

virtualFuns

vtable: A table of function pointers. It is maintained per class.
vptr: A pointer to vtable. It is maintained per object (See this for an example).

Compiler adds additional code at two places to maintain and use *vptr*.
1) Code in every constructor. This code sets vptr of the object being created. This code sets *vptr* to point to *vtable* of the class.
2) Code with polymorphic function call (e.g. *bp->show()* in above code). Wherever a polymorphic call is made, compiler inserts code to first look for *vptr* using base class pointer or reference (In the above example, since pointed or referred object is of derived type, vptr of derived class is accessed). Once *vptr* is fetched, *vtable* of derived class can be accessed. Using *vtable*, address of derived derived class function *show()* is accessed and called.

Is this a standard way for implementation of run-time polymorphism in C++?
The C++ standards do not mandate exactly how runtime polymophism must be implemented, but compilers generally use minor variations on the same basic model.

Quiz on Virtual Functions.

References:
http://en.wikipedia.org/wiki/Virtual_method_table
http://en.wikipedia.org/wiki/Virtual_function
http://www.drbio.cornell.edu/pl47/programming/TICPP-2nd-ed-Vol-one-html/Frames.html

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-functions
cpp-inheritance
cpp-virtual

Default arguments and virtual function

Predict the output of following C++ program.

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

class Base
{
public:
   virtual void fun ( int x = 0 )
   {
      cout << "Base::fun(), x = " << x << endl;
   }
};

class Derived : public Base
{
public:
   virtual void fun ( int x )
   {
      cout << "Derived::fun(), x = " << x << endl;
   }
};
```

```
int main()
{
   Derived d1;
   Base *bp = &d1;
   bp->fun();
   return 0;
}
```

Output:

```
Derived::fun(), x = 0
```

If we take a closer look at the output, we observe that fun() of derived class is called and default value of base class fun() is used.
Default arguments do not participate in signature of functions. So signatures of fun() in base class and derived class are considered same, hence the fun() of base class is overridden. Also, the default value is used at compile time. When compiler sees that an argument is missing in a function call, it substitutes the default value given. Therefore, in the above program, value of x is substituted at compile time, and at run time derived class's fun() is called.

Now predict the output of following program.

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

class Base
{
public:
   virtual void fun ( int x = 0)
   {
      cout << "Base::fun(), x = " << x << endl;
   }
};

class Derived : public Base
{
public:
   virtual void fun ( int x = 10 ) // NOTE THIS CHANGE
   {
      cout << "Derived::fun(), x = " << x << endl;
   }
};


int main()
{
   Derived d1;
   Base *bp = &d1;
   bp->fun();
   return 0;
}
```

The output of this program is same as the previous program. The reason is same, the default value is substituted at compile time. The fun() is called on bp which is a pointer of Base type. So compiler substitutes 0 (not 10).

In general, it is a best practice to avoid default values in virtual functions to avoid confusion (See this for more details)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Virtual functions in derived classes

In C++, once a member function is declared as a virtual function in a base class, it becomes virtual in every class derived from that base class. In other words, it is not necessary to use the keyword virtual in the derived class while declaring redefined versions of the virtual base class function.

Source: http://www.umsl.edu/~subramaniana/virtual2.html

For example, the following program prints “C::fun() called” as B::fun() becomes virtual automatically.

```
#include<iostream>

using namespace std;

class A {
  public:
    virtual void fun()
    { cout<<"\n A::fun() called ";}
};

class B: public A {
  public:
    void fun()
    { cout<<"\n B::fun() called "; }
};

class C: public B {
  public:
    void fun()
    { cout<<"\n C::fun() called "; }
};

int main()
{
   C c; // An object of class C
   B *b = &c; // A pointer of type B* pointing to c
   b->fun();  // this line prints "C::fun() called"
   getchar();
   return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Can static functions be virtual in C++?

In C++, a *static* member function of a class cannot be *virtual*. For example, below program gives compilation error.

```
#include<iostream>

using namespace std;

class Test
{
   public:
      // Error: Virtual member functions cannot be static
      virtual static void fun()  { }
};
```

Also, *static* member function cannot be *const* and *volatile*. Following code also fails in compilation.

```
#include<iostream>

using namespace std;

class Test
{
   public:
      // Error: Static member function cannot be const
      static void fun() const { }
```

```
};
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Virtual Destructor

Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.
Source: https://www.securecoding.cert.org/confluence/display/cplusplus/OOP34-CPP.+Ensure+the+proper+destructor+is+called+for+polymorphic+objects

For example, following program results in undefined behavior. Although the output of following program may be different on different compilers, when compiled using Dev-CPP, it prints following.
Constructing base
Constructing derived
Destructing base

```
// A program without virtual destructor causing undefined behavior
#include<iostream>

using namespace std;

class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called. For example, following program prints:
Constructing base
Constructing derived
Destructing derived
Destructing base

```
// A program with virtual destructor
#include<iostream>

using namespace std;

class base {
  public:
    base()
```

```
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

This article is contributed by **Rahul Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Advanced C++ | Virtual Constructor

Can we make a class constructor *virtual* in C++ to create polymorphic objects? No. C++ being static typed (the purpose of RTTI is different) language, it is meaningless to the C++ compiler to create an object polymorphically. The compiler must be aware of the class type to create the object. In other words, what type of object to be created is a compile time decision from C++ compiler perspective. If we make constructor virtual, compiler flags an error. In fact except *inline*, no other keyword is allowed in the declaration of constructor.

In practical scenarios we would need to create a derived class object in a class hierarchy based on some input. Putting in other words, *object creation and object type are tightly coupled which forces modifications to extended. The objective of virtual constructor is to decouple object creation from it's type*.

How can we create required type of object at runtime? For example, see the following sample program.

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

//// LIBRARY START
class Base
{
public:

    Base() { }

    virtual // Ensures to invoke actual object destructor
    ~Base() { }

    // An interface
    virtual void DisplayAction() = 0;
};

class Derived1 : public Base
{
public:
    Derived1()
    {
        cout << "Derived1 created" << endl;
```

```
    }

    ~Derived1()
    {
        cout << "Derived1 destroyed" << endl;
    }

    void DisplayAction()
    {
        cout << "Action from Derived1" << endl;
    }
};

class Derived2 : public Base
{
public:
    Derived2()
    {
        cout << "Derived2 created" << endl;
    }

    ~Derived2()
    {
        cout << "Derived2 destroyed" << endl;
    }

    void DisplayAction()
    {
        cout << "Action from Derived2" << endl;
    }
};

//// LIBRARY END

class User
{
public:

    // Creates Drived1
    User() : pBase(0)
    {
        // What if Derived2 is required? - Add an if-else ladder (see next sample)
        pBase = new Derived1();
    }

    ~User()
    {
        if( pBase )
        {
            delete pBase;
            pBase = 0;
        }
    }

    // Delegates to actual object
    void Action()
    {
        pBase->DisplayAction();
    }

private:
    Base *pBase;
};

int main()
{
    User *user = new User();

    // Need Derived1 functionality only
    user->Action();

    delete user;
```

```
}
```

In the above sample, assume that the hierarchy *Base*, *Derived1* and *Derived2* are part of library code. The class *User* is utility class trying to make use of the hierarchy. The *main* function is consuming *Base* hierarchy functionality via *User* class.

The *User* class constructor is creating *Derived1* object, always. If the *User*'s consumer (the *main* in our case) needs *Derived2* functionality, *User* needs to create "***new Derived2()***" and it forces recompilation. Recompiling is bad way of design, so we can opt for the following approach.

Before going into details, let us answer, who will dictate to create either of *Derived1* or *Derived2* object? Clearly, it is the consumer of *User* class. The *User* class can make use of if-else ladder to create either *Derived1* or *Derived2*, as shown in the following sample,

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

//// LIBRARY START
class Base
{
public:
   Base() { }

   virtual // Ensures to invoke actual object destructor
   ~Base() { }

   // An interface
   virtual void DisplayAction() = 0;
};

class Derived1 : public Base
{
public:
   Derived1()
   {
      cout << "Derived1 created" << endl;
   }

   ~Derived1()
   {
      cout << "Derived1 destroyed" << endl;
   }

   void DisplayAction()
   {
      cout << "Action from Derived1" << endl;
   }
};

class Derived2 : public Base
{
public:
   Derived2()
   {
      cout << "Derived2 created" << endl;
   }

   ~Derived2()
   {
      cout << "Derived2 destroyed" << endl;
   }

   void DisplayAction()
   {
      cout << "Action from Derived2" << endl;
   }
};

//// LIBRARY END

class User
{
public:
```

```
    // Creates Derived1 or Derived2 based on input
    User() : pBase(0)
    {
        int input; // ID to distinguish between
                   // Derived1 and Derived2

        cout << "Enter ID (1 or 2): ";
        cin  >> input;

        while( (input !=  1) && (input !=  2) )
        {
            cout << "Enter ID (1 or 2 only): ";
            cin  >> input;
        }

        if( input == 1 )
        {
            pBase = new Derived1;
        }
        else
        {
            pBase = new Derived2;
        }

        // What if Derived3 being added to the class hierarchy?
    }

    ~User()
    {
        if( pBase )
        {
            delete pBase;
            pBase = 0;
        }
    }

    // Delegates to actual object
    void Action()
    {
        pBase->DisplayAction();
    }

private:
    Base *pBase;
};

int main()
{
    User *user = new User();

    // Need either Derived1 or Derived2 functionality
    user->Action();

    delete user;
}
```

The above code is *not* open for extension, an inflexible design. In simple words, if the library updates the *Base* class hierarchy with new class *Derived3*. How can the *User* class creates *Derived3* object? One way is to update the if-else ladder that creates *Derived3* object based on new input ID 3 as shown below,

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

class User
{
public:
    User() : pBase(0)
    {
        // Creates Drived1 or Derived2 based on need
```

```
    int input; // ID to distinguish between
            // Derived1 and Derived2

    cout << "Enter ID (1 or 2): ";
    cin  >> input;

    while( (input !=  1) && (input !=  2) )
    {
      cout << "Enter ID (1 or 2 only): ";
      cin  >> input;
    }

    if( input == 1 )
    {
      pBase = new Derived1;
    }
    else if( input == 2 )
    {
      pBase = new Derived2;
    }
    else
    {
      pBase = new Derived3;
    }
  }

  ~User()
  {
    if( pBase )
    {
      delete pBase;
      pBase = 0;
    }
  }

  // Delegates to actual object
  void Action()
  {
    pBase->DisplayAction();
  }

private:
  Base *pBase;
};
```

The above modification forces the users of *User* class to recompile, bad (inflexible) design! And won't close *User* class from further modifications due to *Base* extension.

The problem is with the creation of objects. Addition of new class to the hierarchy forcing dependents of *User* class to recompile. Can't we delegate the action of creating objects to class hierarchy itself or to a function that behaves virtually? By delegating the object creation to class hierarchy (or to a static function) we can avoid the tight coupling between *User* and *Base* hierarchy. Enough theory, see the following code,

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

//// LIBRARY START
class Base
{
public:

  // The "Virtual Constructor"
  static Base *Create(int id);

  Base() { }

  virtual // Ensures to invoke actual object destructor
  ~Base() { }

  // An interface
  virtual void DisplayAction() = 0;
```

```
};

class Derived1 : public Base
{
public:
   Derived1()
   {
      cout << "Derived1 created" << endl;
   }

   ~Derived1()
   {
      cout << "Derived1 destroyed" << endl;
   }

   void DisplayAction()
   {
      cout << "Action from Derived1" << endl;
   }
};

class Derived2 : public Base
{
public:
   Derived2()
   {
      cout << "Derived2 created" << endl;
   }

   ~Derived2()
   {
      cout << "Derived2 destroyed" << endl;
   }

   void DisplayAction()
   {
      cout << "Action from Derived2" << endl;
   }
};

class Derived3 : public Base
{
public:
   Derived3()
   {
      cout << "Derived3 created" << endl;
   }

   ~Derived3()
   {
      cout << "Derived3 destroyed" << endl;
   }

   void DisplayAction()
   {
      cout << "Action from Derived3" << endl;
   }
};

// We can also declare "Create" outside Base
// But it is more relevant to limit it's scope to Base
Base *Base::Create(int id)
{
   // Just expand the if-else ladder, if new Derived class is created
   // User code need not be recompiled to create newly added class objects

   if( id == 1 )
   {
      return new Derived1;
   }
   else if( id == 2 )
   {
```

```
        return new Derived2;
    }
    else
    {
        return new Derived3;
    }
}
//// LIBRARY END

//// UTILITY START
class User
{
public:
    User() : pBase(0)
    {
        // Receives an object of Base heirarchy at runtime

        int input;

        cout << "Enter ID (1, 2 or 3): ";
        cin >> input;

        while( (input != 1) && (input != 2) && (input != 3) )
        {
            cout << "Enter ID (1, 2 or 3 only): ";
            cin >> input;
        }

        // Get object from the "Virtual Constructor"
        pBase = Base::Create(input);
    }

    ~User()
    {
        if( pBase )
        {
            delete pBase;
            pBase = 0;
        }
    }

    // Delegates to actual object
    void Action()
    {
        pBase->DisplayAction();
    }

private:
    Base *pBase;
};

//// UTILITY END

//// Consumer of User (UTILITY) class
int main()
{
    User *user = new User();

    // Action required on any of Derived objects
    user->Action();

    delete user;
}
```

The *User* class is independent of object creation. It delegates that responsibility to *Base*, and provides an input in the form of ID. If the library adds new class *Derived4*, the library modifier will extend the if-else ladder inside *Create* to return proper object. Consumers of *User* need not recompile their code due to extension of *Base*.

Note that the function *Create* used to return different types of *Base* class objects at runtime. It acts like virtual constructor, also referred as *Factory Method* in pattern terminology.

Pattern world demonstrate different ways to implement the above concept. Also there are some potential design issues with the above

code. Our objective is to provide some insights into virtual construction, creating objects dynamically based on some input. We have excellent books devoted to the subject, interested reader can refer them for more information.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

Advanced C++ | Virtual Copy Constructor

In the virtual constructor idiom we have seen the way to construct an object whose type is not determined until runtime. Is it possible to create an object without knowing it's exact class type? The *virtual copy constructor* address this question.

Sometimes we may need to construct an object from another existing object. Precisely the copy constructor does the same. The initial state of new object will be based on another existing object state. The compiler places call to copy constructor when an object being instantiated from another object. However, the compiler needs concrete type information to invoke appropriate copy constructor.

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

class Base
{
public:
   //
};

class Derived : public Base
{
public:
   Derived()
   {
      cout << "Derived created" << endl;
   }

   Derived(const Derived &rhs)
   {
      cout << "Derived created by deep copy" << endl;
   }

   ~Derived()
   {
      cout << "Derived destroyed" << endl;
   }
};

int main()
{
   Derived s1;

   Derived s2 = s1;  // Compiler invokes "copy constructor"
                     // Type of s1 and s2 are concrete to compiler

   // How can we create Derived1 or Derived2 object
   // from pointer (reference) to Base class pointing Derived object?

   return 0;
}
```

What if we can't decide from which type of object, the copy construction to be made? For example, virtual constructor creates an object of class hierarchy at runtime based on some input. When we want to copy construct an object from another object created by virtual constructor, we can't use usual copy constructor. We need a special cloning function that can duplicate the object at runtime.

As an example, consider a drawing application. You can select an object already drawn on the canvas and paste one more instance of the same object. From the programmer perspective, we can't decide which object will be copy-pasted as it is runtime decision. We need virtual copy constructor to help.

Similarly, consider clip board application. A clip board can hold different type of objects, and copy objects from existing objects, pastes them on application canvas. Again, what type of object to be copied is a runtime decision. Virtual copy constructor fills the gap here. See the example below,

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

//// LIBRARY SRART
class Base
{
public:
   Base() { }

   virtual // Ensures to invoke actual object destructor
      ~Base() { }

   virtual void ChangeAttributes() = 0;

   // The "Virtual Constructor"
   static Base *Create(int id);

   // The "Virtual Copy Constructor"
   virtual Base *Clone() = 0;
};

class Derived1 : public Base
{
public:
   Derived1()
   {
      cout << "Derived1 created" << endl;
   }

   Derived1(const Derived1& rhs)
   {
      cout << "Derived1 created by deep copy" << endl;
   }

   ~Derived1()
   {
      cout << "~Derived1 destroyed" << endl;
   }

   void ChangeAttributes()
   {
      cout << "Derived1 Attributes Changed" << endl;
   }

   Base *Clone()
   {
      return new Derived1(*this);
   }
};

class Derived2 : public Base
{
public:
   Derived2()
   {
      cout << "Derived2 created" << endl;
   }

   Derived2(const Derived2& rhs)
   {
      cout << "Derived2 created by deep copy" << endl;
   }

   ~Derived2()
   {
      cout << "~Derived2 destroyed" << endl;
   }
```

```
    void ChangeAttributes()
    {
        cout << "Derived2 Attributes Changed" << endl;
    }

    Base *Clone()
    {
        return new Derived2(*this);
    }
};

class Derived3 : public Base
{
public:
    Derived3()
    {
        cout << "Derived3 created" << endl;
    }

    Derived3(const Derived3& rhs)
    {
        cout << "Derived3 created by deep copy" << endl;
    }

    ~Derived3()
    {
        cout << "~Derived3 destroyed" << endl;
    }

    void ChangeAttributes()
    {
        cout << "Derived3 Attributes Changed" << endl;
    }

    Base *Clone()
    {
        return new Derived3(*this);
    }
};

// We can also declare "Create" outside Base.
// But is more relevant to limit it's scope to Base
Base *Base::Create(int id)
{
    // Just expand the if-else ladder, if new Derived class is created
    // User need not be recompiled to create newly added class objects

    if( id == 1 )
    {
        return new Derived1;
    }
    else if( id == 2 )
    {
        return new Derived2;
    }
    else
    {
        return new Derived3;
    }
}
//// LIBRARY END

//// UTILITY SRART
class User
{
public:
    User() : pBase(0)
    {
        // Creates any object of Base heirarchey at runtime

        int input;
```

```
    cout << "Enter ID (1, 2 or 3): ";
    cin >> input;

    while( (input != 1) && (input != 2) && (input != 3) )
    {
       cout << "Enter ID (1, 2 or 3 only): ";
       cin >> input;
    }

    // Create objects via the "Virtual Constructor"
    pBase = Base::Create(input);
  }

  ~User()
  {
    if( pBase )
    {
       delete pBase;
       pBase = 0;
    }
  }

  void Action()
  {
    // Duplicate current object
    Base *pNewBase = pBase->Clone();

    // Change its attributes
    pNewBase->ChangeAttributes();

    // Dispose the created object
    delete pNewBase;
  }

private:
  Base *pBase;
};

//// UTILITY END

//// Consumer of User (UTILITY) class
int main()
{
  User *user = new User();

  user->Action();

  delete user;
}
```

User class creating an object with the help of virtual constructor. The object to be created is based on user input. *Action()* is making duplicate of object being created and modifying it's attributes. The duplicate object being created with the help of *Clone()* virtual function which is also considered as *virtual copy constructor*. The concept behind *Clone()* method is building block of *prototype pattern*.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
AdvanceCPP
CPP
About Venki
Software Engineer
View all posts by Venki →

RTTI (Run-time type information) in C++

In C++, RTTI (Run-time type information) is available only for the classes which have at least one virtual function.

For example, dynamic_cast uses RTTI and following program fails with error "*cannot dynamic_cast `b' (of type `class B*') to type `class D*'*

(source type is not polymorphic)" because there is no virtual function in the base class B.

```
#include<iostream>

using namespace std;

class B { };
class D: public B {};

int main()
{
  B *b = new D;
  D *d = dynamic_cast<D*>(b);
  if(d != NULL)
    cout<<"works";
  else
    cout<<"cannot cast B* to D*";
  getchar();
  return 0;
}
```

Adding a virtual function to the base class B makes it working.

```
#include<iostream>

using namespace std;

class B { virtual void fun() {} };
class D: public B { };

int main()
{
  B *b = new D;
  D *d = dynamic_cast<D*>(b);
  if(d != NULL)
    cout<<"works";
  else
    cout<<"cannot cast B* to D*";
  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Can virtual functions be private in C++?

In C++, virtual functions can be private and can be overridden by the derived class. For example, the following program compiles and runs fine.

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

class Derived;

class Base {
private:
   virtual void fun() { cout << "Base Fun"; }
friend int main();
};

class Derived: public Base {
public:
   void fun() { cout << "Derived Fun"; }
```

```
};

int main()
{
  Base *ptr = new Derived;
  ptr->fun();
  return 0;
}
```

Output:

```
Derived fun()
```

There are few things to note in the above program.
1) ptr is a pointer of Base type and points to a Derived class object. When ptr->fun() is called, fun() of Derived is executed.

2) int main() is a friend of Base. If we remove this friendship, the program won't compile (See this). We get compiler error.

Note that this behavior is totally different in Java. In Java, private methods are final by default and cannot be overridden (See this)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-functions
cpp-virtual

Can virtual functions be inlined?

Virtual functions are used to achieve runtime polymorphism or say late binding or dynamic binding. Inline functions are used for efficiency. The whole idea behind the inline functions is that whenever inline function is called code of inline function gets inserted or substituted at the point of inline function call at compile time. Inline functions are very useful when small functions are frequently used and called in a program many times.

By default all the functions defined inside the class are implicitly or automatically considered as inline except virtual functions (Note that inline is a request to the compiler and its compilers choice to do inlining or not).

Whenever virtual function is called using base class reference or pointer it cannot be inlined (because call is resolved at runtime), but whenever called using the object (without reference or pointer) of that class, can be inlined because compiler knows the exact class of the object at compile time.

```
#include <iostream>
using namespace std;
class Base
{
public:
   virtual void who()
   {
      cout << "I am Base\n";
   }
};
class Derived: public Base
{
public:
   void who()
   {
      cout << "I am Derived\n";
   }
};

int main()
{
   // note here virtual function who() is called through
   // object of the class (it will be resolved at compile
   // time) so it can be inlined.
   Base b;
   b.who();
```

```
    // Here virtual function is called through pointer,
    // so it cannot be inlined
    Base *ptr = new Derived();
    ptr->who();

    return 0;
}
```

References:

http://www.parashift.com/c++-faq/inline-virtuals.html

Effective C++, by Scott Meyers

This article is contributed by **Meet Pravasi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-functions
cpp-virtual

Pure Virtual Functions and Abstract Classes in C++

Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. See the following example.

```
// An abstract class
class Test
{
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

   /* Other members */
};
```

A complete example:

A pure virtual function is implemented by classes which are derived from a Abstract class. Following is a simple example to demonstrate the same.

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

class Base
{
   int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

// This class ingerits from Base and implements fun()
class Derived: public Base
{
    int y;
public:
    void fun() { cout << "fun() called"; }
};

int main(void)
{
    Derived d;
```

```
    d.fun();
    return 0;
}
```

Output:

```
fun() called
```

Some Interesting Facts:

1) *A class is abstract if it has at least one pure virtual function.*

In the following example, Test is an abstract class because it has a pure virtual function show().

```
// pure virtual functions make a class abstract
#include<iostream>
using namespace std;

class Test
{
   int x;
public:
    virtual void show() = 0;
    int getX() { return x; }
};

int main(void)
{
    Test t;
    return 0;
}
```

Output:

```
Compiler Error: cannot declare variable 't' to be of abstract
 type 'Test' because the following virtual functions are pure
within 'Test': note:  virtual void Test::show()
```

2) *We can have pointers and references of abstract class type.*

For example the following program works fine.

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

class Base
{
public:
    virtual void show() = 0;
};

class Derived: public Base
{
public:
    void show() { cout << "In Derived \n"; }
};

int main(void)
{
    Base *bp = new Derived();
    bp->show();
    return 0;
}
```

Output:

```
In Derived
```

3) *If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.*

The following example demonstrates the same.

```
#include<iostream>
using namespace std;
class Base
{
public:
   virtual void show() = 0;
};

class Derived : public Base { };

int main(void)
{
  Derived d;
  return 0;
}
```

```
Compiler Error: cannot declare variable 'd' to be of abstract type
'Derived'  because the following virtual functions are pure within
'Derived': virtual void Base::show()
```

4) *An abstract class can have constructors.*
For example, the following program compiles and runs fine.

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

// An abstract class with constructor
class Base
{
protected:
   int x;
public:
  virtual void fun() = 0;
  Base(int i) { x = i; }
};

class Derived: public Base
{
   int y;
public:
   Derived(int i, int j):Base(i) { y = j; }
   void fun() { cout << "x = " << x << ", y = " << y; }
};

int main(void)
{
   Derived d(4, 5);
   d.fun();
   return 0;
}
```

Output:

```
x = 4, y = 5
```

Comparison with Java
In Java, a class can be made abstract by using abstract keyword. Similarly a function can be made pure virtual or abstract by using abstract keyword. See
Abstract Classes in Java for more details.

Interface vs Abstract Classes:
An interface does not have implementation of any of its methods, it can be considered as a collection of method declarations. In C++, an interface can be simulated by making all methods as pure virtual. In Java, there is a separate keyword for interface.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Category: C++

Pure virtual destructor in C++

Do not be anxious about anything, but in everything,
by prayer and petition, with thanksgiving, present
your requests to god.
Philippians 4:6 (Bible)

Can a destructor be pure virtual in C++?

Yes, it is possible to have pure virtual destructor. Pure virtual destructor are legal in standard C++ and one of the most important thing is that if class contains pure virtual destructor it is must to provide a function body for the pure virtual destructor. This seems strange that how a virtual function is pure if it requires a function body? But destructors are always called in the reverse order of the class derivation. That means derived class destructor will be invoked first & then base class destructor will be called. If definition for the pure virtual destructor is not provided then what function body will be called during object destruction? Therefore compiler & linker enforce existence of function body for pure virtual destructor.

Consider following program:

```
#include <iostream>
class Base
{
public:
   virtual ~Base()=0; // Pure virtual destructor
};

class Derived : public Base
{
public:
   ~Derived()
   {
      std::cout << "~Derived() is executed";
   }
};

int main()
{
   Base *b=new Derived();
   delete b;
   return 0;
}
```

The linker will produce following error in the above program.

```
test.cpp:(.text$_ZN7DerivedD1Ev[__ZN7DerivedD1Ev]+0x4c):
undefined reference to `Base::~Base()'
```

Now if the definition for the pure virtual destructor is provided then the program compiles & runs fine.

```
#include <iostream>
class Base
{
public:
   virtual ~Base()=0; // Pure virtual destructor
};
Base::~Base()
{
   std::cout << "Pure virtual destructor is called";
}

class Derived : public Base
```

```
{
public:
   ~Derived()
   {
      std::cout << "~Derived() is executed\n";
   }
};

int main()
{
   Base *b = new Derived();
   delete b;
   return 0;
}
```

Output:

```
~Derived() is executed
Pure virtual destructor is called
```

It is important to note that class becomes abstract class when it contains pure virtual destructor. For example try to compile the below program.

```
#include <iostream>
class Test
{
public:
   virtual ~Test()=0; // Test now becomes abstract class
};
Test::~Test() { }

int main()
{
   Test p;
   Test* t1 = new Test;
   return 0;
}
```

The above program fails in compilation & shows following error messages.
[Error] cannot declare variable ‘p’ to be of abstract type ‘Test’
[Note] because the following virtual functions are pure within ‘Test’:
[Note] virtual Test::~Test()
[Error] cannot allocate an object of abstract type ‘Test’
[Note] since type ‘Test’ has pure virtual functions

Sources:
http://www.bogotobogo.com/cplusplus/virtualfunctions.php
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf

This article is contributed by **Meet Pravasi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-constructor
cpp-virtual

Operator Overloading in C++

In C++, we can make operators to work for user defined classes. For example, we can overload an operator ‘+’ in a class like String so that we can concatenate two strings by just using +.
Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

A simple and complete example

```
#include<iostream>
```

```
using namespace std;

class Complex {
private:
   int real, imag;
public:
   Complex(int r = 0, int i =0)  {real = r;   imag = i;}

   // This is automatically called when '+' is used with
   // between two Complex objects
   Complex operator + (Complex const &obj) {
       Complex res;
       res.real = real + obj.real;
       res.imag = imag + obj.imag;
       return res;
   }
   void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2; // An example call to "operator+"
   c3.print();
}
```

Output:

```
12 + i9
```

What is the difference between operator functions and normal functions?

Operator functions are same as normal functions. The only differences are, name of an operator function is always operator keyword followed by symbol of operator and operator functions are called when the corresponding operator is used.

Following is an example of global operator function.

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

class Complex {
private:
   int real, imag;
public:
   Complex(int r = 0, int i =0)  {real = r;   imag = i;}
   void print() { cout << real << " + i" << imag << endl; }

// The global operator function is made friend of this class so
// that it can access private members
friend Complex operator + (Complex const &, Complex const &);
};


Complex operator + (Complex const &c1, Complex const &c2)
{
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}


int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2; // An example call to "operator+"
   c3.print();
   return 0;
}
```

Can we overload all operators?

Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

```
   . (dot)
   ::
```

```
?:
sizeof
```

Why can't . (dot), ::, ?: and sizeof be overloaded?
See this for answers from Stroustrup himself.

Important points about operator overloading
1) For operator overloading to work, at leas one of the operands must be a user defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of right side to the left side and works fine most of the cases (this behavior is same as copy constructor). See this for more details.

3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

```
#include <iostream>
using namespace std;
class Fraction
{
    int num, den;
public:
    Fraction(int n,  int d) { num = n; den = d; }

    // conversion operator: return float value of fraction
    operator float() const {
        return float(num) / float(den);
    }
};

int main() {
    Fraction f(2, 5);
    float val = f;
    cout << val;
    return 0;
}
```

Output:

```
0.4
```

Overloaded conversion operators must be a member method. Other operators can either be member method or global method.

4) Any constructor that can be called with a single argument works as a conversion constructor, means it can also be used for implicit conversion to the class being constructed.

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

class Point
{
private:
    int x, y;
public:
    Point(int i = 0, int j = 0) {
        x = i;  y = j;
    }
    void print() {
        cout << endl << " x = " << x << ", y = " << y;
    }
};

int main() {
    Point t(20, 20);
    t.print();
    t = 30;   // Member x of t becomes 30
    t.print();
    return 0;
}
```

Output:

```
x = 20, y = 20
x = 30, y = 0
```

We will soon be discussing overloading of some important operators like new, delete, comma, function call, arrow, etc.

References:

http://en.wikipedia.org/wiki/Operator_overloading

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Copy constructor vs assignment operator in C++

Difficulty Level: Rookie

Consider the following C++ program.

```
#include<iostream>
#include<stdio.h>

using namespace std;

class Test
{
public:
  Test() {}
  Test(const Test &t)
  {
    cout<<"Copy constructor called "<<endl;
  }
  Test& operator = (const Test &t)
  {
    cout<<"Assignment operator called "<<endl;
  }
};

int main()
{
  Test t1, t2;
  t2 = t1;
  Test t3 = t1;
  getchar();
  return 0;
}
```

Output:

Assignment operator called

Copy constructor called

Copy constructor is called when a new object is created from an existing object, as a copy of the existing object (see this G-Fact). And assignment operator is called when an already initialized object is assigned a new value from another existing object.

```
t2 = t1;  // calls assignment operator, same as "t2.operator=(t1);"
Test t3 = t1;  // calls copy constructor, same as "Test t3(t1);"
```

References:

http://en.wikipedia.org/wiki/Copy_constructor

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

When should we write our own assignment operator in C++?

The answer is same as Copy Constructor. If a class doesn't contain pointers, then there is no need to write assignment operator and copy constructor. The compiler creates a default copy constructor and assignment operators for every class. The compiler created copy constructor and assignment operator may not be sufficient when we have pointers or any run time allocation of resource like file handle, a network connection..etc. For example, consider the following program.

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

// A class without user defined assignment operator
class Test
{
   int *ptr;
public:
   Test (int i = 0)      { ptr = new int(i); }
   void setValue (int i) { *ptr = i; }
   void print()          { cout << *ptr << endl; }
};

int main()
{
   Test t1(5);
   Test t2;
   t2 = t1;
   t1.setValue(10);
   t2.print();
   return 0;
}
```

Output of above program is "10". If we take a look at main(), we modified 't1' using setValue() function, but the changes are also reflected in object 't2'. This type of unexpected changes cause problems.
Since there is no user defined assignment operator in the above program, compiler creates a default assignment operator, which copies 'ptr' of right hand side to left hand side. So both 'ptr's start pointing to the same location.

We can handle the above problem in two ways.

1) Do not allow assignment of one object to other object. We can create our own dummy assignment operator and make it private.

2) Write your own assignment operator that does deep copy.

Same is true for Copy Constructor.

Following is an example of overloading assignment operator for the above class.

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

class Test
{
   int *ptr;
public:
   Test (int i = 0)      { ptr = new int(i); }
   void setValue (int i) { *ptr = i; }
   void print()          { cout << *ptr << endl; }
   Test & operator = (const Test &t);
};

Test & Test::operator = (const Test &t)
{
  // Check for self assignment
  if(this != &t)
```

```
    *ptr = *(t.ptr);

  return *this;
}

int main()
{
   Test t1(5);
   Test t2;
   t2 = t1;
   t1.setValue(10);
   t2.print();
   return 0;
}
```

Output

```
5
```

We should also add a copy constructor to the above class, so that the statements like “Test t3 = t4;” also don’t cause any problem.

Note the if condition in assignment operator. While overloading assignment operator, we must check for self assignment. Otherwise assigning an object to itself may lead to unexpected results (See this). Self assignment check is not necessary for the above ‘Test’ class, because ‘ptr’ always points to one integer and we may reuse the same memory. But in general, it is a recommended practice to do self-assignment check.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

What are the operators that cannot be overloaded in C++?

In C++, following operators can not be overloaded:

. (Member Access or Dot operator)
?: (Ternary or Conditional Operator)
:: (Scope Resolution Operator)
.* (Pointer-to-member Operator)
sizeof (Object size Operator)
typeid (Object type Operator)

References:
http://en.wikibooks.org/wiki/C++_Programming/Operators/Operator_Overloading

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
GFacts

Use of explicit keyword in C++

Predict the output of following C++ program.

```
#include <iostream>

using namespace std;

class Complex
{
private:
   double real;
   double imag;
```

```
public:
   // Default constructor
   Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

   // A method to compare two Complex numbers
   bool operator == (Complex rhs) {
     return (real == rhs.real && imag == rhs.imag)? true : false;
   }
};

int main()
{
   // a Complex object
   Complex com1(3.0, 0.0);

   if (com1 == 3.0)
     cout << "Same";
   else
     cout << "Not Same";
    return 0;
}
```

Output: The program compiles fine and produces following output.

```
Same
```

As discussed in this GFact, in C++, if a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor because such a constructor allows conversion of the single argument to the class being constructed.
We can avoid such implicit conversions as these may lead to unexpected results. We can make the constructor explicit with the help of explicit keyword. For example, if we try the following program that uses explicit keyword with constructor, we get compilation error.

```
#include <iostream>

using namespace std;

class Complex
{
private:
   double real;
   double imag;

public:
   // Default constructor
   explicit Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

   // A method to compare two Complex numbers
   bool operator== (Complex rhs) {
     return (real == rhs.real && imag == rhs.imag)? true : false;
   }
};

int main()
{
   // a Complex object
   Complex com1(3.0, 0.0);

   if (com1 == 3.0)
     cout << "Same";
   else
     cout << "Not Same";
    return 0;
}
```

Output: Compiler Error

```
no match for 'operator==' in 'com1 == 3.0e+0'
```

We can still typecast the double values to Complex, but now we have to explicitly typecast it. For example, the following program works fine.

```
#include <iostream>

using namespace std;

class Complex
{
private:
   double real;
   double imag;

public:
   // Default constructor
   explicit Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

   // A method to compare two Complex numbers
   bool operator== (Complex rhs) {
      return (real == rhs.real && imag == rhs.imag)? true : false;
   }
};

int main()
{
   // a Complex object
   Complex com1(3.0, 0.0);

   if (com1 == (Complex)3.0)
      cout << "Same";
   else
      cout << "Not Same";
    return 0;
}
```

Output: The program compiles fine and produces following output.

```
Same
```

Also see Advanced C++ | Conversion Operators

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Advanced C++ | Conversion Operators

In C++, the programmer abstracts real world objects using classes as concrete types. Sometimes it is required to convert one concrete type to another concrete type or primitive type implicitly. Conversion operators play smart role in such situations.

For example consider the following class

```
#include <iostream>
#include <cmath>

using namespace std;

class Complex
{
private:
   double real;
   double imag;

public:
   // Default constructor
   Complex(double r = 0.0, double i = 0.0) : real(r), imag(i)
   {}

   // magnitude : usual function style
```

```
    double mag()
    {
        return getMag();
    }

    // magnitude : conversion operator
    operator double ()
    {
        return getMag();
    }

private:
    // class helper to get magnitude
    double getMag()
    {
        return sqrt(real * real + imag * imag);
    }
};

int main()
{
    // a Complex object
    Complex com(3.0, 4.0);

    // print magnitude
    cout << com.mag() << endl;
    // same can be done like this
    cout << com << endl;
}
```

We are printing the magnitude of Complex object in two different ways.

Note that usage of such smart (over smart ?) techniques are discouraged. The compiler will have more control in calling an appropriate function based on type, rather than what the programmer expects. It will be good practice to use other techniques like class/object specific member function (or making use of C++ Variant class) to perform such conversions. At some places, for example in making compatible calls with existing C library, these are unavoidable.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
About Venki
Software Engineer
View all posts by Venki →

Is assignment operator inherited?

In C++, like other functions, assignement operator function is inherited in derived class.

For example, in the following program, base class assignment operator function can be accessed using the derived class object.

```
#include<iostream>

using namespace std;

class A {
 public:
   A & operator= (A &a) {
    cout<<" base class assignment operator called ";
    return *this;
   }
};

class B: public A { };

int main()
{
  B a, b;
  a.A::operator=(b); //calling base class assignment operator function
```

```
                // using derived class
  getchar();
  return 0;
}
```

Output: *base class assignment operator called*

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Default Assignment Operator and References

We have discussed assignment operator overloading for dynamically allocated resources here . This is a an extension of the previous post. In the previous post, we discussed that when we don't write our own assignment operator, compiler created assignment operator does shallow copy and that cause problems. What happens when we have references in our class and there is no user defined assignment operator. For example, predict the output of following program.

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

class Test
{
   int x;
   int &ref;
public:
   Test (int i):x(i), ref(x) {}
   void print() { cout << ref; }
   void setX(int i) { x = i; }
};

int main()
{
   Test t1(10);
   Test t2(20);
   t2 = t1;
   t1.setX(40);
   t2.print();
   return 0;
}
```

Output:

```
Compiler Error: non-static reference member 'int& Test::ref',
          can't use default assignment operator
```

Compiler doesn't creates default assignment operator in following cases

1. Class has a nonstatic data member of a const type or a reference type
2. Class has a nonstatic data member of a type which has an inaccessible copy assignment operator
3. Class is derived from a base class with an inaccessible copy assignment operator

When any of the above conditions is true, user must define assignment operator. For example, if we add an assignment operator to the above code, the code works fine without any error.

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

class Test
{
   int x;
   int &ref;
public:
   Test (int i):x(i), ref(x) {}
   void print() { cout << ref; }
```

```
    void setX(int i) { x = i; }
    Test &operator = (const Test &t) { x = t.x; return *this; }
};

int main()
{
    Test t1(10);
    Test t2(20);
    t2 = t1;
    t1.setX(40);
    t2.print();
    return 0;
}
```

Output:

```
10
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Pre-increment (or pre-decrement) in C++

In C++, pre-increment (or pre-decrement) can be used as l-value, but post-increment (or post-decrement) can not be used as l-value.

For example, following program prints *a = 20* (++a is used as l-value)

```
#include<stdio.h>

int main()
{
  int a = 10;
  ++a = 20; // works
  printf("a = %d", a);
  getchar();
  return 0;
}
```

And following program fails in compilation with error *"non-lvalue in assignment"* (a++ is used as l-value)

```
#include<stdio.h>

int main()
{
  int a = 10;
  a++ = 20; // error
  printf("a = %d", a);
  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Smart Pointers in C++

Consider the following simple C++ code with normal pointers.

```
  MyClass *ptr = new MyClass();
```

```
  ptr->doSomething();
  //  We must do delete(ptr) to avoid memory leak
```

Using smart pointers, we can make pointers to work in way that we don't need to explicitly call delete. Smart pointer is a wrapper class over a pointer with operator like * and -> overloaded. The objects of smart pointer class look like pointer, but can do many things that a normal pointer can't like automatic destruction (yes, we don't have to explicitly use delete), reference counting and more.
The idea is to make a class with a pointer, destructor and overloaded operators like * and ->. Since destructor is automatically called when an object goes out of scope, the dynamically alloicated memory would automatically deleted (or reference count can be decremented). Consider the following simple smartPtr class.

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

class SmartPtr
{
  int *ptr;  // Actual pointer
public:
  // Constructor: Refer http://www.geeksforgeeks.org/g-fact-93/
  // for use of explicit keyword
  explicit SmartPtr(int *p = NULL) { ptr = p; }

  // Destructor
  ~SmartPtr() { delete(ptr); }

  // Overloading dereferencing operator
  int &operator *() {  return *ptr; }
};

int main()
{
   SmartPtr ptr(new int());
   *ptr = 20;
   cout << *ptr;

   // We don't need to call delete ptr: when the object
   // ptr goes out of scope, destructor for it is automatically
   // called and destructor does delete ptr.

   return 0;
}
```

Output:

```
20
```

Can we write one smart pointer class that works for all types?
Yes, we can use templates to write a generic smart pointer class. Following C++ code demonstrates the same.

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

// A generic smart pointer class
template <class T>
class SmartPtr
{
  T *ptr;  // Actual pointer
public:
  // Constructor
  explicit SmartPtr(T *p = NULL) { ptr = p; }

  // Destructor
  ~SmartPtr() { delete(ptr); }

  // Overloading dereferncing operator
  T & operator * () {  return *ptr; }

  // Overloding arrow operator so that members of T can be accessed
  // like a pointer (useful if T represents a class or struct or
  // union type)
```

```
    T * operator -> () { return ptr; }
};

int main()
{
    SmartPtr<int> ptr(new int());
    *ptr = 20;
    cout << *ptr;
    return 0;
}
```

Output:

```
20
```

Smart pointers are also useful in management of resources, such as file handles or network sockets.

C++ libraries provide implementations of smart pointers in the form of auto_ptr, unique_ptr, shared_ptr and weak_ptr

References:

http://en.wikipedia.org/wiki/Smart_pointer

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-pointer

Overloading stream insertion (>) operators in C++

In C++, stream insertion operator “<<” is used for output and extraction operator “>>” is used for input.

We must know following things before we start overloading these operators.

1) cout is an object of ostream class and cin is an object istream class

2) These operators must be overloaded as a global function. And if we want to allow them to access private data members of class, we must make them friend.

Why these operators must be overloaded as global?

In operator overloading, if an operator is overloaded as member, then it must be a member of the object on left side of the operator. For example, consider the statement “ob1 + ob2″ (let ob1 and ob2 be objects of two different classes). To make this statement compile, we must overload ‘+’ in class of ‘ob1′ or make ‘+’ a global function.

The operators ‘> operators.

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

class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)
    {  real = r;   imag = i; }
    friend ostream & operator << (ostream &out, const Complex &c);
    friend istream & operator >> (istream &in,  Complex &c);
};

ostream & operator << (ostream &out, const Complex &c)
{
    out << c.real;
    out << "+i" << c.imag << endl;
    return out;
}

istream & operator >> (istream &in,  Complex &c)
{
    cout << "Enter Real Part ";
```

```
    in >> c.real;
    cout << "Enter Imagenory Part ";
    in >> c.imag;
    return in;
}

int main()
{
  Complex c1;
  cin >> c1;
  cout << "The complex object is ";
  cout << c1;
  return 0;
}
```

Output:

```
Enter Real Part 10
Enter Imagenory Part 20
The complex object is 10+i20
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Overloading Subscript or array index operator [] in C++

We have introduced operator overloading. In this post overloading of index operator [] is discussed.

Following are some useful facts about overloading of [].

1) Overloading of [] may be useful when we want to check for index out of bound.

2) We must return by reference in function because an expression like “arr[i]” can be used an lvalue.

Following is C++ program to demonstrate overloading of array index operator [].

```
// Overloading operators for Array class
#include<iostream>
#include<cstdlib>

using namespace std;

// A class to represent an integer array
class Array
{
private:
    int *ptr;
    int size;
public:
    Array(int *, int);

    // Overloading [] operator to access elements in array style
    int &operator[] (int);

    // Utility function to print contents
    void print() const;
};


// Implementation of [] operator.  This function must return a
// refernce as array element can be put on left side
int &Array::operator[](int index)
```

```
{
    if (index >= size)
    {
        cout << "Array index out of bound, exiting";
        exit(0);
    }
    return ptr[index];
}

// constructor for array class
Array::Array(int *p = NULL, int s = 0)
{
    size = s;
    ptr = NULL;
    if (s != 0)
    {
        ptr = new int[s];
        for (int i = 0; i < s; i++)
            ptr[i] = p[i];
    }
}

void Array::print() const
{
    for(int i = 0; i < size; i++)
        cout<<ptr[i]<<" ";
    cout<<endl;
}

// Driver program to test above methods
int main()
{
    int a[] = {1, 2, 4, 5};
    Array arr1(a, 4);
    arr1[2] = 6;
    arr1.print();
    arr1[8] = 6;
    return 0;
}
```

Output:

```
1 2 6 5
Array index out of bound, exiting
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Exception Handling in C++

One of the advantages of C++ over C is Exception Handling. C++ provides following specialized keywords for this purpose.

try: represents a block of code that can throw an exception.

catch: represents a block of code that is executed when a particular exception is thrown.

throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

Why Exception Handling?

Following are main advantages of exception handling over traditional error handling.

1) *Separation of Error Handling code from Normal Code:* In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

2) *Functions/Methods can handle any exceptions they choose:* A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.
In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

3) *Grouping of Error Types:* In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Exception Handling in C++

1) Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks.

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

int main()
{
  int x = -1;

  // Some code
  cout << "Before try \n";
  try {
    cout << "Inside try \n";
    if (x < 0)
    {
      throw x;
      cout << "After throw (Never executed) \n";
    }
  }
  catch (int x ) {
    cout << "Exception Caught \n";
  }

  cout << "After catch (Will be executed) \n";
  return 0;
}
```

Output:

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

2) There is a special catch block called ‘catch all’ catch(…) that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so catch(…) block will be executed.

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

int main()
{
   try  {
     throw 10;
   }
   catch (char *excp)  {
      cout << "Caught " << excp;
   }
   catch (...)  {
      cout << "Default Exception\n";
   }
   return 0;
}
```

Output:

```
Default Exception
```

3) Implicit type conversion doesn't happen for primitive types. For example, in the following program 'a' is not implicitly converted to int

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

int main()
{
   try {
     throw 'a';
   }
   catch (int x) {
      cout << "Caught " << x;
   }
   catch (...) {
      cout << "Default Exception\n";
   }
   return 0;
}
```

Output:

```
Default Exception
```

4) If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch a char.

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

int main()
{
   try {
     throw 'a';
   }
   catch (int x) {
      cout << "Caught ";
   }
   return 0;
}
```

Output:

```
terminate called after throwing an instance of 'char'

This application has requested the Runtime to terminate it in an
unusual way. Please contact the application's support team for
more information.
```

We can change this abnormal termination behavior by writing our own unexpected function.

5) A derived class exception should be caught before a base class exception. See this for more details.

6) Like Java, C++ library has a standard exception class which is base class for all standard exceptions. All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type

7) Unlike Java, in C++, all exceptions are unchecked. Compiler doesn't check whether an exception is caught or not (See this for details). For example, in C++, it is not necessary to specify all uncaught exceptions in a function declaration. Although it's a recommended practice to do so. For example, the following program compiles fine, but ideally signature of fun() should list unchecked exceptions.

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

// This function signature is fine by the compiler, but not recommended.
// Ideally, the function should specify all uncaught exceptions and function
// signature should be "void fun(int *ptr, int x) throw (int *, int)"
```

```
void fun(int *ptr, int x)
{
   if (ptr == NULL)
      throw ptr;
   if (x == 0)
      throw x;
   /* Some functionality */
}

int main()
{
   try {
     fun(NULL, 0);
   }
   catch(...) {
      cout << "Caught exception from fun()";
   }
   return 0;
}
```

Output:

```
Caught exception from fun()
```

8) In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using “throw; ”

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

int main()
{
   try {
      try  {
         throw 20;
      }
      catch (int n) {
          cout << "Handle Partially ";
          throw;   //Re-throwing an exception
      }
   }
   catch (int n) {
      cout << "Handle remaining ";
   }
   return 0;
}
```

Output:

```
Handle Partially Handle remaining
```

A function can also re-throw a function using same “throw; “. A function can handle a part and can ask the caller to handle remaining.

9) When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

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

class Test {
public:
   Test() { cout << "Constructor of Test " << endl; }
  ~Test() { cout << "Destructor of Test "  << endl; }
};

int main() {
  try {
    Test t1;
    throw 10;
  } catch(int i) {
    cout << "Caught " << i << endl;
```

```
  }
}
```

Output:

```
Constructor of Test
Destructor of Test
Caught 10
```

10) You may like to try Quiz on Exception Handling in C++.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-exception

Stack Unwinding in C++

The process of removing function entries from function call stack at run time is called Stack Unwinding. Stack Unwinding is generally related to Exception Handling. In C++, when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack. So exception handling involves Stack Unwinding if exception is not handled in same function (where it is thrown).

For example, output of the following program is:

```
f3() Start
f2() Start
f1() Start
Caught Exception: 100
f3() End
```

```
#include <iostream>

using namespace std;

// A sample function f1() that throws an int exception
void f1() throw (int) {
  cout<<"\n f1() Start ";
  throw 100;
  cout<<"\n f1() End ";
}

// Another sample function f2() that calls f1()
void f2() throw (int) {
  cout<<"\n f2() Start ";
  f1();
  cout<<"\n f2() End ";
}

// Another sample function f3() that calls f2() and handles exception thrown by f1()
void f3() {
  cout<<"\n f3() Start ";
  try {
    f2();
  }
  catch(int i) {
   cout<<"\n Caught Exception: "<<i;
  }
  cout<<"\n f3() End";
}

// A driver function to demonstrate Stack Unwinding  process
int main() {
  f3();

  getchar();
```

```
  return 0;
}
```

In the above program, when f1() throws exception, its entry is removed from the function call stack (because it f1() doesn't contain exception handler for the thrown exception), then next entry in call stack is looked for exception handler. The next entry is f2(). Since f2() also doesn't have handler, its entry is also removed from function call stack. The next entry in function call stack is f3(). Since f3() contains exception handler, the catch block inside f3() is executed, and finally the code after catch block is executed. Note that the following lines inside f1() and f2() are not executed at all.

```
//inside f1()
  cout<<"\n f1() End ";

//inside f2()
  cout<<"\n f2() End ";
```

On a side note, if there were some local class objects inside f1() and f2(), destructors for those local objects would have been called in Stack Unwinding process.

Stack Unwinding also happens in Java when exception is not handled in same function.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Catching base and derived classes as exceptions

Exception Handling – catching base and derived classes as exceptions:

If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.

If we put base class first then the derived class catch block will never be reached. For example, following C++ code prints *"Caught Base Exception"*

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

class Base {};
class Derived: public Base {};
int main()
{
  Derived d;
  // some other stuff
  try {
     // Some monitored code
     throw d;
  }
  catch(Base b) {
      cout<<"Caught Base Exception";
  }
  catch(Derived d) {  //This catch block is NEVER executed
      cout<<"Caught Derived Exception";
  }
  getchar();
  return 0;
}
```

In the above C++ code, if we change the order of catch statements then both catch statements become reachable. Following is the modifed program and it prints *"Caught Derived Exception"*

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

class Base {};
class Derived: public Base {};
int main()
```

```
{
  Derived d;
  // some other stuff
  try {
     // Some monitored code
     throw d;
  }
  catch(Derived d) {
      cout<<"Caught Derived Exception";
  }
  catch(Base b) {
      cout<<"Caught Base Exception";
  }
  getchar();
  return 0;
}
```

In Java, catching a base class exception before derived is not allowed by the compiler itself. In C++, compiler might give warning about it, but compiles the code.
For example, following Java code fails in compilation with error message *"exception Derived has already been caught"*

```
//filename Main.java
class Base extends Exception {}
class Derived extends Base  {}
public class Main {
  public static void main(String args[]) {
    try {
      throw new Derived();
    }
    catch(Base b) {}
    catch(Derived d) {}
  }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
CPP

Catch block and type conversion in C++

Predict the output of following C++ program.

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

int main()
{
   try
   {
      throw 'x';
   }
   catch(int x)
   {
      cout << " Caught int " << x;
   }
   catch(...)
   {
      cout << "Defaule catch block";
   }
}
```

```
Defaule catch block
```

In the above program, a character 'x' is thrown and there is a catch block to catch an int. One might think that the int catch block could be

matched by considering ASCII value of ‘x’. But such conversions are not performed for catch blocks. Consider the following program as another example where conversion constructor is not called for thrown object.

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

class MyExcept1 {};

class MyExcept2
{
public:

   // Conversion constructor
   MyExcept2 (const MyExcept1 &e )
   {
      cout << "Conversion constructor called";
   }
};

int main()
{
   try
   {
      MyExcept1 myexp1;
      throw myexp1;
   }
   catch(MyExcept2 e2)
   {
      cout << "Caught MyExcept2 " << endl;
   }
   catch(...)
   {
      cout << " Defaule catch block " << endl;
   }
   return 0;
}
```

```
Defaule catch block
```

As a side note, the derived type objects are converted to base typr when a derived object is thrown and there is a catch block to catch base type. See this GFact for more details.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Exception handling and object destruction | Set 1

Predict the output of following C++ program.

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

class Test {
public:
  Test() { cout << "Constructing an object of Test " << endl; }
  ~Test() { cout << "Destructing an object of Test "  << endl; }
};

int main() {
  try {
    Test t1;
    throw 10;
  } catch(int i) {
    cout << "Caught " << i << endl;
```

```
  }
}
```

Output:

```
Constructing an object of Test
Destructing an object of Test
Caught 10
```

When an exception is thrown, destructors of the objects (whose scope ends with the try block) is automatically called before the catch block gets exectuted. That is why the above program prints “Destructing an object of Test” before “Caught 10”.

What happens when an exception is thrown from a constructor? Consider the following program.

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

class Test1 {
public:
  Test1() { cout << "Constructing an Object of Test1" << endl; }
  ~Test1() { cout << "Destructing an Object of Test1" << endl; }
};

class Test2 {
public:
  // Following constructor throws an integer exception
  Test2() { cout << "Constructing an Object of Test2" << endl;
          throw 20; }
  ~Test2() { cout << "Destructing an Object of Test2" << endl; }
};

int main() {
  try {
    Test1 t1;  // Constructed and destructed
    Test2 t2;  // Partially constructed
    Test1 t3;  // t3 is not constructed as this statement never gets executed
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

Output:

```
Constructing an Object of Test1
Constructing an Object of Test2
Destructing an Object of Test1
Caught 20
```

Destructors are only called for the completely constructed objects. When constructor of an object throws an exception, destructor for that object is not called.

As an excecise, predict the output of following program.

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

class Test {
  static int count;
  int id;
public:
  Test() {
    count++;
    id = count;
    cout << "Constructing object number " << id << endl;
    if(id == 4)
      throw 4;
  }
  ~Test() { cout << "Destructing object number " << id << endl; }
};
```

```
int Test::count = 0;

int main() {
  try {
    Test array[5];
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

We will be covering more of this topic in a separate post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Templates in C++

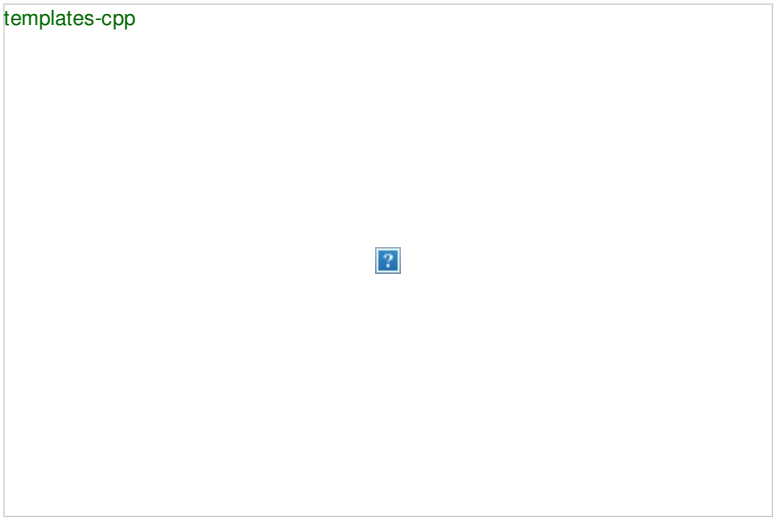
Template is simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write same code for different data types. For example a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

C++ adds two new keywords to support templates: *'template'* and *'typename'*. The second keyword can always be replaced by keyword 'class'.

How templates work?

Templates are expended at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

templates-cpp



Function Templates We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray()

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

// One function works for all data types.  This would work
// even for user defined types if operator '>' is overloaded
template <typename T>
T myMax(T x, T y)
{
  return (x > y)? x: y;
}

int main()
{
  cout << myMax<int>(3, 7) << endl;  // Call myMax for int
  cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double
```

```
  cout << myMax<char>('g', 'e') << endl;   // call myMax for char

  return 0;
}
```

Output:

```
7
7
g
```

Class Templates Like function templates, class templates are useful when a class defines something that is independent of data type. Can be useful for classes like LinkedList, BinaryTre, Stack, Queue, Array, etc.

Following is a simple example of template Array class.

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

template <typename T>
class Array {
private:
    T *ptr;
    int size;
public:
    Array(T arr[], int s);
    void print();
};

template <typename T>
Array<T>::Array(T arr[], int s) {
    ptr = new T[s];
    size = s;
    for(int i = 0; i < size; i++)
        ptr[i] = arr[i];
}

template <typename T>
void Array<T>::print() {
    for (int i = 0; i < size; i++)
        cout<<" "<<*(ptr + i);
    cout<<endl;
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    Array<int> a(arr, 5);
    a.print();
    return 0;
}
```

Output:

```
1 2 3 4 5
```

Can there be more than one arguments to templates?

Yes, like normal parameters, we can pass more than one data types as arguments to templates. The following example demonstrates the same.

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

template<class T, class U>
class A  {
    T x;
    U y;
Public:
    A() {    cout<<"Constructor Called"<<endl;   }
};
```

```
int main()  {
   A<char, char> a;
   A<int, double> b;
   return 0;
}
```

Output:

```
Constructor Called
Constructor Called
```

Can we specify default value for template arguments?

Yes, like normal parameters, we can specify default arguments to templates. The following example demonstrates the same.

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

template<class T, class U = char>
class A  {
public:
    T x;
    U y;
    A() {   cout<<"Constructor Called"<<endl;  }
};

int main()  {
   A<char> a;  // This will call A<char, char>
   return 0;
}
```

Output:

```
Constructor Called
```

What is the difference between function overloading and templates?

Both function overloading and templates are examples of polymorphism feature of OOP. Function overloading is used when multiple functions do similar operations, templates are used when multiple functions do identical operations.

What happens when there is static member in a template class/function?

Each instance of a template contains its own static variable. See Templates and Static variables for more details.

What is template specialization?

Template specialization allows us to have different code for a particular data type. See Template Specialization for more details.

Can we pass nontype parameters to templates?

We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of template. The important thing to note about non-type parameters is, they must be const. Compiler must know the value of non-type parameters at compile time. Because compiler needs to create functions/classes for a specified non-type value at compile time. In below program, if we replace 10000 or 25 with a variable, we get compiler error. Please see this.

Below is a C++ program.

```
// A C++ program to demonstrate working of non-type
// parameters to templates in C++.
#include <iostream>
using namespace std;

template <class T, int max>
int arrMin(T arr[], int n)
{
   int m = max;
   for (int i = 0; i < n; i++)
      if (arr[i] < m)
         m = arr[i];

   return m;
}

int main()
```

```
{
  int arr1[]  = {10, 20, 15, 12};
  int n1 = sizeof(arr1)/sizeof(arr1[0]);

  char arr2[] = {1, 2, 3};
  int n2 = sizeof(arr2)/sizeof(arr2[0]);

  // Second template parameter to arrMin must be a constant
  cout << arrMin<int, 10000>(arr1, n1) << endl;
  cout << arrMin<char, 256>(arr2, n2);
  return 0;
}
```

Output:

```
10
1
```

What is template metaprogramming?
See Template Metaprogramming

You may also like to take a quiz on templates.

Java also support these features. Java calls it generics .

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Category: C++

Templates and Static variables in C++

Function templates and static variables:

Each instantiation of function template has its own copy of local static variables. For example, in the following program there are two instances: *void fun(int)* and *void fun(double)*. So two copies of static variable *i* exist.

```
#include <iostream>

using namespace std;

template <typename T>
void fun(const T& x)
{
  static int i = 10;
  cout << ++i;
  return;
}

int main()
{
  fun<int>(1);  // prints 11
  cout << endl;
  fun<int>(2);  // prints 12
  cout << endl;
  fun<double>(1.1); // prints 11
  cout << endl;
  getchar();
  return 0;
}
```

Output of the above program is:

```
11
12
11
```

Class templates and static variables:

The rule for class templates is same as function templates

Each instantiation of class template has its own copy of member static variables. For example, in the following program there are two instances *Test* and *Test*. So two copies of static variable *count* exist.

```
#include <iostream>

using namespace std;

template <class T> class Test
{
private:
  T val;
public:
  static int count;
  Test()
  {
    count++;
  }
  // some other stuff in class
};

template<class T>
int Test<T>::count = 0;

int main()
{
  Test<int> a;  // value of count for Test<int> is 1 now
  Test<int> b;  // value of count for Test<int> is 2 now
  Test<double> c;  // value of count for Test<double> is 1 now
  cout << Test<int>::count   << endl;  // prints 2
  cout << Test<double>::count << endl; //prints 1

  getchar();
  return 0;
}
```

Output of the above program is:

```
2
1
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Templates and Default Arguments

Default parameters for templates in C++:

Like function default arguments, templates can also have default arguments. For example, in the following program, the second parameter U has the default value as char.

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

template<class T, class U = char> class A
{
public:
   T x;
   U y;
```

```
};

int main()
{
   A<char> a;
   A<int, int> b;
   cout<<sizeof(a)<<endl;
   cout<<sizeof(b)<<endl;
   return 0;
}
```

Output: (char takes 1 byte and int takes 4 bytes)
2
8

Also, similar to default function arguments, if one template parameter has a default argument, then all template parameters following it must also have default arguments. For example, the compiler will not allow the following program:

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

template<class T = char, class U, class V = int> class A  // Error
{
  // members of A
};

int main()
{

}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Template Metaprogramming in C++

Predict the output of following C++ program.

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

template<int n> struct funStruct
{
   enum { val = 2*funStruct<n-1>::val };
};

template<> struct funStruct<0>
{
   enum { val = 1 };
};

int main()
{
   cout << funStruct<8>::val << endl;
   return 0;
}
```

Output:

```
256
```

The program calculates “2 raise to the power 8 (or 2^8)”. In fact, the structure *funStruct* can be used to calculate 2^n for any known n (or constant n). The special thing about above program is: **calculation is done at compile time**. So, it is compiler that calculates 2^8. To

understand how compiler does this, let us consider the following facts about templates and enums:

1) We can pass nontype parameters (parameters that are not data types) to class/function templates.
2) Like other const expressions, values of enumaration constants are evaluated at compile time.
3) When compiler sees a new argument to a template, compiler creates a new instance of the template.

Let us take a closer look at the original program. When compiler sees *funStruct::val*, it tries to create an instance of *funStruct* with parameter as 8, it turns out that *funStruct* must also be created as enumaration constant *val* must be evaluated at compile time. For *funStruct*, compiler need *funStruct* and so on. Finally, compiler uses *funStruct::val* and compile time recursion terminates. So, using templates, we can write programs that do computation at compile time, such programs are called template metaprograms. Template metaprogramming is in fact Turing-complete, meaning that any computation expressible by a computer program can be computed, in some form, by a template metaprogram. Template Metaprogramming is generally not used in practical programs, it is an interesting conecpt though.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
C++

Template Specialization in C++

Template in C++is a feature. We write code once and use it for any data type including user defined data types. For example, sort() can be written and used to sort any data type items. A class stack can be created that can be used as a stack of any data type.
What if we want a different code for a particular data type? Consider a big project that needs a function sort() for arrays of many different data types. Let Quick Sort be used for all datatypes except char. In case of char, total possible values are 256 and counting sort may be a better option. Is it possible to use different code only when sort() is called for char data type?
It is possible in C++ to get a special behavior for a particular data type. This is called template specialization.

```
// A generic sort function
template <class T>
void sort(T arr[], int size)
{
   // code to implement Quick Sort
}

// Template Specialization: A function specialized for char data type
template <>
void sort<char>(char arr[], int size)
{
   // code to implement counting sort
}
```

Another example could be a class *Set* that represents a set of elements and supports operations like union, intersection, etc. When the type of elements is char, we may want to use a simple boolean array of size 256 to make a set. For other data types, we have to use some other complex technique.

An Example Program for function template specialization
For example, consider the following simple code where we have general template fun() for all data types except int. For int, there is a specialized version of fun().

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

template <class T>
void fun(T a)
{
  cout << "The main template fun(): " << a << endl;
}

template<>
void fun(int a)
{
   cout << "Specialized Template for int type: " << a << endl;
}
```

```
int main()
{
   fun<char>('a');
   fun<int>(10);
   fun<float>(10.14);
}
```

Output:

```
The main template fun(): a
Specialized Template for int type: 10
The main template fun(): 10.14
```

An Example Program for class template specialization

In the following program, a specialized version of class Test is written for int data type.

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

template <class T>
class Test
{
  // Data memnbers of test
public:
   Test()
   {
      // Initializstion of data memnbers
      cout << "General template object \n";
   }
   // Other methods of Test
};

template <>
class Test <int>
{
public:
   Test()
   {
      // Initializstion of data memnbers
      cout << "Specialized template object\n";
   }
};

int main()
{
   Test<int> a;
   Test<char> b;
   Test<float> c;
   return 0;
}
```

Output:

```
Specialized template object
General template object
General template object
```

How does template specialization work?

When we write any template based function or class, compiler creates a copy of that function/class whenever compiler sees that being used for a new data type or new set of data types(in case of multiple template arguments).

If a specialized version is present, compiler first checks with the specialized version and then the main template. Compiler first checks with the most specialized version by matching the passed parameter with the data type(s) specified in a specialized version.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Namespace in C++ | Set 1 (Introduction)

Consider following C++ program.

```
// A program to demonstrate need of namespace
int main()
{
   int value;
   value = 0;
   double value; // Error here
   value = 0.0;
}
```

Output：

```
Compiler Error:
'value' has a previous declaration as 'int value'
```

In each scope, a name can only represent one entity. So, there cannot be two variables with the same name in the same scope. Using namespaces, we can create two variables or member functions having the same name.

```
// Here we can see that more than one variables
// are being used without reporting any error.
// That is because they are declared in the
// different namespaces and scopes.
#include <iostream>
using namespace std;

// Variable created inside namespace
namespace first
{
   int val = 500;
}

// Global variable
int val = 100;

int main()
{
   // Local variable
   int val = 200;

   // These variables can be accessed from
   // outside the namespace using the scope
   // operator ::
   cout << first::val << '\n';

   return 0;
}
```

Output:

```
500
```

Definition and Creation:

Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

- Namespace is a feature added in C++ and not present in C.
- A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.
- Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

```
namespace namespace_name
{
  int x, y; // code declarations where
        // x and y are declared in
        // namespace_name's scope
}
```

- Namespace declarations appear only at global scope.
- Namespace declarations can be nested within another namespace.
- Namespace declarations don't have access specifiers. (Public or private)
- No need to give semicolon after the closing brace of definition of namespace.
- We can split the definition of namespace over several units.

```
// Creating namespaces
#include <iostream>
using namespace std;
namespace ns1
{
   int value()    { return 5; }
}
namespace ns2
{
   const double x = 100;
   double value() {  return 2*x; }
}

int main()
{
   // Access value function within ns1
   cout << ns1::value() << '\n';

   // Access value function within ns2
   cout << ns2::value() << '\n';

   // Access variable x directly
   cout << ns2::x << '\n';

   return 0;
}
```

Output:

```
5
200
100
```

Classes and Namespace:

Following is a simple way to create classes in a name space

```
// A C++ program to demonstrate use of class
// in a namespace
#include <iostream>
using namespace std;

namespace ns
{
   // A Class in a namespace
   class geek
   {
   public:
      void display()
      {
         cout << "ns::geek::display()\n";
      }
   };
}
```

```
int main()
{
   // Creating Object of student Class
   ns::geek obj;

   obj.display();

   return 0;
}
```

Output:

```
ns::geek::display()
```

Class can also be declared inside namespace and defined outside namespace using following syntax

```
// A C++ program to demonstrate use of class
// in a namespace
#include <iostream>
using namespace std;

namespace ns
{
   // Only declaring class here
   class geek;
}

// Defining class outside
class ns::geek
{
public:
   void display()
   {
      cout << "ns::geek::display()\n";
   }
};

int main()
{
   //Creating Object of student Class
   ns::geek obj;
   obj.display();
   return 0;
}
```

Output:

```
ns::geek::display()
```

We can **define methods also outside the namespace**. Following is an example code.

```
// A C++ code to demonstrate that we can define
// methods outside namespace.
#include <iostream>
using namespace std;

// Creating a namespace
namespace ns
{
   void display();
   class geek
   {
   public:
      void display();
   };
}

// Defining methods of namespace
void ns::geek::display()
```

```
{
   cout << "ns::geek::display()\n";
}
void ns::display()
{
   cout << "ns::display()\n";
}

// Driver code
int main()
{
   ns::geek obj;
   ns::display();
   obj.display();
   return 0;
}
```

Output:

```
ns::display()
ns::geek::display()
```

Can namespaces be nested in C++?

Reference:
http://www.cplusplus.com/doc/tutorial/namespaces/

This article is contributed by **Abhinav Tiwari**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-namespaces

Can namespaces be nested in C++?

In C++, namespaces can be nested, and resolution of namespace variables is hierarchical. For example, in the following code, namespace *inner* is created inside namespace *outer*, which is inside the global namespace. In the line *"int z = x"*, *x* refers to *outer::x*. If x would not have been in *outer* then this *x* would have referred to *x* in global namespace.

```
#include <iostream>

int x = 20;
namespace outer {
  int x = 10;
  namespace inner {
    int z = x; // this x refers to outer::x
  }
}

int main()
{
  std::cout<<outer::inner::z; //prints 10
  getchar();
  return 0;
}
```

Output of the above program is 10.

On a side node, unlike C++ namespaces, Java packages are not hierarchical.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

The C++ Standard Template Library (STL)

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms and iterators. It is a generalized library and so, its components are parameterized. A working knowledge of template classes is a prerequisite for working with STL.

STL has four components

- Algorithms
- Containers
- Functions
- Iterators

Algorithms

The header algorithm defines a collection of functions especially designed to be used on ranges of elements.They act on containers and provide means for various operations for the contents of the containers.

- Algorithm
  - Sorting
  - Searching
  - Important STL Algorithms
  - Useful Array algorithms
  - Partition Operations
- Numeric
  - valarray class

Containers

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

- Sequence Containers: implement data structures which can be accessed in a sequential manner.
  - vector
  - list
  - deque
  - arrays
  - forward_list(Introduced in C++11)
- Container Adaptors : provide a different interface for sequential containers.
  - queue
  - priority_queue
  - stack
- Associative Containers : implement sorted data structures that can be quickly searched (O(log n) complexity).
  - set
  - multiset
  - map
  - multimap

Functions

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.

- Functors

Iterators

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

- Iterators

Utility Library

Defined under <utility header>

- pair

References:

- http://en.cppreference.com/w/cpp/
- http://cs.stmarys.ca/~porter/csc/ref/stl/headers.html

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Category: C++ Tags: STL

Sort : Algorithms – The C++ Standard Template Library (STL)

Sorting is one of the most basic functions applied on data.

The prototype for sort is :

```
sort(startaddress, endaddress)
```

```
#include <iostream>
#include <algorithm>

using namespace std;

void show(int a[])
{
   for(int i = 0; i < 10; ++i)
      cout << '\t' << a[i];
}

int main()
{
   int a[10]= {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
   cout << "\n The array before sorting is : ";
   show(a);

   sort(a, a+10);

   cout << "\n\n The array after sorting is : ";
   show(a);

   return 0;

}
```

The outut of the above program is :

```
The array before sorting is :  1 5 8
9 6 7 3 4 2 0

 The array after sorting is :  0 1 2
3 4 5 6 7 8 9
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Binary Search : Algorithms – The C++ Standard Template Library (STL)

Binary search is a widely used searching algorithm that requires the array to be sorted before search is applied.

The prototype for binary search is :

```
binary_search(startaddress, endaddress, valuetofind)
```

```
#include <iostream>
#include <algorithm>

using namespace std;

void show(int a[], int arraysize)
{
   for(int i = 0; i < arraysize; ++i)
      cout << '\t' << a[i];
}


int main()
{
   int a[]= {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
   int asize = sizeof(a) / sizeof(a[0]);
   cout << "\n The array is : ";
   show(a, asize);

   cout << "\n\nLet's say we want to search for 2 in the array";
   cout << "\n So, we first sort the array";
   sort(a, a + 10);
   cout << "\n\n The array after sorting is : ";
   show(a, asize);
   cout << "\n\nNow, we do the binary search";
   if (binary_search(a, a + 10, 2))
     cout << "\nElement found in the array";
   else
     cout << "\nElement not found in the array";

   cout << "\n\nNow, say we want to search for 10";
    if (binary_search(a, a + 10, 10))
     cout << "\nElement found in the array";
   else
     cout << "\nElement not found in the array";

   return 0;

}
```

The output of the above program is :

```
The array is :  1 5 8 9
6 7 3 4 2 0
```

```
Let's say we want to search for 2 in the array
 So, we first sort the array

 The array after sorting is :  0 1 2
3 4 5 6 7 8 9

Now, we do the binary search
Element found in the array

Now, say we want to search for 10
Element not found in the array
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Pair : Simple Containers – The C++ Standard Template Library (STL)

The pair container is a simple container defined in **<utility>** header consisting of two data elements or objects.

- The first element is referenced as 'first' and the second element as 'second' and the order is fixed (first, second).
- Pair is used to combine together two values which may be different in type. Pair provides a way to store two heterogeneous objects as a single unit.
- Pair can be assigned, copied and compared. The array of objects allocated in a map or hash_map are of type 'pair' by default in which all the 'first' elements are unique keys associated with their 'second' value objects.
- To access the elements, we use variable name followed by dot operator followed by the keyword first or second.

Syntax :

```
pair (data_type1, data_type2) Pair_name;
```

```
//CPP program to illustrate pair STL
#include <iostream>
#include <utility>
using namespace std;

int main()
{
   pair <int, char> PAIR1 ;

   PAIR1.first = 100;
   PAIR1.second = 'G' ;

   cout << PAIR1.first << " " ;
   cout << PAIR1.second << endl ;

   return 0;
}
```

Output:

```
100 G
```

Initializing a pair

We can also initialize a pair.
Syntax :

```
pair (data_type1, data_type2) Pair_name (value1, value2) ;
```

Different ways to initialize pair:

```
pair  g1;       //default
pair  g2(1, 'a');  //initialized,  different data type
pair  g3(1, 10);   //initialized,  same data type
pair  g4(g3);    //copy of g3
```

Another way to initialize a pair is by using the make_pair() function.

```
g2 = make_pair(1, 'a');
```

```
//CPP program to illustrate Initializing of pair STL
#include <iostream>
#include <utility>
using namespace std;

int main()
{
 pair <string,double> PAIR2 ("GeeksForGeeks", 1.23);

 cout << PAIR2.first << " " ;
 cout << PAIR2.second << endl ;

return 0;
}
```

Output:

```
GeeksForGeeks 1.23
```

Note: If not initialized, the first value of the pair gets automatically initialized.

```
//CPP program to illustrate auto-initializing of pair STL
#include <iostream>
#include <utility>

using namespace std;

int main()
{
   pair <int, double> PAIR1 ;
   pair <string, char> PAIR2 ;

   cout << PAIR1.first ;  //it is initialised to 0
   cout << PAIR1.second ; //it is initialised to 0

   cout << " ";

   cout << PAIR2.first ;  //it prints nothing i.e NULL
   cout << PAIR2.second ; //it prints nothing i.e NULL

   return 0;
}
```

Output:

```
00
```

Member Functions

1. **make_pair()** : This template function allows to create a value pair without writing the types explicitly.
   Syntax :

```
Pair_name = make_pair (value1,value2);
```

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

int main()
{
   pair <int, char> PAIR1 ;
   pair <string, double> PAIR2 ("GeeksForGeeks", 1.23) ;
   pair <string, double> PAIR3 ;

   PAIR1.first = 100;
   PAIR1.second = 'G' ;

   PAIR3 = make_pair ("GeeksForGeeks is Best",4.56);

   cout << PAIR1.first << " " ;
   cout << PAIR1.second << endl ;

   cout << PAIR2.first << " " ;
   cout << PAIR2.second << endl ;

   cout << PAIR3.first << " " ;
   cout << PAIR3.second << endl ;

   return 0;
}
```

Output:

```
100 G
GeeksForGeeks 1.23
GeeksForGeeks is Best 4.56
```

2. **operators(=, ==, !=, >=, <=) :** We can use operators with pairs as well.
   - **using equal(=) :** It assigns new object for a pair object.
     Syntax :

     ```
     pair& operator= (const pair& pr);
     ```

     This Assigns pr as the new content for the pair object. The first value is assigned the first value of pr and the second value is assigned the second value of pr .
   - **Comparison (==) operator with pair :** For given two pairs say pair1 and pair2, the comparison operator compares the first value and second value of those two pairs i.e. if pair1.first is equal to pair2.first or not AND if pair1.second is equal to pair2.second or not .
   - **Not equal (!=) operator with pair :** For given two pairs say pair1 and pair2, the != operator compares the first values of those two pairs i.e. if pair1.first is equal to pair2.first or not, if they are equal then it checks the second values of both.
   - **Logical(>=, <=)operators with pair :** For given two pairs say pair1 and pair2, the =, >, can be used with pairs as well.

.

```
//CPP code to illustrate operators in pair
#include <iostream>
#include<utility>
using namespace std;

int main()
{
   pair<int, int>pair1 = make_pair(1, 12);
   pair<int, int>pair2 = make_pair(9, 12);


   cout << (pair1 == pair2) << endl;
   cout << (pair1 != pair2) << endl;
   cout << (pair1 >= pair2) << endl;
   cout << (pair1 <= pair2) << endl;
   cout << (pair1 > pair2) << endl;
```

```
    cout << (pair1 < pair2) << endl;

    return 0;
}
```

Output:

```
0
1
0
1
0
1
```

3. **swap :** This function swaps the contents of one pair object with the contents of another pair object. The pairs must be of same type.
   Syntax :

```
pair1.swap(pair2) ;
```

For two given pairs say pair1 and pair2 of same type, swap function will swap the pair1.first with pair2.first and pair1.second with pair2.second.

```
#include <iostream>
#include<utility>

using namespace std;

int main()
{
    pair<char, int>pair1 = make_pair('A', 1);
    pair<char, int>pair2 = make_pair('B', 2);

    cout << "Before swapping:\n " ;
    cout << "Contents of pair1 = " << pair1.first << " " << pair1.second ;
    cout << "Contents of pair2 = " << pair2.first << " " << pair2.second ;
    pair1.swap(pair2);

    cout << "\nAfter swapping:\n ";
    cout << "Contents of pair1 = " << pair1.first << " " << pair1.second ;
    cout << "Contents of pair2 = " << pair2.first << " " << pair2.second ;

    return 0;
}
```

Output:

```
Before swapping:
Contents of pair1 = (A, 1)
Contents of pair2 = (B, 2)

After swapping:
Contents of pair1 = (B, 2)
Contents of pair2 = (A, 1)
```

```
//CPP program to illustrate pair in STL
#include <iostream>
#include <utility>
#include <string>
using namespace std;

int main()
{
 pair <string, int> g1;
 pair <string, int> g2("Quiz", 3);
 pair <string, int> g3(g2);
 pair <int, int> g4(5, 10);

 g1 = make_pair(string("Geeks"), 1);
 g2.first = ".com";
```

```
 g2.second = 2;

 cout << "This is pair g" << g1.second << " with "
  << "value " << g1.first << "." << endl << endl;

 cout << "This is pair g" << g3.second
  << " with value " << g3.first
  << "This pair was initialized as a copy of "
  << "pair g2" << endl << endl;

 cout << "This is pair g" << g2.second
  << " with value " << g2.first
  << "\nThe values of this pair were"
  << " changed after initialization."
  << endl << endl;

 cout << "This is pair g4 with values "
  << g4.first << " and " << g4.second
  << " made for showing addition. \nThe "
  << "sum of the values in this pair is "
  << g4.first+g4.second
  << "." << endl << endl;

 cout << "We can concatenate the values of"
  << " the pairs g1, g2 and g3 : "
  << g1.first + g3.first + g2.first << endl << endl;

 cout << "We can also swap pairs "
  << "(but type of pairs should be same) : " << endl;
 cout << "Before swapping, " << "g1 has " << g1.first
  << " and g2 has " << g2.first << endl;
 swap(g1, g2);
 cout << "After swapping, "
  << "g1 has " << g1.first << " and g2 has " << g2.first;

 return 0;
}
```

Output:

```
This is pair g1 with value Geeks.

This is pair g3 with value QuizThis pair was initialized as a copy of pair g2

This is pair g2 with value .com
The values of this pair were changed after initialization.

This is pair g4 with values 5 and 10 made for showing addition.
The sum of the values in this pair is 15.

We can concatenate the values of the pairs g1, g2 and g3 : GeeksQuiz.com

We can also swap pairs (but type of pairs should be same) :
Before swapping, g1 has Geeks and g2 has .com
After swapping, g1 has .com and g2 has Geeks
```

This article is contributed by **MAZHAR IMAM KHAN**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Category: C++

Vector : Sequence Containers – The C++ Standard Template Library (STL) – Set 1

Vector

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array.Removing the last element takes only constant time, because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

Certain functions are associated with vector :

Iterators

1. begin() – Returns an iterator pointing to the first element in the vector
2. end() – Returns an iterator pointing to the theoretical element that follows last element in the vector
3. rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
4. rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector <int> g1;
   vector <int> :: iterator i;
   vector <int> :: reverse_iterator ir;

   for (int i = 1; i <= 5; i++)
      g1.push_back(i);

   cout << "Output of begin and end\t:\t";
   for (i = g1.begin(); i != g1.end(); ++i)
      cout << *i << '\t';

   cout << endl << endl;
   cout << "Output of rbegin and rend\t:\t";
   for (ir = g1.rbegin(); ir != g1.rend(); ++ir)
      cout << '\t' << *ir;

   return 0;

}
```

The output of the above program is :

```
Output of begin and end : 1 2 3 4 5

Output of rbegin and rend  : 5 4 3 2 1
```

Capacity

1. size() – Returns the number of elements in the vector
2. max_size() – Returns the maximum number of elements that the vector can hold
3. capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements
4. resize(size_type g) – Resizes the container so that it contains ‘g’ elements
5. empty() – Returns whether the container is empty

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector <int> g1;

   for (int i = 1; i <= 5; i++)
      g1.push_back(i);

   cout << "Size : " << g1.size();
   cout << "\nCapacity : " << g1.capacity();
   cout << "\nMax_Size : " << g1.max_size();

   return 0;

}
```

The output of the above program is :

```
Size : 5
Capacity : 8
Max_Size : 4611686018427387903
```

Accessing the elements

1. reference operator [g] – Returns a reference to the element at position ‘g’ in the vector
2. at(g) – Returns a reference to the element at position ‘g’ in the vector
3. front() – Returns a reference to the first element in the vector
4. back() – Returns a reference to the last element in the vector

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

int main()
{
   vector <int> g1;

   for (int i = 1; i <= 10; i++)
      g1.push_back(i * 10);

   cout << "Reference operator [g] : g1[2] = " << g1[2];
   cout << endl;
   cout << "at : g1.at(4) = " << g1.at(4);
   cout << endl;
   cout << "front() : g1.front() = " << g1.front();
   cout << endl;
   cout << "back() : g1.back() = " << g1.back();
   cout << endl;

   return 0;

}
```

The output of the above program is :

```
Reference operator [g] : g1[2] = 30
at : g1.at(4) = 50
front() : g1.front() = 10
back() : g1.back() = 100
```

Click here for Set 2 of Vectors.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Category: C++ Tags: STL

List : Sequence Containers – The C++ Standard Template Library (STL)

List

Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about doubly linked list. For implementing a singly linked list, we use forward list.

Functions used with List :
front() – Returns reference to the first element in the list
back() – Returns reference to the last element in the list
push_front(g) – Adds a new element 'g' at the beginning of the list
push_back(g) – Adds a new element 'g' at the end of the list
pop_front() – Removes the first element of the list, and reduces size of the list by 1
pop_back() – Removes the last element of the list, and reduces size of the list by 1
begin() – Returns an iterator pointing to the first element of the list
end() – Returns an iterator pointing to the theoretical last element which follows the last element
empty() – Returns whether the list is empty(1) or not(0)
insert() – Inserts new elements in the list before the element at a specified position
erase() – Removes a single element or a range of elements from the list
assign() – Assigns new elements to list by replacing current elements and resizes the list
remove() – Removes all the elements from the list, which are equal to given element
reverse() – Reverses the list
size() – Returns the number of elements in the list
sort() – Sorts the list in increasing order

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

//function for printing the elements in a list
void showlist(list <int> g)
{
   list <int> :: iterator it;
   for(it = g.begin(); it != g.end(); ++it)
      cout << '\t' << *it;
   cout << '\n';
}

int main()
{

   list <int> gqlist1, gqlist2;


   for (int i = 0; i < 10; ++i)
   {
      gqlist1.push_back(i * 2);
      gqlist2.push_front(i * 3);
   }
   cout << "\nList 1 (gqlist1) is : ";
```

```
    showlist(gqlist1);

    cout << "\nList 2 (gqlist2) is : ";
    showlist(gqlist2);

    cout << "\ngqlist1.front() : " << gqlist1.front();
    cout << "\ngqlist1.back() : " << gqlist1.back();

    cout << "\ngqlist1.pop_front() : ";
    gqlist1.pop_front();
    showlist(gqlist1);

    cout << "\ngqlist2.pop_back() : ";
    gqlist2.pop_back();
    showlist(gqlist2);

    cout << "\ngqlist1.reverse() : ";
    gqlist1.reverse();
    showlist(gqlist1);

    cout << "\ngqlist2.sort(): ";
    gqlist2.sort();
    showlist(gqlist2);

    return 0;

}
```

The output of the above program is :

```
List 1 (gqlist1) is :  0 2 4 6
8 10 12 14 16 18

List 2 (gqlist2) is :  27 24 21 18
15 12 9 6 3 0

gqlist1.front() : 0
gqlist1.back() : 18
gqlist1.pop_front() :  2 4 6 8
10 12 14 16 18

gqlist2.pop_back() :  27 24 21 18
15 12 9 6 3

gqlist1.reverse() :  18 16 14 12
10 8 6 4 2

gqlist2.sort():  3 6 9 12
15 18 21 24 27
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Deque : Sequence Containers – The C++ Standard Template Library (STL)

Double ended queues are sequence containers with the feature of expansion and contraction on both the ends. They are similar to vectors, but are more efficient in case of insertion and deletion of elements at the end, and also the beginning. Unlike vectors, contiguous storage allocation may not be guaranteed.

The functions for deque are same as vector, with an addition of push and pop operations for both front and back.

```
#include <iostream>
#include <deque>

using namespace std;

void showdq(deque <int> g)
{
   deque <int> :: iterator it;
   for (it = g.begin(); it != g.end(); ++it)
      cout << '\t' << *it;
   cout << '\n';
}

int main()
{
   deque <int> gquiz;
   gquiz.push_back(10);
   gquiz.push_front(20);
   gquiz.push_back(30);
   gquiz.push_front(15);
   cout << "The deque gquiz is : ";
   showdq(gquiz);

   cout << "\ngquiz.size() : " << gquiz.size();
   cout << "\ngquiz.max_size() : " << gquiz.max_size();

   cout << "\ngquiz.at(2) : " << gquiz.at(2);
   cout << "\ngquiz.front() : " << gquiz.front();
   cout << "\ngquiz.back() : " << gquiz.back();

   cout << "\ngquiz.pop_front() : ";
   gquiz.pop_front();
   showdq(gquiz);

   cout << "\ngquiz.pop_back() : ";
   gquiz.pop_back();
   showdq(gquiz);

   return 0;
}
```

The output of the above program is :

```
The deque gquiz is :  15 20 10 30

gquiz.size() : 4
gquiz.max_size() : 4611686018427387903
gquiz.at(2) : 10
gquiz.front() : 15
gquiz.back() : 30
gquiz.pop_front() :  20 10 30

gquiz.pop_back() :  20 10
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Queue : Container Adaptors – The C++ Standard Template Library (STL)

Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front.

The functions supported by queue are :
empty() – Returns whether the queue is empty
size() – Returns the size of the queue
front() – Returns a reference to the first element of the queue
back() – Returns a reference to the last element of the queue
push(g) – Adds the element 'g' at the end of the queue
pop() – Deletes the first element of the queue

```
#include <iostream>
#include <queue>

using namespace std;

void showq(queue <int> gq)
{
   queue <int> g = gq;
   while (!g.empty())
   {
      cout << '\t' << g.front();
      g.pop();
   }
   cout << '\n';
}

int main()
{
   queue <int> gquiz;
   gquiz.push(10);
   gquiz.push(20);

   cout << "The queue gquiz is : ";
   showq(gquiz);

   cout << "\ngquiz.size() : " << gquiz.size();
   cout << "\ngquiz.front() : " << gquiz.front();
   cout << "\ngquiz.back() : " << gquiz.back();

   cout << "\ngquiz.pop() : ";
   gquiz.pop();
   showq(gquiz);

   return 0;
}
```

The output of the above program is :

```
The queue gquiz is :  10 20

gquiz.size() : 2
gquiz.front() : 10
gquiz.back() : 20
gquiz.pop() :  20
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Category: C++

Priority Queue : Container Adaptors – The C++ Standard Template Library (STL)

Priority queues are a type of container adaptors, specifically designed such that the first element of the queue is the greatest of all elements in the queue.

The functions associated with priority queue are:
empty() – Returns whether the queue is empty
size() – Returns the size of the queue
top() – Returns a reference to the top most element of the queue
push(g) – Adds the element ‘g’ at the end of the queue
pop() – Deletes the first element of the queue

```
#include <iostream>
#include <queue>

using namespace std;

void showpq(priority_queue <int> gq)
{
   priority_queue <int> g = gq;
   while (!g.empty())
   {
      cout << '\t' << g.top();
      g.pop();
   }
   cout << '\n';
}

int main ()
{
   priority_queue <int> gquiz;
   gquiz.push(10);
   gquiz.push(30);
   gquiz.push(20);
   gquiz.push(5);
   gquiz.push(1);

   cout << "The priority queue gquiz is : ";
   showpq(gquiz);

   cout << "\ngquiz.size() : " << gquiz.size();
   cout << "\ngquiz.top() : " << gquiz.top();


   cout << "\ngquiz.pop() : ";
   gquiz.pop();
   showpq(gquiz);

   return 0;
}
```

The output of the above programs is :

```
The priority queue gquiz is :  30 20 10 5 1

gquiz.size() : 5
gquiz.top() : 30
gquiz.pop() :  20 10 5 1
```

How to implement Min Heap using priority queue?

Prim's algorithm using priority_queue in STL

Dijkstra's Shortest Path Algorithm using priority_queue of STL

Greedy Algorithms | Set 3 (Huffman Coding)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Stack : Container Adaptors – The C++ Standard Template Library (STL)

Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end and (top) an element is removed from that end only.

The functions associated with stack are:
empty() – Returns whether the stack is empty
size() – Returns the size of the stack
top() – Returns a reference to the top most element of the stack
push(g) – Adds the element 'g' at the top of the stack
pop() – Deletes the top most element of the stack

```
#include <iostream>
#include <stack>

using namespace std;

void showstack(stack <int> gq)
{
   stack <int> g = gq;
   while (!g.empty())
   {
      cout << '\t' << g.top();
      g.pop();
   }
   cout << '\n';
}

int main ()
{
   stack <int> gquiz;
   gquiz.push(10);
   gquiz.push(30);
   gquiz.push(20);
   gquiz.push(5);
   gquiz.push(1);

   cout << "The stack gquiz is : ";
   showstack(gquiz);
```

```
    cout << "\ngquiz.size() : " << gquiz.size();
    cout << "\ngquiz.top() : " << gquiz.top();


    cout << "\ngquiz.pop() : ";
    gquiz.pop();
    showstack(gquiz);

    return 0;
}
```

The output of the above program is :

```
The stack gquiz is :  1 5 20 30 10

gquiz.size() : 5
gquiz.top() : 1
gquiz.pop() :  5 20 30 10
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Set : Associative Containers – The C++ Standard Template Library (STL)

Sets are a type of associative containers in which each element has to be unique, because the value of the element identifies it. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

Functions associated with Set:
begin() – Returns an iterator to the first element in the set
end() – Returns an iterator to the theoretical element that follows last element in the set
size() – Returns the number of elements in the set
max_size() – Returns the maximum number of elements that the set can hold
empty() – Returns whether the set is empty
pair <iterator, bool> insert(const g) – Adds a new element 'g' to the set
iterator insert (iterator position, const g) – Adds a new element 'g' at the position pointed by iterator
erase(iterator position) – Removes the element at the position pointed by the iterator
erase(const g)- Removes the value 'g' from the set
clear() – Removes all the elements from the set
key_comp() / value_comp() – Returns the object that determines how the elements in the set are ordered ('<' by default)
find(const g) – Returns an iterator to the element 'g' in the set if found, else returns the iterator to end
count(const g) – Returns the number of matches to element 'g' in the set
lower_bound(const g) – Returns an iterator to the first element that is equivalent to 'g' or definitely will not go before the element 'g' in the set
upper_bound(const g) – Returns an iterator to the first element that is equivalent to 'g' or definitely will go after the element 'g' in the set

```
#include <iostream>
#include <set>
#include <iterator>

using namespace std;

int main()
```

```
{
    // empty set container
    set <int, greater <int> > gquiz1;

    // insert elements in random order
    gquiz1.insert(40);
    gquiz1.insert(30);
    gquiz1.insert(60);
    gquiz1.insert(20);
    gquiz1.insert(50);
    gquiz1.insert(50); // only one 50 will be added to the set
    gquiz1.insert(10);

    // printing set gquiz1
    set <int, greater <int> > :: iterator itr;
    cout << "\nThe set gquiz1 is : ";
    for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr)
    {
        cout << '\t' << *itr;
    }
    cout << endl;

    // assigning the elements from gquiz1 to gquiz2
    set <int> gquiz2(gquiz1.begin(), gquiz1.end());

    // print all elements of the set gquiz2
    cout << "\nThe set gquiz2 after assign from gquiz1 is : ";
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout << '\t' << *itr;
    }
    cout << endl;

    // remove all elements up to 30 in gquiz2
    cout << "\ngquiz2 after removal of elements less than 30 : ";
    gquiz2.erase(gquiz2.begin(), gquiz2.find(30));
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout << '\t' << *itr;
    }

    // remove all elements with value 50 in gquiz2
    int num;
    num = gquiz2.erase (50);
    cout << "\ngquiz2.erase(50) : ";
    cout << num << " removed \t" ;
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout << '\t' << *itr;
    }

    cout << endl;

    //lower bound and upper bound for set gquiz1
    cout << "gquiz1.lower_bound(40) : "
         << *gquiz1.lower_bound(40) << endl;
    cout << "gquiz1.upper_bound(40) : "
         << *gquiz1.upper_bound(40) << endl;

    //lower bound and upper bound for set gquiz2
    cout << "gquiz2.lower_bound(40) : "
         << *gquiz2.lower_bound(40) << endl;
    cout << "gquiz2.upper_bound(40) : "
         << *gquiz2.upper_bound(40) << endl;

    return 0;

}
```

The output of the above program is :

```
The set gquiz1 is :     60 50 40 30 20 10

The set gquiz2 after assign from gquiz1 is : 10   20 30 40 50 60

gquiz2 after removal of elements less than 30 : 30 40 50 60
gquiz2.erase(50) : 1 removed  30 40 60
gquiz1.lower_bound(40) : 40
gquiz1.upper_bound(40) : 30
gquiz2.lower_bound(40) : 40
gquiz2.upper_bound(40) : 60
```

Dijkstra's shortest path algorithm using set in STL

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++ Tags: STL

Multiset : Associative Containers – The C++ Standard Template Library (STL)

Multisets are a type of associative containers similar to set, with an exception that multiple elements can have same values.

Functions associated with multiset:
begin() – Returns an iterator to the first element in the multiset
end() – Returns an iterator to the theoretical element that follows last element in the multiset
size() – Returns the number of elements in the multiset
max_size() – Returns the maximum number of elements that the multiset can hold
empty() – Returns whether the multiset is empty
pair insert(const g) – Adds a new element 'g' to the multiset
iterator insert (iterator position,const g) – Adds a new element 'g' at the position pointed by iterator
erase(iterator position) – Removes the element at the position pointed by the iterator
erase(const g)- Removes the value 'g' from the multiset
clear() – Removes all the elements from the multiset
key_comp() / value_comp() – Returns the object that determines how the elements in the multiset are ordered (' find(const g) – Returns an iterator to the element 'g' in the multiset if found, else returns the iterator to end
count(const g) – Returns the number of matches to element 'g' in the multiset
lower_bound(const g) – Returns an iterator to the first element that is equivalent to 'g' or definitely will not go before the element 'g' in the multiset
upper_bound(const g) – Returns an iterator to the first element that is equivalent to 'g' or definitely will go after the element 'g' in the multiset

```
#include <iostream>
#include <set>
#include <iterator>

using namespace std;

int main()
{
   // empty multiset container
   multiset <int, greater <int> > gquiz1;

   // insert elements in random order
   gquiz1.insert(40);
```

```
    gquiz1.insert(30);
    gquiz1.insert(60);
    gquiz1.insert(20);
    gquiz1.insert(50);
    gquiz1.insert(50); // 50 will be added again to the multiset unlike set
    gquiz1.insert(10);

    // printing multiset gquiz1
    multiset <int, greater <int> > :: iterator itr;
    cout << "\nThe multiset gquiz1 is : ";
    for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr)
    {
        cout << '\t' << *itr;
    }
    cout << endl;

    // assigning the elements from gquiz1 to gquiz2
    multiset <int> gquiz2(gquiz1.begin(), gquiz1.end());

    // print all elements of the multiset gquiz2
    cout << "\nThe multiset gquiz2 after assign from gquiz1 is : ";
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout << '\t' << *itr;
    }
    cout << endl;

    // remove all elements up to element with value 30 in gquiz2
    cout << "\ngquiz2 after removal of elements less than 30 : ";
    gquiz2.erase(gquiz2.begin(), gquiz2.find(30));
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout << '\t' << *itr;
    }

    // remove all elements with value 50 in gquiz2
    int num;
    num = gquiz2.erase(50);
    cout << "\ngquiz2.erase(50) : ";
    cout << num << " removed \t" ;
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout << '\t' << *itr;
    }

    cout << endl;

    //lower bound and upper bound for multiset gquiz1
    cout << "gquiz1.lower_bound(40) : "
         << *gquiz1.lower_bound(40) << endl;
    cout << "gquiz1.upper_bound(40) : "
         << *gquiz1.upper_bound(40) << endl;

    //lower bound and upper bound for multiset gquiz2
    cout << "gquiz2.lower_bound(40) : "
         << *gquiz2.lower_bound(40) << endl;
    cout << "gquiz2.upper_bound(40) : "
         << *gquiz2.upper_bound(40) << endl;

         return 0;

}
```

The output of the above program is :

```
The multiset gquiz1 is :  60  50 50 40 30 20 10

The multiset gquiz2 after assign from gquiz1 is :  10  20 30 40 50 50 60
```

```
gquiz2 after removal of elements less than 30 :  30 40 50 50 60
gquiz2.erase(50) : 2 removed   30 40 60
gquiz1.lower_bound(40) : 40
gquiz1.upper_bound(40) : 30
gquiz2.lower_bound(40) : 40
gquiz2.upper_bound(40) : 60
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Map : Associative Containers – The C++ Standard Template Library (STL)

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have same key values.

Functions associated with Map:
begin() – Returns an iterator to the first element in the map
end() – Returns an iterator to the theoretical element that follows last element in the map
size() – Returns the number of elements in the map
max_size() – Returns the maximum number of elements that the map can hold
empty() – Returns whether the map is empty
pair insert(keyvalue,mapvalue) – Adds a new element to the map
erase(iterator position) – Removes the element at the position pointed by the iterator
erase(const g)- Removes the key value ‘g’ from the map
clear() – Removes all the elements from the map
key_comp() / value_comp() – Returns the object that determines how the elements in the map are ordered (‘ find(const g) – Returns an iterator to the element with key value ‘g’ in the map if found, else returns the iterator to end
count(const g) – Returns the number of matches to element with key value ‘g’ in the map
lower_bound(const g) – Returns an iterator to the first element that is equivalent to mapped value with key value ‘g’ or definitely will not go before the element with key value ‘g’ in the map
upper_bound(const g) – Returns an iterator to the first element that is equivalent to mapped value with key value ‘g’ or definitely will go after the element with key value ‘g’ in the map

```
#include <iostream>
#include <map>
#include <iterator>

using namespace std;

int main()
{
   map <int, int> gquiz1;       // empty map container

   // insert elements in random order
   gquiz1.insert(pair <int, int> (1, 40));
   gquiz1.insert(pair <int, int> (2, 30));
   gquiz1.insert(pair <int, int> (3, 60));
   gquiz1.insert(pair <int, int> (4, 20));
   gquiz1.insert(pair <int, int> (5, 50));
   gquiz1.insert(pair <int, int> (6, 50));
   gquiz1.insert(pair <int, int> (7, 10));
```

```
    // printing map gquiz1
    map <int, int> :: iterator itr;
    cout << "\nThe map gquiz1 is : \n";
    cout << "\tKEY\tELEMENT\n";
    for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr)
    {
        cout  <<  '\t' << itr->first
             <<  '\t' << itr->second << '\n';
    }
    cout << endl;

    // assigning the elements from gquiz1 to gquiz2
    map <int, int> gquiz2(gquiz1.begin(), gquiz1.end());

    // print all elements of the map gquiz2
    cout << "\nThe map gquiz2 after assign from gquiz1 is : \n";
    cout << "\tKEY\tELEMENT\n";
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout  <<  '\t' << itr->first
             <<  '\t' << itr->second << '\n';
    }
    cout << endl;

    // remove all elements up to element with key=3 in gquiz2
    cout << "\ngquiz2 after removal of elements less than key=3 : \n";
    cout << "\tKEY\tELEMENT\n";
    gquiz2.erase(gquiz2.begin(), gquiz2.find(3));
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout  <<  '\t' << itr->first
             <<  '\t' << itr->second << '\n';
    }

    // remove all elements with key = 4
    int num;
    num = gquiz2.erase (4);
    cout << "\ngquiz2.erase(4) : ";
    cout << num << " removed \n" ;
    cout << "\tKEY\tELEMENT\n";
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout  <<  '\t' << itr->first
             <<  '\t' << itr->second << '\n';
    }

    cout << endl;

    //lower bound and upper bound for map gquiz1 key = 5
    cout << "gquiz1.lower_bound(5) : " << "\tKEY = ";
    cout << gquiz1.lower_bound(5)->first << '\t';
    cout << "\tELEMENT = " << gquiz1.lower_bound(5)->second << endl;
    cout << "gquiz1.upper_bound(5) : " << "\tKEY = ";
    cout << gquiz1.upper_bound(5)->first << '\t';
    cout << "\tELEMENT = " << gquiz1.upper_bound(5)->second << endl;

    return 0;

}
```

The output of the above program is :

```
The map gquiz1 is :
 KEY ELEMENT
 1 40
 2 30
 3 60
 4 20
```

```
 5 50
 6 50
 7 10


The map gquiz2 after assign from gquiz1 is :
 KEY ELEMENT
 1 40
 2 30
 3 60
 4 20
 5 50
 6 50
 7 10


gquiz2 after removal of elements less than key=3 :
 KEY ELEMENT
 3 60
 4 20
 5 50
 6 50
 7 10

gquiz2.erase(4) : 1 removed
 KEY ELEMENT
 3 60
 5 50
 6 50
 7 10

gquiz1.lower_bound(5) :  KEY = 5  ELEMENT = 50
gquiz1.upper_bound(5) :  KEY = 6  ELEMENT = 50
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: C++

Multimap : Associative Containers – The C++ Standard Template Library (STL)

Multimap is similar to mapwith an addition that multiple elements can have same keys. Rather than each element being unique, the key value and mapped value pair has to be unique in this case.

Functions associated with multimap:

- **begin()** – Returns an iterator to the first element in the multimap
- **end()** – Returns an iterator to the theoretical element that follows last element in the multimap
- **size()** – Returns the number of elements in the multimap
- **max_size()** – Returns the maximum number of elements that the multimap can hold
- **empty()** – Returns whether the multimap is empty
- **pair<int,int> insert(keyvalue,multimapvalue**) – Adds a new element to the multimap
- **erase(iterator position)** – Removes the element at the position pointed by the iterator
- **erase(const g)**– Removes the key value ‘g’ from the multimap
- **clear() –** Removes all the elements from the multimap
- **key_comp()** / **value_comp()** – Returns the object that determines how the elements in the multimap are ordered (‘<‘ by default)
- **find(const g)** – Returns an iterator to the element with key value ‘g’ in the multimap if found, else returns the iterator to end
- **count(const g)** – Returns the number of matches to element with key value ‘g’ in the multimap

- **lower_bound(const g)** – Returns an iterator to the first element that is equivalent to multimapped value with key value ‘g’ or definitely will not go before the element with key value ‘g’ in the multimap
- **upper_bound(const g)** – Returns an iterator to the first element that is equivalent to multimapped value with key value ‘g’ or definitely will go after the element with key value ‘g’ in the multimap

C++ implementation to illustrate above functions

```
#include <iostream>
#include <map>
#include <iterator>

using namespace std;

int main()
{
   multimap <int, int> gquiz1;       // empty multimap container

   // insert elements in random order
   gquiz1.insert(pair <int, int> (1, 40));
   gquiz1.insert(pair <int, int> (2, 30));
   gquiz1.insert(pair <int, int> (3, 60));
   gquiz1.insert(pair <int, int> (4, 20));
   gquiz1.insert(pair <int, int> (5, 50));
   gquiz1.insert(pair <int, int> (6, 50));
   gquiz1.insert(pair <int, int> (6, 10));

   // printing multimap gquiz1
   multimap <int, int> :: iterator itr;
   cout << "\nThe multimap gquiz1 is : \n";
   cout << "\tKEY\tELEMENT\n";
   for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr)
   {
      cout  <<  '\t' << itr->first
           <<  '\t' << itr->second << '\n';
   }
   cout << endl;

   // assigning the elements from gquiz1 to gquiz2
   multimap <int, int> gquiz2(gquiz1.begin(),gquiz1.end());

   // print all elements of the multimap gquiz2
   cout << "\nThe multimap gquiz2 after assign from gquiz1 is : \n";
   cout << "\tKEY\tELEMENT\n";
   for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
   {
      cout << '\t' << itr->first
           << '\t' << itr->second << '\n';
   }
   cout << endl;

   // remove all elements up to element with value 30 in gquiz2
   cout << "\ngquiz2 after removal of elements less than key=3 : \n";
   cout << "\tKEY\tELEMENT\n";
   gquiz2.erase(gquiz2.begin(), gquiz2.find(3));
   for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
   {
      cout << '\t' << itr->first
           << '\t' << itr->second << '\n';
   }

   // remove all elements with key = 4
   int num;
   num = gquiz2.erase(4);
   cout << "\ngquiz2.erase(4) : ";
   cout << num << " removed \n" ;
   cout << "\tKEY\tELEMENT\n";
   for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
   {
      cout << '\t' << itr->first
           << '\t' << itr->second << '\n';
   }
```

```
    cout << endl;

    //lower bound and upper bound for multimap gquiz1 key = 5
    cout << "gquiz1.lower_bound(5) : " << "\tKEY = ";
    cout << gquiz1.lower_bound(5)->first << '\t';
    cout << "\tELEMENT = " << gquiz1.lower_bound(5)->second << endl;
    cout << "gquiz1.upper_bound(5) : " << "\tKEY = ";
    cout << gquiz1.upper_bound(5)->first << '\t';
    cout << "\tELEMENT = " << gquiz1.upper_bound(5)->second << endl;

    return 0;

}
```

Output:

```
The multimap gquiz1 is :
 KEY ELEMENT
 1 40
 2 30
 3 60
 4 20
 5 50
 6 50
 6 10


The multimap gquiz2 after assign from gquiz1 is :
 KEY ELEMENT
 1 40
 2 30
 3 60
 4 20
 5 50
 6 50
 6 10


gquiz2 after removal of elements less than key=3 :
 KEY ELEMENT
 3 60
 4 20
 5 50
 6 50
 6 10

gquiz2.erase(4) : 1 removed
 KEY ELEMENT
 3 60
 5 50
 6 50
 6 10

gquiz1.lower_bound(5) :  KEY = 5  ELEMENT = 50
gquiz1.upper_bound(5) :  KEY = 6  ELEMENT = 50
```

Application :

Sort an array according to absolute difference with given value

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Local Classes in C++

A class declared inside a function becomes local to that function and is called Local Class in C++. For example, in the following program, Test is a local class in fun().

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

void fun()
{
    class Test  // local to fun
    {
      /* members of Test class */
    };
}

int main()
{
   return 0;
}
```

Following are some interesting facts about local classes.

1) *A local class type name can only be used in the enclosing function.* For example, in the following program, declarations of t and tp are valid in fun(), but invalid in main().

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

void fun()
{
    // Local class
    class Test
    {
      /* ... */
    };

    Test t;  // Fine
    Test *tp;  // Fine
}

int main()
{
   Test t;  // Error
   Test *tp;  // Error
   return 0;
}
```

2) *All the methods of Local classes must be defined inside the class only.* For example, program 1 works fine and program 2 fails in compilation.

```
// PROGRAM 1
#include<iostream>
using namespace std;

void fun()
{
   class Test  // local to fun
   {
   public:

     // Fine as the method is defined inside the local class
     void method() {
       cout << "Local Class method() called";
```

```
    }
  };

  Test t;
  t.method();
}

int main()
{
  fun();
  return 0;
}
```

Output:

```
Local Class method() called
```

```
// PROGRAM 2
#include<iostream>
using namespace std;

void fun()
{
  class Test  // local to fun
  {
  public:
    void method();
  };

  // Error as the method is defined outside the local class
  void Test::method()
  {
    cout << "Local Class method()";
  }
}

int main()
{
  return 0;
}
```

Output:

```
Compiler Error:
 In function 'void fun()':
 error: a function-definition is not allowed here before '{' token
```

3) *A Local class cannot contain static data members. It may contain static functions though.* For example, program 1 fails in compilation, but program 2 works fine.

```
// PROGRAM 1
#include<iostream>
using namespace std;

void fun()
{
   class Test  // local to fun
   {
     static int i;
   };
}

int main()
{
  return 0;
}
```

```
Compiler Error:
```

```
In function 'void fun()':
error: local class 'class fun()::Test' shall not have static data member 'int fun()::Test::i'
```

```
// PROGRAM 2
#include<iostream>
using namespace std;

void fun()
{
   class Test  // local to fun
   {
   public:
      static void method()
      {
         cout << "Local Class method() called";
      }
   };

   Test::method();
}

int main()
{
   fun();
   return 0;
}
```

Output:

```
Local Class method() called
```

4) *Member methods of local class can only access static and enum variables of the enclosing function. Non-static variables of the enclosing function are not accessible inside local classes.* For example, the program 1 compiles and runs fine. But, program 2 fails in compilation.

```
// PROGRAM 1
#include<iostream>
using namespace std;

void fun()
{
     static int x;
     enum {i = 1, j = 2};

     // Local class
     class Test
     {
       public:
         void method() {
            cout << "x = " << x << endl;   // fine as x is static
            cout << "i = " << i << endl;   // fine as i is enum
         }
     };

     Test t;
     t.method();
}

int main()
{
   fun();
   return 0;
}
```

Output:

```
x = 0
i = 1
```

```
// PROGRAM 2
#include<iostream>
using namespace std;

void fun()
{
    int x;

    // Local class
    class Test
    {
      public:
        void method() {
            cout << "x = " << x << endl;
        }
    };

    Test t;
    t.method();
}

int main()
{
   fun();
   return 0;
}
```

Output:

```
In member function 'void fun()::Test::method()':
error: use of 'auto' variable from containing function
```

5) *Local classes can access global types, variables and functions. Also, local classes can access other local classes of same function..*
For example, following program works fine.

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

int x;

void fun()
{

    // First Local class
    class Test1 {
    public:
      Test1() { cout << "Test1::Test1()" << endl; }
    };

    // Second Local class
    class Test2
    {
        // Fine: A local class can use other local classes of same function
        Test1 t1;
    public:
       void method() {
          // Fine: Local class member methods can access global variables.
          cout << "x = " << x << endl;
       }
    };

    Test2 t;
    t.method();
}

int main()
{
   fun();
```

```
    return 0;
}
```

Output:

```
Test1::Test1()
x = 0
```

Also see Nested Classes in C++

References:
Local classes (C++ only)
Local Classes

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Nested Classes in C++

A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

For example, program 1 compiles without any error and program 2 fails in compilation.

Program 1

```
#include<iostream>

using namespace std;

 /* start of Enclosing class declaration */
class Enclosing {

  int x;

  /* start of Nested class declaration */
  class Nested {
    int y;
    void NestedFun(Enclosing *e) {
      cout<<e->x;  // works fine: nested class can access
                   // private members of Enclosing class
    }
  }; // declaration Nested class ends here
}; // declaration Enclosing class ends here

int main()
{

}
```

Program 2

```
#include<iostream>

using namespace std;

 /* start of Enclosing class declaration */
class Enclosing {

  int x;

  /* start of Nested class declaration */
  class Nested {
```

```
    int y;
  }; // declaration Nested class ends here

  void EnclosingFun(Nested *n) {
      cout<<n->y;  // Compiler Error: y is private in Nested
  }
}; // declaration Enclosing class ends here

int main()
{

}
```

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Casting operators in C++ | Set 1 (const_cast)

C++ supports following 4 types of casting operators:

1. const_cast
2. static_cast
3. dynamic_cast
4. reinterpret_cast

1. const_cast

const_cast is used to cast away the constness of variables. Following are some interesting facts about const_cast.

1) const_cast can be used to change non-const class members inside a const member function. Consider the following code snippet. Inside const member function fun(), 'this' is treated by the compiler as 'const student* const this', i.e. 'this' is a constant pointer to a constant object, thus compiler doesn't allow to change the data members through 'this' pointer. const_cast changes the type of 'this' pointer to 'student* const this'.

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

class student
{
private:
   int roll;
public:
   // constructor
   student(int r):roll(r) {}

   // A const function that changes roll with the help of const_cast
   void fun() const
   {
      ( const_cast <student*> (this) )->roll = 5;
   }

   int getRoll()  { return roll; }
};

int main(void)
{
   student s(3);
   cout << "Old roll number: " << s.getRoll() << endl;

   s.fun();

   cout << "New roll number: " << s.getRoll() << endl;
```

```
    return 0;
}
```

Output:

```
Old roll number: 3
New roll number: 5
```

2) const_cast can be used to pass const data to a function that doesn't receive const. For example, in the following program fun() receives a normal pointer, but a pointer to a const can be passed with the help of const_cast.

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

int fun(int* ptr)
{
    return (*ptr + 10);
}

int main(void)
{
    const int val = 10;
    const int *ptr = &val;
    int *ptr1 = const_cast <int *>(ptr);
    cout << fun(ptr1);
    return 0;
}
```

Output:

```
20
```

3) It is undefined behavior to modify a value which is initially declared as const. Consider the following program. The output of the program is undefined. The variable 'val' is a const variable and the call 'fun(ptr1)' tries to modify 'val' using const_cast.

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

int fun(int* ptr)
{
    *ptr = *ptr + 10;
    return (*ptr);
}

int main(void)
{
    const int val = 10;
    const int *ptr = &val;
    int *ptr1 = const_cast <int *>(ptr);
    fun(ptr1);
    cout << val;
    return 0;
}
```

Output:

```
 Undefined Behavior
```

It it fine to modify a value which is not initially declared as const. For example, in the above program, if we remove const from declaration of val, the program will produce 20 as output.

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

int fun(int* ptr)
{
```

```
    *ptr = *ptr + 10;
    return (*ptr);
}

int main(void)
{
    int val = 10;
    const int *ptr = &val;
    int *ptr1 = const_cast <int *>(ptr);
    fun(ptr1);
    cout << val;
    return 0;
}
```

4) const_cast is considered safer than simple type casting. It'safer in the sense that the casting won't happen if the type of cast is not same as original object. For example, the following program fails in compilation because 'int *' is being typecasted to 'char *'

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

int main(void)
{
    int a1 = 40;
    const int* b1 = &a1;
    char* c1 = const_cast <char *> (b1); // compiler error
    *c1 = 'A';
    return 0;
}
```

output:

```
prog.cpp: In function 'int main()':
prog.cpp:8: error: invalid const_cast from type 'const int*' to type 'char*'
```

5) const_cast can also be used to cast away volatile attribute. For example, in the following program, the typeid of b1 is PVKi (pointer to a volatile and constant integer) and typeid of c1 is Pi (Pointer to integer)

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

int main(void)
{
    int a1 = 40;
    const volatile int* b1 = &a1;
    cout << "typeid of b1 " << typeid(b1).name() << '\n';
    int* c1 = const_cast <int *> (b1);
    cout << "typeid of c1 " << typeid(c1).name() << '\n';
    return 0;
}
```

Output:

```
typeid of b1 PVKi
typeid of c1 Pi
```

Exercise

Predict the output of following programs. If there are compilation errors, then fix them.

Question 1

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

int main(void)
```

```
{
   int a1 = 40;
   const int* b1 = &a1;
   char* c1 = (char *)(b1);
   *c1 = 'A';
   return 0;
}
```

Question 2

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

class student
{
private:
   const int roll;
public:
   // constructor
   student(int r):roll(r) {}

   // A const function that changes roll with the help of const_cast
   void fun() const
   {
      ( const_cast <student*> (this) )->roll = 5;
   }

   int getRoll()  { return roll; }
};

int main(void)
{
   student s(3);
   cout << "Old roll number: " << s.getRoll() << endl;

   s.fun();

   cout << "New roll number: " << s.getRoll() << endl;

   return 0;
}
```

—Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Simulating final class in C++

Ever wondered how can you design a class in C++ which can't be inherited. Java and C# programming languages have this feature built-in. You can use final keyword in java, sealed in C# to make a class non-extendable.

Below is a mechanism using which we can achieve the same behavior in C++. It makes use of private constructor, virtual inheritance and friend class.

In the following code, we make the *Final* class non-inheritable. When a class *Derived* tries to inherit from it, we get compilation error.

An extra class *MakeFinal* (whose default constructor is private) is used for our purpose. Constructor of *Final* can call private constructor of *MakeFinal* as *Final* is a friend of *MakeFinal* .

Note that *MakeFinal* is also a virtual base class. The reason for this is to call the constructor of *MakeFinal* through the constructor of *Derived*, not *Final* (The constructor of a virtual base class is not called by the class that inherits from it, instead the constructor is called by the constructor of the concrete class).

```
/* A program with compilation error to demonstrate that Final class cannot
   be inherited */
```

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

class Final;  // The class to be made final

class MakeFinal // used to make the Final class final
{
private:
   MakeFinal() { cout << "MakFinal constructor" << endl; }
friend class Final;
};

class Final : virtual MakeFinal
{
public:
   Final() { cout << "Final constructor" << endl; }
};

class Derived : public Final // Compiler error
{
public:
   Derived() { cout << "Derived constructor" << endl; }
};

int main(int argc, char *argv[])
{
   Derived d;
   return 0;
}
```

Output: *Compiler Error*

```
 In constructor 'Derived::Derived()':
 error: 'MakeFinal::MakeFinal()' is private
```

In the above example, *Derived*'s constructor directly invokes *MakeFinal's* constructor, and the constructor of *MakeFinal* is private, therefore we get the compilation error.

You can create the object of *Final* class as it is friend class of *MakeFinal* and has access to its constructor. For example, the following program works fine.

```
/* A program without any compilation error to demonstrate that instances of
  the Final class can be created */
#include<iostream>
using namespace std;

class Final;

class MakeFinal
{
private:
   MakeFinal() { cout << "MakeFinal constructor" << endl; }
   friend class Final;
};

class Final : virtual MakeFinal
{
public:
   Final() { cout << "Final constructor" << endl; }
};

int main(int argc, char *argv[])
{
   Final f;
   return 0;
}
```

Output: *Compiles and runs fine*

```
MakeFinal constructor
```

```
Final constructor
```

This article is compiled by Gopal Gorthi and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Const member functions in C++

A function becomes const when const keyword is used in function's declaration. The idea of const functions is not allow them to modify the object on which they are called. It is recommended practice to make as many functions const as possible so that accidental changes to objects are avoided.

Following is a simple example of const function.

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

class Test {
    int value;
public:
    Test(int v = 0) {value = v;}

    // We get compiler error if we add a line like "value = 100;"
    // in this function.
    int getValue() const {return value;}
};

int main() {
    Test t(20);
    cout<<t.getValue();
    return 0;
}
```

Output:

```
20
```

When a function is declared as const, it can be called on any type of object. Non-const functions can only be called by non-const objects.

For example the following program has compiler errors.

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

class Test {
    int value;
public:
    Test(int v = 0) {value = v;}
    int getValue() {return value;}
};

int main() {
    const Test t;
    cout << t.getValue();
    return 0;
}
```

Output:

```
 passing 'const Test' as 'this' argument of 'int
Test::getValue()' discards qualifiers
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Category: C++

Comparison of static keyword in C++ and Java

Static keyword is used for almost same purpose in both C++ and Java. There are some differences though. This post covers similarities and differences of static keyword in C++ and Java.

Static Data Members: Like C++, static data members in Java are class members and shared among all objects. For example, in the following Java program, static variable *count* is used to count the number of objects created.

```
class Test {
   static int count = 0;

   Test() {
      count++;
   }
   public static void main(String arr[]) {
      Test t1 = new Test();
      Test t2 = new Test();
      System.out.println("Total " + count + " objects created");
   }
}
```

Output:

```
Total 2 objects created
```

Static Member Methods: Like C++, methods declared as static are class members and have following restrictions:

1) They can only call other static methods. For example, the following program fails in compilation. fun() is non-static and it is called in static main()

```
class Main {
   public static void main(String args[]) {
      System.out.println(fun());
   }
   int fun() {
      return 20;
   }
}
```

2) They must only access static data.

3) They cannot access *this* or *super* . For example, the following program fails in compilation.

```
class Base {
   static int x = 0;
}

class Derived extends Base
{
  public static void fun() {
     System.out.println(super.x); // Compiler Error: non-static variable
                                  // cannot be referenced from a static context
  }
}
```

Like C++, static data members and static methods can be accessed without creating an object. They can be accessed using class name. For example, in the following program, static data member count and static method fun() are accessed without any object.

```
class Test {
   static int count = 0;
   public static void fun() {
     System.out.println("Static fun() called");
   }
}

class Main
{
   public static void main(String arr[]) {
     System.out.println("Test.count = " + Test.count);
     Test.fun();
   }
}
```

Static Block: Unlike C++, Java supports a special block, called static block (also called static clause) which can be used for static initialization of a class. This code inside static block is executed only once. See Static blocks in Java for details.

Static Local Variables: Unlike C++, Java doesn't support static local variables. For example, the following Java program fails in compilation.

```
class Test {
  public static void main(String args[]) {
    System.out.println(fun());
  }
  static int fun()  {
    static int x= 10; //Compiler Error: Static local variables are not allowed
    return x--;
  }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Java
C++
Java

How does default virtual behavior differ in C++ and Java ?

Default virtual behavior of methods is opposite in C++ and Java:

In C++, class member methods are non-virtual by default. They can be made virtual by using *virtual* keyword. For example, *Base::show()* is non-virtual in following program and program prints *"Base::show() called"*.

```
#include<iostream>

using namespace std;

class Base {
public:

   // non-virtual by default
   void show() {
       cout<<"Base::show() called";
   }
};

class Derived: public Base {
public:
   void show() {
       cout<<"Derived::show() called";
   }
};

int main()
{
```

```
  Derived d;
  Base &b = d;
  b.show();
  getchar();
  return 0;
}
```

Adding *virtual* before definition of *Base::show()* makes program print *"Derived::show() called"*

In Java, methods are virtual by default and can be made non-virtual by using *final* keyword. For example, in the following java program, *show()* is by default virtual and the program prints *"Derived::show() called"*

```
class Base {

   // virtual by default
   public void show() {
     System.out.println("Base::show() called");
   }
}

class Derived extends Base {
   public void show() {
     System.out.println("Derived::show() called");
   }
}

public class Main {
   public static void main(String[] args) {
      Base b = new Derived();;
      b.show();
   }
}
```

Unlike C++ non-virtual behavior, if we add *final* before definition of show() in *Base* , then the above program fails in compilation.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Java
Java

Comparison of Exception Handling in C++ and Java

Both languages use *try*, *catch* and *throw* keywords for exception handling, and meaning of *try*, *catch* and *free* blocks is also same in both languages. Following are the differences between Java and C++ exception handling.

1) In C++, all types (including primitive and pointer) can be thrown as exception. But in Java only throwable objects (Throwable objects are instances of any subclass of the Throwable class) can be thrown as exception. For example, following type of code works in C++, but similar code doesn't work in Java.

```
#include <iostream>
using namespace std;
int main()
{
  int x = -1;

  // some other stuff
  try {
    // some other stuff
    if( x < 0 )
    {
      throw x;
    }
  }
  catch (int x ) {
    cout << "Exception occurred: thrown value is " << x << endl;
```

```
  }
  getchar();
  return 0;
}
```

Output:

Exception occurred: thrown value is -1

2) In C++, there is a special catch called “catch all” that can catch all kind of exceptions.

```
#include <iostream>
using namespace std;
int main()
{
  int x = -1;
  char *ptr;

  ptr = new char[256];

  // some other stuff
  try {
    // some other stuff
    if( x < 0 )
    {
      throw x;
    }
    if(ptr == NULL)
    {
      throw " ptr is NULL ";
    }
  }
  catch (...) // catch all
  {
    cout << "Exception occurred: exiting "<< endl;
    exit(0);
  }

  getchar();
  return 0;
}
```

Output:

Exception occurred: exiting

In Java, for all practical purposes, we can catch Exception object to catch all kind of exceptions. Because, normally we do not catch Throwable(s) other than Exception(s) (which are Errors)

```
catch(Exception e){
  .......
}
```

3) In Java, there is a block called *finally* that is always executed after the try-catch block. This block can be used to do cleanup work. There is no such block in C++.

```
// creating an exception type
class Test extends Exception { }

class Main {
  public static void main(String args[]) {

    try {
      throw new Test();
    }
    catch(Test t) {
      System.out.println("Got the Test Exception");
    }
    finally {
```

```
        System.out.println("Inside finally block ");
    }
  }
}
```

Output:
Got the error
Inside finally block

4) In C++, all exceptions are unchecked. In Java, there are two types of exceptions – checked and unchecked. See this for more details on checked vs Unchecked exceptions.

5) In Java, a new keyword *throws* is used to list exceptions that can be thrown by a function. In C++, there is no *throws* keyword, the same keyword *throw* is used for this purpose also.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
Java
cpp-exception
Java-Exceptions

Can we call an undeclared function in C++?

Calling an undeclared function is poor style in C (See this) and illegal in C++. So is passing arguments to a function using a declaration that doesn't list argument types:

If we save the below program in a .c file and compile it, it works without any error. But, if we save the same in a .cpp file, it doesn't compile.

```
#include<stdio.h>

void f(); /* Argument list is not mentioned */

int main()
{
  f(2); /* Poor style in C, invalid in C++*/
  getchar();
  return 0;
}

void f(int x)
{
  printf("%d", x);
}
```

Source: http://www2.research.att.com/~bs/bs_faq.html#C-is-subset

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Can we access global variable if there is a local variable with same name?

In C, we cannot access a global variable if we have a local variable with same name, but it is possible in C++ using scope resolution operator (::).

```
#include<iostream>

using namespace std;
```

```
int x;  // Global x

int main()
{
  int x = 10; // Local x
  cout<<"Value of global x is "<<::x<<endl;
  cout<<"Value of local x is "<<x;
  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect in the above GFact or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Can we use function on left side of an expression in C and C++?

In C, it might not be possible to have function names on left side of an expression, but it's possible in C++.

```
#include<iostream>

using namespace std;

/* such a function will not be safe if x is non static variable of it */
int &fun()
{
  static int x;
  return x;
}

int main()
{
  fun() = 10;

  /* this line prints 10 on screen */
  printf(" %d ", fun());

  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect or want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

Can we access private data members of a class without using a member or a friend function?

The idea of Encapsulation is to bundle data and methods (that work on the data) together and restrict access of private data members outside the class. In C++, a friend function or friend class can also access private data members.

Is it possible to access private members outside a class without friend?
Yes, it is possible using pointers. See the following program as an example.

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

```
class Test
{
private:
   int data;
public:
   Test() { data = 0; }
   int getData() { return data; }
};

int main()
{
   Test t;
   int* ptr = (int*)&t;
   *ptr = 10;
   cout << t.getData();
   return 0;
}
```

Output:

```
10
```

Note that the above way of accessing private data members is not at all a recommended way of accessing members and should never be used. Also, it doesn't mean that the encapsulation doesn't work in C++. The idea of making private members is to avoid accidental changes. The above change to data is not accidental. It's an intentionally written code to fool the compiler.

This article is contributed by **Ashish Kumar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-class

How to make a C++ class whose objects can only be dynamically allocated?

The problem is to create a class such that the non-dynamic allocation of object causes compiler error. For example, create a class 'Test' with following rules.

```
   Test t1;  // Should generate compiler error
   Test *t3 = new Test; // Should work fine
```

The idea is to create a private destructor in the class. When we make a private destructor, the compiler would generate a compiler error for non-dynamically allocated objects because compiler need to remove them from stack segment once they are not in use.
Since compiler is not responsible for deallocation of dynamically allocated objects (programmer should explicitly deallocate them), compiler won't have any problem with them. To avoid memory leak, we create a friend function *destructTest()* which can be called by users of class to destroy objects.

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

// A class whose object can only be dynamically created
class Test
{
private:
   ~Test() { cout << "Destroying Object\n"; }
public:
    Test() { cout << "Object Created\n"; }
friend void destructTest(Test* );
};

// Only this function can destruct objects of Test
void destructTest(Test* ptr)
{
```

```
    delete ptr;
    cout << "Object Destroyed\n";
}

int main()
{
    /* Uncommenting following following line would cause compiler error */
    // Test t1;

    // create an object
    Test *ptr = new Test;

    // destruct the object to avoid memory leak
    destructTest(ptr);

    return 0;
}
```

Output:

```
Object Created
Destroying Object
Object Destroyed
```

If we don't want to create a friend function, we can also overload delete and delete[] operators in Test, this way we don't have to call a specific function to delete dynamically allocated objects.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-constrcutor

How to print "GeeksforGeeks" with empty main() in C++?

Write a C++ program that prints "GeeksforGeeks" with empty main() function. You are not allowed to write anything in main().

One way of doing this is to apply constructor attribute to a function so that it executes before main() (See this for details).

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

/* Apply the constructor attribute to myStartupFun() so that it
   is executed before main() */
void myStartupFun (void) __attribute__ ((constructor));

/* implementation of myStartupFun */
void myStartupFun (void)
{
    cout << "GeeksforGeeks";
}

int main ()
{

}
```

Output:

```
GeeksforGeeks
```

The above approach would work only with GCC family of compilers. Following is an interesting way of doing it in C++.

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

class MyClass
```

```
{
public:
   MyClass()
   {
      cout << "GeeksforGeeks";
   }
}m;

int main()
{

}
```

Output:

```
GeeksforGeeks
```

The idea is to create a class, have a cout statement in constructor and create a global object of the class. When the object is created, constructor is called and “GeeksforGeeks” is printed.

This article is contributed by **Viki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
C++
CPP

Print 1 to 100 in C++, without loop and recursion

Following is a C++ program that prints 1 to 100 without loop and without recursion.

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

template<int N>
class PrintOneToN
{
public:
   static void print()
   {
      PrintOneToN<N-1>::print();  // Note that this is not recursion
      cout << N << endl;
   }
};

template<>
class PrintOneToN<1>
{
public:
   static void print()
   {
      cout << 1 << endl;
   }
};
int main()
{
   const int N = 100;
   PrintOneToN<N>::print();
   return 0;
}
```

Output:

```
1
2
3
```

```
..
..
98
99
100
```

The program prints all numbers from 1 to n without using a loop and recursion. The concept used in this program is Template Metaprogramming.
Let us see how this works. Templates in C++ allow non-datatypes also as parameter. Non-datatype means a value, not a datatype. For example, in the above program, N is passed as a value which is not a datatype. A new instance of a generic class is created for every parameter and these classes are created at compile time. In the above program, when compiler sees the statement "PrintOneToN>::print()" with N = 100, it creates an instance PrintOneToN. In function PrintOneToN::print(), another function PrintOneToN::print() is called, therefore an instance PrintOneToN is created. Similarly, all instances from PrintOneToN to PrintOneToN are created. PrintOneToN::print() is already there and prints 1. The function PrintOneToN prints 2 and so on. Therefore we get all numbers from 1 to N printed on the screen.

Following is **another approach** to print 1 to 100 without loop and recursion.

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

class A
{
public:
   static int a;
   A()
   { cout<<a++<<endl; }
};

int A::a = 1;

int main()
{
   int N = 100;
   A obj[N];
   return 0;
}
```

The output of this program is same as above program. In the above program, class A has a static variable 'a', which is incremented with every instance of A. The default constructor of class A prints the value of 'a'. When we create an array of objects of type A, the default constructor is called for all objects one by one. Value of 'a' is printed and incremented with every call. Therefore, we get all values from 1 to 100 printed on the scree.

Thanks to *Lakshmanan* for suggesting this approach.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-puzzle

How to restrict dynamic allocation of objects in C++?

C++ programming language allows both auto(or stack allocated) and dynamically allocated objects. In java & C#, all objects must be dynamically allocated using new.
C++ supports stack allocated objects for the reason of runtime efficiency. Stack based objects are implicitly managed by C++ compiler. They are destroyed when they go out of scope and dynamically allocated objects must be manually released, using delete operator otherwise memory leak occurs. C++ doesn't support automatic garbage collection approach used by languages such as Java & C#.

How do we achieve following behavior from a class 'Test' in C++?

```
Test* t = new Test; // should produce compile time error
Test t;   // OK
```

The idea of is to keep new operator function private so that new cannot be called. See the following program. Objects of 'Test' class cannot be created using new as new operator function is private in 'Test'. If we uncomment the 2nd line of main(), the program would produce compile time error.

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

// Objects of Test can not be dynamically allocated
class Test
{
   // Notice this, new operator function is private
   void* operator new(size_t size);
   int x;
public:
   Test()        { x = 9; cout << "Constructor is called\n"; }
   void display()  { cout << "x = " << x << "\n"; }
   ~Test()       { cout << "Destructor is executed\n"; }
};

int main()
{
   // Uncommenting following line would cause a compile time error.
   // Test* obj=new Test();
   Test t;        // Ok, object is allocated at compile time
   t.display();
   return 0;
} // object goes out of scope, destructor will be called
```

Output:

```
Constructor is called
x = 9
Destructor is executed
```

Reference:

Design and evolution of C++, Bjarne Stroustrup

This article is contributed by **Pravasi Meet**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles
cpp-pointer

Output of C++ Program | Set 1

Predict the output of below C++ programs.

Question 1

```
// Assume that integers take 4 bytes.
#include<iostream>

using namespace std;

class Test
{
  static int i;
  int j;
};

int Test::i;

int main()
{
   cout << sizeof(Test);
   return 0;
```

```
}
```

Output: 4 (size of integer)
static data members do not contribute in size of an object. So ‘i’ is not considered in size of Test. Also, all functions (static and non-static both) do not contribute in size.

Question 2

```
#include<iostream>

using namespace std;
class Base1 {
 public:
    Base1()
    { cout << " Base1's constructor called" << endl; }
};

class Base2 {
 public:
    Base2()
    { cout << "Base2's constructor called" << endl; }
};

class Derived: public Base1, public Base2 {
  public:
    Derived()
    {  cout << "Derived's constructor called" << endl; }
};

int main()
{
  Derived d;
  return 0;
}
```

Ouput:
Base1’s constructor called
Base2’s constructor called
Derived’s constructor called

In case of Multiple Inheritance, constructors of base classes are always called in derivation order from left to right and Destructors are called in reverse order.

GATE CS Corner Company Wise Coding Practice

Output
CPP-Output

Output of C++ Program | Set 2

Predict the output of below C++ programs.

Question 1

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

class A {
 public:
    A(int ii = 0) : i(ii) {}
    void show() { cout << "i = " << i << endl;}
 private:
    int i;
};

class B {
 public:
    B(int xx) : x(xx) {}
```

```
    operator A() const { return A(x); }
  private:
    int x;
};

void g(A a)
{  a.show(); }

int main() {
  B b(10);
  g(b);
  g(20);
  getchar();
  return 0;
}
```

Output:
i = 10
i = 20

Since there is a Conversion constructor in class A, integer value can be assigned to objects of class A and function call g(20) works. Also, there is a conversion operator overloaded in class B, so we can call g() with objects of class B.

Question 2

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

class base {
   int arr[10];
};

class b1: public base { };

class b2: public base { };

class derived: public b1, public b2 {};

int main(void)
{
  cout<<sizeof(derived);
  getchar();
  return 0;
}
```

Output: If integer takes 4 bytes, then 80.

Since *b1* and *b2* both inherit from class *base*, two copies of class *base* are there in class *derived*. This kind of inheritance without virtual causes wastage of space and ambiguities. virtual base classes are used to save space and avoid ambiguities in such cases. For example, following program prints 48. 8 extra bytes are for bookkeeping information stored by the compiler (See this for details)

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

class base {
  int arr[10];
};

class b1: virtual public base { };

class b2: virtual public base { };

class derived: public b1, public b2 {};

int main(void)
{
  cout<<sizeof(derived);
  getchar();
  return 0;
```

```
}
```

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above

GATE CS Corner Company Wise Coding Practice

Output
CPP-Output

Output of C++ Program | Set 3

Predict the output of below C++ programs.

Question 1

```
#include<iostream>

using namespace std;
class P {
public:
  void print()
  { cout <<" Inside P::"; }
};

class Q : public P {
public:
  void print()
  { cout <<" Inside Q"; }
};

class R: public Q {
};

int main(void)
{
 R r;

 r.print();
 return 0;
}
```

Output:
Inside Q

The print function is not defined in class R. So it is looked up in the inheritance hierarchy. *print()* is present in both classes *P* and *Q*, which of them should be called? The idea is, if there is multilevel inheritance, then function is linearly searched up in the inheritance heirarchy until a matching function is found.

Question 2

```
#include<iostream>
#include<stdio.h>

using namespace std;

class Base
{
public:
 Base()
 {
   fun(); //note: fun() is virtual
 }
 virtual void fun()
 {
   cout<<"\nBase Function";
 }
```

```
};

class Derived: public Base
{
public:
  Derived(){}
  virtual void fun()
  {
    cout<<"\nDerived Function";
  }
};

int main()
{
  Base* pBase = new Derived();
  delete pBase;
  return 0;
}
```

Output:
Base Function

See following excerpt from C++ standard for explanation.

When a virtual function is called directly or indirectly from a constructor (including from the mem-initializer for a data member) or from a destructor, and the object to which the call applies is the object under construction or destruction, the function called is the one defined in the constructor or destructor's own class or in one of its bases, but not a function overriding it in a class derived from the constructor or destructor's class, or overriding it in one of the other base classes of the most derived object.

Because of this difference in behavior, it is recommended that object's virtual function is not invoked while it is being constructed or destroyed. See this for more details.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

GATE CS Corner Company Wise Coding Practice

Output
CPP-Output

Output of C++ Program | Set 4

Difficulty Level: Rookie

Predict the output of below C++ programs.

Question 1

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

int x = 10;
void fun()
{
   int x = 2;
   {
      int x = 1;
      cout << ::x << endl;
   }
}

int main()
{
   fun();
   return 0;
}
```

Output: 10
If Scope Resolution Operator is placed before a variable name then the global variable is referenced. So if we remove the following line

from the above program then it will fail in compilation.

```
int x = 10;
```

Question 2

```
#include<iostream>
using namespace std;
class Point {
private:
   int x;
   int y;
public:
   Point(int i, int j);  // Constructor
};

Point::Point(int i = 0, int j = 0)  {
   x = i;
   y = j;
   cout << "Constructor called";
}

int main()
{
  Point t1, *t2;
  return 0;
}
```

Output: Constructor called.
If we take a closer look at the statement “Point t1, *t2;:” then we can see that only one object is constructed here. t2 is just a pointer variable, not an object.

Question 3

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

class Point {
private:
   int x;
   int y;
public:
   Point(int i = 0, int j = 0);    // Normal Constructor
   Point(const Point &t); // Copy Constructor
};

Point::Point(int i, int j)  {
   x = i;
   y = j;
   cout << "Normal Constructor called\n";
}

Point::Point(const Point &t) {
  y = t.y;
  cout << "Copy Constructor called\n";
}

int main()
{
  Point *t1, *t2;
  t1 = new Point(10, 15);
  t2 = new Point(*t1);
  Point t3 = *t1;
  Point t4;
  t4 = t3;
  return 0;
}
```

Output:
Normal Constructor called
Copy Constructor called
Copy Constructor called
Normal Constructor called

See following comments for explanation:

```
Point *t1, *t2;  // No constructor call
t1 = new Point(10, 15); // Normal constructor call
t2 = new Point(*t1);  // Copy constructor call
Point t3 = *t1; // Copy Constructor call
Point t4;  // Normal Constructor call
t4 = t3;  // Assignment operator call
```

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above

GATE CS Corner Company Wise Coding Practice

Output
CPP-Output

Output of C++ Program | Set 5

Difficulty Level: Rookie

Predict the output of below C++ programs.

Question 1

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

class Test {
   int value;
public:
   Test(int v);
};

Test::Test(int v) {
   value = v;
}

int main() {
   Test t[100];
   return 0;
}
```

Output: Compiler error

The class Test has one user defined constructor “Test(int v)” that expects one argument. It doesn’t have a constructor without any argument as the compiler doesn’t create the default constructor if user defines a constructor (See this). Following modified program works without any error.

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

class Test {
   int value;
public:
   Test(int v = 0);
};

Test::Test(int v) {
   value = v;
}
```

```
int main() {
   Test t[100];
   return 0;
}
```

Question 2

```
#include<iostream>
using namespace std;
int &fun() {
  static int a = 10;
  return a;
}

int main() {
  int &y = fun();
  y = y +30;
  cout<<fun();
  return 0;
}
```

Output: 40
The program works fine because ‘a’ is static. Since ‘a’ is static, memory location of it remains valid even after fun() returns. So a reference to static variable can be returned.

Question 3

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

class Test
{
public:
  Test();
};

Test::Test()  {
   cout<<"Constructor Called \n";
}

int main()
{
   cout<<"Start \n";
   Test t1();
   cout<<"End \n";
   return 0;
}
```

Output:
Start
End

Note that the line “Test t1();” is not a constructor call. Compiler considers this line as declaration of function t1 that doesn’t recieve any parameter and returns object of type Test.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above

GATE CS Corner Company Wise Coding Practice

Output
CPP-Output

Output of C++ Program | Set 6

Predict the output of below C++ programs.

Question 1

```
#include<iostream>

using namespace std;

class Test {
   int value;
public:
   Test (int v = 0) {value = v;}
   int getValue() { return value; }
};

int main() {
   const Test t;
   cout << t.getValue();
   return 0;
}
```

Output: Compiler Error.

A const object cannot call a non-const function. The above code can be fixed by either making getValue() const or making t non-const. Following is modified program with getValue() as const, it works fine and prints 0.

```
#include<iostream>

using namespace std;

class Test {
   int value;
public:
   Test (int v = 0) { value = v; }
   int getValue() const { return value; }
};

int main() {
   const Test t;
   cout << t.getValue();
   return 0;
}
```

Question 2

```
#include<iostream>

using namespace std;

class Test {
   int &t;
public:
   Test (int &x) { t = x; }
   int getT() { return t; }
};

int main()
{
   int x = 20;
   Test t1(x);
   cout << t1.getT() << " ";
   x = 30;
   cout << t1.getT() << endl;
   return 0;
}
```

Output: Compiler Error

Since t is a reference in Test, it must be initialized using Initializer List. Following is the modified program. It works and prints “20 30”.

```
#include<iostream>

using namespace std;

class Test {
   int &t;
public:
   Test (int &x):t(x) {  }
   int getT() { return t; }
};

int main() {
   int x = 20;
   Test t1(x);
   cout << t1.getT() << " ";
   x = 30;
   cout << t1.getT() << endl;
   return 0;
}
```

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above

GATE CS Corner Company Wise Coding Practice

Output
CPP-Output

Output of C++ Program | Set 7

Predict the output of following C++ programs.

Question 1

```
class Test1 {
   int y;
};

class Test2 {
   int x;
   Test1 t1;
public:
   operator Test1() { return t1; }
   operator int() { return x; }
};

void fun ( int x)  { };
void fun ( Test1 t ) { };

int main() {
   Test2 t;
   fun(t);
   return 0;
}
```

Output: Compiler Error
There are two conversion operators defined in the Test2 class. So Test2 objects can automatically be converted to both int and Test1. Therefore, the function call fun(t) is ambiguous as there are two functions void fun(int) and void fun(Test1), compiler has no way to decide which function to call. In general, conversion operators must be overloaded carefully as they may lead to ambiguity.

Question 2

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

class X {
private:
  static const int a = 76;
```

```
public:
  static int getA() { return a; }
};

int main() {
  cout <<X::getA()<<endl;
  return 0;
}
```

Output: The program compiles and prints 76
Generally, it is not allowed to initialize data members in C++ class declaration, but static const integral members are treated differently and can be initialized with declaration.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

GATE CS Corner Company Wise Coding Practice

Output
CPP-Output

Output of C++ Program | Set 8

Predict the output of following C++ programs.

Question 1

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

class Test1
{
   int x;
public:
   void show() { }
};

class Test2
{
   int x;
public:
   virtual void show() { }
};

int main(void)
{
   cout<<sizeof(Test1)<<endl;
   cout<<sizeof(Test2)<<endl;
   return 0;
}
```

Output:
4
8
There is only one difference between Test1 and Test2. show() is non-virtual in Test1, but virtual in Test2. When we make a function virtual, compiler adds an extra pointer vptr to objects of the class. Compiler does this to achieve run time polymorphism (See chapter 15 of Thinking in C++ book for more details). The extra pointer vptr adds to the size of objects, that is why we get 8 as size of Test2.

Question 2

```
#include<iostream>
using namespace std;
class P
{
public:
   virtual void show() = 0;
```

```
};

class Q : public P {
  int x;
};

int main(void)
{
   Q q;
   return 0;
}
```

Output: Compiler Error
We get the error because we can't create objects of abstract classes. P is an abstract class as it has a pure virtual method. Class Q also becomes abstract because it is derived from P and it doesn't implement show().

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

GATE CS Corner Company Wise Coding Practice

Output
CPP-Output

Output of C++ Program | Set 9

Predict the output of following C++ programs.

Question 1

```
template <class S, class T> class Pair
{
private:
   S x;
   T y;
/* ... */
};

template <class S> class Element
{
private:
   S x;
/* ... */
};

int main ()
{
   Pair <Element<int>, Element<char>> p;
   return 0;
}
```

Output:

```
Compiler Error: '>>' should be '> >' within a nested template argument list
```

When we use nested templates in our program, we must put a space between two closing angular brackets, otherwise it conflicts with operator >>. For example, following program compiles fine.

```
template <class S, class T> class Pair
{
private:
   S x;
   T y;
/* ... */
};

template <class S> class Element
```

```
{
private:
   S x;
/* ... */
};

int main ()
{
   Pair <Element<int>, Element<char> > p;   // note the space between '>' and '>'
   return 0;
}
```

Question 2

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

class Test
{
private:
   static int count;
public:
   static Test& fun();
};

int Test::count = 0;

Test& Test::fun()
{
   Test::count++;
   cout<<Test::count<<" ";
   return *this;
}

int main()
{
   Test t;
   t.fun().fun().fun().fun();
   return 0;
}
```

Output:

```
Compiler Error: 'this' is unavailable for static member functions
```

this pointer is not available to static member methods in C++, as static methods can be called using class name also. Similarly in Java, static member methods cannot access this and super (super is for base or parent class).
If we make fun() non-static in the above program, then the program works fine.

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

class Test
{
private:
   static int count;
public:
   Test& fun(); // fun() is non-static now
};

int Test::count = 0;

Test& Test::fun()
{
   Test::count++;
   cout<<Test::count<<" ";
   return *this;
}
```

```
int main()
{
   Test t;
   t.fun().fun().fun().fun();
   return 0;
}
```

Output:

```
Output:
1 2 3 4
```

Please write comments if you find any of the answers/explanations incorrect, or want to share more information about the topics discussed above.

GATE CS Corner Company Wise Coding Practice

Output
CPP-Output

Output of C++ Program | Set 10

Predict the output of following C++ programs.

Question 1

```
#include<iostream>
#include<string.h>
using namespace std;

class String
{
   char *p;
   int len;
public:
   String(const char *a);
};

String::String(const char *a)
{
   int length = strlen(a);
   p = new char[length +1];
   strcpy(p, a);
   cout << "Constructor Called " << endl;
}

int main()
{
   String s1("Geeks");
   const char *name = "forGeeks";
   s1 = name;
   return 0;
}
```

Output:

```
Constructor called
Constructor called
```

The first line of output is printed by statement “String s1(“Geeks”);” and the second line is printed by statement “s1 = name;”. The reason for the second call is, a single parameter constructor also works as a conversion operator (See this and this for deatils).

Question 2

```
#include<iostream>
```

```
using namespace std;

class A
{
   public:
   virtual void fun() {cout << "A" << endl ;}
};
class B: public A
{
   public:
   virtual void fun() {cout << "B" << endl;}
};
class C: public B
{
   public:
   virtual void fun() {cout << "C" << endl;}
};

int main()
{
   A *a = new C;
   A *b = new B;
   a->fun();
   b->fun();
   return 0;
}
```

Output:

```
C
B
```

A base class pointer can point to objects of children classes. A base class pointer can also point to objects of grandchildren classes. Therefor, the line “A *a = new C;” is valid. The line “a->fun();” prints “C” because the object pointed is of class C and fun() is declared virtual in both A and B (See this for details). The second line of output is printed by statement “b->fun();”.

Please write comments if you find any of the answers/explanations incorrect, or want to share more information about the topics discussed above.

GATE CS Corner Company Wise Coding Practice

Output
C++
CPP-Output

Output of C++ Program | Set 11

Predict the output of following C++ programs.

Question 1

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

class Point
{
private:
   int x;
   int y;
public:
   Point(const Point&p) { x = p.x; y = p.y; }
   void setX(int i) {x = i;}
   void setY(int j) {y = j;}
   int getX() {return x;}
   int getY() {return y;}
   void print() { cout << "x = " << getX() << ", y = " << getY(); }
};
```

```
int main()
{
   Point p1;
   p1.setX(10);
   p1.setY(20);
   Point p2 = p1;
   p2.print();
   return 0;
}
```

Output: Compiler Error in first line of main(), i.e., “Point p1;”

Since there is a user defined constructor, compiler doesn’t create the default constructor (See this GFact). If we remove the copy constructor from class Point, the program works fine and prints the output as “x = 10, y = 20”

Question 2

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

int main()
{
   int *ptr = new int(5);
   cout << *ptr;
   return 0;
}
```

Output: 5
The new operator can also initialize primitive data types. In the above program, the value at address ‘ptr ‘ is initialized as 5 using the new operator.

Question 3

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

class Fraction
{
private:
   int den;
   int num;
public:
  void print() { cout << num << "/" << den; }
  Fraction() { num = 1; den = 1; }
  int &Den() { return den; }
  int &Num() { return num; }
};

int main()
{
  Fraction f1;
  f1.Num() = 7;
  f1.Den() = 9;
  f1.print();
  return 0;
}
```

Output: 7/9
The methods Num() and Den() return references to num and den respectively. Since references are returned, the returned values can be uses as an lvalue, and the private members den and num are modified. The program compiles and runs fine, but this kind of class design is strongly discouraged (See this). Returning reference to private variable allows users of the class to change private data directly which defeats the purpose of encapsulation.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

Output of C++ Program | Set 12

Predict the output of following C++ programs.

Question 1

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

int fun(int a, int b  = 1, int c =2)
{
   return (a + b + c);
}

int main()
{
   cout << fun(12, ,2);
   return 0;
}
```

Output: Compiler Error in function call fun(12, ,2)

With default arguments, we cannot skip an argument in the middle. Once an argument is skipped, all the following arguments must be skipped. The calls fun(12) and fun(12, 2) are valid.

Question 2

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

/* local variable is same as a member's name */
class Test
{
private:
   int x;
public:
   void setX (int x) { Test::x = x; }
   void print() { cout << "x = " << x << endl; }
};

int main()
{
   Test obj;
   int x = 40;
   obj.setX(x);
   obj.print();
   return 0;
}
```

Output:

```
x = 40
```

Scope resolution operator can always be used to access a class member when it is made hidden by local variables. So the line “Test::x = x” is same as “this->x = x”

Question 3

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

class Test
```

```
{
private:
   int x;
   static int count;
public:
   Test(int i = 0) : x(i) {}
   Test(const Test& rhs) : x(rhs.x) { ++count; }
   static int getCount() { return count; }
};

int Test::count = 0;

Test fun()
{
   return Test();
}

int main()
{
   Test a = fun();
   cout<< Test::getCount();
   return 0;
}
```

Output: Compiler Dependent
The line “Test a = fun()” may or may not call copy constructor. So output may be 0 or 1. If copy elision happens in your compiler, the copy constructor will not be called. If copy elision doesn’t happen, copy constructor will be called. The gcc compiler produced the output as 0.

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

GATE CS Corner Company Wise Coding Practice

Output
CPP-Output

Output of C++ Program | Set 13

Predict the output of following C++ program.

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

class A
{
   // data members of A
public:
   A ()         { cout << "\n A's constructor"; /* Initialize data members */ }
   A (const A &a) { cout << "\n A's Copy constructor";  /* copy data members */}
   A& operator= (const A &a) // Assignemt Operator
   {
      // Handle self-assignment:
      if(this == &a) return *this;

      // Copy data members
      cout << "\n A's Assignment Operator";  return *this;
   }
};

class B
{
   A a;
   // Other members of B
public:
   B(A &a) { this->a = a; cout << "\n B's constructor"; }
};

int main()
{
```

```
    A a1;
    B b(a1);
    return 0;
}
```

Output:

```
A's constructor
A's constructor
A's Assignment Operator
B's constructor
```

The first line of output is printed by the statement “A a1;” in main().
The second line is printed when B’s member ‘a’ is initialized. This is important.
The third line is printed by the statement “this->a = a;” in B’s constructor.
The fourth line is printed by cout statement in B’s constructor.

If we take a look a closer look at the above code, the constructor of class B is not efficient as member ‘a’ is first constructed with default constructor, and then the values from the parameter are copied using assignment operator. It may be a concern when class A is big, which generally is the case with many practical classes. See the following optimized code.

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

class A
{
    // data members of A
public:
    A()         { cout << "\n A's constructor"; /* Initialize data members */ }
    A(const A &a) { cout << "\n A's Copy constructor"; /* Copy data members */ }
    A& operator= (const A &a) // Assignemt Operator
    {
        // Handle self-assignment:
        if(this == &a) return *this;

        // Copy data members
        cout << "\n A's Assignment Operator";  return *this;
    }
};

class B
{
    A a;
    // Other members of B
public:
    B(A &a):a(a) {  cout << "\n B's constructor"; }
};

int main()
{
    A a;
    B b(a);
    return 0;
}
```

Output:

```
A's constructor
A's Copy constructor
B's constructor
```

The constructor of class B now uses initializer list to initialize its member ‘a’. When Initializer list is used, the member ‘a’ of class B is initialized directly from the parameter. So a call to A’s constructor is reduced.
In general, it is a good idea to use Initializer List to initialize all members of a class, because it saves one extra assignment of members. See point 6 of this post for more details.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Output of C++ Program | Set 14

Predict the output of following C++ program.

Difficulty Level: Rookie

Question 1

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

class A
{
   int id;
public:
   A (int i) { id = i; }
   void print () { cout << id << endl; }
};

int main()
{
   A a[2];
   a[0].print();
   a[1].print();
   return 0;
}
```

There is a compilation error in line “A a[2]”. There is no default constructor in class A. When we write our own parameterzied constructor or copy constructor, compiler doesn’t create the default constructor (See this Gfact). We can fix the error, either by creating a default constructor in class A, or by using the following syntax to initialize array member using parameterzied constructor.

```
// Initialize a[0] with value 10 and a[1] with 20
A a[2] = { A(10),  A(20) }
```

Question 2

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

class A
{
   int id;
   static int count;
public:
   A()
   {
      count++;
      id = count;
      cout << "constructor called " << id << endl;
   }
   ~A()
   {
      cout << "destructor called " << id << endl;
   }
};

int A::count = 0;

int main()
{
   A a[2];
   return 0;
}
```

Output:

```
constructor called 1
constructor called 2
destructor called 2
destructor called 1
```

In the above program, object a[0] is created first, but the object a[1] is destroyed first. Objects are always destroyed in reverse order of their creation. The reason for reverse order is, an object created later may use the previously created object. For example, consider the following code snippet.

```
A a;
B b(a);
```

In the above code, the object ‘b’ (which is created after ‘a’), may use some members of ‘a’ internally. So destruction of ‘a’ before ‘b’ may create problems. Therefore, object ‘b’ must be destroyed before ‘a’.

Question 3

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

class A
{
  int aid;
public:
  A(int x)
  { aid = x; }
  void print()
  { cout << "A::aid = " <<aid; }
};

class B
{
   int bid;
public:
   static A a;
   B (int i) { bid = i; }
};

int main()
{
 B b(10);
 b.a.print();
 return 0;
}
```

Compiler Error: undefined reference to `B::a’
The class B has a static member ‘a’. Since member ‘a’ is static, it must be defined outside the class. Class A doesn’t have Default constructor, so we must pass a value in definition also. Adding a line “A B::a(10);” will make the program work.

The following program works fine and produces the output as “A::aid = 10”

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

class A
{
  int aid;
public:
  A(int x)
  { aid = x; }
  void print()
  { cout << "A::aid = " <<aid; }
};

class B
```

```
{
   int bid;
public:
   static A a;
   B (int i) { bid = i; }
};

A B::a(10);

int main()
{
  B b(10);
  b.a.print();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Output
C++
CPP-Output

Output of C++ Program | Set 15

Predict the output of following C++ programs.

Question 1

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

class A
{
public:
   void print() { cout << "A::print()"; }
};

class B : private A
{
public:
   void print() { cout << "B::print()"; }
};

class C : public B
{
public:
   void print() { A::print(); }
};

int main()
{
   C b;
   b.print();
}
```

Output: Compiler Error: 'A' is not an accessible base of 'C'
There is multilevel inheritance in the above code. Note the access specifier in "class B : private A". Since private access specifier is used, all members of 'A' become private in 'B'. Class 'C' is a inherited class of 'B'. An inherited class can not access private data members of the parent class, but print() of 'C' tries to access private member, that is why we get the error.

Question 2

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

```
class base
{
public:
   virtual void show()  { cout<<" In Base \n"; }
};

class derived: public base
{
   int x;
public:
   void show() { cout<<"In derived \n"; }
   derived()   { x = 10; }
   int getX() const { return x;}
};

int main()
{
   derived d;
   base *bp = &d;
   bp->show();
   cout << bp->getX();
   return 0;
}
```

Output: Compiler Error: 'class base' has no member named 'getX'
In the above program, there is pointer 'bp' of type 'base' which points to an object of type derived. The call of show() through 'bp' is fine because 'show()' is present in base class. In fact, it calls the derived class 'show()' because 'show()' is virtual in base class. But the call to 'getX()' is invalid, because getX() is not present in base class. When a base class pointer points to a derived class object, it can access only those methods of derived class which are present in base class and are virtual.

Question 3

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

class Test
{
   int value;
public:
   Test(int v = 0) { value = v; }
   int getValue()  { return value; }
};

int main()
{
   const Test t;
   cout << t.getValue();
   return 0;
}
```

Output: Compiler Error
In the above program, object 't' is declared as a const object. A const object can only call const functions. To fix the error, we must make getValue() a const function.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Output
CPP-Output

Output of C++ Program | Set 16

Predict the output of following C++ programs.

Question 1

```
#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(char x)   { cout << "Derived::fun(char ) called"; }
};

int main()
{
   Derived d;
   d.fun();
   return 0;
}
```

Output: Compiler Error.
In the above program, fun() of base class is not accessible in the derived class. If a derived class creates a member method with name same as one of the methods in base class, then all the base class methods with this name become hidden in derived class (See this for more details)

Question 2

```
#include<iostream>
using namespace std;
class Base
{
  protected:
     int x;
  public:
     Base (int i){ x = i;}
};

class Derived : public Base
{
  public:
     Derived (int i):x(i) { }
     void print() { cout << x ; }
};

int main()
{
   Derived d(10);
   d.print();
}
```

Output: Compiler Error
In the above program, x is protected, so it is accessible in derived class. Derived class constructor tries to use initializer list to directly initialize x, which is not allowed even if x is accessible. The members of base class can only be initialized through a constructor call of base class. Following is the corrected program.

```
#include<iostream>
using namespace std;
class Base {
  protected:
     int x;
  public:
     Base (int i){ x = i;}
};

class Derived : public Base {
```

```
  public:
    Derived (int i):Base(i) { }
    void print() { cout << x; }
};

int main()
{
  Derived d(10);
  d.print();
}
```

Output:

```
10
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Output
C++
CPP-Output

Output of C++ Program | Set 17

Predict the output of following C++ programs.

Question 1

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

class A
{
  public:
  A& operator=(const A&a)
  {
    cout << "A's assignment operator called" << endl;
    return *this;
  }
};

class B
{
  A a[2];
};

int main()
{
  B b1, b2;
  b1 = b2;
  return 0;
}
```

Output:

```
A's assignment operator called
A's assignment operator called
```

The class B doesn't have user defined assignment operator. If we don't write our own assignment operator, compiler creates a default assignment operator. The default assignment operator one by one copies all members of right side object to left side object. The class B has 2 members of class A. They both are copied in statement "b1 = b2", that is why there are two assignment operator calls.

Question 2

```
#include<stdlib.h>
#include<iostream>
```

```
using namespace std;

class Test {
public:
   void* operator new(size_t size);
   void operator delete(void*);
   Test() { cout<<"\n Constructor called"; }
   ~Test() { cout<<"\n Destructor called"; }
};

void* Test::operator new(size_t size)
{
   cout<<"\n new called";
   void *storage = malloc(size);
   return storage;
}

void Test::operator delete(void *p )
{
   cout<<"\n delete called";
   free(p);
}

int main()
{
   Test *m = new Test();
   delete m;
   return 0;
}
```

```
new called
Constructor called
Destructor called
delete called
```

Let us see what happens when below statement is executed.

```
   Test *x = new Test;
```

When we use new keyword to dynamically allocate memory, two things happen: memory allocation and constructor call. The memory allocation happens with the help of operator new. In the above program, there is a user defined operator new, so first user defined operator new is called, then constructor is called.
The process of destruction is opposite. First, destructor is called, then memory is deallocated.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Output
CPP-Output

Output of C++ Program | Set 18

Predict the output of following C++ programs.

Question 1

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

template <int N>
class A {
  int arr[N];
public:
  virtual void fun() { cout << "A::fun()"; }
};
```

```
class B : public A<2> {
public:
  void fun() { cout << "B::fun()"; }
};

class C : public B { };

int main() {
  A<2> *a = new C;
  a->fun();
  return 0;
}
```

Output:

```
B::fun()
```

In general, the purpose of using templates in C++ is to avoid code redundancy. We create generic classes (or functions) that can be used for any datatype as long as logic is identical. Datatype becomes a parameter and an instance of class/function is created at compile time when a data type is passed. C++ Templates also allow nontype (a parameter that represents a value, not a datatype) things as parameters. In the above program, there is a generic class A which takes a nontype parameter N. The class B inherits from an instance of generic class A. The value of N for this instance of A is 2. The class B overrides fun() of class A. The class C inherits from B. In main(), there is a pointer 'a' of type A that points to an instance of C. When 'a->fun()' is called, the function of class B is executed because fun() is virtual and virtual functions are called according to the actual object, not according to pointer. In class C, there is no function 'fun()', so it is looked up in the hierarchy and found in class B.

Question 2

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

template <int i>
int fun()
{
  i =20;
}

int main() {
  fun<4>();
  return 0;
}
```

Output:

```
Compiler Error
```

The value of nontype parameters must be constant as they are used at compile time to create instance of classes/functions. In the above program, templated fun() receives a nontype parameter and tries to modify it which is not possible. Therefore, compiler error.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Output
C++
CPP-Output

Question 1
What is the return value of f(p, p) if the value of p is initialized to 5 before the call? Note that the first parameter is passed by reference, whereas the second parameter is passed by value.

```
int f(int &x, int c) {
  c  = c - 1;
  if (c == 0) return 1;
```

```
  x = x + 1;
  return f(x, c) * x;
}
```

A3024
B6561
C55440
D161051

References

Discuss it

Question 1 Explanation:
Since c is passed by value and x is passed by reference, all functions will have same copy of x, but different copies of c. f(5, 5) = f(x, 4)*x = f(x, 3)*x*x = f(x, 2)*x*x*x = f(x, 1)*x*x*x*x = 1*x*x*x*x = x^4 Since x is incremented in every function call, it becomes 9 after f(x, 2) call. So the value of expression x^4 becomes 9^4 which is 6561. 1
Question 2
Which of the following is FALSE about references in C++
AReferences cannot be NULL
BA reference must be initialized when declared
COnce a reference is created, it cannot be later made to reference another object; it cannot be reset.
DReferences cannot refer to constant value

References

Discuss it

Question 2 Explanation:
We can create a constant reference that refers to a constant. For example, the following program compiles and runs fine.

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

int main()
{
  const int x = 10;
  const int& ref = x;

  cout << ref;
  return 0;
}
```

Question 3
Which of the following functions must use reference.
AAssignment operator function
BCopy Constructor
CDestructor
DParameterized constructor

References

Discuss it

Question 3 Explanation:
A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn't allow parameters to be pass by value. See http://geeksquiz.com/copy-constructor-in-cpp/ for details.
Question 4
Predict the output of following C++ program.

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

int &fun()
{
   static int x = 10;
   return x;
}
int main()
{
   fun() = 30;
   cout << fun();
   return 0;
}
```

ACompiler Error: Function cannot be used as lvalue
B10
C30

References

Discuss it

Question 4 Explanation:

When a function returns by reference, it can be used as lvalue. Since x is a static variable, it is shared among function calls and the initialization line "static int x = 10;" is executed only once. The function call fun() = 30, modifies x to 30. The next call "cout

Question 5

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

int &fun()
{
   int x = 10;
   return x;
}
int main()
{
   fun() = 30;
   cout << fun();
   return 0;
}
```

ACompiler Error: Function cannot be used as lvalue

B10

C30

D0

References

Discuss it

Question 5 Explanation:

When a function returns by reference, it can be used as lvalue. Since x is a local variable, every call to fun() will have use memory for x and call “fun() = 30" will not effect on next call.

Question 6

Output of following C++ program?

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

int main()
{
  int x = 10;
  int& ref = x;
  ref = 20;
  cout << "x = " << x << endl ;
  x = 30;
  cout << "ref = " << ref << endl;
  return 0;
}
```

A
```
x = 20
ref = 30
```

B
```
x = 20
ref = 20
```

C
```
x = 10
ref = 30
```

D
```
x = 30
ref = 30
```

References

Discuss it

Question 6 Explanation:

ref is an alias of x, so if we change either of them, we can see the change in other as well.

There are 6 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Load Comments

Question 1
What is the difference between struct and class in C++?
AAll members of a structure are public and structures don't have constructors and destructors
BMembers of a class are private by default and members of struct are public by default. When deriving a struct from a class/struct, default access-specifier for a base class/struct is public and when deriving a class, default access specifier is private.
CAll members of a structure are public and structures don't have virtual functions
DAll of the above
Class and Object
Discuss it
Question 1 Explanation:
See Structure vs class in C++
Question 2
Predict the output of following C++ program

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

class Empty {};

int main()
{
  cout << sizeof(Empty);
  return 0;
}
```

AA non-zero value
B0
CCompiler Error
DRuntime Error
Class and Object
Discuss it
Question 2 Explanation:
See Why is the size of an empty class not zero in C++
Question 3

```
class Test {
    int x;
};
int main()
{
  Test t;
  cout << t.x;
  return 0;
}
```

A0
BGarbage Value
CCompiler Error
Class and Object
Discuss it
Question 3 Explanation:
In C++, the default access is private. Since x is a private member of Test, it is compiler error to access it outside the class.
Question 4
Which of the following is true?
AAll objects of a class share all data members of class
BObjects of a class do not share non-static members. Every object has its own copy.
CObjects of a class do not share codes of non-static methods, they have their own copy
DNone of the above
Class and Object

Discuss it

Question 4 Explanation:

Every object maintains a copy of non-static data members. For example, let Student be a class with data members as name, year, batch. Every object of student will have its own name, year and batch. On a side note, static data members are shared among objects. All objects share codes of all methods. For example, every student object uses same logic to find out grades or any other method.

Question 5

Assume that an integer and a pointer each takes 4 bytes. Also, assume that there is no alignment in objects. Predict the output following program.

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

class Test
{
   static int x;
   int *ptr;
   int y;
};

int main()
{
   Test t;
   cout << sizeof(t) << " ";
   cout << sizeof(Test *);
}
```

A12 4

B12 12

C8 4

D8 8

Class and Object

Discuss it

Question 5 Explanation:

For a compiler where pointers take 4 bytes, the statement "sizeof(Test *)" returns 4 (size of the pointer ptr). The statement "sizeof(t)" returns 8. Since static is not associated with each object of the class, we get (8 not 12).

Question 6

Which of the following is true about the following program

```
#include <iostream>
class Test
{
public:
   int i;
   void get();
};
void Test::get()
{
   std::cout << "Enter the value of i: ";
   std::cin >> i;
}
Test t;  // Global object
int main()
{
   Test t;  // local object
   t.get();
   std::cout << "value of i in local t: "<<t.i<<'\n';
   ::t.get();
   std::cout << "value of i in global t: "<<::t.i<<'\n';
   return 0;
}
```

Contributed by **Pravasi Meet**

ACompiler Error: Cannot have two objects with same class name

BCompiler Error in Line "::t.get();"

CCompiles and runs fine

Class and Object

Discuss it

Question 6 Explanation:

The above program compiles & runs fine. Like variables it is possible to create 2 objects having same name & in different scope.

There are 6 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Which of the followings is/are automatically added to every class, if we do not write our own.
ACopy Constructor
BAssignment Operator
CA constructor without any parameter
DAll of the above
Constructors
Discuss it
Question 1 Explanation:
In C++, if we do not write our own, then compiler automatically creates a default constructor, a copy constructor and a assignment operator for every class.
Question 2
When a copy constructor may be called?
AWhen an object of the class is returned by value.
BWhen an object of the class is passed (to a function) by value as an argument.
CWhen an object is constructed based on another object of the same class
DWhen compiler generates a temporary object.
EAll of the above
Constructors
Discuss it
Question 2 Explanation:
See When is copy constructor called?
Question 3
Output of following program?

```
#include<iostream>
using namespace std;
class Point {
    Point() { cout << "Constructor called"; }
};

int main()
{
  Point t1;
  return 0;
}
```

ACompiler Error
BRuntime Error
CConstructor called
Constructors
Discuss it
Question 3 Explanation:
By default all members of a class are private. Since no access specifier is there for Point(), it becomes private and it is called outside the class when t1 is constructed in main.
Question 4

```
#include<iostream>
using namespace std;
class Point {
public:
    Point() { cout << "Constructor called"; }
};

int main()
{
  Point t1, *t2;
  return 0;
}
```

ACompiler Error
BConstructor called

Constructor called
CConstructor called
Constructors
Discuss it
Question 4 Explanation:
Only one object t1 is constructed here. t2 is just a pointer variable, not an object
Question 5
Output of following program?

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

class Point {
public:
    Point() { cout << "Normal Constructor called\n"; }
    Point(const Point &t) { cout << "Copy constructor called\n"; }
};

int main()
{
   Point *t1, *t2;
   t1 = new Point();
   t2 = new Point(*t1);
   Point t3 = *t1;
   Point t4;
   t4 = t3;
   return 0;
}
```

ANormal Constructor called
Normal Constructor called
Normal Constructor called
Copy Constructor called
Copy Constructor called
Normal Constructor called
Copy Constructor called
BNormal Constructor called
Copy Constructor called
Copy Constructor called
Normal Constructor called
Copy Constructor called
CNormal Constructor called
Copy Constructor called
Copy Constructor called
Normal Constructor called
Constructors
Discuss it
Question 5 Explanation:
See following comments for explanation:

```
Point *t1, *t2;   // No constructor call
t1 = new Point(10, 15);  // Normal constructor call
t2 = new Point(*t1);   // Copy constructor call
Point t3 = *t1;  // Copy Constructor call
Point t4;   // Normal Constructor call
t4 = t3;   // Assignment operator call
```

Question 6

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

class X
{
public:
    int x;
};

int main()
{
    X a = {10};
```

```
    X b = a;
    cout << a.x << " " << b.x;
    return 0;
}
```

ACompiler Error
B10 followed by Garbage Value
C10 10
D10 0
Constructors
Discuss it
Question 6 Explanation:
The following may look like an error, but it works fine. X a = {10}; Like structures, class objects can be initialized. The line "X b = a;" calls copy constructor and is same as "X b(a);". Please note that, if we don't write our own copy constructor, then compiler creates a default copy constructor which assigns data members one object to other object.
Question 7
What is the output of following program?

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

class Point
{
    int x, y;
public:
   Point(const Point &p) { x = p.x; y = p.y; }
   int getX() { return x; }
   int getY() { return y; }
};

int main()
{
    Point p1;
    Point p2 = p1;
    cout << "x = " << p2.getX() << " y = " << p2.getY();
    return 0;
}
```

Ax = garbage value y = garbage value
Bx = 0 y = 0
CCompiler Error
Constructors
Discuss it
Question 7 Explanation:
There is compiler error in line "Point p1;". The class Point doesn't have a constructor without any parameter. If we write any constructor, then compiler doesn't create the default constructor. It is not true other way, i.e., if we write a default or parameterized constructor, then compiler creates a copy constructor. See the next question.
Question 8

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

class Point
{
    int x, y;
public:
   Point(int i = 0, int j = 0) { x = i; y = j; }
   int getX() { return x; }
   int getY() { return y; }
};

int main()
{
    Point p1;
    Point p2 = p1;
    cout << "x = " << p2.getX() << " y = " << p2.getY();
    return 0;
}
```

ACompiler Error
Bx = 0 y = 0
Cx = garbage value y = garbage value
Constructors
Discuss it
Question 8 Explanation:
Compiler creates a copy constructor if we don't write our own. Compiler writes it even if we have written other constructors in class. So the above program works fine. Since we have default arguments, the values assigned to x and y are 0 and 0.
Question 9
Predict the output of following program.

```
#include<iostream>
#include<stdlib.h>
using namespace std;

class Test
{
public:
  Test()
  { cout << "Constructor called"; }
};

int main()
{
   Test *t = (Test *) malloc(sizeof(Test));
   return 0;
}
```

AConstructor called
BEmpty
CCompiler Error
DRuntime error
Constructors
Discuss it
Question 9 Explanation:
Unlike new, malloc() doesn't call constructor (See this) If we replace malloc() with new, the constructor is called, see this.
Question 10

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

class Test
{
public:
     Test() { cout << "Hello from Test() "; }
} a;

int main()
{
   cout << "Main Started ";
   return 0;
}
```

AMain Started
BMain Started Hello from Test()
CHello from Test() Main Started
DCompiler Error: Global objects are not allowed
Constructors
Discuss it
Question 10 Explanation:
Output is

```
Hello from Test() Main Started
```

There is a global object 'a' which is constructed before the main functions starts, so the constructor for a is called first, then main()' execution begins.
Question 11
Output?

```
#include<iostream>
#include<string.h>
using namespace std;
```

```
class String
{
    char *str;
public:
     String(const char *s);
     void change(int index, char c) { str[index] = c; }
     char *get() { return str; }
};

String::String(const char *s)
{
    int l = strlen(s);
    str = new char[l+1];
    strcpy(str, s);
}

int main()
{
   String s1("geeksQuiz");
   String s2 = s1;
   s1.change(0, 'G');
   cout << s1.get() << " ";
   cout << s2.get();
}
```

AGeeksQuiz geeksQuiz
BGeeksQuiz GeeksQuiz
CgeeksQuiz geeksQuiz
DgeeksQuiz GeeksQuiz

Constructors

Discuss it

Question 11 Explanation:
Since there is no copy constructor, the compiler creates a copy constructor. The compiler created copy constructor does shallow copy in line " String s2 = s1;" So str pointers of both s1 and s2 point to the same location. There must be a user defined copy constructor in classes with pointers ot dynamic memory allocation.

Question 12
Predict the output of following program.

```
#include<iostream>
using namespace std;
class Point {
    int x;
public:
    Point(int x) { this->x = x; }
    Point(const Point p) { x = p.x;}
    int getX() { return x; }
};

int main()
{
   Point p1(10);
   Point p2 = p1;
   cout << p2.getX();
   return 0;
}
```

A10
BCompiler Error: p must be passed by reference
CGarbage value
DNone of the above

Constructors

Discuss it

Question 12 Explanation:
Objects must be passed by reference in copy constructors. Compiler checks for this and produces compiler error if not passed by reference. The following program compiles fine and produces output as 10.

```
#include <iostream >
using namespace std;
class Point {
    int x;
```

```
public:
   Point(int x) { this->x = x; }
   Point(const Point &p) { x = p.x;}
   int getX() { return x; }
};

int main()
{
  Point p1(10);
  Point p2 = p1;
  cout


The reason is simple, if we don't pass by reference, then argument p1 will be copied to p.  So there will be a copy constructor call to call the copy cor
```

Question 13
We must use initializer list in a constructor when
AThere is a reference variable in class
BThere is a constant variable in class
CThere is an object of another class. And the other class doesn't have default constructor.
DAll of the above
Constructors
Discuss it
Question 13 Explanation:
See When do we use Initializer List in C++?
Question 14
Which of the following is true about constructors. 1) They cannot be virtual. 2) They cannot be private. 3) They are automatically called by new operator.
AAll 1, 2, and 3
BOnly 1 and 3
COnly 1 and 2
DOnly 2 and 3
Constructors
Discuss it
Question 14 Explanation:
1) True: Virtual constructors don't make sense, it is meaningless to the C++ compiler to create an object polymorphically. 2) False: Constructors can be private, for example, we make copy constructors private when we don't want to create copyable objects. The reason for not making copyable object could be to avoid shallow copy. 3) True: Constructors are automatically called by new operator, we can in-fact pass parameters to constructors.
Question 15

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

class Test
{
public:
  Test();
};

Test::Test()  {
   cout << " Constructor Called. ";
}

void fun() {
  static Test t1;
}

int main() {
   cout << " Before fun() called. ";
   fun();
   fun();
   cout << " After fun() called. ";
   return 0;
}
```

AConstructor Called. Before fun() called. After fun() called.
BBefore fun() called. Constructor Called. Constructor Called. After fun() called.
CBefore fun() called. Constructor Called. After fun() called.

DConstructor Called. Constructor Called. After fun() called.Before fun() called.

Constructors

Discuss it

Question 15 Explanation:

Note that t is static in fun(), so constructor is called only once.

Question 16

Predict the output of following program?

```
#include <iostream>
using namespace std;
class Test
{
private:
   int x;
public:
   Test(int i)
   {
      x = i;
      cout << "Called" << endl;
   }
};

int main()
{
   Test t(20);
   t = 30; // conversion constructor is called here.
   return 0;
}
```

ACompiler Error

B
```
Called
Called
```

C
```
Called
```

Constructors

Discuss it

Question 16 Explanation:

If a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor because such a constructor allows automatic conversion to the class being constructed. A conversion constructor can be called anywhere when the type of single argument is assigned to the object. The output of the given program is

```
Called
Called
```

Question 17

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

class Test
{
public:
  Test(Test &t) { }
  Test()       { }
};

Test fun()
{
   cout << "fun() Called\n";
   Test t;
   return t;
}

int main()
{
   Test t1;
   Test t2 = fun();
   return 0;
}
```

Afun() Called
BEmpty Output
CCompiler Error: Because copy constructor argument is non-const
Constructors
Discuss it
Question 17 Explanation:
See following for details: Why copy constructor argument should be const in C++?
There are 17 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Can destuctors be private in C++?
AYes
BNo
Destructors
Discuss it
Question 1 Explanation:
Destructors can be private. See Private Destructor for examples and uses of private destructors in C++.

Question 2
Predict the output of following C++ progran

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

int i;

class A
{
public:
   ~A()
   {
      i=10;
   }
};

int foo()
{
   i=3;
   A ob;
   return i;
}

int main()
{
   cout << foo() << endl;
   return 0;
}
```

A0
B3
C10
DNone of the above
Destructors
Discuss it
Question 2 Explanation:
While returning from a function, destructor is the last method to be executed. The destructor for the object “ob” is called after the value of i is copied to the return value of the function. So, before destructor could change the value of i to 10, the current value of i gets copied & hence the output is i = 3. See this for more details.

Question 3
Like constructors, can there be more than one destructors in a class?
AYes
BNo
Destructors
Discuss it
Question 3 Explanation:
There can be only one destructor in a class. Destructor's signature is always *~ClassNam()* and they can not be passed arguments.
Question 4

```
#include <iostream>
using namespace std;
class A
{
   int id;
   static int count;
public:
   A() {
      count++;
      id = count;
      cout << "constructor for id " << id << endl;
   }
   ~A() {
      cout << "destructor for id " << id << endl;
   }
};

int A::count = 0;

int main() {
   A a[3];
   return 0;
}
```

A
```
constructor for id 1
constructor for id 2
constructor for id 3
destructor for id 3
destructor for id 2
destructor for id 1
```

B
```
constructor for id 1
constructor for id 2
constructor for id 3
destructor for id 1
destructor for id 2
destructor for id 3
```

C
```
Compiler Dependent.
```

D
```
constructor for id 1
destructor for id 1
```

Destructors
Discuss it
Question 4 Explanation:
In the above program, id is a static variable and it is incremented with every object creation. Object a[0] is created first, but the object a[2] is destroyed first. Objects are always destroyed in reverse order of their creation. The reason for reverse order is, an object created later may use the previously created object. For example, consider the following code snippet.

```
A a;
B b(a);
```

In the above code, the object ‘b’ (which is created after ‘a’), may use some members of ‘a’ internally. So destruction of ‘a’ before ‘b’ may create problems. Therefore, object ‘b’ must be destroyed before ‘a’.
Question 5
Can destructors be virtual in C++?
AYes

BNo
Destructors
Discuss it
Question 5 Explanation:
See http://www.geeksforgeeks.org/g-fact-37/
There are 5 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
How can we restrict dynamic allocation of objects of a class using new?
ABy overloading new operator
BBy making an empty private new operator.
CBy making an empty private new and new[] operators
DBy overloading new operator and new[] operators
Operator Overloading
Discuss it
Question 1 Explanation:
If we declare *new* and *[] new* operators, then the objects cannot be created anywhere (within the class and outside the class) See the following example. We can not allocate an object of type Test using new.

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream>

using namespace std;

class Test {
private:
   void* operator new(size_t size) {}
   void* operator new[](size_t size) {}
};

int main()
{
   Test *obj = new Test;
   Test *arr = new Test[10];
   return 0;
}
```

Question 2
Which of the following operators cannot be overloaded
A. (Member Access or Dot operator)
B?: (Ternary or Conditional Operator)
C:: (Scope Resolution Operator)
D.* (Pointer-to-member Operator)
EAll of the above
Operator Overloading
Discuss it
Question 2 Explanation:
See What are the operators that cannot be overloaded in C++?
Question 3
Which of the following operators are overloaded by default by the compiler in every user defined classes even if user has not written?

```
1) Comparison Operator ( == )
2) Assignment Operator ( = )
```

ABoth 1 and 2
BOnly 1
COnly 2
DNone of the two
Operator Overloading
Discuss it

Question 3 Explanation:
Assign operator is by default available in all user defined classes even if user has not implemented. The default assignement does shallow copy. But comparison operator "==" is not overloaded.

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

class Complex {
private:
   int real, imag;
public:
   Complex(int r = 0, int i =0)  {real = r;   imag = i;}
};

int main()
{
   Complex c1(10, 5), c2(2, 4);

   // For example, below code works fine
   c1 = c2;

   // But this code throws compiler error
   if (c1 == c2)
     cout
```

Question 4
Which of the following operators should be preferred to overload as a global function rather than a member method?
APostfix ++
BComparison Operator
CInsertion Operator
DPrefix++
Operator Overloading
Discuss it
Question 4 Explanation:
cout is an object of ostream class which is a compiler defined class. When we do "cout #include <iostream> using namespace std; class Complex { private: int real; int imag; public: Complex(int r = 0, int i =0) { real = r; imag = i; } friend ostream & operator
Question 5
How does C++ compiler differs between overloaded postfix and prefix operators?
AC++ doesn't allow both operators to be overlaoded in a class
BA postfix ++ has a dummy parameter
CA prefix ++ has a dummy parameter
DBy making prefix ++ as a global function and postfix as a member function.
Operator Overloading
Discuss it
Question 5 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;
}
```

ACompiler Error

B
```
10
20
```

C
```
20
20
```

D
```
10
10
```

Operator Overloading

Discuss it

Question 6 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 7

Output of following program?

```
#include <iostream>
using namespace std;
class Test2
{
   int y;
};

class Test
{
   int x;
```

```
    Test2 t2;
public:
    operator Test2 ()  { return t2; }
    operator int () { return x; }
};

void fun ( int x) { cout << "fun(int) called"; }
void fun ( Test2 t ) { cout << "fun(Test 2) called"; }

int main()
{
    Test t;
    fun(t);
    return 0;
}
```

Afun(int) called
Bfun(Test 2) called
CCompiler Error: Ambiguous call to fun()
Operator Overloading
Discuss it
Question 7 Explanation:
The class Test has two conversion operators overloaded, int and Test2. And there are two fun() for int and Test2.
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
```

Operator Overloading
Discuss it
Question 8 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;
}
```

Ax = 3, y = 2
BCompiler Error
Cx = 2, y = 3
Operator Overloading
Discuss it
Question 9 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.
Anew
Bdelete
CConverstion Operator
DAll of the above
Operator Overloading
Discuss it
Question 10 Explanation:
new and delete can be global, see following example.

```
#include
#include
#include

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
```

There are 10 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Which of the following is true about this pointer?
AIt is passed as a hidden argument to all function calls
BIt is passed as a hidden argument to all non-static function calls
CIt is passed as a hidden argument to all static functions
DNone of the above
this pointer
Discuss it
Question 1 Explanation:
The ‘this’ pointer is passed as a hidden argument to all non-static member function calls and is available as a local variable within the body of all non-static functions. ‘this’ pointer is a constant pointer that holds the memory address of the current object. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name). Source: ‘this’ pointer in C++
Question 2
What is the use of this pointer?
AWhen local variable’s name is same as member’s name, we can access member using this pointer.
BTo return reference to the calling object
CCan be used for chained function calls on an object

DAll of the above
this pointer
Discuss it
Question 2 Explanation:
See following example for first use.

```
/* local variable is same as a member's name */
class Test
{
private:
  int x;
public:
  void setX (int x)
  {
```

```
    // The 'this' pointer is used to retrieve the object's x
    // hidden by the local variable 'x'
    this->x = x;
  }
  void print() { cout
```

And following example for second and third point.

```
#include
using namespace std;

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test &setX(int a) { x = a; return *this; }
  Test &setY(int b) { y = b; return *this; }
  void print() { cout
```

Question 3

Predict the output of following C++ program.

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

class Test
{
private:
  int x;
public:
  Test(int x = 0) { this->x = x; }
  void change(Test *t) { this = t; }
  void print() { cout << "x = " << x << endl; }
};

int main()
{
  Test obj(5);
  Test *ptr = new Test (10);
  obj.change(ptr);
  obj.print();
  return 0;
}
```

Ax = 5

Bx = 10

CCompiler Error

DRuntime Error

this pointer

Discuss it

Question 3 Explanation:

this is a const pointer, so there is an error in line "this = t;"

Question 4

Predict the output of following C++ program

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

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  static void fun1() { cout << "Inside fun1()"; }
```

```
  static void fun2() { cout << "Inside fun2()"; this->fun1(); }
};

int main()
{
  Test obj;
  obj.fun2();
  return 0;
}
```

AInside fun2() Inside fun1()
BInside fun2()
CInside fun1() Inside fun2()
DCompiler Error
this pointer
Discuss it
Question 4 Explanation:
There is error in fun2(). It is a static function and tries to access this pointer. this pointer is not available to static member functions as static member function can be called without any object.
Question 5
Predict the output of following C++ program?

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

class Test
{
private:
  int x;
public:
  Test() {x = 0;}
  void destroy()  { delete this; }
  void print() { cout << "x = " << x; }
};

int main()
{
  Test obj;
  obj.destroy();
  obj.print();
  return 0;
}
```

Ax = 0
Bundefined behavior
Ccompiler error
this pointer
Discuss it
Question 5 Explanation:
delete operator works only for objects allocated using operator new (See http://geeksforgeeks.org/?p=8539). If the object is created using new, then we can do delete this, otherwise behavior is undefined. See "delete this" in C++ for more examples.
There are 5 questions to complete.

GATE CS Corner

Question 1
How to create a dynamic array of pointers (to integers) of size 10 using new in C++? Hint: We can create a non-dynamic array using int *arr[10]
Aint *arr = new int *[10];
Bint **arr = new int *[10];
Cint *arr = new int [10];
DNot Possible
new and delete

Discuss it

Question 2

Which of the following is true about new when compared with malloc. 1) new is an operator, malloc is a function 2) new calls constructor, malloc doesn't 3) new returns appropriate pointer, malloc returns void * and pointer needs to typecast to appropriate type.

A1 and 3

B2 and 3

C1 and 2

DAll 1, 2 and 3

new and delete

Discuss it

Question 2 Explanation:

See malloc() vs new

Question 3

Predict the output?

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

class Test
{
  int x;
  Test() { x = 5;}
};

int main()
{
   Test *t = new Test;
   cout << t->x;
}
```

ACompiler Error

B5

CGarbage Value

D0

new and delete

Discuss it

Question 3 Explanation:

There is compiler error: Test::Test() is private. new makes call to the constructor. In class Test, constructor is private (note that default access is private in C++).

Question 4

What happens when delete is used for a NULL pointer?

```
int *ptr = NULL;
delete ptr;
```

ACompiler Error

BRun-time Crash

CNo Effect

new and delete

Discuss it

Question 4 Explanation:

Deleting a null pointer has no effect, so it is not necessary to check for a null pointer before calling delete.

Question 5

Is it fine to call delete twice for a pointer?

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

int main()
{
   int *ptr = new int;
   delete ptr;
   delete ptr;
   return 0;
}
```

AYes

BNo

new and delete

Discuss it

Question 5 Explanation:

It is undefined behavior to call delete twice on a pointer. Anything can happen, the program may crash or produce nothing.
There are 5 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Which of the following in Object Oriented Programming is supported by Function overloading and default arguments features of C++.
AInheritance
BPolymorphism
CEncapsulation
DNone of the above
Function Overloading and Default Arguments
Discuss it
Question 1 Explanation:
Both of the features allow one function name to work for different parameter.
Question 2
Output?

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

int fun(int x = 0, int y = 0, int z)
{  return (x + y + z); }

int main()
{
  cout << fun(10);
  return 0;
}
```

A10
B0
C20
DCompiler Error
Function Overloading and Default Arguments
Discuss it
Question 2 Explanation:
All default arguments must be the rightmost arguments. The following program works fine and produces 10 as output.

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

int fun(int x, int y = 0, int z = 0)
{  return (x + y + z); }

int main()
{
  cout
```

Question 3
Which of the following overloaded functions are NOT allowed in C++? 1) Function declarations that differ only in the return type

```
int fun(int x, int y);
    void fun(int x, int y);
```

2) Functions that differ only by static keyword in return type

```
int fun(int x, int y);
    static int fun(int x, int y);
```

3)Parameter declarations that differ only in a pointer * versus an array []

```
int fun(int *ptr, int n);
int fun(int ptr[], int n);
```

4) Two parameter declarations that differ only in their default arguments

```
int fun( int x, int y);
int fun( int x, int y = 10);
```

AAll of the above
BAll except 2)
CAll except 1)
DAll except 2 and 4

Function Overloading and Default Arguments

Discuss it

Question 3 Explanation:
See Function overloading in C++
Question 4
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 " << endl; }
   void fun()       {  cout << "fun() " << endl;    }
};

int main()
{
   Test t1 (10);
   const Test t2 (20);
   t1.fun();
   t2.fun();
   return 0;
}
```

ACompiler Error
Bfun()
  fun() const
Cfun() const
  fun() const
Dfun()
  fun()

Function Overloading and Default Arguments

Discuss it

Question 4 Explanation:
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 following for more details. Function overloading and const keyword
Question 5
Output of following program?

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

int fun(int=0, int = 0);

int main()
{
  cout << fun(5);
  return 0;
}

int fun(int x, int y) { return (x+y); }
```

ACompiler Error
B5

C0
D10

Function Overloading and Default Arguments

Discuss it

Question 5 Explanation:
The statement "int fun(int=0, int=0)" is declaration of a function that takes two arguments with default values as 0 and 0. The last statement is definition of fun(). When we make a call fun(5), x gets the value 5 and y gets 0. So the returned value is 5.
There are 5 questions to complete.

GATE CS Corner

Question 1
Predict the output of following C++ program.

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

class Test
{
   static int x;
public:
   Test() { x++; }
   static int getX() {return x;}
};

int Test::x = 0;

int main()
{
   cout << Test::getX() << " ";
   Test t[5];
   cout << Test::getX();
}
```

A0 0
B5 5
C0 5
DCompiler Error

Static Keyword

Discuss it

Question 1 Explanation:
Static functions can be called without any object. So the call "Test::getX()" is fine. Since x is initialized as 0, the first call to getX() returns 0. Note the statement x++ in constructor. When an array of 5 objects is created, the constructor is called 5 times. So x is incremented to 5 before the next call to getX().
Question 2

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

class Player
{
private:
   int id;
   static int next_id;
public:
   int getID() { return id; }
   Player() { id = next_id++; }
};
int Player::next_id = 1;

int main()
{
  Player p1;
  Player p2;
```

```
  Player p3;
  cout << p1.getID() << " ";
  cout << p2.getID() << " ";
  cout << p3.getID();
  return 0;
}
```

ACompiler Error
B1 2 3
C1 1 1
D3 3 3
E0 0 0

Static Keyword

Discuss it

Question 2 Explanation:
If a member variable is declared static, all objects of that class have access to a single instance of that variable. Static variables are sometimes called class variables, class fields, or class-wide fields because they don't belong to a specific object; they belong to the class. In the above code, static variable next_id is used to assign a unique id to all objects.

Question 3
Which of the following is true?
AStatic methods cannot be overloaded.
BStatic data members can only be accessed by static methods.
CNon-static data members can be accessed by static methods.
DStatic methods can only access static members (data and methods)

Static Keyword

Discuss it

Question 4
Predict the output of following C++ program.

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

class A
{
private:
   int x;
public:
   A(int _x) { x = _x; }
   int get() { return x; }
};

class B
{
   static A a;
public:
  static int get()
  { return a.get(); }
};

int main(void)
{
   B b;
   cout << b.get();
   return 0;
}
```

A0
BLinker Error: Undefined reference B::a
CLinker Error: Cannot access static a
DLinker Error: multiple functions with same name get()

Static Keyword

Discuss it

Question 4 Explanation:
There is a compiler error because static member a is not defined in B. To fix the error, we need to explicitly define a. The following program works fine.

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

class A
{
```

```
private:
   int x;
public:
   A(int _x)  {  x = _x; }
   int get()  { return x; }
};

class B
{
   static A a;
public:
  static int get()
  {  return a.get(); }
};

A B::a(0);

int main(void)
{
   B b;
   cout
```

Question 5

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

class Test
{
private:
   static int count;
public:
   Test& fun();
};

int Test::count = 0;

Test& Test::fun()
{
   Test::count++;
   cout << Test::count << " ";
   return *this;
}

int main()
{
   Test t;
   t.fun().fun().fun().fun();
   return 0;
}
```

ACompiler Error
B4 4 4 4
C1 1 1 1
D1 2 3 4
Static Keyword
Discuss it
Question 5 Explanation:
Static members are accessible in non-static functions, so no problem with accessing count in fun(). Also, note that fun() returns the same object by reference.
Question 6
Output of following C++ program?

```
#include <iostream>
class Test
{
public:
   void fun();
};
static void Test::fun()
{
   std::cout<<"fun() is static\n";
```

```
}
int main()
{
   Test::fun();
   return 0;
}
```

Contributed by **Pravasi Meet**
Afun() is static
BEmpty Screen
CCompiler Error
Static Keyword
Discuss it
Question 6 Explanation:
The above program fails in compilation and shows below error messages. [Error] cannot declare member function 'void Test::fun()' to have static linkage [-fpermissive] In function 'int main()': [Error] cannot call member function 'void Test::fun()' without object If the static function is to be defined outside the class then static keyword must be present in function declaration only not in the definition outside the class. Following program is now correct. 1 #include class Test { public: static void fun(); }; void Test::fun() { std::cout
There are 6 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Load Comments

Question 1
Predict the output of following program.

```
#include <iostream>
using namespace std;
class A
{
protected:
   int x;
public:
   A() {x = 0;}
   friend void show();
};

class B: public A
{
public:
   B() : y (0) {}
private:
   int y;
};

void show()
{
   A a;
   B b;
   cout << "The default value of A::x = " << a.x << " ";
   cout << "The default value of B::y = " << b.y;
}
```

ACompiler Error in show() because x is protected in class A
BCompiler Error in show() because y is private in class b
CThe default value of A::x = 0 The default value of B::y = 0
DCompiler Dependent
friend keyword
Discuss it
Question 1 Explanation:
Please note that show() is a friend of class A, so there should not be any compiler error in accessing any member of A in show(). Class B

is inherited from A, the important point to note here is friendship is not inherited. So show() doesn't become a friend of B and therefore can't access private members of B.
Question 2
Predict the output the of following program.

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

class B;
class A {
   int a;
public:
   A():a(0) { }
   void show(A& x, B& y);
};

class B {
private:
   int b;
public:
   B():b(0) { }
   friend void A::show(A& x, B& y);
};

void A::show(A& x, B& y) {
   x.a = 10;
   cout << "A::a=" << x.a << " B::b=" << y.b;
}

int main() {
   A a;
   B b;
   a.show(a,b);
   return 0;
}
```

ACompiler Error
BA::a=10 B::b=0
CA::a=0 B::b=0
friend keyword
Discuss it
Question 2 Explanation:
This is simple program where a function of class A is declared as friend of class B. Since show() is friend, it can access private data members of B.
There are 2 questions to complete.

GATE CS Corner

Question 1

```
#include<iostream>

using namespace std;
class Base1 {
 public:
    Base1()
    { cout << " Base1's constructor called" << endl; }
};

class Base2 {
 public:
    Base2()
    { cout << "Base2's constructor called" << endl; }
};
```

```
class Derived: public Base1, public Base2 {
  public:
    Derived()
    { cout << "Derived's constructor called" << endl; }
};

int main()
{
  Derived d;
  return 0;
}
```

ACompiler Dependent
BBase1's constructor called
Base2's constructor called
Derived's constructor called
CBase2's constructor called
Base1's constructor called
Derived's constructor called
DCompiler Error
Inheritance
Discuss it
Question 1 Explanation:
When a class inherits from multiple classes, constructors of base classes are called in the same order as they are specified in inheritance.
Question 2
Output?

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

class Base1 {
 public:
    ~Base1() { cout << " Base1's destructor" << endl; }
};

class Base2 {
 public:
    ~Base2() { cout << " Base2's destructor" << endl; }
};

class Derived: public Base1, public Base2 {
  public:
    ~Derived() { cout << " Derived's destructor" << endl; }
};

int main()
{
  Derived d;
  return 0;
}
```

A
```
Base1's destructor
Base2's destructor
Derived's destructor
```

B
```
Derived's destructor
Base2's destructor
Base1's destructor
```

C
```
Derived's destructor
```

DCompiler Dependent
Inheritance
Discuss it
Question 2 Explanation:
Destructors are always called in reverse order of constructors.
Question 3

Assume that an integer takes 4 bytes and there is no alignment in following classes, predict the output.

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

class base {
   int arr[10];
};

class b1: public base { };

class b2: public base { };

class derived: public b1, public b2 {};

int main(void)
{
  cout << sizeof(derived);
  return 0;
}
```

A40
B80
C0
D4
Inheritance
Discuss it
Question 3 Explanation:
Since b1 and b2 both inherit from class base, two copies of class base are there in class derived. This kind of inheritance without virtual causes wastage of space and ambiguities. virtual base classes are used to save space and avoid ambiguities in such cases. For example, following program prints 48. 8 extra bytes are for bookkeeping information stored by the compiler (See this for details)

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

class base {
  int arr[10];
};

class b1: virtual public base { };

class b2: virtual public base { };

class derived: public b1, public b2 {};

int main(void)
{
  cout
```

Question 4

```
#include<iostream>

using namespace std;
class P {
public:
   void print()  { cout <<" Inside P"; }
};

class Q : public P {
public:
   void print() { cout <<" Inside Q"; }
};

class R: public Q { };

int main(void)
{
  R r;
  r.print();
  return 0;
}
```

AInside P
BInside Q
CCompiler Error: Ambiguous call to print()
Inheritance
Discuss it
Question 4 Explanation:
The print function is not present in class R. So it is looked up in the inheritance hierarchy. print() is present in both classes P and Q, which of them should be called? The idea is, if there is multilevel inheritance, then function is linearly searched up in the inheritance hierarchy until a matching function is found.
Question 5
Output?

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

class Base {
private:
    int i, j;
public:
   Base(int _i = 0, int _j = 0): i(_i), j(_j) { }
};
class Derived: public Base {
public:
    void show(){
      cout<<" i = "<<i<<"  j = "<<j;
    }
};
int main(void) {
  Derived d;
  d.show();
  return 0;
}
```

Ai = 0 j = 0
BCompiler Error: i and j are private in Base
CCompiler Error: Could not call constructor of Base
Inheritance
Discuss it
Question 6

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

class Base {};

class Derived: public Base {};

int main()
{
   Base *bp = new Derived;
   Derived *dp = new Base;
}
```

ANo Compiler Error
BCompiler Error in line "Base *bp = new Derived;"
CCompiler Error in line " Derived *dp = new Base;"
DRuntime Error
Inheritance
Discuss it
Question 6 Explanation:
A Base class pointer/reference can point/refer to a derived class object, but the other way is not possible.
Question 7

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

class Base
{
public:
   void show()
   {
      cout<<" In Base ";
```

```
    }
};

class Derived: public Base
{
public:
    int x;
    void show()
    {
        cout<<"In Derived ";
    }
    Derived()
    {
        x = 10;
    }
};

int main(void)
{
    Base *bp, b;
    Derived d;
    bp = &d;
    bp->show();
    cout << bp->x;
    return 0;
}
```

ACompiler Error in line " bp->show()"
BCompiler Error in line " cout x"
CIn Base 10
DIn Derived 10
Inheritance
Discuss it
Question 7 Explanation:
A base class pointer can point to a derived class object, but we can only access base class member or virtual functions using the base class pointer.
Question 8

```
#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);
    return 0;
}
```

ABase::fun(int i) called
BDerived::fun() called
CBase::fun() called
DCompiler Error
Inheritance
Discuss it
Question 8 Explanation:
If a derived class writes its own method, then all functions of base class with same name become hidden, even if signaures of base class functions are different. In the above question, when fun() is rewritten in Derived, it hides both fun() and fun(int) of base class.

Question 9

```
#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.Base::fun(5);
   return 0;
}
```

ACompiler Error
BBase::fun(int i) called
Inheritance
Discuss it
Question 9 Explanation:
We can access base class functions using scope resolution operator even if they are made hidden by a derived class function.
Question 10
Output of following program?

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

class Base
{
public:
   virtual string print() const
   {
      return "This is Base class";
   }
};

class Derived : public Base
{
public:
   virtual string print() const
   {
      return "This is Derived class";
   }
};

void describe(Base p)
{
   cout << p.print() << endl;
}

int main()
{
   Base b;
   Derived d;
   describe(b);
   describe(d);
   return 0;
}
```

A
```
This is Derived class
This is Base class
```

B
```
This is Base class
```

This is Derived class

C
```
This is Base class
This is Base class
```

DCompiler Error

Inheritance

Discuss it

Question 10 Explanation:

Note that an object of Derived is passed in describe(d), but print of Base is called. The describe function accepts a parameter of Base type. This is a typical example of object slicing, when we assign an object of derived class to an object of base type, the derived class object is sliced off and all the data members inherited from base class are copied. Object slicing should be ovoided as there may be surprising results like above. As a side note, object slicing is not possible in Java. In Java, every non-primitive variable is actually a reference.

Question 11

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

class Base
{
public :
   int x, y;
public:
   Base(int i, int j){ x = i; y = j; }
};

class Derived : public Base
{
public:
   Derived(int i, int j):x(i), y(j) {}
   void print() {cout << x <<" "<< y; }
};

int main(void)
{
   Derived q(10, 10);
   q.print();
   return 0;
}
```

A10 10

BCompiler Error

C0 0

Inheritance

Discuss it

Question 11 Explanation:

The base class members cannot be directly assigned using initializer list. We should call the base class constructor in order to initialize base class members. Following is error free program and prints "10 10" 1 #include using namespace std; class Base { public : int x, y; public: Base(int i, int j){ x = i; y = j; } }; class Derived : public Base { public: Derived(int i, int j): Base(i, j) {} void print() {cout

Question 12

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

class Base
{
protected:
   int a;
public:
   Base() {a = 0;}
};

class Derived1: public Base
{
public:
   int c;
};


class Derived2: public Base
```

```
{
public:
   int c;
};

class DerivedDerived: public Derived1, public Derived2
{
public:
   void show() {   cout << a; }
};

int main(void)
{
   DerivedDerived d;
   d.show();
   return 0;
}
```

ACompiler Error in Line "cout
B0
CCompiler Error in Line "class DerivedDerived: public Derived1, public Derived2"
Inheritance
Discuss it
Question 12 Explanation:
This is a typical example of diamond problem of multiple inheritance. Here the base class member 'a' is inherited through both *Derived1* and *Derived2*. So there are two copies of 'a' in *DerivedDerived* which makes the statement "cout using namespace std; class Base { protected: int a; public: Base() {a = 0;} }; class Derived1: virtual public Base { public: int c; }; class Derived2: virtual public Base { public: int c; }; class DerivedDerived: public Derived1, public Derived2 { public: void show() { cout
Question 13

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

class Base1
{
public:
   char c;
};

class Base2
{
public:
   int c;
};

class Derived: public Base1, public Base2
{
public:
   void show()  { cout << c; }
};

int main(void)
{
   Derived d;
   d.show();
   return 0;
}
```

ACompiler Error in "cout
BGarbage Value
CCompiler Error in "class Derived: public Base1, public Base2"
Inheritance
Discuss it
Question 13 Explanation:
The variable 'c' is present in both super classes of Derived. So the access to 'c' is ambiguous. The ambiguity can be removed by using scope resolution operator. 1 #include using namespace std; class Base1 { public: char c; }; class Base2 { public: int c; }; class Derived: public Base1, public Base2 { public: void show() { cout
Question 14
Consider the below C++ program.

```
#include<iostream>
```

```
using namespace std;
class A
{
public:
    A(){ cout <<"1";}
    A(const A &obj){ cout <<"2";}
};

class B: virtual A
{
public:
   B(){cout <<"3";}
   B(const B & obj){cout<<"4";}
};

class C: virtual A
{
public:
  C(){cout<<"5";}
  C(const C & obj){cout <<"6";}
};

class D:B,C
{
public:
   D(){cout<<"7";}
   D(const D & obj){cout <<"8";}
};

int main()
{
  D d1;
  D d(d1);
}
```

Which of the below is not printed? *This question is contributed by Sudheendra Baliga*
A2
B4
C6
DAll of the above
Inheritance
Discuss it
Question 14 Explanation:
Output will be 13571358 as 1357 (for D d1) and as 1358 (for D d(d1))......reason is that .....during inheritance we need to explicitly call copy constructor of base class otherwise only default constructor of base class is called. One more thing, as we are using virtual before base class, there will be only one copy of base class in multiple inheritance. And without virtual output will be.....13157....&...13158 as (1315713158) respectively for each derived class object.
There are 14 questions to complete.

GATE CS Corner

Question 1
Predict the output of following program

```
#include <iostream>
using namespace std;
int main()
{
   const char* p = "12345";
   const char **q = &p;
   *q = "abcde";
   const char *s = ++p;
   p = "XYZWVU";
   cout << *++s;
   return 0;
}
```

ACompiler Error
Bc
Cb
DGarbage Value
const keyword
Discuss it
Question 1 Explanation:
Output is ‘c’ const char* p = “12345″ declares a pointer to a constant. So we can’t assign something else to *p, but we can assign new value to p. const char **q = &p; declares a pointer to a pointer. We can’t assign something else to **q, but we can assign new values to q and *q. *q = “abcde”; changes p to point to “abcde” const char *s = ++p; assigns address of literal ”bcde” to s. Again *s can’t be assigned a new value, but s can be changed. The statement printf(“%cn”, *++s) changes s to “cde” and first character at s is printed.
Question 2
In C++, const qualifier can be applied to 1) Member functions of a class 2) Function arguments 3) To a class data member which is declared as static 4) Reference variables
AOnly 1, 2 and 3
BOnly 1, 2 and 4
CAll
DOnly 1, 3 and 4
const keyword
Discuss it
Question 2 Explanation:

When a function is declared as const, it cannot modify data members of its class. When we don't want to modify an argument and pass it as reference or pointer, we use const qualifier so that the argument is not accidentally modified in function. Class data members can be declared as both const and static for class wide constants. Reference variables can be const when they refer a const location.

Question 3

Predict the output of following program.

```
#include <iostream>
using namespace std;
class Point
{
    int x, y;
public:
 Point(int i = 0, int j =0)
   { x = i; y = j; }
   int getX() const { return x; }
   int getY() {return y;}
};

int main()
{
    const Point t;
    cout << t.getX() << " ";
    cout << t.gety();
    return 0;
}
```

AGarbage Values

B0 0

CCompiler Error in line cout

DCompiler Error in line cout

const keyword

Discuss it

Question 3 Explanation:

There is compiler Error in line cout

Question 4

```
#include <stdio.h>
int main()
{
   const int x;
   x = 10;
   printf("%d", x);
   return 0;
}
```

ACompiler Error

B10

C0

DRuntime Error

const keyword

Discuss it

Question 4 Explanation:

One cannot change the value of 'const' variable except at the time of initialization. Compiler does check this.

Question 5

Output of C++ program?

```
#include <iostream>
int const s=9;
int main()
{
    std::cout << s;
    return 0;
}
```

Contributed by **Pravasi Meet**

A9

BCompiler Error

const keyword

Discuss it

Question 5 Explanation:

The above program compiles & runs fine. Const keyword can be put after the variable name or before variable name. But most

programmers prefer to put const keyword before the variable name.
There are 5 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Load Comments

Question 1
Which of the following is true about virtual functions in C++.
AVirtual functions are functions that can be overridden in derived class with the same signature.
BVirtual functions enable run-time polymorphism in a inheritance hierarchy.
CIf a function is 'virtual' in the base class, the most-derived class's implementation of the function is called according to the actual type of the object referred to, regardless of the declared type of the pointer or reference. In non-virtual functions, the functions are called according to the type of reference or pointer.
DAll of the above
Virtual Functions
Discuss it
Question 1 Explanation:
See http://en.wikipedia.org/wiki/Virtual_function
Question 2
Predict output of the following program

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

class Base
{
public:
   virtual void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
public:
   void show() { cout<<"In Derived \n"; }
};

int main(void)
{
   Base *bp = new Derived;
   bp->show();

   Base &br = *bp;
   br.show();

   return 0;
}
```

A
```
In Base
In Base
```

B
```
In Base
In Derived
```

C
```
In Derived
In Derived
```

D
```
In Derived
In Base
```

Virtual Functions
Discuss it
Question 2 Explanation:
Since show() is virtual in base class, it is called according to the type of object being referred or pointed, rather than the type of pointer or reference.
Question 3
Output of following program

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

class Base
{
public:
   virtual void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
public:
   void show() { cout<<"In Derived \n"; }
};

int main(void)
{
   Base *bp, b;
   Derived d;
   bp = &d;
   bp->show();
   bp = &b;
   bp->show();
   return 0;
}
```

A
```
In Base
In Base
```

B
```
In Base
In Derived
```

C
```
In Derived
In Derived
```

D
```
In Derived
In Base
```

Virtual Functions
Discuss it
Question 3 Explanation:
Initially base pointer points to a derived class object. Later it points to base class object,
Question 4
Which of the following is true about pure virtual functions? 1) Their implementation is not provided in a class where they are declared. 2) If a class has a pure virtual function, then the class becomes abstract class and an instance of this class cannot be created.
ABoth 1 and 2
BOnly 1
COnly 2
DNeither 1 nor 2
Virtual Functions
Discuss it
Question 4 Explanation:
S e e http://en.wikipedia.org/wiki/Virtual_function#Abstract_classes_and_pure_virtual_functions See http://stackoverflow.com/questions/5481941/c-pure-virtual-function-have-body
Question 5

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

class Base
```

```
{
public:
   virtual void show() = 0;
};

int main(void)
{
   Base b;
   Base *bp;
   return 0;
}
```

AThere are compiler errors in lines "Base b;" and "Base bp;"
BThere is compiler error in line "Base b;"
CThere is compiler error in line "Base bp;"
DNo compiler Error
Virtual Functions
Discuss it
Question 5 Explanation:
Since Base has a pure virtual function, it becomes an abstract class and an instance of it cannot be created. So there is an error in line "Base b". Note that there is no error in line "Base *bp;". We can have pointers or references of abstract classes.
Question 6
Predict the output of following program.

```
#include<iostream>
using namespace std;
class Base
{
public:
   virtual void show() = 0;
};

class Derived : public Base { };

int main(void)
{
   Derived q;
   return 0;
}
```

ACompiler Error: there cannot be an empty derived class
BCompiler Error: Derived is abstract
CNo compiler Error
Virtual Functions
Discuss it
Question 6 Explanation:
If we don't override the pure virtual function in derived class, then derived class also becomes abstract class.
Question 7

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

class Base
{
public:
   virtual void show() = 0;
};

class Derived: public Base
{
public:
   void show() { cout<<"In Derived \n"; }
};

int main(void)
{
   Derived d;
   Base &br = d;
   br.show();
   return 0;
}
```

ACompiler Error in line "Base &br = d;"
BEmpty Output
CIn Derived
Virtual Functions
Discuss it
Question 7 Explanation:
Please refer Pure Virtual Functions and Abstract Classes in C++
Question 8
Can a constructor be virtual? Will the following program compile?

```
#include <iostream>
using namespace std;
class Base {
public:
  virtual Base() {}
};
int main() {
   return 0;
}
```

AYes
BNo
Virtual Functions
Discuss it
Question 8 Explanation:
There is nothing like Virtual Constructor. Making constructors virtual doesn't make sense as constructor is responsible for creating an object and it can't be delegated to any other object by virtual keyword means.
Question 9
Can a destructor be virtual? Will the following program compile?

```
#include <iostream>
using namespace std;
class Base {
public:
  virtual ~Base() {}
};
int main() {
   return 0;
}
```

AYes
BNo
Virtual Functions
Discuss it
Question 9 Explanation:
A destructor can be virtual. We may want to call appropriate destructor when a base class pointer points to a derived class object and we delete the object. If destructor is not virtual, then only the base class destructor may be called. For example, consider the following program.

```
// Not good code as destructor is not virtual
#include<iostream>
using namespace std;

class Base  {
public:
   Base()    { cout
```

Question 10

```
#include<iostream>
using namespace std;
class Base  {
public:
   Base()    { cout<<"Constructor: Base"<<endl; }
   virtual ~Base()   { cout<<"Destructor : Base"<<endl; }
};
class Derived: public Base {
public:
   Derived()   { cout<<"Constructor: Derived"<<endl; }
   ~Derived()  { cout<<"Destructor : Derived"<<endl; }
};
int main()  {
```

```
    Base *Var = new Derived();
    delete Var;
    return 0;
}
```

A
```
Constructor: Base
Constructor: Derived
Destructor : Derived
Destructor : Base
```

B
```
Constructor: Base
Constructor: Derived
Destructor : Base
```

C
```
Constructor: Base
Constructor: Derived
Destructor : Derived
```

D
```
Constructor: Derived
Destructor : Derived
```

Virtual Functions

Discuss it

Question 10 Explanation:

Since the destructor is vitrual, the derived class destructor is called which in turn calls base class destructor.

Question 11

Can static functions be virtual? Will the following program compile?

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

class Test
{
  public:
    virtual static void fun() { }
};
```

AYes

BNo

Virtual Functions

Discuss it

Question 11 Explanation:

Static functions are class specific and may not be called on objects. Virtual functions are called according to the pointed or referred object.

Question 12

Predict the output of following C++ program. Assume that there is no alignment and a typical implementation of virtual functions is done by the compiler.

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

class A
{
public:
   virtual void fun();
};

class B
{
public:
  void fun();
};

int main()
{
   int a = sizeof(A), b = sizeof(B);
   if (a == b) cout << "a == b";
   else if (a > b) cout << "a > b";
   else cout << "a < b";
   return 0;
```

```
}
```

Aa > b
Ba == b
Ca
DCompiler Error

Virtual Functions

Discuss it

Question 12 Explanation:
Class A has a VPTR which is not there in class B. In a typical implementation of virtual functions, compiler places a VPTR with every object. Compiler secretly adds some code in every constructor to this.

Question 13

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

class A
{
public:
   virtual void fun() { cout << "A::fun() "; }
};

class B: public A
{
public:
  void fun() { cout << "B::fun() "; }
};

class C: public B
{
public:
  void fun() { cout << "C::fun() "; }
};

int main()
{
   B *bp = new C;
   bp->fun();
   return 0;
}
```

AA::fun()
BB::fun()
CC::fun()

Virtual Functions

Discuss it

Question 13 Explanation:
The important thing to note here is B::fun() is virtual even if we have not uses virtual keyword with it. When a class has a virtual function, functions with same signature in all descendant classes automatically become virtual. We don't need to use virtual keyword in declaration of fun() in B and C. They are anyways virtual.

Question 14

Predict the output of following C++ program

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

class Base
{
public:
   virtual void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
public:
   void show() { cout<<"In Derived \n"; }
};

int main(void)
{
   Base *bp = new Derived;
```

```
    bp->Base::show(); // Note the use of scope resolution here
    return 0;
}
```

AIn Base
BIn Derived
CCompiler Error
DRuntime Error
Virtual Functions
Discuss it
Question 14 Explanation:
A base class function can be accessed with scope resolution operator even if the function is virtual.
There are 14 questions to complete.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Template is simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write same code for different data types. For example a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter....Read More

Question 1
Which of the following is true about templates.
1) Template is a feature of C++ that allows us to write one code for different data types.
2) We can write one function that can be used for all data types including user defined types. Like sort(), max(), min(), ..etc.
3) We can write one class or struct that can be used for all data types including user defined types. Like Linked List, Stack, Queue ..etc.
4) Template is an example of compile time polymorphism.
A1 and 2
B1, 2 and 3
C1, 2 and 4
D1, 2, 3 and 4
Templates
Discuss it
Question 1 Explanation:
see http://en.wikipedia.org/wiki/Template_(C%2B%2B)
Question 2
Predict the output?

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

template <typename T>
void fun(const T&x)
{
   static int count = 0;
   cout << "x = " << x << " count = " << count << endl;
   ++count;
   return;
}

int main()
{
   fun<int> (1);
   cout << endl;
   fun<int>(1);
   cout << endl;
   fun<double>(1.1);
   cout << endl;
   return 0;
}
```

A
```
x = 1 count = 0
```

```
x = 1 count = 1

x = 1.1 count = 0
```

B
```
x = 1 count = 0

x = 1 count = 0

x = 1.1 count = 0
```

C
```
x = 1 count = 0

x = 1 count = 1

x = 1.1 count = 2
```

DCompiler Error

Templates

Discuss it

Question 2 Explanation:

Compiler creates a new instance of a template function for every data type. So compiler creates two functions in the above example, one for int and other for double. Every instance has its own copy of static variable. The int instance of function is called twice, so count is incremented for the second call.

Question 3

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

template <typename T>
T max(T x, T y)
{
   return (x > y)? x : y;
}
int main()
{
   cout << max(3, 7) << std::endl;
   cout << max(3.0, 7.0) << std::endl;
   cout << max(3, 7.0) << std::endl;
   return 0;
}
```

A
```
7
7.0
7.0
```

BCompiler Error in all cout statements as data type is not specified.

CCompiler Error in last cout statement as call to max is ambiguous.

DNone of the above

Templates

Discuss it

Question 4

Output of following program?

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

template <class T>
class Test
{
private:
   T val;
public:
   static int count;
   Test() {   count++;  }
};

template<class T>
int Test<T>::count = 0;
```

```
int main()
{
   Test<int> a;
   Test<int> b;
   Test<double> c;
   cout << Test<int>::count   << endl;
   cout << Test<double>::count << endl;
   return 0;
}
```

A
```
0
0
```

B
```
1
1
```

C
```
2
1
```

D
```
1
0
```

Templates
Discuss it
Question 4 Explanation:
There are two classes created by the template: Test and Test. Since count is a static member, every class has its own copy of it. Also, count gets incremented in constructor.
Question 5
Output of following program? Assume that the size of char is 1 byte and size of int is 4 bytes, and there is no alignment done by the compiler.

```
#include<iostream>
#include<stdlib.h>
using namespace std;

template<class T, class U>
class A  {
   T x;
   U y;
   static int count;
};

int main()  {
  A<char, char> a;
  A<int, int> b;
  cout << sizeof(a) << endl;
  cout << sizeof(b) << endl;
  return 0;
}
```

A
```
6
12
```

B
```
2
8
```

CCompiler Error: There can not be more than one template arguments.

D
```
8
8
```

Templates
Discuss it
Question 5 Explanation:
Since count is static, it is not counted in sizeof.

Question 6
Output of following program? Assume that the size of int is 4 bytes and size of double is 8 bytes, and there is no alignment done by the compiler.

```
#include<iostream>
#include<stdlib.h>
using namespace std;

template<class T, class U, class V=double>
class A {
   T x;
   U y;
   V z;
   static int count;
};

int main()
{
  A<int, int> a;
  A<double, double> b;
  cout << sizeof(a) << endl;
  cout << sizeof(b) << endl;
  return 0;
}
```

A
```
16
24
```

B
```
8
16
```

C
```
20
28
```

DCompiler Error: template parameters cannot have default values.

Templates

Discuss it

Question 6 Explanation:
templates can also have default parameters. The rule is same all default values must be on the rightmost side. Since count is static, it is not counted in sizeof.

Question 7
Output of following program.

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

template <class T, int max>
int arrMin(T arr[], int n)
{
  int m = max;
  for (int i = 0; i < n; i++)
    if (arr[i] < m)
      m = arr[i];

  return m;
}

int main()
{
  int arr1[]  = {10, 20, 15, 12};
  int n1 = sizeof(arr1)/sizeof(arr1[0]);

  char arr2[] = {1, 2, 3};
  int n2 = sizeof(arr2)/sizeof(arr2[0]);

  cout << arrMin<int, 10000>(arr1, n1) << endl;
  cout << arrMin<char, 256>(arr2, n2);
  return 0;
}
```

ACompiler error, template argument must be a data type.

B
```
10
1
```

C
```
10000
256
```

D
```
1
1
```

Templates
Discuss it
Question 7 Explanation:
We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of template. The important thing to note about non-type parameters is, they must be const. Compiler must know the value of non-type parameters at compile time. Because compiler needs to create functions/classes for a specified non-type value at compile time. Following is another example of non-type parameters.

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

template
 T fun (T arr[], int size)
{
   if (size > N)
     cout  (arr, 3);
}
```

Question 8
Output?

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

template <int i>
void fun()
{
  i = 20;
  cout << i;
}

int main()
{
  fun<10>();
  return 0;
}
```

A10
B20
CCompiler Error
Templates
Discuss it
Question 8 Explanation:
Compiler error in line "i = 20;" Non-type parameters must be const, so they cannot be modified.
Question 9
Output?

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

template <class T>
T max (T &a, T &b)
{
   return (a > b)? a : b;
}

template <>
int max <int> (int &a, int &b)
{
   cout << "Called ";
```

```
    return (a > b)? a : b;
}

int main ()
{
    int a = 10, b = 20;
    cout << max <int> (a, b);
}
```

A20
BCalled 20
CCompiler Error
Templates
Discuss it
Question 9 Explanation:
Above program is an example of template specialization. Sometime we want a different behaviour of a function/class template for a particular data type. For this, we can create a specialized version for that particular data type.
Question 10
Output?

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

template<int n> struct funStruct
{
    static const int val = 2*funStruct<n-1>::val;
};

template<> struct funStruct<0>
{
    static const int val = 1 ;
};

int main()
{
    cout << funStruct<10>::val << endl;
    return 0;
}
```

ACompiler Error
B1024
C2
D1
Templates
Discuss it
Question 10 Explanation:
This is an example of template metaprogramming. The program mainly calculates 2^10.
There are 10 questions to complete.

Article on Templates in C++

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1

```
#include <iostream>
using namespace std;
int main()
{
  int x = -1;
  try {
    cout << "Inside try \n";
    if (x < 0)
```

```
    {
      throw x;
      cout << "After throw \n";
    }
  }
  catch (int x ) {
    cout << "Exception Caught \n";
  }

  cout << "After catch \n";
  return 0;
}
```

A
```
Inside try
Exception Caught
After throw
After catch
```

B
```
Inside try
Exception Caught
After catch
```

C
```
Inside try
Exception Caught
```

D
```
Inside try
After throw
After catch
```

Exception Handling

Discuss it

Question 1 Explanation:

When an exception is thrown, lines of try block after the throw statement are not executed. When exception is caught, the code after catch block is executed. Catch blocks are generally written at the end through.

Question 2

What is the advantage of exception handling?

```
1) Remove error-handling code from the software's main line of code.

2) A method writer can chose to handle certain exceptions and delegate
   others to the caller.

3) An exception that occurs in a function can be handled anywhere in
   the function call stack.
```

AOnly 1

B1, 2 and 3

C1 and 3

D1 and 2

Exception Handling

Discuss it

Question 3

What should be put in a *try* block?

1. Statements that might cause exceptions
2. Statements that should be skipped in case of an exception

AOnly 1

BOnly 2

CBoth 1 and 2

Exception Handling

Discuss it

Question 3 Explanation:

The statements which may cause problems are put in try block. Also, the statements which should not be executed after a problem accursed, are put in try block. Note that once an exception is caught, the control goes to the next line after the catch block.

Question 4

Output of following program

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

class Base {};
class Derived: public Base {};
int main()
{
  Derived d;
  try {
     throw d;
  }
  catch(Base b) {
      cout<<"Caught Base Exception";
  }
  catch(Derived d) {
      cout<<"Caught Derived Exception";
  }
  return 0;
}
```

ACaught Derived Exception
BCaught Base Exception
CCompiler Error

Exception Handling

Discuss it

Question 4 Explanation:
If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class. If we put base class first then the derived class catch block will never be reached. In Java, catching a base class exception before derived is not allowed by the compiler itself. In C++, compiler might give warning about it, but compiles the code.

Question 5

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

int main()
{
   try
   {
     throw 'a';
   }
   catch (int param)
   {
      cout << "int exception\n";
   }
   catch (...)
   {
      cout << "default exception\n";
   }
   cout << "After Exception";
   return 0;
}
```

A
```
default exception
After Exception
```

B
```
int exception
After Exception
```

C
```
int exception
```

D
```
default exception
```

Exception Handling

Discuss it

Question 5 Explanation:
The block *catch(...)* is used for catch all, when a data type of a thrown exception doesn't match with any other catch block, the code inside catch(...) is executed. Note that the implicit type conversion doesn't happen when exceptions are caught. The character 'a' is not

automatically converted to int.
Question 6

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

int main()
{
   try
   {
     throw 10;
   }
   catch (...)
   {
      cout << "default exception\n";
   }
   catch (int param)
   {
      cout << "int exception\n";
   }

   return 0;
}
```

Adefault exception
Bint exception
CCompiler Error
Exception Handling
Discuss it
Question 6 Explanation:
It is compiler error to put catch all block before any other catch. The catch(...) must be the last catch block.
Question 7

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

int main()
{
   try
   {
      try
      {
         throw 20;
      }
      catch (int n)
      {
         cout << "Inner Catch\n";
         throw;
      }
   }
   catch (int x)
   {
      cout << "Outer Catch\n";
   }
   return 0;
}
```

A
```
Outer Catch
```

B
```
Inner Catch
```

C
```
Inner Catch
Outer Catch
```

DCompiler Error
Exception Handling
Discuss it
Question 7 Explanation:
The statement 'throw;' is used to re-throw an exception. This is useful when a function can handles some part of the exception handling and

then delegates the remaining part to the caller. A catch block cleans up resources of its function, and then rethrows the exception for handling elsewhere.

Question 8

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

class Test {
public:
  Test() { cout << "Constructing an object of Test " << endl; }
 ~Test() { cout << "Destructing an object of Test "  << endl; }
};

int main() {
 try {
   Test t1;
   throw 10;
 } catch(int i) {
   cout << "Caught " << i << endl;
 }
}
```

A
```
Caught 10
```

B
```
Constructing an object of Test
Caught 10
```

C
```
Constructing an object of Test
Destructing an object of Test
Caught 10
```

DCompiler Errror

Exception Handling

Discuss it

Question 8 Explanation:

When an object is created inside a try block, destructor for the object is called before control is transferred to catch block.

Question 9

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

class Test {
 static int count;
 int id;
public:
 Test() {
   count++;
   id = count;
   cout << "Constructing object number " << id << endl;
   if(id == 4)
     throw 4;
 }
 ~Test() { cout << "Destructing object number " << id << endl; }
};

int Test::count = 0;

int main() {
 try {
   Test array[5];
 } catch(int i) {
   cout << "Caught " << i << endl;
 }
}
```

A
```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
```

```
Destructing object number 1
Destructing object number 2
Destructing object number 3
Destructing object number 4
Caught 4
```

B
```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 3
Destructing object number 2
Destructing object number 1
Caught 4
```

C
```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 4
Destructing object number 3
Destructing object number 2
Destructing object number 1
Caught 4
```

D
```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 1
Destructing object number 2
Destructing object number 3
Caught 4
```

Exception Handling
Discuss it
Question 9 Explanation:
The destructors are called in reverse order of constructors. Also, after the try block, the destructors are called only for completely constructed objects.
Question 10
Which of the following is true about exception handling in C++? 1) There is a standard exception class like Exception class in Java. 2) All exceptions are unchecked in C++, i.e., compiler doesn't check if the exceptions are caught or not. 3) In C++, a function can specify the list of exceptions that it can throw using comma separated list like following.

```
void fun(int a, char b) throw (Exception1, Exception2, ..)
```

A1 and 3
B1, 2 and 3
C1 and 2
D2 and 3
Exception Handling
Discuss it
Question 10 Explanation:
See Comparison of Exception Handling in C++ and Java
Question 11
What happens in C++ when an exception is thrown and not caught anywhere like following program.

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

int fun() throw (int)
{
   throw 10;
}

int main() {

  fun();
```

```
  return 0;
}
```

ACompiler error
BAbnormal program termination
CProgram doesn't print anything and terminates normally
DNone of the above
Exception Handling
Discuss it
Question 11 Explanation:
When an exception is thrown and not caught, the program terminates abnormally.
Question 12
What happens when a function throws an error but doesn't specify it in the list of exceptions it can throw. For example, what is the output of following program?

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

// Ideally it should have been "int fun() (int)"
int fun()
{
   throw 10;
}

int main()
{
   try
   {
      fun();
   }
   catch (int )
   {
      cout << "Caught";
   }
   return 0;
}
```

ACompiler Error
BNo compiler Error. Output is "Caught"
Exception Handling
Discuss it
Question 12 Explanation:
C++ compiler doesn't check enforce a function to list the exceptions that it can throw. In Java, it is enforced. It is up to the programmer to specify. Being a civilized programmer, a programmer should specify the list.
There are 12 questions to complete.

GATE CS Corner

Question 1
Inline functions are useful when
AFunction is large with many nested loops
BFunction has many static variables
CFunction is small and we want to avoid function call overhead.
DNone of the above
Misc C++
Discuss it
Question 1 Explanation:
Inline functions are generally used in place of small macros. They are substitute to macros and better than macros. See following for details. http://en.wikipedia.org/wiki/Inline_function#Comparison_with_macros
Question 2

```
#include<iostream>
```

```
using namespace std;
int x = 1;
void fun()
{
   int x = 2;
   {
      int x = 3;
      cout << ::x << endl;
   }
}
int main()
{
   fun();
   return 0;
}
```

A1
B2
C3
D0
Misc C++
Discuss it
Question 2 Explanation:
The value of ::x is 1. The scope resolution operator when used with a variable name, always refers to global variable.
Question 3
Predict the output of following C++ program

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

union A {
  int a;
  unsigned int b;
  A() { a = 10; }
  unsigned int getb() {return b;}
};

int main()
{
   A obj;
   cout << obj.getb();
   return 0;
}
```

ACompiler Error: union can't have functions
BCompiler Error: can't access private members of A
C10
Dgarbage value
Misc C++
Discuss it
Question 3 Explanation:
Like struct and class, union can have methods. Like struct and unlike class, members of union are public by default. Since data members of union share memory, the value of b becomes same as a.
Question 4
Which of the following is true about inline functions and macros.
AInline functions do type checking for parameters, macros don't
BMacros are processed by pre-processor and inline functions are processed in later stages of compilation.
CMacros cannot have return statement, inline functions can.
DMacros are prone to bugs and errors, inline functions are not.
EAll of the above
Misc C++
Discuss it
Question 5
How can we make a C++ class such that objects of it can only be created using new operator? If user tries to create an object directly, the program produces compiler error.
ANot possible
BBy making destructor private
CBy making constructor private

DBy making both constructor and destructor private
Misc C++

Discuss it

Question 5 Explanation:

See the following example.

```
// Objects of test can only be created using new
class Test
{
private:
   ~Test() {}
friend void destructTest(Test* );
};

// Only this function can destruct objects of Test
void destructTest(Test* ptr)
{
   delete ptr;
}

int main()
{
   // create an object
   Test *ptr = new Test;

   // destruct the object
   destructTest (ptr);

   return 0;
}
```

See http://www.geeksforgeeks.org/private-destructor/ for more details.

Question 6

Would destructor be called, if yes, then due to which vector?

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

class a
{
public :
   ~a()
   {
      cout << "destroy";
   }
};
int main()
{
  vector <a*> *v1  = new vector<a*>;
  vector <a> *v2  = new vector<a>;
  return 0;
}
```

Av1

Bv2

Cv1 and v2

Dno destructor call

Misc C++

Discuss it

Question 7

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

int x[100];
int main()
{
   cout << x[99] << endl;
}
```

This question is contributed by Sudheendra Baliga

AUnpredictable

BRuntime error

C0
D99
Misc C++
Discuss it
Question 7 Explanation:
The correct answer is c. In C++ all the uninitialized global variables are initialized to 0.
Question 8

```
#include<iostream>
using namespace std;
int main ()
{
    int cin;
    cin >> cin;
    cout << "cin" << cin;
    return 0;
}
```

Thanks to Gokul Kumar for contributing this question.
Aerror in using cin keyword
Bcin+junk value
Ccin+input
DRuntime error
Misc C++
Discuss it
There are 8 questions to complete.

GATE CS Corner