**PRE-PROCESSOR DIRECTIVES (instructions to compiler) – MACROS/\_\_FILE\_\_/\_\_LINE\_\_/#pragma**

1. Macros are only replaced.
2. Include means copy the include file’s code in the current file.
3. ‘\’ can be used for multiple lines in macros. Don’t use it in the last line else error.
4. Warnings are issued when we redefine/redeclare a preprocessor directive, but in case of variables, error will be thrown.
5. There should be no space in preprocessor directive unit, else whole of the rest of the part will be considered as the replacement string.
6. #pragma once and include guards are used to avoid inclusion of header files more than once.
   1. Writing #pragma once on top of a file will include it only once.
   2. Benefits compared to include guards are:
      1. less code
      2. avoidance of name clashes - suppose for a file system\_calls.h, system\_call\_h has been defined, then erroneously the same macro name could be used for another file.
      3. and sometimes improvement in compilation speed -  improve compilation speed since it is a higher-level mechanism; the compiler itself can compare filenames or [inodes](https://en.wikipedia.org/wiki/Inode) without having to invoke the [C preprocessor](https://en.wikipedia.org/wiki/C_preprocessor) to scan the header for #ifndef and #endif.
7. Use do while in macros for multiline macros. This is because do-while makes it a single entity.
8. Macros arguments are not evaluated and they can cause unexpected results if used in cases like #define square(x) x\*x.

#define INIT\_LIST\_HEAD(ptr) do {\

(ptr)->next = (ptr); (ptr)->prev = (ptr); \

} while (0);

if (foo)

INIT\_LIST\_HEAD(bar); // if it was written as two separate lines, then this would have resulted in compilation error

else

(bar)->next = (bar);

(bar)->prev = (bar);

1. ## is used for merging arguments of macro.
2. # is used for converting macro argument to string.
3. Undefined macros are initialized to 0.
4. #pragma startup and #pragma exit are used to call functions before and after main respectively. Basically, it tells compiler to indicate some rules.

**SINGLETON PATTERN**

In [software engineering](https://en.wikipedia.org/wiki/Software_engineering), the singleton pattern is a [software design pattern](https://en.wikipedia.org/wiki/Software_design_pattern) that restricts the [instantiation](https://en.wikipedia.org/wiki/Instantiation_(computer_science)) of a [class](https://en.wikipedia.org/wiki/Class_(computer_programming)) to one [object](https://en.wikipedia.org/wiki/Object_(computer_science)). We keep constructor private so that no object can be created on its own.

**// Eager instantiation**

**// Create a static object when the class is loaded**

**class** **Singleton** {

**static** Singleton INSTANCE = **new** Singleton();

**Singleton** () {}

**// Only one thread can execute this at a time**

**public:**

**static** Singleton **getInstance**() {

**return** INSTANCE;

}

}

C++ implementation:

class Singleton {

static Singleton \*INSTANCE;

Singleton () {}

// Only one thread can execute this at a time

public:

static Singleton\* getInstance() {

return INSTANCE;

}

};

Singleton\* Singleton::INSTANCE = new Singleton();

**// Lazy Instantiation**

**// Create object when first time request comes**

**class Singleton**

**{**

**static Singleton obj;**

**Singleton() {}**

**// Only one thread can execute this at a time**

**public:**

**static Singleton getInstance()**

**{**

**if (obj==null)**

**obj = new Singleton();**

**return obj;**

**}**

**}**

How to implement multi-ton?

1. Basic idea is put instances in map and return them if already present.
2. The map is static.
3. Always non-NULL object is returned.
4. Objects are returned based on <key> value. In singleton since there is only one object, no need for storing *key*.
5. A simple array of instances where key is the index can also be used to implement multiton.

**FACTORY PATTERN**

We create object without exposing the creation logic to client and the client use the same common interface to create new type of object.

Non-factory:

Object type is chosen at client side. For adding ThreeWheeler() object, client needs to change it and compile again.

class Client {

public:

    Client(int type)  {

        if (type == 1)

            pVehicle = new TwoWheeler();

        else if (type == 2)

            pVehicle = new FourWheeler();

/\* Add ThreeWheeler() case here \*/

        else

            pVehicle = NULL;

    }

}

Factory:

Code at library side. Only library needs to be compiled in this case.

Vehicle::Create(VehicleType type) {

    if (type == VT\_TwoWheeler)

        return new TwoWheeler();

    /\* else if (type == VT\_ThreeWheeler)

        return new ThreeWheeler(); \*/

    else if (type == VT\_FourWheeler)

        return new FourWheeler();

    else return NULL;

}

Client will use the same interface Create(). It would not need to compile its code and would give only the type of the object.

class Client {

public:

    // Client doesn't explicitly create objects

    // but passes type to factory method "Create()"

    Client()

    {

        VehicleType type = VT\_ThreeWheeler;

        pVehicle = Vehicle::Create(type);

    }

}

Note: The client program includes (binds) the file which contains the code of Create(). Whenever there is a change in the Create()function, only the file containing it needs to be compiled again. This scenario is valid in runtime binding.

**Struct in C vs C++**

1. No access specifiers supported in C.
2. Size of empty struct in C is 0, whereas in C++ 1 (to uniquely identify an object).
3. In C++, a struct can have member functions.
4. In C++, a struct can have static members whereas in C it cannot.
5. In C++, we can directly initialize structure data members whereas in C we cannot.

struct Record

{

   int x = 7;

};

1. In C++, struct keyword is not required during declaration of an object.

**Struct vs Class in C++**

1. There’s no difference in struct and class in C++, except in struct members are public by default. Inheritance and function definition, both are there in struct too.
2. When deriving a struct from a class/struct, default access-specifier for a base class/struct is public. And when deriving a class from a class/struct, default access specifier is private.
3. a struct with no modifiers or methods is called a POD struct.
4. use struct as plain-old-data structures without any class-like features, and class as aggregate data structures with private data and member functions.
5. Suppose Foo is a class and Bar is a struct, then both of below expressions would return true.
   1. std::is\_class<Foo>::value
   2. std::is\_class<Bar>::value
6. Declaration of a class (class Foo;) can also be defined by

struct Foo

{

int x;

};

Example 1:

class Base {

public:

    int x;

};

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

int main()

{

  Derived d;

  d.x = 20; // compiler error because inheritance is private

  getchar();

  return 0;

}

Example 2:

class Base {

public:

    int x;

};

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

int main()

{

  Derived d;

  d.x = 20; // works fine because inheritance is public

  getchar();

  return 0;

}

**malloc/calloc/realloc/new/free/delete**

1. malloc is a function which allocates memory and returns void \* pointer to that memory location while new is an operator that allocates the memory, calls the constructor of the class and returns pointer of type <type> in new <type>().
2. malloc doesn’t create instance while new does. In new, memory allocation is done and then constructor is called.
3. When malloc fails, it returns NULL pointer, and new fails: bad\_alloc exception is thrown. This exception must be handled in order to avoid abort.

#include <iostream> // std::cout

#include <new> // std::bad\_alloc

int main () {

try

{

int\* myarray= new int[10000];

}

catch (std::bad\_alloc& ba)

{

std::cerr << "bad\_alloc caught: " << ba.what() << '\n';

}

return 0;

}

1. Both have junk as default values unless there is some constructor defining values in class for new.
2. calloc = malloc + putting 0’s to memory allocated, hence calloc is slow.
3. calloc(n\_blocks, size of 1 block), malloc(size in bytes).
4. realloc allocates a memory of newer size.

void\* realloc (void\* ptr, size\_t size);

* 1. In case it fails to allocate, it returns NULL without destructing previous memory.
  2. The new memory address may or may not be the same.
  3. The contents are copied if the new address is different.
  4. If the size of newer size is 0, it behaves like free and NULL pointer is returned.
  5. If ptr was already NULL, pointer to new allocated memory is returned.

1. free and delete both can be called on NULL pointer. Like opposite of malloc, free just deallocates the memory pointed to by the argument pointer, and like opposite of new, delete deallocates the memory and calls the destructor.
2. When you use free, you are deallocating the memory, that is, you are actually telling the computer that you don't need that space anymore, so it marks that space as available for other data.
3. free or delete cannot be called twice on a pointer. It will lead to program crash.

The pointer still points to that memory address. At this point that same space in the heap can be returned by another malloc call. When you invoke free a second time, you are not freeing the previous allocation, but the new allocation, and this may not be good for your program.

1. calling free for new and delete for malloc is undefined. Similarly realloc on new.
2. new/delete can be overridden, whereas malloc/free cannot be. We can override new/delete to check for counts of new and delete calls or do some extra work like setting the memory with 0.
3. Required size is calculated by compiler in new, whereas in malloc, it should be specified in bytes.
4. By making destructor private, we can restrict creation of object by only new.

If the object is created by new, it is not destructed on its own. It’s the programmers responsibility to destroy it. Hence it won’t throw compiler error as we can call destructor through some other means, but in case of stack variables, it will be destroyed automatically when the variable goes out of scope.

Example:

char \*pBuffer = new char[1024];

// delete pBuffer; //wrong, it may delete only first element

delete[] pBuffer;

* **uint64\_t, uintptr\_t** can be used for printing the decimal values of pointers.
* **Dangling pointers** when a memory is deleted/freed, but the pointer pointing to that memory still points to the deallocated memory.

**Function pointers**

These are basically pointers to functions. They point to the start of executable code and not to any data. No allocation de-allocation in function pointers.

Uses – pass as arguments to functions, return value from functions

used in virtual functions (static vtable).

int \*foo(int);

// wrong declaration, operator precedence comes into play

int (\*foo)(int);

foo = &func;

void (\*func\_ptr\_array[]) (int, int) = {add, sub};

**Pointer and string in C++**

int main(){

char \*ptr = "prashant"; // warning of conversion from string literal to char\* which is non-const

// ptr[0] = 'a'; // this will give bus error at runtime as changing read only section

// ptr[8] is '\0';

// sizeof(ptr) = 8 bytes // as pointer on 64-bit machine

string str = "prashant";

// str[0] = 'a'; // no issue

// str[8] is '\0'

// sizeof(str) // prints 24 this is the standard size of string class. For size of the data (string) stored by str, str.size()

}

**Type conversions**

Type conversion means changing one data type to another. This can be between built-in/user-defined to built-in/user-defined data types (4 combinations).

1. **Implicit type conversion**
   1. Standard type: between built-in to built-in (int, char, float; NULL, 0; etc)

Base\* d = new Derived();

This is standard type because we are changing pointer to another pointer.

* 1. User-defined type: between user-defined data type to built-in/user-defined:

There are 3 ways for type b.

* **Single-argument constructors:** allow implicit conversion from another type to initialize an object. If we put *explicit* in front of constructor, this conversion will not happen. Without explicit, it is a constructor + an implicit converter.
* **Assignment operator:** allow implicit conversion from another type on assignments.
* **Type-cast operator:** allow implicit conversion to a particular type. Converts objects of your class to another type.

// implicit conversion of classes:

#include <iostream>

using namespace std;

class A {};

class B {

public:

B (const A& x) {} // conversion from A (constructor):

B& operator= (const A& x) {return \*this;} // conversion from A (assignment):

operator A() {return A();} // conversion to A (type-cast operator)

};

int main ()

{

A foo;

B bar = foo; // calls constructor

bar = foo; // calls assignment

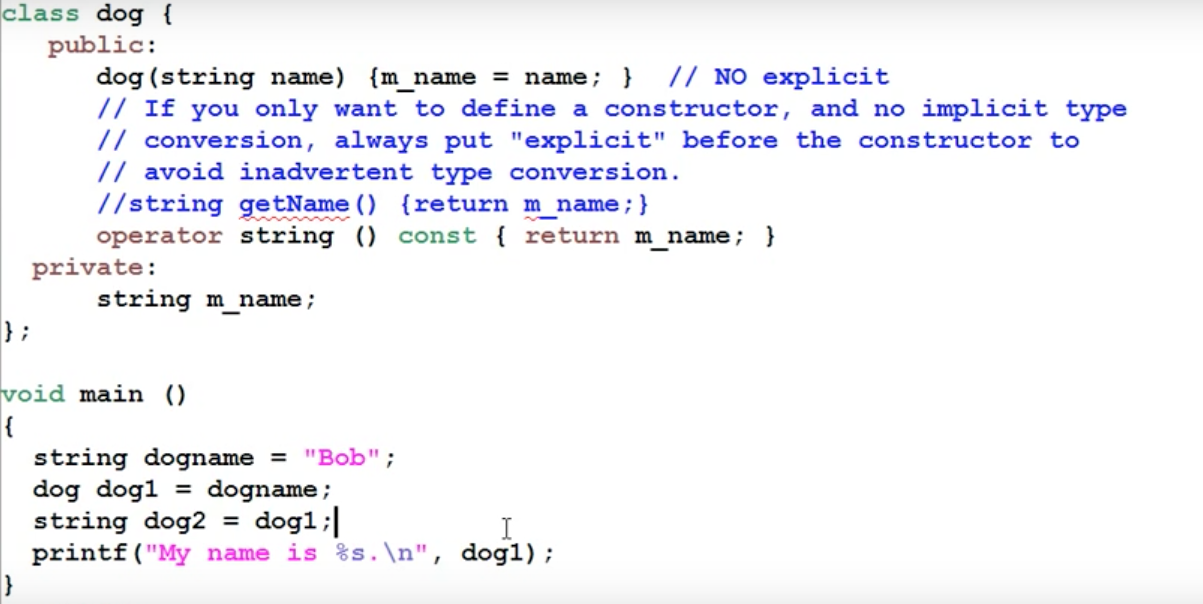
foo = bar; // calls type-cast operator

return 0;

}

If assignment operator was not overloaded, B’s constructor is used for assigning bar = foo.

One more example: Here A is a string class, and B is user-defined type.



Here getName() function returns a string, but since we have defined an implicit converter from our class type to another type, getName() is not needed.

class Dog{

public:

operator string(){

return "Whatsup";

}

};

int main(int argc, char const \*argv[])

{

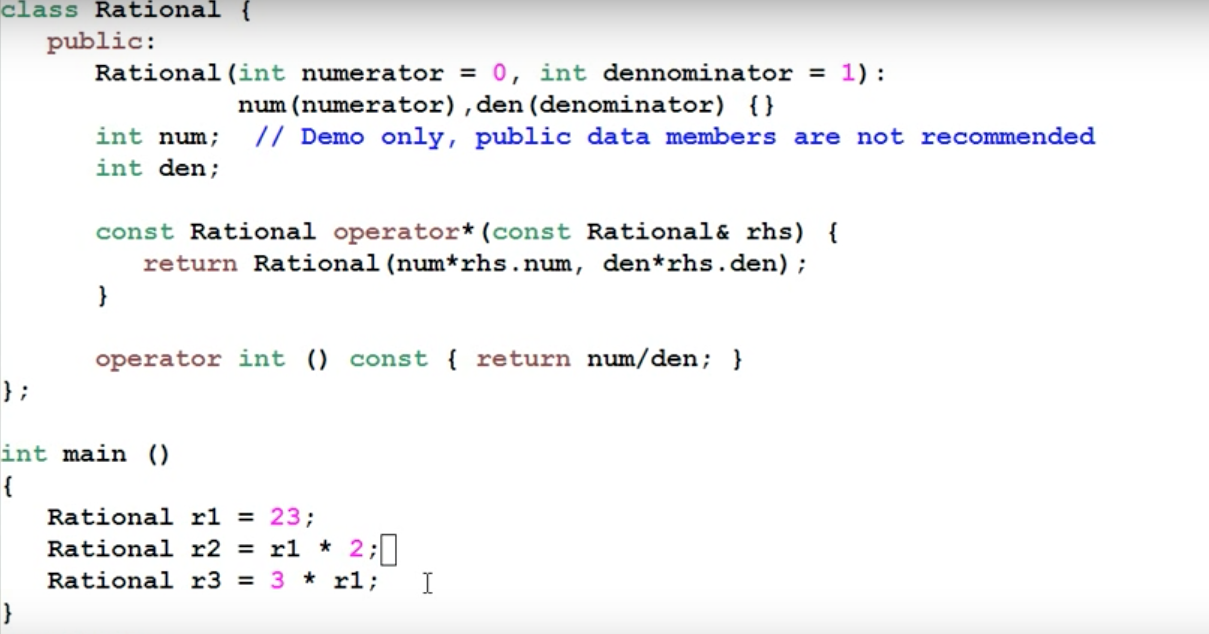
string s = Dog(); // If we are giving only the class name, () is required, else object could also // be assigned

cout<<s<<endl; // Whatsup is printed

return 0;

}

**Use cases of implicit conversions**

****

Points:

* Here an integer is converted to a Rational object implicitly because the constructor still takes 1 argument.
* Second line, r1 \* 2 will work because compiler converts 2 into Rational object and then *sees operator \* in class*.
* Third line, 3 \* r1 will work because compiler returns r1 as an integer due to operator int() converter, and *finds global operator \** and returns the results into r3 after again converting the result into Rational object.

1. **Explicit type conversion**

This is also known as casting.

* 1. **const\_cast**
* const\_cast is used to cast away the constness of variables.

Example 1:

Inside class, "this" pointer is a constant pointer. It does not point to any other location/object. And inside a const member function in a class, this is *const <class\_type\*> const this*, that is, it cannot even change the values of the object’s variables.

But we can cast away the constness of this pointer and change the member variables’ values.

// A const function that changes roll with the help of const\_cast

void fun() const

{

( const\_cast <student\*> (this) )->roll = 5;

}

*Here roll must not be a const type. This is because we are casting away constness of a pointer. This doesn’t mean we can change the value of a const int.*

Consider the below example:

#include <iostream>

using namespace std;

     const int a1 = 40; // global const

int main(void)

{

    const int a1 = 40; // local const

    const int\* b1 = &a1;

    char\* c1 = (char \*)(b1);

    \*c1 = 'A';

    return 0;

}

1. If *a1* was *global constant*, then we would have got Bus error (no compilation error) as *this depends on the constness of the data-type and not the pointer or reference to it.*
2. If *a1* was *local constant*, then *volatile const* would give the updated value whereas only *const* would have given the same cached value. However, it’s implementation defined.

* const\_cast can be used to pass const value to a function that doesn’t receive const. Only pointers or references need constness matching in function arguments.
* const\_cast is only used with pointers and references.
* Consider following code:

const int a = 5;

const int \*ptr = &a;

int \*ptr1 = const\_cast<int\*>ptr;

\*ptr1 = \*ptr1 + 10;

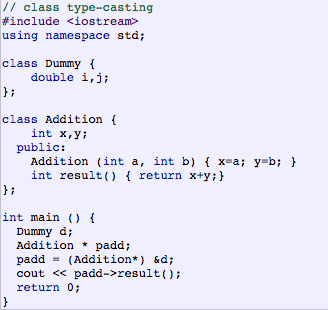
This is undefined behavior although the value in 'a' retains as tested by a program. If ‘a’ was volatile, it would have changed to new value.

* In const\_cast, the data types should be same.

char\* c1 = const\_cast <char \*> (ptr1);

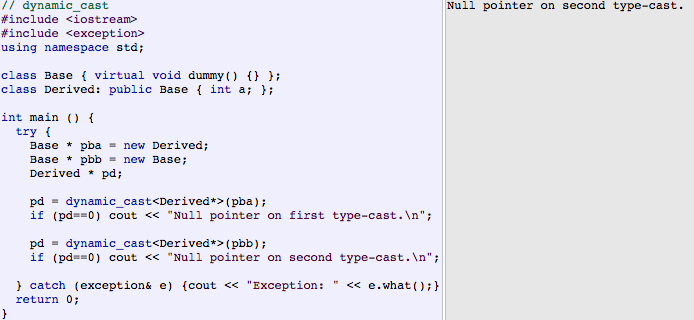
// invalid const\_cast from type 'const int\*' to type 'char\*'

* 1. **dynamic\_cast**
* unrestricted explicit type-casting can lead to errors because any pointer can be converted to any other pointer. Even when two pointers point to different types of objects (unrelated with respect to inheritance), the compiler doesn’t throw any error, but the program would show undefined behavior at runtime. Consider below program:



To control such casting to allow pointer conversions of objects only within related (with respect to inheritance) objects, dynamic\_cast is used. It either does *upcast* (converting from pointer-to-derived to pointer-to-base), or *downcast* (convert from pointer-to-base to pointer-to-derived).

However, downcast doesn’t work in following scenario.



This is because pd is pointing to an incomplete Derived object.

* dynamic\_cast can convert NULL pointers between any types, and any type to void\* pointer but not void\* to any type giving error (source is not a pointer to a class).
* dynamic\_cast downcast only works when there is polymorphism happening between base and derived classes, i.e., at least one virtual function in base class. upcast however doesn’t need this condition. Necessity for virtual function could be taken as:

dynamic\_cast uses run-time type checking which is done via *vptr* which is only present when a class has a virtual function.

Also, if a Base pointer pointing to a Base object is converted by dynamic\_cast to Derived pointer, then Derived pointer will be NULL because the object type is incomplete. It uses RTTI do type checking.

When dynamic\_cast cannot cast a pointer because ***it is not a complete object of the required class*** - it returns a null pointer to indicate the failure. This also means that upcast in dynamic\_cast will succeed if the pointers are related, and downcast will require

* 1. **static\_cast**
* static\_cast doesn’t do type checking at runtime to guarantee that the object being converted is in fact a full object of the destination type
* It can convert from any pointer type to another provided the classes are related by inheritance. It doesn’t allow pointer conversions between unrelated classes.
* static\_cast can convert from one object type to another provided conversion function is defined.
* static\_cast can convert void\* to any other type and reverse and the value will be same when converted back. It can convert any type to void.
* static\_cast unlike dynamic\_cast doesn’t need polymorphism in downcasting.

class Base {};

class Derived: public Base {};

Base \* a = new Base;

Derived \* b = static\_cast<Derived\*>(a);

* static\_cast can also be used to perform any other non-pointer conversion that could also be performed implicitly:

|  |  |  |
| --- | --- | --- |
| double d=3.14159265;  int i = static\_cast<int>(d); | 1 2 |  |

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

* 1. **reinterpret\_cast**
* can convert any type pointer to another type.
* can convert pointer to int provided int is long enough.
* Used at lower level conversions compared to static.

1. When two classes are not related by inheritance, dynamic\_cast and static\_cast will throw error but reinterpret\_cast will simply copy the address.
2. When two classes are unrelated but the first class has a polymorphic function (virtual function), dynamic\_cast will not return error, but it will return NULL.
3. When two classes are related but does not have a virtual function, static and reinterpret cast will work and copy pointer values, but dynamic\_cast will not work in case of downcast. It gives error.
4. When a Base\* is pointing to a Base object, and that is converted to Derived\*, following is behavior:

dynamic\_cast: it returns NULL as object is incomplete. This is due to run-time type checking.

static\_cast: no error since no run-time type checking. Copies pointer value.

reinterpret\_cast: no error. Copies pointer value.

RTTI refers to the C++ mechanism of getting object’s type at runtime. It is available only for classes which are polymorphic, which means they have at least one virtual method. In practice, this is not a limitation because base classes must have a virtual destructor to allow objects of derived classes to perform proper cleanup if they are deleted from a base pointer. This is because RTTI uses *vptr* to get the class type at runtime.

**typeid**

* returns an object of type *type\_info* whose name() function returns a string with type name. It depends on compiler what will get printed.

**decltype**

* extracts the type of an entity or an expression.

int fun1()   { return 10;}

decltype(fun1()) x;

x = fun1();

cout<<typeid(x).name<<endl; // i is printed

**Storing negative numbers**

2’s complement of a *number* that is n-bit large is ***2ˆn - number*.**

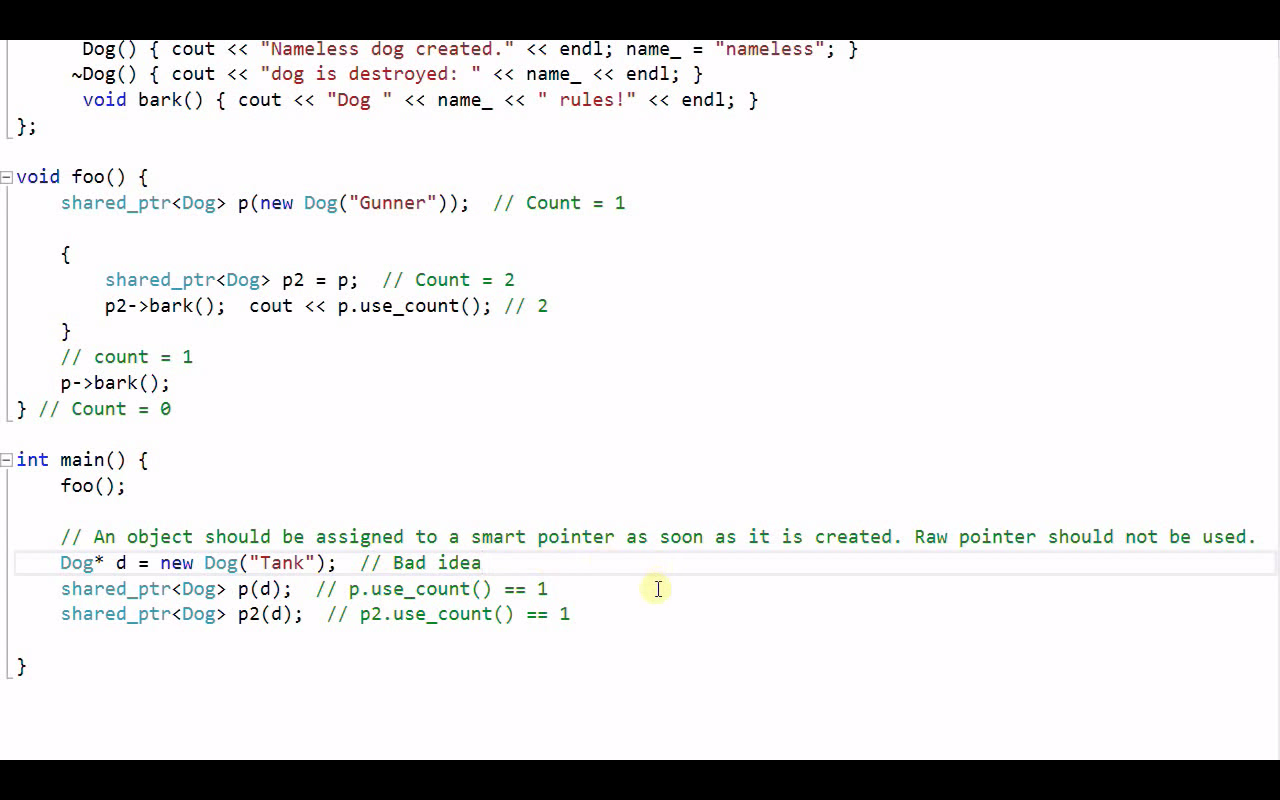
Computer stores negative numbers in 2’s complement. *Hence -1 is stored as 2’s complement of 1 that is 2ˆn – (1) and the MSB is used to store sign as 2^n-(1) will not require the MSB.*

Hence unsigned int a = -1 gives a very large value, 2ˆ32-1 (this occupies 32 bits, all 1’s).

**Shared pointers**

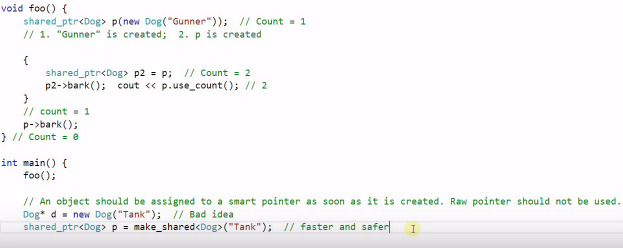
* shared\_ptr creation

shared\_ptr<int> ptr(new int(10));



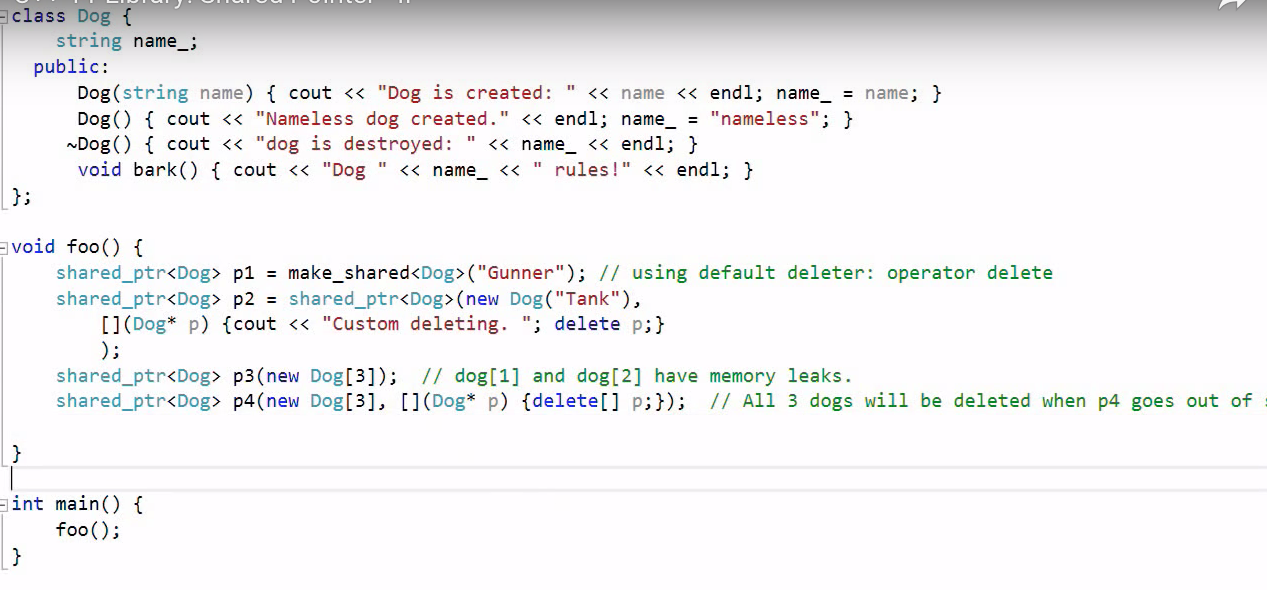
In foo(), correct usage of shared\_ptr is done. While in main(), both *p* and *p2* point to the same memory but the memory has reference counts of 1 instead of having 3 pointers pointing to that memory. So, it will be destroyed if *p* was written within braces and went out of scope.

Hence shared\_ptr should be created either as done in foo(), or the following way.



* shared\_ptr deletion

shared\_ptrs are automatically deleted (their destructor is invoked in case of class pointers), but we can add a custom deleter as shown.



We pass second argument as a lambda function which invokes custom delete upon shared\_ptr deletion and then *delete p* invokes destructor of Dog.

If we don't call delete from custom deleter, then destructor won't be invoked. Custom deleter is used to delete array of shared\_ptrs.

* Using raw pointer from shared\_ptr

1. Suppose we have

shared\_ptr<Dog> p = make\_shared< Dog>(“Gunner”);

1. Then we extract raw pointer from it

Dog \*d = p.get();

Now we need to use d with extra care.

1. We cannot call *delete* on d because when p will go out of scope, the pointer will be deleted again.
2. If we make another shared\_ptr using d, then there are two independent shared\_ptrs referencing same memory. Hence when both go out of scope, pointer will be deleted twice.

**Unique pointer**

Unique pointer unlike shared\_ptr has sole ownership of the allocated memory. When it is assigned to any other memory, the old memory is automatically deleted.

When a pointer is allocated inside a function and if the function throws exception, then without reaching ‘delete pointer’ statement, the function is unwounded. This leads to memory leak.

* Unique pointer creation

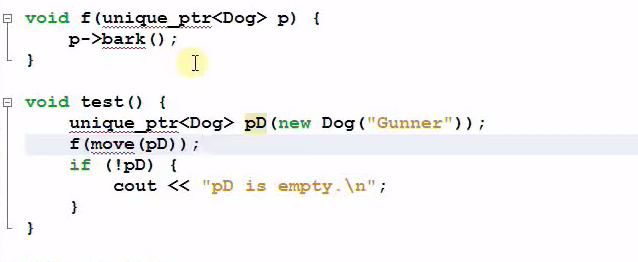
unique\_ptr<int> ptr(new int(10));

Here pt r is a stack variable which will be deallocated when the stack unwinds. Hence, there will be no reference to the allocated memory (new int(10)) and eventually the heap allocated memory will be freed.

int \*ptr2 = ptr.release();

// this does not destroy the memory as it is being pointed by another pointer

// now the memory should be freed by delete(ptr2)



Here, move() changes the ownership from one unique\_ptr to another and doesn’t free.

Sole ownership can be implemented by making operator= in unique\_ptr implementation to be private.

weak\_ptr can only access by weak\_ptr.lock(). This converts it into shared\_ptr.

Doesn’t take ownership.

**Volatile keyword:**

*Volatile means fetching from memory every time.*

Const: value cannot be changed. Program will throw compilation error in C++.

However, in C, this can be changed through pointer to the variable.

Const volatile:

If the value of a const variable is changed with the help of a pointer to the memory location,

* There will be a mismatch in cache value and memory value if Const is changed via its address.
* Volatile means fetch the value from RAM every time. Do not store it in cache. This is useful for avoiding compiler optimizations.
* –O option is used for explicitly asking for optimization.
* Other scenarios could be DMA writing onto a buffer which overwrites the memory.

volatile const int a = 5;

int b = 10;

memset(&b, 0, 100);

cout<<&a<<" "<<&b<<endl; // a and b are not very far

cout<<a<<" "<<b<<endl; // 0 and 0 for both

In the above code, ‘a’ is reflected to 0 only if it’s const and volatile.

Const volatile: The programmer is prevented from modifying the object after its initialization (this is the const part).

* External mechanisms may still modify its value, which therefore must be retrieved from "memory" each time a read is requested rather than being cached by optimizations (this is the volatile part)

**Global const variables** *including pointers* are stored in text section. As a result, modifying them with the help of their pointers would result in Bus error as we are trying to modify read-only memory. It won’t give compiler error though.

In C++, Global const have static linkage. If multiple files want to use a global const, then define it in a common header file and include the same in every translation unit.

**Pass by reference**

Array is a legacy concept inherited from C. In practice, it will be automatically casted to a pointer when we pass it as a function parameter. Thus, it is light weighted enough and there is no point to pass-by-reference. Vector, on the other hand, is a C++ class. Passing it by value typically means calling to the copy-constructor, which is expensive as it clones the entire vector. Thus, we tend to pass it by reference to save the cost.

**Return by reference in C++**

When returned by reference, pointer to the variable (its address) is passed by the function.

1. Local variable reference shouldn’t be returned as it goes out of scope and it is destroyed when the stack is destroyed. However, reference to a local static variable can be returned as its lifetime is program’s lifetime.
2. A const reference can be received by any variable w.r.t. to const-ness or reference/non-reference nature except a non-const reference.
3. If a function is returning a reference which is non-const, func() = 5 is okay, but if it returns a const reference, func() = 5 will give compilation error as a constant reference is being modified.

**Storage classes in C++**

1. Auto. Local variable inside function.
2. Static local. Declaration hits only once. Lifetime is program’s lifetime.
3. Register. Variable will be stored in processor’s register if available. Deprecated since C++11.
4. Extern. It specifies that the variable is only declared in this file and may not be defined. If we don’t use extern, then the variable is defined by default.
5. Global variable. Const global stored in text segment.
6. Thread Local Storage - one instance of the variable per extant thread. Keyword thread\_local is used for this purpose.

**Static members in Class**

Suppose a class A is having a static member object of class B, then this static member object will be allocated memory only when it will be initialized. Till then, it is only declared.

class B

{

public:

int x;

B(){

cout<<"B\n";

}

};

class A

{

public:

static B a;

};

B A::a = B(); // without this, and if used, it gives linker error

int main(int argc, char const \*argv[])

{

A t;

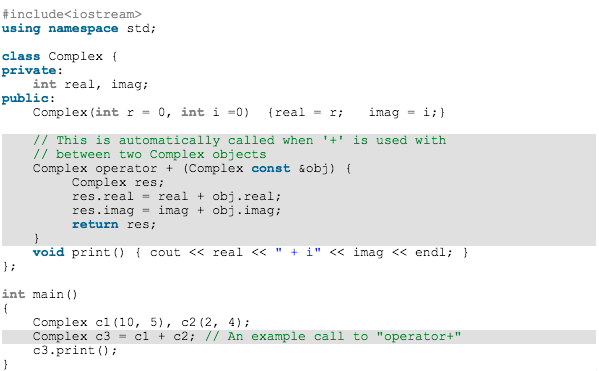
cout<<t.a.x<<endl;

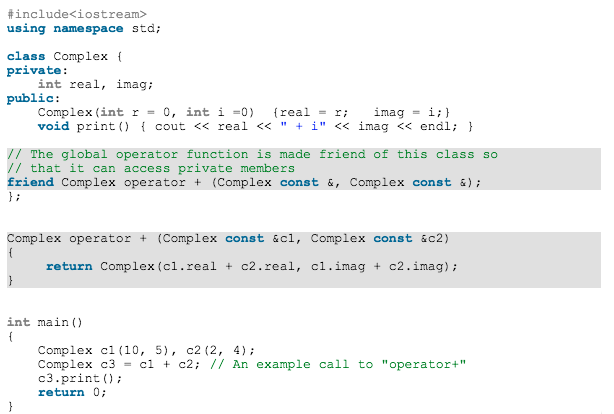
return 0;

}

**Operator Overloading in C++**

1. Copy constructor and overloaded assignment operator: these are implemented to avoid shallow copy.
2. Operator overloading: one of the operands should be of user defined data type.
   1. Creating a class member function for operator overloading.



* 1. Creating a global function for operator overloading.

1. Assignment operator can also be overloaded, and it should be overloaded in case we don’t want shallow copy of objects during assignation.
2. . (dot), ::, ?:, sizeof cannot be overloaded.

**OOPs**

**Inheritance:**

* public members – every public member of the class can be accessed by anywhere in the program with valid scope.
* protected members – only children classes can access those members.
* private members – only that particular class can access private members.
* private inheritance – everything is inherited except private and converted to private for the rest of the program.
* protected inheritance – everything is inherited except private and converted to protected for the rest of the program.
* public inheritance – everything is inherited except private.

**friend class and function:**

* 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;    friend class LinkedList;  // Now class LinkedList can access private members of Node  };  class A  {  int x;  friend class B;  };  class B  {  public:  A a;  void f(){  cout<<a.x;  }  }; |

* Friend Function – when we want to write a function outside a class, but also want to access the protected and the private members of that class, we declare the function as friend inside that class.
* This function is not a member of that class.
* A friend function can be:  
  a) A method of another class  
  b) A global function

1. Friendship is not mutual. If a class A is friend of B, then B doesn’t become friend of A automatically.
2. Friendship is not inherited.

Friend function in another class:

class A;

class B

{

public:

void f (A& a){};

};

class A

{

public:

friend void B::f(A& a); // f has to be a public member of class B

};

Here class B doesn’t need to know full definition of A as it is not accessing any class members yet. However, A needs to know full definition of B as it accessing B::f(A&).

Where are class functions stored?

They may be stored in the same way other functions are stored and would take *this* parameter implicitly that can be used for checking argument types between various functions with same names in different classes or scopes.

**Multiple inheritance –** When a class inherits from more than one parent.

**Diamond problem**. 

1. func\_A () is inherited in both B and C, and two copies are then inherited in D. When an object of D calls func\_A (), there is ambiguity.
2. When object of D is created, constructor of A is called twice.

* Diamond problem doesn’t occur unless an ambiguous call is made.

**Virtual inheritance** *to solve diamond problem*

* To avoid the diamond problem, we make the inheritance of A in B and C to be *virtual*. This ensures that only one copy of A members is inherited by both B and C.
* When D inherits from B and C, all D, B, and C share the same copy of A.
* Constructor of A will be called only once. To call parameterized constructor of A, we have to call it from the class whose object is in the context. For example: if object of D is created, then parameterized constructor of A can only be called from D’s initializer list, and any initializer list in B and C will be ignored.
* Hence, if we use *virtual* keyword, if D’s initializer list doesn’t contain A’s initialization, then the default constructor of grandparent class (A) is called even if the parent classes (B and C) had explicitly called A’s parameterized constructor.

Class A{

A(int);

};

Class B: public virtual A{

B(int): A(int); // initialization is ignored when D’s object is created

};

Class C: public virtual A{

C(int): A(int); // initialization is ignored when D’s object is created

};

Class D{

D (int x, int y, int z): A (int x), B (int y), C (int z){}

};

* <https://en.wikipedia.org/wiki/Virtual_inheritance>

**Runtime Polymorphism**

Polymorphism means one name and multiple definitions. Virtual functions implement the idea of polymorphism. Idea is, [**virtual functions**](http://en.wikipedia.org/wiki/Virtual_function)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. This is also called *Late Resolution* or *Late Binding*. *Dynamic dispatch*is the process of selecting which implementation of a [polymorphic](https://en.wikipedia.org/wiki/Subtyping) operation function to call at [run time](https://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)).

Base \*bp = new Derived;

    bp->show();

    Base &br = \*bp;

    br.show();

In both cases above, derived class function will be called.

This is not valid in case of variables, of course, because they are not overridden.

*vptr* is present in classes that have at least one virtual function. When an object is created, vptr points to vtable which contains pointers to all the functions which are virtual. This pointing happens in the constructor code. vptr is maintained per object and vtable is maintained per class.

Suppose there are two functions in Base class: virtual a(), and normal b() which are also overridden in Derived class. So vtable of Base and Derived will have pointer only to a(). That means only a() will exhibit runtime polymorphism.

*vtables* are static as all the instances of a class can use the same vtable for finding addresses of functions.

**Virtual table internals**

|  |  |
| --- | --- |
| **class** **B1** {  **public**:  **virtual** ~B1() {}  **virtual** void f1() {}*// not overridden*  void f0() {} *// not overridden*  int int\_in\_b1;  }; | **class** **B2** {  **public**:  **virtual** ~B2() {}  **virtual** void f2() {} *// overridden in D*  int int\_in\_b2;  }; |
| **class** **D** : **public** B1, **public** B2 {  **public**:  void d() {}  void f2() {} *// override B2::f2()*  int int\_in\_d;  };  object memory layout of d:  +0: pointer to virtual method table of D (for B1)  +4: value of int\_in\_b1  +8: pointer to virtual method table of D (for B2)  +12: value of int\_in\_b2  +16: value of int\_in\_d  Total size: 20 Bytes.  virtual method table of D (for B1):  +0: B1::f1() *// B1::f1() is not overridden*  virtual method table of D (for B2):  +0: D::f2() *// B2::f2() is overridden by D::f2()* | object memory layout of b2:  +0: pointer to virtual method table of B2 (vptr)  +4: value of int\_in\_b2  virtual method table of B2:  +0: B2::f2()  Overriding of the method f2() in class D is implemented by duplicating the virtual method table of B2 and replacing the pointer to B2::f2() with a pointer to D::f2().  Functions not carrying the keyword virtual in their declaration (such as f0() and d()) do not generally appear in the virtual method table. |

So, in effect, vtables are copied in derived classes and the functions that are overridden, the pointers are replaced. This means only one vptr is used here to point to the vtable for one inheritance. If a Derived class is inheriting from two Base classes, it may be possible that the entries for virtual functions of Derived class (the ones which are present in Derived class only) may be stored in first inherited base class’ vtable and hence, there is no need to replicate in other base class’ vtable.

**Multiple inheritance**

The g++ compiler implements the [multiple inheritance](https://en.wikipedia.org/wiki/Multiple_inheritance) of the classes B1 and B2 in class D using two virtual method tables, one for each base class. This leads to the necessity for "pointer fixups", also called [thunks](https://en.wikipedia.org/wiki/Thunk_(programming)), when [casting](https://en.wikipedia.org/wiki/Type_conversion).

Consider the following C++ code:

D \*d = **new** D();

B1 \*b1 = d;

B2 \*b2 = d;

While d and b1 will point to the same memory location after execution of this code, b2 will point to the location d+8 (eight bytes beyond the memory location of d). Thus, b2 points to the region within d which "looks like" an instance of B2, i.e., has the same memory layout as an instance of B2.

**Diamond inheritance**

struct A { int r; int s; }; struct B { int t; int u; };

struct M: A, B { int v; int w; };

M:

+-----+-----+-----+-----+-----+-----+

| A | B | v | w |

+-----+-----+-----+-----+-----+-----+

Using these diagrams, let's see what happens when casting a derived pointer to a base pointer:

M\* pm = new M();

A\* pa = pm; // points to the A subpart of M

B\* pb = pm; // points to the B subpart of M

M:

+-----+-----+-----+-----+-----+-----+

| A | B | v | w |

+-----+-----+-----+-----+-----+-----+

^ ^

pm pb

pa

When an object of type M is created but the pointer to it if of type B, then *thunking* (pointer resetting) helps in getting the correct offset to the vtable of B subobject.

**Implementing virtual inheritance and solving diamond problem**

struct V { int t; };

struct B: virtual V { int u; };

struct C: virtual V { int v; };

struct D: B, C { int w; };

B and C reserve space for a pointer to V other than their own members, and:

* if you build a stand-alone B, the constructor will allocate a V on the heap, which will be handled automatically
* if you build B as part of a D, the B subpart will expect the D constructor to pass the pointer to the location of V

B: (and C is similar)

+-----+-----+

| V\* | u |

+-----+-----+

D:

+-----+-----+-----+-----+-----+-----+

| B | C | w | A |

+-----+-----+-----+-----+-----+-----+

Compiler would behave as if A was directly inherited by D. Hence, only D is able to call parameterized constructor of A.



The “most derived” class XY alone knows where exactly a sub-object of the virtual base class Base is to be allocated. That’s why it is the most derived class which is responsible for initializing all the sub-objects of virtual base classes.

XY constructors initialize the Base sub-object and pointers to it in X and Y. After that, all the rest members of the classes X, Y and XY are initialized.

Refer VirtualInheritance.cpp in github/prashantmarshal

**Why is vptr not static**

There could be two reasons:

1. If vptr was static, it would be stored in data section.

Considering this code: Base\* ptr = new Derived(); ptr->func();

Now to resolve func() we need vptr so that ptr->vptr->func() can be invoked. ptr points to a memory where vptr is not present, and it has not mechanism for how to get the location of the static vptr.

1. We cannot create any non-static function getType() to fetch vptr as it would itself need a vptr.
2. We cannot create a static getType() to get a static vptr because we don’t know whether we want to call Base class getType() or Derived class getType(). So, we can’t resolve getType().

vptr identifies object’s type at runtime.

**Pure virtual functions** (= 0)

The objective of pure virtual function is to force implementation of this function in derived classes.

Virtual functions can be private. The following program won’t give compilation error.

#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;

}

1. If main wasn’t friend, this would have given compilation error. This is because ptr->func() could point to either Base or Derived, and Base::fun() is private.

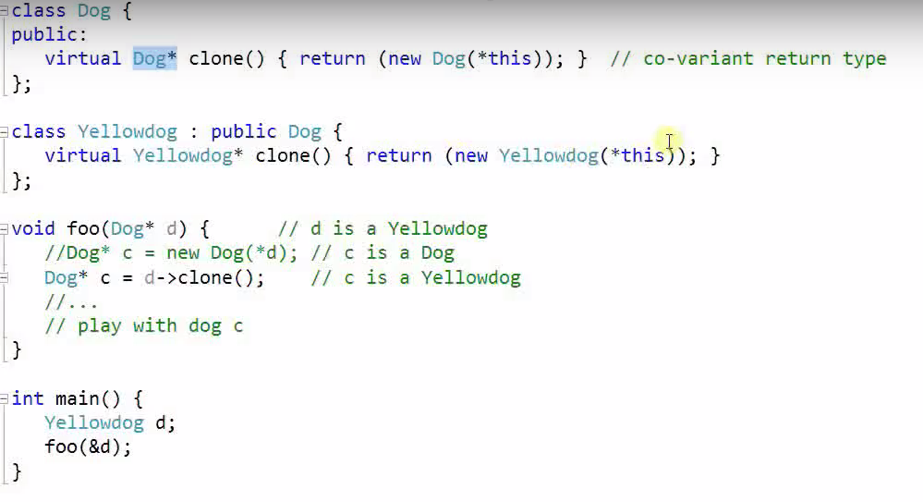
2. If Base \*ptr was Derived \*ptr, main didn’t need to be friend as fun() in Derived isn’t private.

There can’t be **virtual constructor** because:

1. Compile time and run time specifications for an object are same. They do not change, hence there can’t be any runtime polymorphism for constructors.
2. For a virtual constructor, there should be overriding constructor in derived class. That means overriding class itself has the same name unless they are different namespace. It is against OOP.
3. To implement it we need to select which class should be selected at runtime via pointer mechanism, but classes don’t have pointers.

**Virtual constructor – clone ()**

Consider the following scenario, when we need to make a copy object. In foo(), the object in the parameter passed will be different than the object pointed by Dog \*c had clone() wouldn’t have been implemented. So to ensure that even though coding is done by using base pointers only, but during copying of objects via pointers, correct object gets copied, we use clone() function to ensure that the object returned is the correct object type being pointed by the pointer.



<https://www.youtube.com/watch?v=UHP-DKrxgBs>

Another example:

struct A {

virtual ~A() {}

virtual A \* Clone() { return new A; }

};

struct B : public A {

virtual A \* Clone() { return new B; }

};

int main() {

A \* a1 = new B;

A \* a2 = a1->Clone(); // virtual construction

delete a2;

delete a1;

}

**this pointer**

* Hidden argument to all non-static functions. It is not passed in static functions as static functions are not associated with any object. static functions only access static variables.
* *this* is a constant pointer.
* 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’.

**Initializer list**

* to assign values to non-static const class members.
* to assign values to reference variables.
* to assign values to base class members.
* to assign values to other class' subobjects.
* <https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/>

**Static and const**

* *static* functions cannot be virtual as they aren’t tied to any object. They are tied to classes. For example: you cannot polymorphically call class\_name::foo() when you have explicitly bounded foo().
  + But when static functions are called via objects of a class, then it could have made sense to call them polymorphically, but then, it’s just not supported.
* static function can’t be overridden, but it can be hidden by the same name function in derived class.
* A *virtual final* function can’t be overridden. Putting final without virtual will give error in C++. This is because non-virtual functions cannot be overridden, hence no use for keeping them final.
  + The use case of virtual and final together is – suppose you have classes parent, child, grandchild. A certain function foo() in parent is virtual which you want to override in child but not in grandchild. Hence you mark foo() in parent as virtual, and final in child (it will implicitly be virtual). So now foo() in child becomes virtual and final.
* *private* *virtual* functions can be overridden in C++, but not in JAVA. In JAVA, private functions are final by default.
* *static* member functions cannot use *this* pointer to access any non-static data members. They have to create an instance first as they are not associated with any object. “this” pointer is initialized in the constructor code.
* *static* objects are destroyed at program termination and not at the time of returning from function. Local non-static objects are not destroyed if we *exit* from main, but *static* objects will be destroyed. All objects created on heap will not be destroyed whether their pointers are static/non-static unless they are deleted via *delete* or the program exits.
* *static* functions cannot use this pointer. this pointer is passed as a hidden argument which is not passed in case of static functions.
* *const* variables in C++ classes are initialized via *initializer* list.
* non-member functions cannot have 'const' qualifier.
* *static* objects outside *main* are created before *main* is called. Static objects inside *main* will be given address during compile time but their construction will happen only after the code hits their line.

**C++ classes**

* A class cannot have an object of its own type which is non-static in its own class definition. It will show error as incomplete type. However, it can have a static object as they do not contribute to the size of the class. It can also have a pointer as pointers are fixed sizes.
* Forward declaration – a forward declaration is a [declaration](https://en.wikipedia.org/wiki/Declaration_(computer_science)) of an [identifier](https://en.wikipedia.org/wiki/Identifier_(computer_programming)) for which the programmer has not yet given a complete [definition](https://en.wikipedia.org/wiki/Definition). It is required for a [compiler](https://en.wikipedia.org/wiki/Compiler) to know certain properties of an identifier (size for [memory allocation](https://en.wikipedia.org/wiki/Memory_allocation)), but no other details, like the particular value it holds or definition (in case of functions).
  + For instance, if a class A has a member object of type class B and B is forward declared, then this will give compiler error as compiler doesn’t know size of class B. This is because compiler needs to know size of class A. However, we can use pointers or references of a forward-declared data type.
  + In case of functions only declarations are allowed to use a forward-declared type. It throws error if defined.

**Overloading and Overriding:**

* Overloaded functions are static bounded. Compiler uses the reference type or pointer type of the declared variable to resolve binding.
* Overriding means giving different definition across inheritance (in derived classes). Overridden functions are dynamic bounded. In Java, @Overridden is used while in C++, virtual is used.

**Why destructor call order is reverse?**

1. The derived class destructor may access base class members, so it needs the base class object to be in a valid state.
2. Suppose you have a class Y which inherits from X, and a class Z that inherits from Y. By the principles of object-oriented inheritance, every Y is an X. And every Z is a Y and an X. If base object part is destroyed first, it violates above rule.

**Little and Big endianness** can be calculated in the following way:

Lower values on lower address is Little endianness, and big endianness is reverse.

#include <stdio.h>

int main()

{

unsigned int i = 1;

char \*c = (char\*)&i;

    if (\*c)

        printf("Little endian");

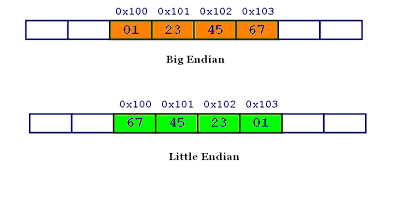
    else

        printf("Big endian");

    getchar();

    return 0;

}



To reverse:

int myreversefunc(int num)  
{  
int byte0, byte1, byte2, byte3;   
  
byte0 = (num & x000000FF) >> 0 ;  
byte1 = (num & x0000FF00) >> 8 ;  
byte2 = (num & x00FF0000) >> 16 ;  
byte3 = (num & xFF000000) >> 24 ;  
  
return((byte0 << 24) | (byte1 << 16) | (byte2 << 8) | (byte3 << 0));  
}

Another solution:

short convert\_short(short in)

{

short out;

char \*p\_in = (char \*) &in;

char \*p\_out = (char \*) &out;

p\_out[0] = p\_in[1];

p\_out[1] = p\_in[0];

return out;

}

long convert\_long(long in)

{

long out;

char \*p\_in = (char \*) &in;

char \*p\_out = (char \*) &out;

p\_out[0] = p\_in[3];

p\_out[1] = p\_in[2];

p\_out[2] = p\_in[1];

p\_out[3] = p\_in[0];

return out;

}

**Constructors and Destructors:**

Once we define our own constructors and destructors, the default constructors are no longer available. If we have to create an object, then we need to have a corresponding C-or D-or signature present in the class.

* Crashing C++ program before main: If we write crash prone code in a constructor and define global objects, then program will crash before main() is called.

**explicit**

class Foo

{

public:

Foo(int \_x, int \_y = 0){}

private:

int x;

int y;

};

Foo bar1(10);

Foo bar2 = 20; // In this, the compiler does implicit conversion and calls Foo’s constructor with \_x = 20, \_y = 0

Suppose we have a pointer in our class. If we have to create a copy object, that pointer will be copied too. This way whenever pointer value is changed in one object, it will be changed in another too. By having our own *copy constructor,* we can allocate new memory for the pointer and avoid the aforementioned scenario.

Assignment operator and Copy constructor should be defined when the class contains pointers.

**Copy constructor and Return value optimization and Copy Elision**

Since a const cannot be bound to a non-const reference (because this non-const reference can illegally change the const’s value), a temporary object can be bound to a const reference but not to a non-const reference (because a temporary object is a const object, and it can be destroyed any time after its usage).

* Types of copy constructors:

X(**const** X& copy\_from\_me);

X(X& copy\_from\_me);

X(**volatile** X& copy\_from\_me);

X(**const** **volatile** X& copy\_from\_me);

X(X& copy\_from\_me, int = 0);

X(**const** X& copy\_from\_me, double = 1.0, int = 42);

1.

X a = X();

This is valid given X(const X& copy\_from\_me) but not valid given X(X& copy\_from\_me) because the second wants a non-const X& to create a, the compiler first creates a temporary by invoking the default constructor of X, then uses the copy constructor to initialize a as a copy of that temporary.

Temporary objects created during program execution are always of const type. So, const keyword is required*.*

2.

**const** X a;

X b = a;

This is valid given X(const X& copy\_from\_me) but not valid given X(X& copy\_from\_me) because the second wants a non-const X&

* Return value optimization:

The following cases may result in a call to a copy constructor:

1. When an object is passed (to a function) by value as an argument or returned by value.
2. When an object is thrown, caught
3. When an object is placed in a brace-enclosed initializer list

How are large objects returned by a function?

Whenever a large object is to be returned by a function, compiler may perform a trick. This could be keeping an object’s reference onto the function’s stack and assigning the return value to that reference, and then using that reference to assign to the lvalue (which was to be assigned the function’s return value).

**User code:**

struct Data {

char bytes[16];

};

Data f () {

Data result = {};

return result;

}

int main () {

Data d = f ();

}

**Optimized code:**

Data \* f (Data \* \_hiddenAddress) {

Data result = {};

\*\_hiddenAddress = result;

return \_hiddenAddress;

}

int main () {

Data \_hidden; // create hidden object

Data d = \*f(&\_hidden); // copy the result into d

}

* Copy Elision - copy elision refers to a [compiler optimization](https://en.wikipedia.org/wiki/Compiler_optimization) technique that eliminates unnecessary [copying of objects](https://en.wikipedia.org/wiki/Object_copy)

#include <iostream>

struct C {

C() {}

C(const C&) { std::cout << "A copy was made.\n"; }

};

C f() {

return C();

}

int main() {

C obj = f();

}

Output of above code could be:

Hello World!

A copy was made.

A copy was made.

Hello World!

A copy was made.

Hello World!

1.

T foo(){

return T(“\_attributes\_”);

}

int main(){

T b1 = foo();

}

Here, copy constructor isn’t invoked and also, constructor is invoked once. The possible optimization could be:

T b1(“\_attributes\_”); instead of T b1 = foo();

2.

Suppose in main(), it was like this:

T b1; b1 = foo();

Then constructor would have got invoked for b1, and foo(), and then assignment operator would have got invoked for assigning foo() to b1.

3.

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.

To prevent optimization,

g++ copy\_elision.cpp -fno-elide-constructors

<https://www.geeksforgeeks.org/copy-elision-in-c/>

**Virtual functions in constructor:**

Suppose there is some virtual function being called from the constructor in Base class, and that function is overridden in Derived class. So, if the pointer is of Base class, but the object is of Derived, during constructor call of Base class, Base class’s function will be called as the Derived class constructor isn’t invoked yet, or in other words, Derived class part of the object isn’t created yet.

Similarly, if some virtual function is called from Base class's destructor, Base class's function will be called. This is because when a Derived class object is destroyed, Derived class' destructor is called first and then the Base class's. Hence, by that time, only Base class's part will be left in the object.

<https://www.youtube.com/watch?v=mTE5jaXaOuE&index=8&list=PLE28375D4AC946CC3>

**Private Constructors**

1. Using friend class: when we want only friend class to create objects of this class. friend class can access constructor of this class and hence can create objects as its subobjects.
2. Singleton pattern.
3. Named constructor.

**Private Destructors**

Destructors can be private. In such a scenario, an object can be created only if its destruction doesn’t depend upon destructor’s privacy.

<https://www.geeksforgeeks.org/playing-with-destructors-in-c/>

Use of private destructor?

Basically, any time you want some other class to be responsible for the life cycle of your class' objects, or you have reason to prevent the destruction of an object, you can make the destructor private.

For instance, if you're doing some sort of reference counting thing, you can have the object (or manager that has been "friend"ed) responsible for counting the number of references to itself and delete it when the number hits zero. A private dtor would prevent anybody else from deleting it when there were still references to it.

#include <iostream>

class a {

~a() {}

friend void delete\_a(a\* p);

};

void delete\_a(a\* p) {

delete p;

}

int main()

{

a \*p = new a;

delete\_a(p);

return 0;

}

Such objects cannot be created on stack.

**Pure virtual destructor**

It can make a class abstract, but its definition is to be provided because when an object is destructed, Base class needs to have destructor defined.

**Destructor and Exception Handling**

#include <iostream>

using namespace std;

class A

{

public:

A(){

printf("A constructor\n");

}

~A(){

printf("A destructor\n");

}

};

int main(void)

{

A a;

try {

A b;

}catch(...){

printf("Inside catch\n");

}

printf("Program\n");

}

A constructor

A constructor

A destructor

Program

A destructor

Throwing exception from destructor can cause program crashing when there are multiple objects going out of scope at a time. This is because destructors of all the objects will be called one by one, and while the second destructor will throw an exception, the exception from the first one would have already been present, and two exceptions cannot be handled at the same time in C++.

**Deep copy and Shallow copy**

When one object is assigned to another, all members are copied, but in case of pointers, they refer to the same memory in the older object. So, that’s *shallow copy*. User defined copy constructors and overloaded assignment operators can allow *deep copy*.

But this is not the case in case of arrays. They are always deep copied.

**Java Exception Handling**

* finally block always executed unless exit is called.
* Chained exceptions in Java – one exception can throw
* catch block hierarchy in multiple catch blocks one after the another.
* throw, throws
* when there are multiple catches of base class object and derived class object, derived class object exception should come first, and then base class. Else error will be thrown. In C++, error won’t be thrown but derived class object catch block will not be reached.
* User-defined custom exceptions can be created in Java by extending Exception class.
* We can rethrow exception from catch block, and also from one method to another.
* Exception and Error both are sub classes of java.lang.Throwable class. Error is different from exception in the sense they are mostly system side, like OutOfMemoryError, StackOverflowError, etc.

**Data structure alignment and padding**

*Data alignment* refers to aligning elements according to their natural alignment. To ensure natural alignment, it may be necessary to insert some *padding* between structure elements or after the last element of a structure.

A memory address *a*, is said to be *n-byte aligned* when *a* is a multiple of *n* [bytes](https://en.wikipedia.org/wiki/Byte) (where *n* is a power of 2). A memory access is said to be *aligned* when the size of data is *n* bytes long and its address is *n*-byte aligned. Byte memory accesses are always aligned.

1. A computer accesses memory by one *word* at a time. If the word size is as large as the largest [primitive data type](https://en.wikipedia.org/wiki/Primitive_data_type) supported by the computer, aligned accesses will always access a single memory word. For example: if the word size if 4 bytes and unsigned long is 8 bytes, then 2 words are required to be accessed, i.e., processor needs to read two times (two words) from memory.
2. Suppose for reading a variable whose address is unaligned, let say, 12 to 20, so the processor reads the variable in two accesses – first till 16 and then from 16 to 20. But by the time 16 to 20 is read, it got changed by a device. This leads to insufficient read followed by incorrect read.
3. The alignment of any struct is equal to the largest alignment of its members.

#pragma pack instructs the compiler to pack structure members with particular alignment. Argument can be given.

struct Test

{

char AA;

int BB;

char CC;

};

The compiler could choose to lay the struct out in memory like this:

| 1 | 2 | 3 | 4 |

| AA(1) | pad.................. |

| BB(1) | BB(2) | BB(3) | BB(4) |

| CC(1) | pad.................. |

sizeof(Test) would be 4 × 3 = 12, even though it only contains 6 bytes of data.

#pragma pack(1), the struct above would be laid out like this:

| 1 |

| AA(1) |

| BB(1) |

| BB(2) |

| BB(3) |

| BB(4) |

| CC(1) |

And sizeof(Test) would be 1 × 6 = 6.

With #pragma pack(2), the struct above would be laid out like this:

| 1 | 2 |

| AA(1) | pad.. |

| BB(1) | BB(2) |

| BB(3) | BB(4) |

| CC(1) | pad.. |

And sizeof(Test) would be 2 × 4 = 8.

**struct** MixedData

{

char Data1;

short Data2;

int Data3;

char Data4;

};

Padding:

**struct** MixedData */\* After compilation in 32-bit x86 machine \*/*

{

char Data1; */\* 1 byte \*/*

char Padding1[1]; */\* 1 byte for the following 'short' to be aligned on a 2-byte boundary*

*assuming that the address where structure begins is an even number \*/*

short Data2; */\* 2 bytes \*/*

int Data3; */\* 4 bytes - largest structure member \*/*

char Data4; */\* 1 byte \*/*

char Padding2[3]; */\* 3 bytes to make total size of the structure 12 bytes \*/*

};

Reordering (by compilers):

**struct** MixedData */\* after reordering \*/*

{

char Data1;

char Data4; */\* reordered \*/*

short Data2;

int Data3;

};

**Bitfields**

Bitfields are used to specify how much space a variable is going to take so that optimization could be done.

* <type> <variable>: <value> is used to specify the length = value.
* Alignment is according to the largest data type present in bitfield or normally.

struct s

{

short b: 15; // address 0 byte

char x: 1; // address 15th bit

int: 0; // start from next address = address 0 + 4 bytes

short a: 1; // address 4 byte

};

Size of struct = 6

struct s

{

short b: 15; // address 0

char x: 1; // address 15th bit

long long: 0; // start from next address = address 0 + 8

short a: 1; // address 8 byte

};

Size of struct = 10

* If we give more size than data type, the compiler throws error. If we give more value than size, then it’s implementation dependent.
* Bitfields cannot be static nor they can be in array.

|  |
| --- |
| x |
| padding |
| y |
| y |
| z |
| padding |

struct test

{

unsigned int x;

long int y: 33;

unsigned int z;

};

Suppose each entry is 4 bytes

Padding = 4 bytes

x = 0, y = 8, z = 16

Size is 24 bytes.

struct test

|  |
| --- |
| x |
| y |
| z |
| padding |

{

unsigned int x;

long int y: 1;

unsigned int z;

};

Size is 16 bytes

clang -cc1 -fdump-record-layouts BitFields.cpp

**In structure every element should be aligned according to its own natural alignment and the structure in whole too.**

**Templates**

Geeksforgeeks

**Garbage Collection**

* When all the references to an object are removed from a program, there is a daemon thread called garbage collector which calls System.gc() to clear the object memories. However, collection of objects as soon as they are unreferenced can’t be guaranteed.