Contents

[This Pointer 2](#_Toc475530104)

[Const 2](#_Toc475530105)

[Static Members 3](#_Toc475530106)

[Overloading Vs Overriding 4](#_Toc475530107)

[Type difference of character literals in C and C++ 4](#_Toc475530108)

[References in C++ 5](#_Toc475530109)

[Storage Classes 8](#_Toc475530110)

[“Volatile” qualifier 8](#_Toc475530111)

[Function Overloading 9](#_Toc475530112)

[Constructor and Destructor 11](#_Toc475530113)

[*Copy Constructor* 11](#_Toc475530114)

[*When is copy constructor called?* 11](#_Toc475530115)

[*Virtual* Functions 12](#_Toc475530116)

[Operator Overloading 13](#_Toc475530117)

[Casting operators 15](#_Toc475530118)

# This Pointer

* ‘this’ pointer is passed as a **hidden argument to all non-static member** function calls
* ‘this’ 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**
* Since this is constant pointer

Test setX(int a) { x = a; return this;} will throw an error

Test &setX(int a) { x = a; return \*this; } will work

* deleteoperator works only for objects allocated using operator new. 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;  } |

# Const

* 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**.
* **A const object *can only call const functions*.**

|  |
| --- |
| 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();// error      return 0;  } |

* **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.

|  |
| --- |
| int **const** s=9;  int main()  {      std::cout << s;      return 0;  } |

# ****Static Members****

http://www.geeksforgeeks.org/static-objects-destroyed/

* Static member functions ***do not have this pointer***.
* A static member function **cannot be virtual**
* 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.

|  |
| --- |
| class Test {     static void fun() {}     void fun() {} // compiler error  }; |

* A static member function **cannot be declared const, volatile, or const volatile.**

Default values of static variables

* If it has **pointer** type, it is initialized to a **NULL** pointer;
* If it has **arithmetic** type, it is initialized to (positive or unsigned) **zero**;
* If it is an **aggregate**, **every member is initialized (recursively)** **according to these rules**;
* If it is a **union**, the **first named member is initialized (recursively)** **according to these rules**.

**Model–view–controller** (**MVC**) is a [software design pattern](https://en.wikipedia.org/wiki/Software_design_pattern) for designing software and web applications. It divides a given application into three interconnected parts in order to separate internal representations of information from the ways that information is presented to and interacted with by the user. The MVC design pattern decouples these major components allowing for efficient [code reuse](https://en.wikipedia.org/wiki/Code_reuse) and parallel development.

What are the real applications of linked list?

Front-end frameworks?

Promises. Calling multiple AJAX calls parallelly.

Give an example for oops. What is OO Javascript.

# Overloading Vs Overriding

1. Function **Overloading** is when **multiple functions with same name but with different parameters exist in a class**. Function **Overriding** is when function have **same prototype in base class as well as derived class**.
2. Function **Overloading** can occur without inheritance.

Function **Overriding** occurs when one class is inherited from another class.

Name Mangling and extern “C” in C++

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.

|  |
| --- |
| // Save file as .cpp and use C++ compiler to compile it  int **printf**(const char \*format,...);    int main()  {  **printf**("GeeksforGeeks"); // throw an error for C++ compiler      return 0;  } |
| *// Allow C code to compile in C style in C++ compiler*  **extern** "C"  {       int **printf**(const char \*format,...);  } |

Void pointer in C & C++

**C** allows a **void\*** pointer to be **assigned to any pointer type without a cast**, whereas **C++** does not.

|  |
| --- |
| *// C code*  **void**\* ptr;  **int** \*i = ptr; */\* Implicit conversion from void\* to int\* \*/* |
| *//C++ code and also work in C*  **void**\* ptr;  **int** \*i = (int \*) ptr; ***// Type cast is important***  **int** \*j = (int \*) malloc(sizeof(int) \* 5); |

# Type difference of character literals in C and C++

* 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 in C***  **sizeof**(‘V’) = **4** **sizeof**(char) = **1** |
| ***// Result in C++***  **sizeof**(‘V’) = **1** **sizeof**(char) = **1** |

# References in C++

* When a variable is declared as reference, it becomes an **alternative name** for an existing variable.

|  |
| --- |
| **int** x = 10;    *// ref is a reference to x.*  **int**& ref = x; |
| **void** swap (**int**& first, **int**& second)  {  **int** temp = first;      first = second;      second = temp;  } |

References vs Pointers

http://www.geeksforgeeks.org/references-in-c/

Similarity

1. To **modify local variables** of the caller function
2. For **passing large sized arguments & return big object**, to get efficiency gain
3. To **avoid Object Slicing**
4. To achieve **Run Time Polymorphism** in a function

Differences

1. A pointer can be declared as void but a **reference can never be void**

|  |
| --- |
| **int** a = 10;  **void**\* aa = &a; *//it is valid*  **void** &ar = a; *// it is not valid* |

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

Places where Pointers cannot be used

1. Copy constructor argument where pointer cannot be used. **Reference** must be used **pass the argument in copy constructor.**
2. **References** must be used for **overloading** **some operators like ++**

Invalid reference examples

|  |
| --- |
| *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;  } |

Inline Functions

<http://quiz.geeksforgeeks.org/inline-functions-cpp/>

* When **function execution time is less than the switching time** from the caller function to called function (callee).
* Substitution is performed **by the C++ compiler** at compile time.
* Inline function may **increase efficiency** if it is small.
* Inlining is only a **request to the compiler**, not a command.

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**.

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.

*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.

***What is wrong with macro?***

* Macros are **error prone**.
* Macro **cannot access private members** of class.
* Macros **looks like function call but they are actually not**.

|  |
| --- |
| **class** S  {  **int** m;  **public**:  #define MAC(S::m)    *// error - access private*  }; |

* Macros are **managed by preprocessor** and inline functions are managed by C++ compiler
* Look link for more info

# Storage Classes

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Storage Class | Keyword | Lifetime | Visibility | Initial Value |
|  |  | 1 |  |  | Automatic | auto | Function Block | Local | Garbage |
|  |  | 2 |  |  | External | extern | Whole Program | Global | Zero |
|  |  | 3 |  |  | Static | static | Whole Program | Local | Zero |
|  |  | 4 |  |  | Register | register | Function Block | Local | Garbage |
|  |  | 5 |  |  | Mutable | mutable | Class | Local | Garbage |

Mutable Storage Class

https://www.programtopia.net/cplusplus/docs/storage-classes

In C++, a class object can be kept constant using keyword *const*. This doesn't allow the data members of the class object to be modified during program execution. But, there are cases when some data members of this constant object must be changed.

 Mutable = they can change what they hold

It **allows changing constant object member variable & also allows change in const function**.

|  |  |
| --- | --- |
| **class** test{  **mutable** int a;  **int** b;  **public**:  test(int x,int y){a=x;b=y;}  **void** square\_a() **const**{  a=a\*a;  }  **void** display() **const**{  cout<<"a = "<<a<<" b = "<<b<<endl;  }  };  **int** main(){  **const** test x(2,3);  cout<<"Initial value"<<endl;  x.display();  x.square\_a();  cout<<"Final value"<<endl;  x.display();  **return** 0;  } | **Output**  Initial value  **a = 2**  **b = 3**  Final value  **a = 4**  **b = 3** |

# “Volatile” qualifier

<http://www.geeksforgeeks.org/understanding-volatile-qualifier-c-set-1-introduction/>

<http://www.geeksforgeeks.org/understanding-volatile-qualifier-in-c/>

* Intended to **prevent the (pseudo)compiler from applying any optimizations** on the code that assume values of variables cannot change "on their own."
* An object **can be modified** in the program by something such as the **operating system, the hardware, or a concurrently executing thread**.

|  |  |
| --- | --- |
| **int** some\_int = 100;  **while**(some\_int == 100){  *//your code*  } | it may be tempted to optimize the while loop by changing it from  **while**(some\_int == 100)  to simply  **while**(true) |

# Function Overloading

* Where two or more functions can have the same name but different parameters.
* Overloading is allowed with function const type **void** func() **const**.

Functions that cannot be overloaded

1. Function declarations that **differ only in the return type**.

|  |
| --- |
| **class** Test {  **int** fun() {**return** 10;}  **char** fun() { **return** ‘V’;}  }; |

1. 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.

|  |
| --- |
| **class** Test {  **static void** fun(int i) {}  **void** fun(int i) {}  }; |

1. Parameter declarations that differ only in a **pointer \* versus an array []** are equivalent.

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

1. 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())* | **void** **closure**(){  cout<<" closure called"<<endl;  }  **void** **caller**(**void** fun()){  cout<<"caller called"<<endl;  fun();  }  **int** main(){  **caller**(closure);  return 0;  } |

1. **Parameter declarations** that differ only in the **presence or absence of const and/or volatile** are equivalent. But overloading **works when const parameter is a reference or a pointer**.

|  |  |
| --- | --- |
| *// will not work and throw and error*  **int** f ( **int** x) { **return** x+10;}  **int** f ( **const** **int** x) { **return** x+10;} | *// Will work*  **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;  } |
| *// will work*  **void** fun() **const**{  cout << "fun() const called " << endl;  }  **void** fun(){  cout << "fun() called " << endl;  } |  |

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

|  |
| --- |
| **int** f ( **int** x, **int** y) { **return** x+10;}  **int** f ( **int** x, **int** y = 10) { **return** x+y;} |

# Constructor and Destructor

# *Copy Constructor*

http://quiz.geeksforgeeks.org/copy-constructor-in-cpp/

A copy constructor is a member function which initializes an object using another object of the same class.

|  |
| --- |
| ClassName (**const** ClassName **&**old\_obj); |

# *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**.

*Shallow copy Vs Deep copy*

|  |  |
| --- | --- |
| Default copy constructor | Pointers (or references) of copied object point to new memory locations |
|  |  |

*Copy constructor vs Assignment Operator*

MyClass t1, t2;

MyClass t3 = t1; *// ----> (1) call copy constructor*

t2 = t1; *// -----> (2) call assignment operator*

# *Virtual* Functions

Virtual Constructor

http://www.geeksforgeeks.org/advanced-c-virtual-constructor/

There is **no concept** of Virtual Constructor as C++ being static typed.

Virtual Copy Constructor

http://www.geeksforgeeks.org/advanced-c-virtual-copy-constructor/

There is **no concept** of Virtual Copy Constructor as C++ being static typed.

Need to Discuss with Anil.

Virtual Destructor

Virtual destructor **guarantees that the object of derived class is destructed properly**.

|  |  |
| --- | --- |
| **int** main(void)  {    derived \*d = new derived();    base \*b = d;  **delete** b;    getchar();    return 0;  } | **Constructing base**  **Constructing derived**  **Destructing derived**  **Destructing base** |

Can a destructor be pure virtual in C++?

**Yes**, it is possible to have pure virtual destructor. But it has an body

Class **becomes abstract class when it contains pure virtual destructor.**

|  |
| --- |
| **class** Test{  **public**:  virtual ~Test()=0; *// Test now becomes abstract class*  };  Test::~Test() { }  **int** main(){  Test p; *// cannot declare variable ‘p’ to be of abstract type ‘Test’*  Test\* t1 = new Test; *//* *cannot allocate an object of abstract type ‘Test’*  **return** 0;  } |

Can static functions be virtual in C++?

**No**. staticmember function cannot be **const**and **volatile** also. Static function can have body inside class definition.

*Interface vs abstract classes*

* A class is abstract if it has **at least one pure virtual** function.
* We **can have pointers and references** of abstract class type.
* If we **do not override the pure virtual function in derived class**, then **derived class also becomes abstract class**.
* An abstract class **can have constructors**.

|  |  |
| --- | --- |
| class MyInterface  {  public:  *// Empty virtual destructor for proper cleanup*  virtual ~MyInterface() {}  virtual void Method1() = 0;  virtual void Method2() = 0;  }; | class MyAbstractClass  {  public:  virtual ~MyAbstractClass();  virtual void Method1();  virtual void Method2();  void Method3();  *// Make MyAbstractClass not* ***instantiable***  virtual void Method4() = 0;  }; |
| Interface = **all pure virtual function** | Abstract = **at least one pure virtual function** which make it instantiable (object can’t be created) |

# ****Operator Overloading****

|  |
| --- |
| *// 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;  } |
| *// The global operator function is made friend of this class so*  *// that it can access private members*  friend Complex **operator +** (Complex **const &**, Complex **const** **&**); |

**Below operators can’t be overload**

. (Member Access or Dot operator)

?: (Ternary or Conditional Operator )

:: (Scope Resolution Operator)

.\* (Pointer-to-member Operator)

sizeof (Object size Operator)

typeid (Object type Operator)

Signature for overloading operator in class

|  |
| --- |
| Test & **operator =** (const Test &t); |
| Complex **operator +** (Complex const &obj) |
| *// Overloading* ***[] operator*** *to access elements in array style*  int **&operator[]** (int); |
| friend ostream & **operator <<** (ostream &out, const Complex &c);  friend istream & **operator >>** (istream &in, Complex &c); |
| bool **operator ==** (Complex c) |
| *//* ***Type conversion*** *operator can be a class or primitive type*  **operator double** () |
| *// Overloading* ***dereferencing*** *operator*  int & **operator \***() { return \*ptr; } |

explicit

Use **explicit** keyword on a constructor stop implicit conversions of an object.

|  |
| --- |
| 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) *// w/o conversion it will throw an error*  cout << "Same";  else  cout << "Not Same";  return 0;  } |

# Casting operators

1. const\_cast – remove constantness of the const object
2. static\_cast – Normal Type cast first check type cast is possible then type cast the object
3. dynamic\_cast – It type cast if the object could be cast otherwise return null.
4. reinterpret\_cast – It type cast the memory in to said object. Therefore memory laid must be a valid object. It can be said REINTERPRET the MEMORY for OBJECT.

# constexpr

<https://msdn.microsoft.com/en-us/library/dn956974.aspx>