**FRIEND FUNCTION:**

A friend functionis a function that can access the private members of a class as though it were a member of that class. In all other regards, the friend function is just like a normal function. A friend function may be either a normal function, or a member function of another class.

A function can be friend of more than 1 class

It is also possible to make an entire class a friend of another class. This gives all of the members of the friend class access to the private members of the other class

**OPERATOR OVERLOADING:**

The following rules of thumb can help you determine which form is best for a given situation:

If you’re overloading assignment (=), subscript ([]), function call (()), or member selection (->), do so as a member function.

If you’re overloading a unary operator, do so as a member function.

If you’re overloading a binary operator that modifies its left operand (e.g. operator+=), do so as a member function if you can.

If you’re overloading a binary operator that does not modify its left operand (e.g. operator+), do so as a normal function or friend function.

the parenthesis operator must be implemented as a member function

**overloading<< operator**

friend std::ostream& operator<< (std::ostream &out, const Point &point)

{

out<<point.member1<<point.member2;

}

**overloading>>operator**

friend std::istream& operator>> (std::istream &in, Point &point)

{

in>>point.member1;

in>>point.member2;

However, we are not able to overloaded operator<< as a member function. Why not? Because the overloaded operator must be added as a member of the left operand. In this case, the left operand is an object of type std::ostream. std::ostream is fixed as part of the standard library. We can’t modify the class declaration to add the overload as a member function of std::ostream.

This necessitates that operator<< be overloaded as a friend.

Similarly, although we can overload operator+(Class, int) as a member function (as we did above), we can’t overload operator+(int, Class) as a member function, because int isn’t a class we can add members to.

**COMPOSITION**:

process of building complex objects from simpler ones is called composition

You simply create objects of your existing

class inside the new class. This is called composition because the new

class is composed of objects of existing classes.

#define USEFUL\_H

class X {

int i;

public:

X() { i = 0; }

void set(int ii) { i = ii; }

int read() const { return i; }

int permute() { return i = i \* 47; }

};

#endif // USEFUL\_H ///

Composition.cpp

// Reuse code with composition

#include "Useful.h"

class Y {

int i;

public:

X x; // Embedded object

Y() { i = 0; }

void f(int ii) { i = ii; }

int g() const { return i; }

};

int main() {

Y y;

y.f(47);

y.x.set(37); // Access the embedded object

}

//Composition2.cpp

// Private embedded objects

#include "Useful.h"

class Y {

int i;

X x; // Embedded object

public:

Y() { i = 0; }

void f(int ii) { i = ii; x.set(ii); }

int g() const { return i \* x.read(); }

void permute() { x.permute(); }

};

int main() {

Y y;

y.f(47);

y.permute();

}

When a composition is destroyed, all of the subobjects are destroyed as well.

**AGGREGATION**:

An aggregation is a specific type of composition where no ownership between the complex object and the subobjects is implied. When an aggregate is destroyed, the subobjects are not destroyed.

In composition class A uses private or public object of class B, hence the complex object is destroyed along with the lifetime of class A.

In aggregation, complex object is created such that it lives out of the scope of aggregate class A (inside a member function or using pointer variables that point to an object.)

b

#include<iostream>

using namespace std;

class B

{

public:

B(int id)

{

this->ID = id;

cout<<"construct a B object: "<<id<<endl;

}

~B()

{

cout<<"destruct the B class object: "<<this->ID<<endl;

}

private:

int ID;

};

class C

{

public:

C()

{

cout<<"construct C object"<<endl;

}

~C()

{

cout<<"destruct C object"<<endl;

}

private:

};

class A

{

public:

A()

{

cout<<"construct A object"<<endl;

}

~A()

{

cout<<"destruct A object"<<endl;

}

void Aggregation()

{

bPtr = new B(1);

}

private:

B\* bPtr;//aggregation

C composition;//composition

};

void main()

{

cout<<"Demo for class aggregation relationship"<<endl;

A a;

a.Aggregation();

}

e.g.2.

#include <string>

using namespace std;

class Teacher

{

private:

string m\_strName;

public:

Teacher(string strName)

: m\_strName(strName)

{

}

string GetName() { return m\_strName; }

};

class Department

{

private:

Teacher \*m\_pcTeacher; // This dept holds only one teacher

public:

Department(Teacher \*pcTeacher=NULL)

: m\_pcTeacher(pcTeacher)

{

}

};

int main()

{

// Create a teacher outside the scope of the Department

Teacher \*pTeacher = new Teacher("Bob"); // create a teacher

{

// Create a department and use the constructor parameter to pass

// the teacher to it.

Department cDept(pTeacher);

} // cDept goes out of scope here and is destroyed

// pTeacher still exists here because cDept did not destroy it

delete pTeacher;

}

**Compositions:**

Typically use normal member variables

Can use pointer values if the composition class automatically handles allocation/deallocation

Responsible for creation/destruction of subclasses

**Aggregations**:

Typically use pointer variables that point to an object that lives outside the scope of the aggregate class

Can use reference values that point to an object that lives outside the scope of the aggregate class

Not responsible for creating/destroying subclasses

**ASSOCIATION:**

There is no parent/child relationship between 2 classes but they can communicate with each other. (may be through anaother class)

e.g.:

Class A

{

};

Class B

{

};

Class C

{

A \*a =new A(this);

B \*b =new B(this);

public:

A\* method()

{

return a;

}

}

In above e.g. Class B can access class A methods as itwill get A's ptr through C's method.

Hence Class A and B have association relationship. Class A and C or B and C have composition relationship.

**COPY CONSTRUCTOR:**

**SHALLOW COPY:**

copies each member of a class individually. Member to member copy. Useful when classes are simple i.e. Without dynamic memory allocation

for classes which have dynamically memory allocated members shallow copies of a pointer just copy the address of the pointer -- it does not allocate any memory or copy the contents being pointed to.

So both copied pointer and original pointer points to same memory location and change in value pointed by original ptr will also change value at copied ptr. This is disadvantage.

**DEEP COPY:**

A deep copy allocates memory for the copy and then copies the actual value, so that the copy lives in distinct memory from the source. This way, the copy and source are distinct and will not affect each other in any way. Doing deep copies requires that we write our own copy constructors and overloaded assignment operators. Because default copy constructor and assignment operators do shallow copy.

Here we will first allocate the memory to hold copy and then will manually copy the content.

The default copy constructor and default assignment operators do shallow copies, which is fine for classes that contain no dynamically allocated variables.

* Classes with dynamically allocated variables need to have a copy constructor and assignment operator that do a deep copy.
* Favor using classes in the standard library over doing your own memory management.

To write own copy constructor and assignment operator : <http://www.learncpp.com/cpp-tutorial/915-shallow-vs-deep-copying/>

Copy constructor is called when a new object is created from an existing object, as a copy of the existing object (see [this](http://geeksforgeeks.org/?p=6883)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);" |

**INHERITANCE:**

First, the base class sets its access specifiers. The base class can always access its own members. The access specifiers only affect whether outsiders and derived classes can access those members.

Second, derived classes have access to base class members based on the access specifiers of the immediate parent. The way a derived class accesses inherited members is not affected by the inheritance method used!

Finally, derived classes can change the access type of inherited members based on the inheritance method used. This does not affect the derived classes’ members, which have their own access specifiers. It only affects whether outsiders and classes derived from the derived class can access those inherited members.

**Diamond problem:**

class A

{

};

class B: public A

{

};

class C: public A

{

};

class D: public B, public C

{

};

when object of class D is created, 2 copies of A will get created, A will be constructed twice or will givw compilaton error. To solve this, insert the “virtual” keyword in the inheritance list of the derived class. This creates what is called avirtual base class, which means there is only one base object that is shared.

therefore

class A

{

};

class B: virtual public A

{

};

class C: virtual public A

{

};

class D: public B, public C

{

};

obj of D will now construct class A once only. But in D constructor we also pass parametes to A constructor :

<http://www.learncpp.com/cpp-tutorial/118-virtual-base-classes/>

It is possible to change the access specifier of base class member in derived class. e.g we can change protected member of a base class to public or private in derived class.

**you can never change the access specifier of a base member from private to protected or public, because derived classes do not have access to private members of the base class.**

**If a ptr or reference of base class is assigned with an object of derived, the ptr will call methods of base class only even though those are overriden in derived.**

**It turns out that because rBase and pBase are a Base reference and pointer, they can only see members of Base (or any classes that Base inherited). So even though Derived::GetName() is an override of Base::GetName(), the Base pointer/reference can not see Derived::GetName(). Consequently, they call Base::GetName(), which is why rBase and pBase report that they are a Base rather than a Derived.**

**See e.g inheritance.cpp for understanding**

**VIRTUAL FUNCTIONS AND VIRTUAL DESTRUCTORS:**

A virtual function is a special type of function that resolves to the most-derived version of the function with the same signature. This capability is known as polymorphism

(Binding refers to the process that is used to convert identifiers (such as variable and function names) into machine language addresses. )

Early binding: direct function calls, static or compile time binding

While prforming Early Binding the compiler can ensure at compile time that the function will exist and be callable at runtime

Advantage: performance and ease of development.

Late binding: function calls with function pointer, dynamic binding

Advantage: Objects of this type can hold references to any object,

Most script languages use late binding, and compiled languages use early binding.

Method Overloading happens at compile time (Early Binding) while Overriding happens at runtime (Late Binding). In method overloading, method call to its definition has happens at compile time (Static Binding) while in method overriding, method call to its definition happens at runtime (Dynamic Binding).

Virtual table(vtable):

The virtual table is a lookup table of functions used to resolve function calls in a dynamic/late binding manner

A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.

Compiler adds \*\_vptr pointer to class which points to the vitual table of that class.

when these virtual tables are filled out, each entry is filled out with the most-derived function an object of that class type can call.

Pure virtual functions and abstract class:

virtual function = 0 in base. Must be redefined in derived. Can not instantiate base class due to pure virtual function -> abstract class. If derived class doesnt define pure virtual function, then it is also abstract class

Interface classes

An interface class is a class that has no members variables, and where all of the functions are pure virtual! In other words, the class is purely a definition, and has no actual implementation. Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

**Functions tht are never inherited**

* Constructors and Destructors are never inherited and hence never overrided.
* Also, assignment operator = is never inherited. It can be overloaded but can't be inherited by sub class.

**UPCASTING:**

**Upcasting** is converting a derived-class reference or pointer to a base-class. In other words, upcasting allows us to treat a derived type as though it were its base type. It is always allowed for **public** inheritance, without an explicit type cast

e.g.

Class Shape{

};

class circle:public Shape{

};

class square:public Shape{

};

void play(**Shape& s)**

**{**

s.draw();

s.move();

}

void main()

{

circle c;

square sq;

play(c);

play(sq)

}

here function play expects obj(ptr) of shape but we are passing it's children. This is upcasting. Circle is automatically upcasted to parent shape.

**class Parent {**

**public:**

**void sleep() {}**

**};**

**class Child: public Parent {**

**public:**

**void gotoSchool(){}**

**};**

**int main( )**

**{**

**Parent parent;**

**Child child;**

**// upcast - implicit type cast allowed**

**Parent \*pParent = &child;**

**// downcast - explicit type case required**

**Child \*pChild = (Child \*) &parent;**

**pParent -> sleep();**

**pChild -> gotoSchool();**

**return 0;**

**}**

**Upcasting** can cause [object slicing](http://www.bogotobogo.com/cplusplus/slicing.php) when a derived class object is passed by value as a base class object

**DOWNCASTING:**

The opposite process, converting a base-class pointer (reference) to a derived-class pointer (reference) is called **downcasting**. Downcasting is not allowed without an explicit type cast. The reason for this restriction is that the **is-a**relationship is not, in most of the cases, symmetric. A derived class could add new data members, and the class member functions that used these data members wouldn't apply to the base class.

**In above e.g. If we change**

**Child \*pChild = &parent; // actually this won't compile**

// error: cannot convert from 'Parent \*' to 'Child \*'

Downcasting is done with the help of dynamic\_cast

**Dynamic casting:**

<http://www.bogotobogo.com/cplusplus/dynamic_cast.php>

## static\_cast< Type\* >(ptr)

This takes the pointer in ptr and tries to safely cast it to a pointer of type Type\*. This cast is done at compile time. It will only perform the cast if the type types are related. If the types are not related, you will get a compiler error

## dynamic\_cast< Type\* >(ptr)

This again tries to take the pointer in ptr and safely cast it to a pointer of type Type\*. But this cast is executed at runtime, not compile time. Because this is a run-time cast, it is useful especially when combined with polymorphic classes. In fact, in certian cases the classes must be polymorphic in order for the cast to be legal.

Reintrerprete cast:

between two differenbt type conversions(char to int)

cons cast:

casting from cont to non const and vice versa

**Syntax**

**dynamic\_cast < new\_type > ( expression )**

new\_type - pointer to complete class type, reference to complete class type, or pointer to (optionally cv-qualified) void

expression - lvalue of a complete class type if new\_type is a reference, prvalue of a pointer to complete class type if new\_type is a pointer.

If the cast is successful, dynamic\_cast returns a value of type new\_type. If the cast fails and new\_type is a pointer type, it returns a null pointer of that type. If the cast fails and new\_type is a reference type, it throws an exception that matches a handler of type std::bad\_cast

**Object slicing:**

Object Slicing is occurred when a derived class object is **passed by value**as a **BaseCls  classobject**, the base class **copy constructor**is called. Here the derived class object are **sliced off**.  if we upcast ([Upcasting and downcasting](http://www.cpptutorials.com/2014/11/upcasting-and-downcasting-in-c-with.html)) to  an **object**instead of a [**pointer**](http://www.cpphub.com/search/label/Pointers)or **reference**, the object is **sliced**.

<http://www.bogotobogo.com/cplusplus/slicing.php>

**Function templates:**

In C++, function templatesare functions that serve as a pattern for creating other similar functions. The basic idea behind function templates is to create a function without having to specify the exact type(s) of some or all of the variables. Instead, we define the function using placeholder types, called template type parameters.

e.g.

template <typename Type> // this is the template parameter declaration

Type max(Type tX, Type tY)

{

return (tX > tY) ? tX : tY;

}

Here type can be anythhing (int,float,char etc.)

at compile time, when the compiler encounters a call to a template function, it replicates the template function and replaces the template type parameters with actual types! The function with actual types is called a **function template instance**.

**Q: What are the differences between a C++ struct and C++ class?**

A: The default member and base class access specifies are different. This is one of the

commonly misunderstood aspects of C++. Believe it or not, many programmers think that a C++

struct is just like a C struct, while a C++ class has inheritance, access specifes, member

functions, overloaded operators, and so on. Actually, the C++ struct has all the features of the

class. The only differences are that a struct defaults to public member access and public base

class inheritance, and a class defaults to the private access specified and private base-class

inheritance.

**Constant class members and functions:**

constant member function can be called by constant object as well as non constant objects. It will not change any class var or not call any const member function.

Constant objects can call only const member functions.

e.g:const\_class.cpp

**Exception handling:**

An exception is a problem that arises during the execution of a program

throw: A program throws an exception when a problem shows up. This is done using a throw keyword.

catch: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

try: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

try

{

// protected code

}catch( ExceptionName e1 )

{

// catch block

}catch( ExceptionName e2 )

{

// catch block

}

e.g

#include <iostream>

using namespace std;

double division(int a, int b)

{

if( b == 0 )

{

throw "Division by zero condition!";

}

return (a/b);

}

int main ()

{

int x = 50;

int y = 0;

double z = 0;

try {

z = division(x, y);

cout << z << endl;

}catch (const char\* msg) {

cerr << msg << endl;

}

return 0;

}

e.g. Defining own exception:

#include <iostream>

#include <exception>

using namespace std;

struct MyException : public exception

{

const char \* what () const throw ()

{

return "C++ Exception";

}

};

int main()

{

try

{

throw MyException();

}

catch(MyException& e)

{

std::cout << "MyException caught" << std::endl;

std::cout << e.what() << std::endl;

}

catch(std::exception& e)

{

//Other errors

}

}

what is public method provided by exception class which is overriden

Cpp has many std exceptions too.(in standard exception class)

If ellipsis is given in catch statement then all exceptions will get catched.

e.g

catch(...)

catch all must be last catch block otherwise compilation error

if exception is thrown but not catched, then program will terminate abnormally.

e.g.

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {  **try**  {  **throw** 'a';      }  **catch** (**int** x)  {          cout << "Caught ";      }  **return** 0;  } |

This program will terminate.

If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.

Otherwise derived class exception will never be reached.

When an exception is thrown, lines of try block after the throw statement are not executed.

When an object is created inside a try block, destructor for the object is called before control is transferred to catch block. Destructors will be called only for completely created objects.

Quiz:<http://quiz.geeksforgeeks.org/c-plus-plus/exception-handling/>

**Pass by reference and pass by value:**

1.learncpp.

2.see e.g. Pass.cpp

**VOLATILE:**

noramlly a variable is stored in a cpu register to fast access. But this variable then can not be accessed by outsiders of code i.e.by code the compiler doesn't know about as they dont have access to those register.

For. e.g variable is to be used bet two apps.

If the variable is declared as volatile it will be stord in main memory rather than register memory.

That is cpu will not optimize it. Time consumption will be more but outsiders can access it.

e.g. A variable is set by your code. And also external timer is modifiing it. Then it should be volatile.

Useful in embedded, when intrface with hardware.

Both of these declarations declare foo to be a pointer to a volatile integer:

Pointer to volatile variaable

volatile int \* foo;   
int volatile \* foo;

Volatile pointers to non-volatile variables

int \* volatile foo;

volatile pointer to a volatile variable, :

int volatile \* volatile foo;

**Mutable:**

The keyword mutable is mainly used to allow a particular data member of const object to be modified. [When we declare a function as const, the this pointer passed to function becomes const](http://www.geeksforgeeks.org/g-fact-77/). Adding mutable to a variable allows a const pointer to change members.

If we make any variable of class or structure mutable, constant object or constant function can modify it. In normal case const obj will not modify any thing from class members and const function can not update class vars

**functors:**

class object can be used as a function if ( ) operator is overloaded as a member function.

Function operatioe “( )”

Class calc{

public:

int operator( )(int a, int b)

{

int c= a+b;

return c;

}

};

int main()

{

calc sum;

int c=sum(2,4);

}

**Standard Conversions:**  
The C++ language defines conversions between its fundamental types. It also defines conversions for pointer, reference, and pointer-to-member derived types. These conversions are called "standard conversions

**Conversion functions and conversion constructors:**

User-defined conversions perform conversions between user-defined types, or between user-defined types and built-in types. You can implement them as [Conversion constructors](https://msdn.microsoft.com/en-us/library/wwywka61.aspx" \l "ConvCTOR) or as [Conversion functions](https://msdn.microsoft.com/en-us/library/wwywka61.aspx" \l "ConvFunc).

**Conversion function: also called Conversion operator**

**Conversion functions define conversions from a user-defined type to other types**

**The target type of the conversion must be declared prior to the declaration of the conversion function. Classes, structures, enumerations, and typedefs cannot be declared within the declaration of the conversion function.**

**Conversion functions take no arguments. Specifying any parameters in the declaration is an error.**

* Conversion functions have a return type that is specified by the name of the conversion function, which is also the name of the conversion's target type. Specifying a return type in the declaration is an error.
* Conversion functions can be virtual.
* Conversion functions can be explicit.

**e.g.**

class Money

{

public:

Money() : amount{ 0.0 } {};

Money(double \_amount) : amount{ \_amount } {};

operator double() const { return amount; }

private:

double amount;

};

void display\_balance(const Money balance)

{

std::cout << "The balance is: " << balance << std::endl;

}

**Here to display balance conversion function of double is used.**

Use: when object is not of usable form i.e. in above e.g. obj balance is to be printed but we can not do it.

So here conversion function is called implicitly to convert obj to usable form(built in data type).

**2.**

**class Y {**

int b;

public:

operator int();

};

Y::operator int() {

return b;

}

void f(Y obj) {

int i = int(obj);

int j = (int)obj;

int k = i + obj;

}

**All 3 statements in above function f(Y) use conversion function operator int()**

**When a conversion function is declared to be explicit, it can only be used to perform an explicit cast**

**in e.g.1**

**explicit operator double() const { return amount; }**

**call like (double)balance**

**To call this function it should be mentioned explicitly**

**[https://msdn.microsoft.c](https://msdn.microsoft.com/en-us/library/wwywka61.aspx" \l "ConvFunc)****om/en-us/library/wwywka61.aspx#ConvFunc**

**Conversion Constructor:**

**Conversion constructors define conversions from user-defined or built-in types to a user-defined type**

**e.g**

**class Money**

{

public:

Money() : amount{ 0.0 } {};

Money(double \_amount) : amount{ \_amount } {};

double amount;

};

void display\_balance(const Money balance)

{

std::cout << "The balance is: " << balance.amount << std::endl;

}

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

{

Money payable{ 79.99 };

display\_balance(payable);// directly acceptablr

display\_balance(49.95); // conversion required as funct expects Money obj.doubleto Money

//temp value of Money is created

display\_balance(9.99f);// float to double to Money

return 0;

}

**The target type of the conversion is the user-defined type that's being constructed.**

* Conversion constructors typically take exactly one argument, which is of the source type. However, a conversion constructor can specify additional parameters if each additional parameter has a default value. The source type remains the type of the first parameter.
* Conversion constructors, like all constructors, do not specify a return type. Specifying a return type in the declaration is an error.
* Conversion constructors can be explicit.

**Explicit:**

**explicit Money(double \_amount) : amount{ \_amount } {};**

**display\_balance((Money)9.99f);**

**[https://msdn.microsoft.com/en-us/library/wwywka61.aspx#ConvCTOR](https://msdn.microsoft.com/en-us/library/wwywka61.aspx" \l "ConvCTOR)**

**Difference between conversion function/operator and functor**

Conv func has no return type and doesn’t take arguments

Functor has return type and can take arguments

Call to conversion func doesn’t require any specific syntax i.e. ( ) to be written just obj name will also call it

Calling functor is same as calling function i.e. obj();

**e.g for clarification : functorsAndConversionFunction.cpp**

**DATA ENCAPSULATION AND DATA HIDING:**

Encapsulation means to protect sensitive information in an object by making members protected or private in a class. The user then has to use functions to change the state of the object, rather than [possibly incorrectly] modifying the object directly. This helps assure than an object is always "stable" and can't be corrupted. It also makes classes much harder to misuse and less likely to cause serious problems in a program.  
  
Data hiding is an extreme version of encapsulation where you not only don't want the user to access the data members, but you also don't even want them to be able to see what they are. This is typically accomplished by using a void pointer or a forward declared struct pointer as the class data.  
  
Here's an example of typical encapslation:

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8 | class MyClass  {  public:  // members functions and stuff here  private:  int foo; // members are visible (not hidden)  int bar; // but are still private (encapsulated)  }; |  |

Here's an example of data hiding:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class MyClass  {  public:  // member functions and stuff here  private:  struct Data;  Data\* data; // user cannot see what 'data' this class uses  // therefore it's hidden  };  //================  // then, in the .cpp file... the struct would need to be defined  struct MyClass::Data  {  int foo;  int bar;  }; |

**What breaks encapsulation?**

member function may return a pointer (or reference) to a private data member. There is nothing wrong with this except that in most circumstances it breaks encapsulation.

class Example

{

public:

int\* get() { return &i; }

const int\* get\_const() const { return &i; }

private:

int i;

};

int main()

{

Example e;

int\* p = e.get();

int a = \*p; // yes, we can read the data via the pointer

\*p = 42; // yes, we can modify the data via the pointer

const int\* cp = e.get\_const();

int b = \*cp; // yes, we can read the data via the pointer

\*cp = 42; // error: pointer is to a const int

}

**Can a dynamic cast gives error at compile time or runtime, what is the error string?”**

**runtime**

Unlike other casts, a dynamic\_cast involves a run-time type check. If the object bound to thepointer is not an object of the target type, it fails and the value is 0. If it's a reference type when it fails, then an exception of type bad\_cast is thrown. So, if we want dynamic\_cast to throw an exception (bad\_cast) instead of returning 0, cast to a reference instead of to a pointer. Note also that the dynamic\_cast is the only cast that relies on run-time checking.

**Memory allocation:**

normal functions are allocated memory in code segment.

For class, no memory is allocated at declaration.

When objects are created, memory is allocated for each object for each data member

member functions are stored in program memory only once same as normal function only the difference is compiler adds hidden this pointer at the time of compilation to member function call

size of any object(or class) is equal to sum of sizes of all data members

<http://gradestack.com/Programming-in-C-/Classes-and-Objects/Classes-Objects-and/21199-4331-48216-study-wtw>

**Diff bet const and macro**

1. The statement “#define SIZE 50” is a preprocessor directive with SIZE defined as a MACRO. The advantage of declaring a MACRO in a program is to replace the MACRO name with the Value given to the MACRO in the #define statement wherever MACRO occurs in the program.

2. No memory is allocated to the MACROS. Program is faster in Execution because of no trade-offs due to allocation of memory!

1. Their Primary use in the program is where constant values viz. characters, integers, floating point is to be used. For example: as an array subscripts
2. 4. A MACRO Statement does not terminate with a semicolon ” ; ”

Macro : replaced by value at time of preprocessing

Cosnts: at time of compilation  
CONSTANTS

1. The statement “int const size = 50;” declares and defines size to be a constant integer with the value 50. The const keyword causes the identifier size to be allocated in the read-only memory. This means that the value of the identifier can not be changed by the executing program.

MACROS are efficient than the const statements as they are not given any memory, being more Readable and Faster in execution!

**Memory structure:**  
 Depending on the data segmentation that a particular processor follows, we have five segments:

1. Code Segment /text segment- Stores only code, ROM, read only segment
2. BSS (or Block Started by Symbols)?(block storage start) segment - Stores uninitialised global and static variables  Object file formats distinguish between initialized and uninitialized variables for space efficiency; uninitialized variables do not have to occupy any actual disk space in the object file. Better save space
3. Stack segment - stores all the local varialbles and other informations regarding function return address etc
4. Heap segment - all dynamic allocations happens here
5. Data segment - stores initialised global and static variables

if I have a const int which is local variable, then it is stored in the write protected region of stack segment. and if I have a global that is initialised const var, then it is stored in BSS and if I have an uninitialised const var, then it is stored in data segment...

**Storage classes:**

auto: automatic storage: local vars are auto scope limited to functions

extern: global variables by default extern. Persist throughout code

static: local vars that retain their values.

Register: tells compiler to store variables in processors register

<http://www.tenouk.com/download/pdf/Module20.pdf>

**macro fails e.g.**

# define sqr(x) (x\*x)\

x=4;

sqr(--x)

here we will expect ans. To be (3\*3) = 9

but it would be (--x \* --x)=4

all pre increments will be done first then multiplication  
  
**vectors:**

vector stored on stack elements stored on heap contiguously.

If after allocating size to diff vars, vector elemts are increased, diff memory locations

1. Contiguous memory.
2. Pre-allocates space for future elements, so extra space may be required.
3. Unlike a list where additional space for a pointer is needed, each element only requires the space for the element type itself.
4. Can re-allocate memory for the entire vector at any time that we add an element.
5. Insertions at the end are constant, but insertions elsewhere are a costly O(n).
6. Erasing an element at the end of the vector is constant time, but for the other locations it's O(n).
7. We can randomly access its elements.
8. Iterators, pointers, and references are invalidated if we add or remove elements to or from the vector.

**LIST:**

1. Non-contiguous memory.
2. No pre-allocated memory. The memory overhead for the list itself is constant.
3. Each element requires extra space for the node which holds the element, including pointers to the next and previous elements in the list.
4. Never has to re-allocate memory for the whole list just because we add an element.
5. Insertions and erasures are cheap no matter where in the list they occur.
6. It's cheap to combine lists with splicing.
7. We cannot randomly access elements, so getting at a particular element in the list can be expensive.
8. Iterators remain valid even when we add or remove elements from the list.
9. If we need an array of the elements, we'll have to create a new one and add them all to it, since there is no underlying array.

Why **virtual constructor** cant be created?

The object exists only after the constructor ends.In order for the constructor to be dispatched using the virtual table , there has to be an existing object with a pointer to the virtual table , but how can a pointer to the virtual table exist if the object still doesn't exist?

**Empty class :** always has size of 1 byte so that 2 different objects will have different locations.

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.

<http://www.geeksforgeeks.org/why-is-the-size-of-an-empty-class-not-zero-in-c/>

The **MVC** pattern provides a clean separation of objects into:

* **Models** for maintaining data,
* **Views** for displaying all or a portion of the data, and
* **Controllers** for handling events that affect the model or view(s).

**Singleton class:**

restricts to create only 1 object of it

constructor private.. so that it can not be accessed by main.

i.e. main cannnot create object directly

Define a static member function in class.

As static function can be called only by class name, no need of obj to call it from main

create new object of class through that function.

Return address of obj using static pointer (initialized to null)

create obj only if ptr is null else return previous address of that ptr

ptr will persist address as it is declared static... so only 1 obj will bw created.

**Name mangling:**

In c function overloading is not possible. So linker gets unique name for each function.

In cpp when function overloading is done, compiler converts same function names into different unique names using some method so that linker can link between 2 diff calls.

This technique of adding additional information to function names is called [Name Mangling](http://en.wikipedia.org/wiki/Name_mangling)

These name conversions are compiler dependent and diff for diff compilers.

For most purposes, QList is the right class to use. Its index-based API is more convenient than QLinkedList's iterator-based API, and it is usually faster than QVector because of the way it stores its items in memory. It also expands to less code in your executable.

* If you need a real linked list, with guarantees of constant time insertions in the middle of the list and iterators to items rather than indexes, use QLinkedList.
* If you want the items to occupy adjacent memory positions, use QVector.

**Callbacks and signal slot difference:**

callbacks are tightly coupled. i.e. processing function must know which callback to call

signal slots are loosely coupled.

i.e. when signal is emitted it is not needed to know which slot is to be called.

Any slot can be called depending on signature matching.

Callbacks are always called in the context of the calling thread

signal slots can be in diferent threads

signal slots are slower than callbacks

the biggest strength of the signals and slots concept is that it makes it possible to do truly loose coupling. A slot foo( int bar ), can be connected to any signal emitting and integer such as baz( int x ) or y( int z ) without having to worry about the source of the signal. This makes it possible to interconnect objects in a very flexible manner and this makes Qt objects very reuseable.