
بسم الله الرحمن الرحيم

مقام معظم رهبری:

علم، پایه‌ی پیشرفت همه‌جانبه‌ی یک کشور است.

۱۳۹۰/۱۱/۱۴

Qt Training in C++

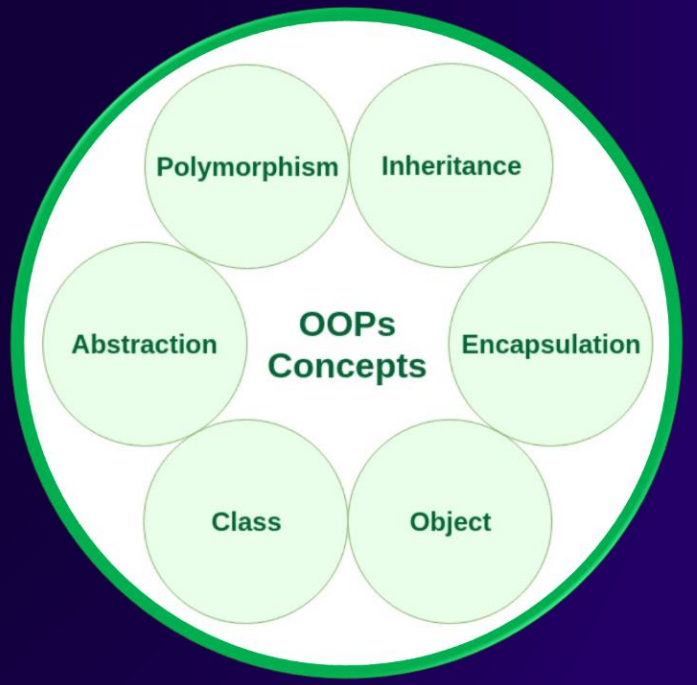
Lecturer: Ali Panahi

Spring 1402

Object Oriented Programming in C++

Object Oriented

- Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.
- Basic concepts:
  - Class
  - Objects
  - Encapsulation
  - Abstraction
  - Polymorphism
  - Inheritance
  - Dynamic Binding
  - Message Passing



Object Oriented Programming in C++

Class

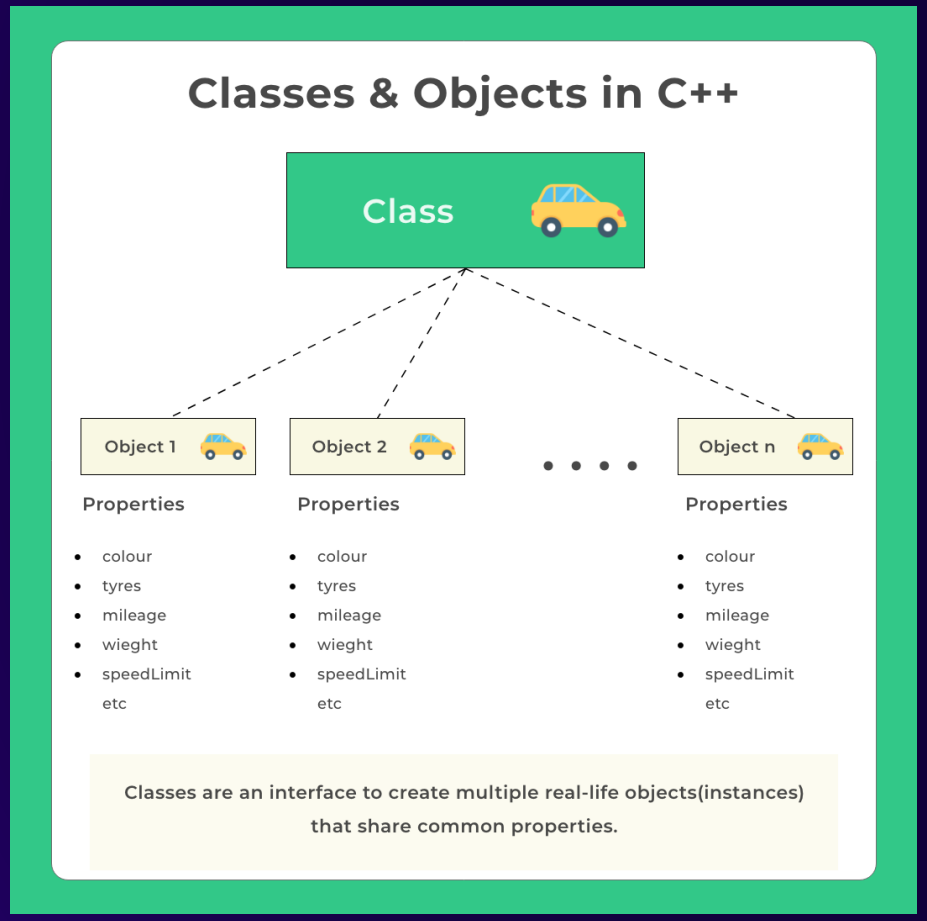
- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables together these data members and member functions define the properties and behavior of the objects in a Class.
  - For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, the Car is the class, and wheels, speed limits, and mileage are their properties.

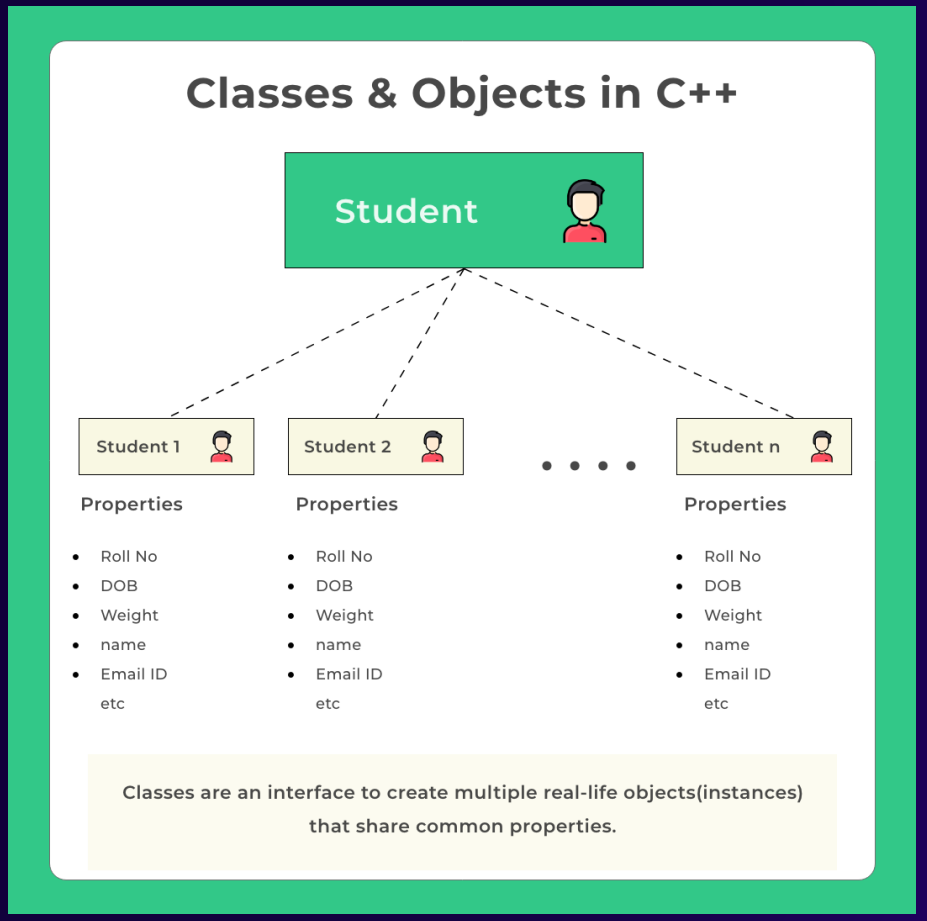
Object Oriented Programming in C++

Object

- An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.
- When a program is executed the objects interact by sending messages to one another. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and the type of response returned by the objects.

Class vs Object

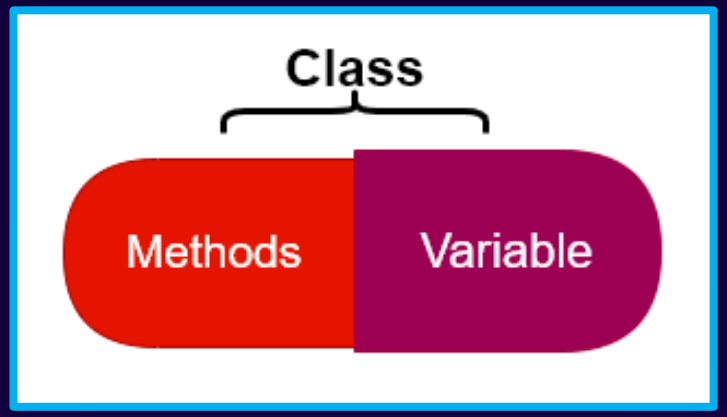




Object Oriented Programming in C++

Encapsulation

- Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.
- Data encapsulation is a mechanism of bundling the data, and the functions that use them and data abstraction is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

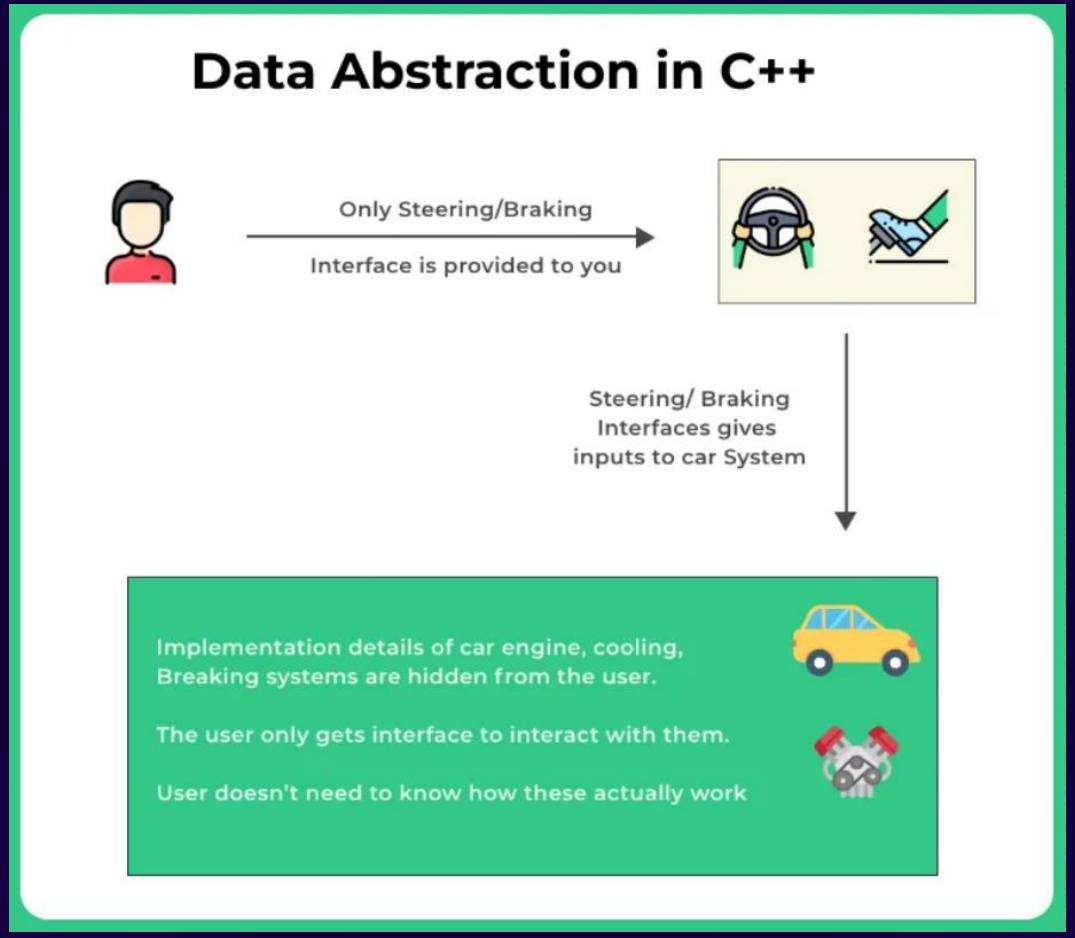


Object Oriented Programming in C++

Abstraction

- Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
  - Abstraction using Classes: We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
  - Abstraction in Header files: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

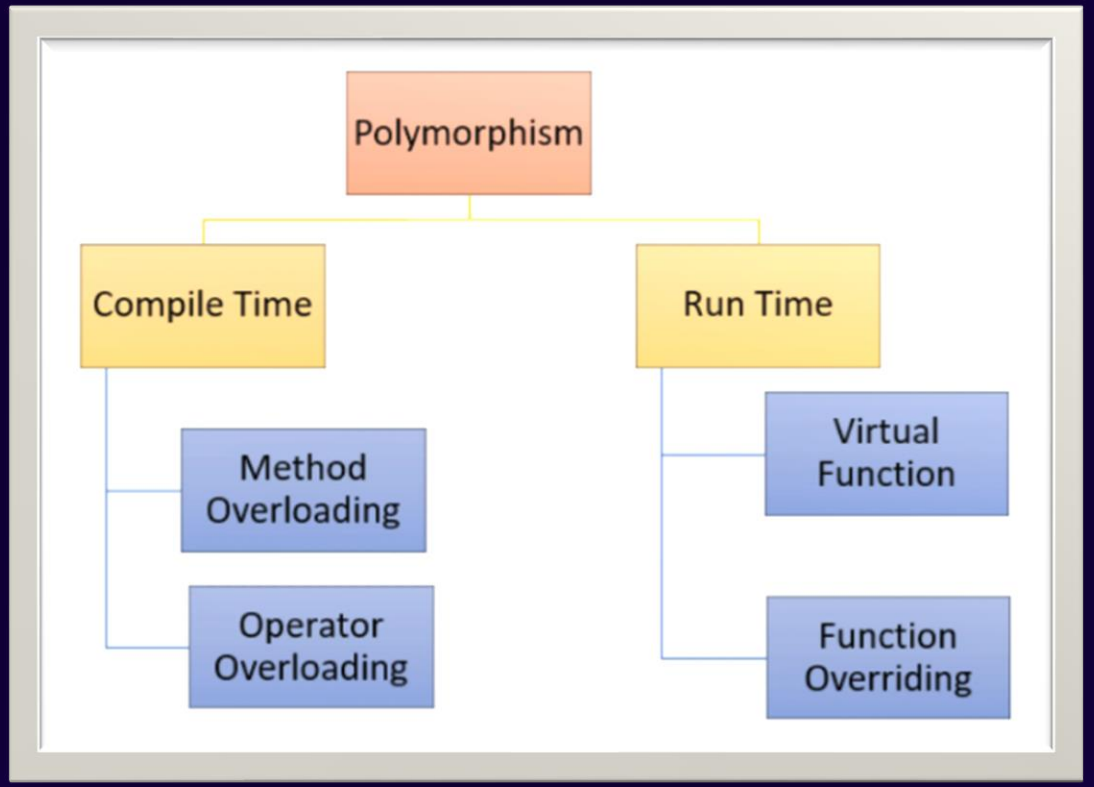
Abstraction



Polymorphism

- Polymorphism in C++ means, the same entity (function or object) behaves differently in different scenarios.
  1. Compile Time Polymorphism
     - Function Overloading
     - Operator Overloading
  2. Runtime Polymorphism
     - Function overriding
     - Virtual Function

Polymorphism

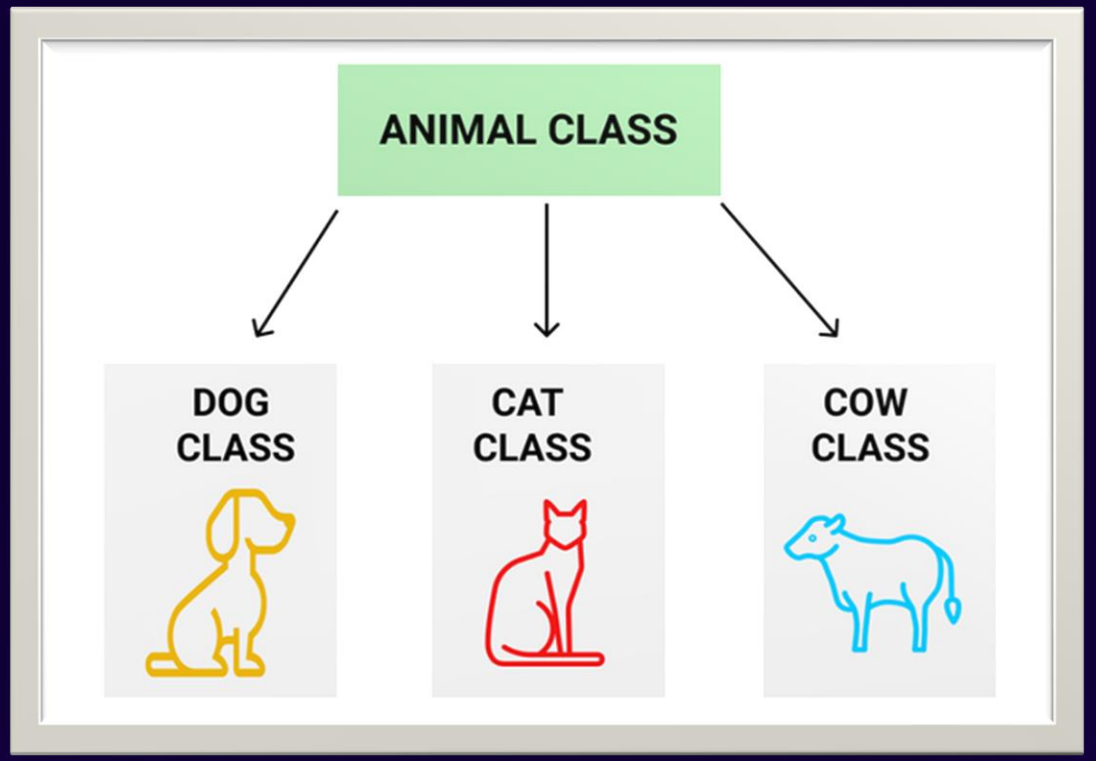


Object Oriented Programming in C++

Inheritance

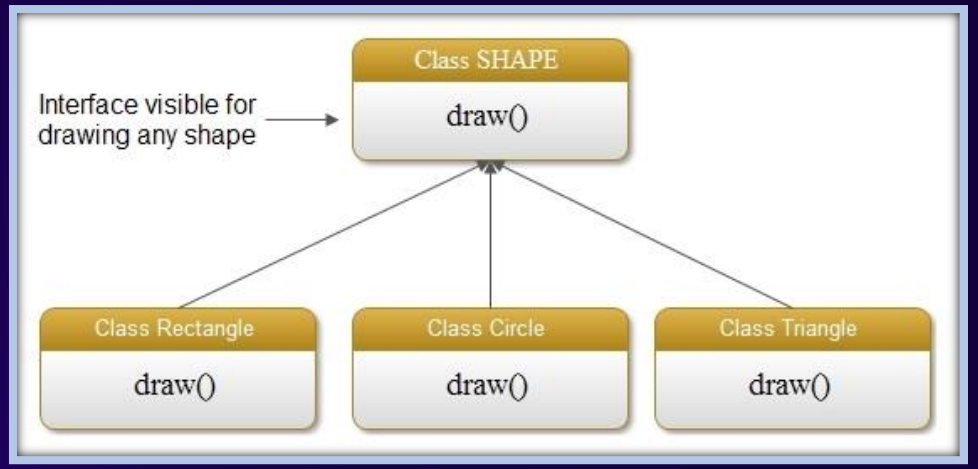
- The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.
  - Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.
  - Super Class: The class whose properties are inherited by a sub-class is called Base Class or Superclass.
  - Reusability: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

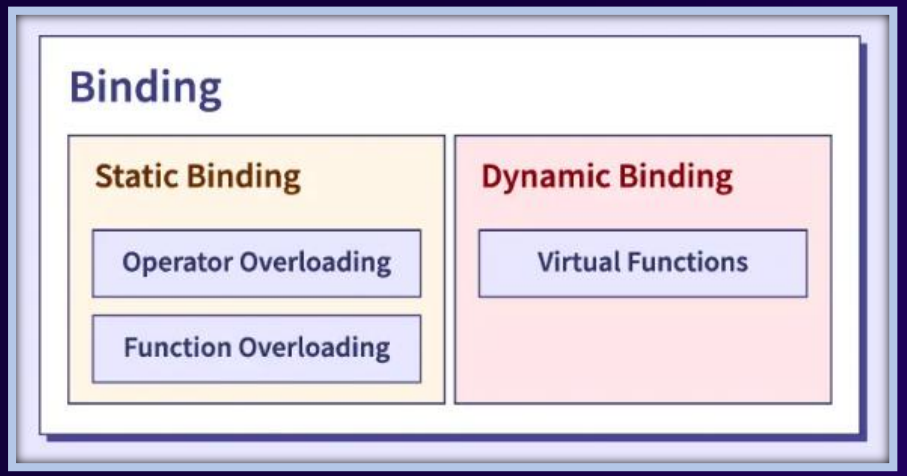
Inheritance



Dynamic Binding

- In dynamic binding, the code to be executed in response to the function call is decided at runtime. C++ has virtual functions to support this. Because dynamic binding is flexible, it avoids the drawbacks of static binding, which connected the function call and definition at build time.

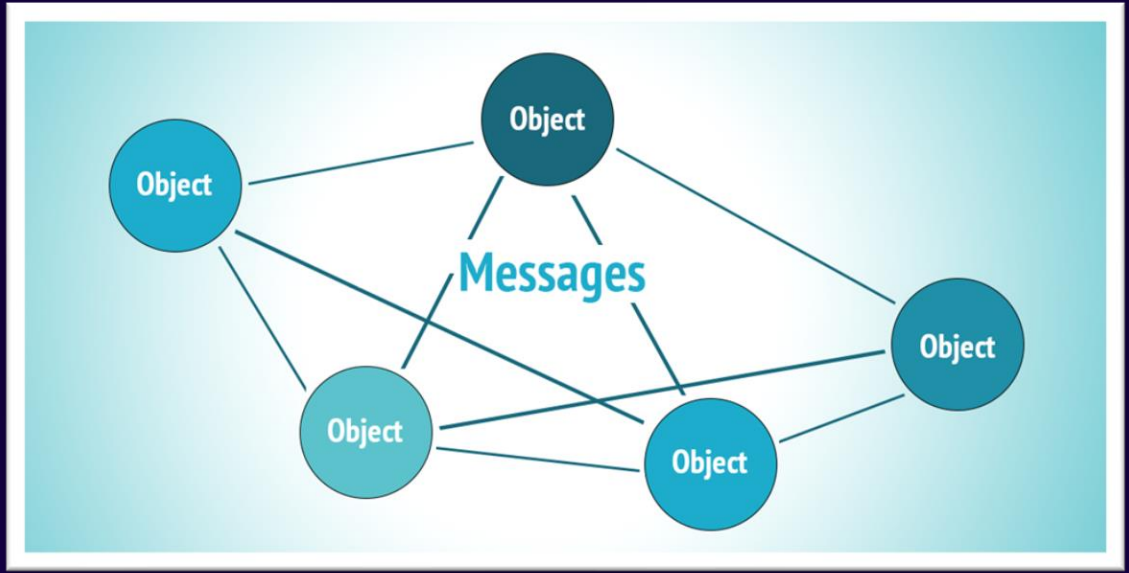




Object Oriented Programming in C++

Message Passing

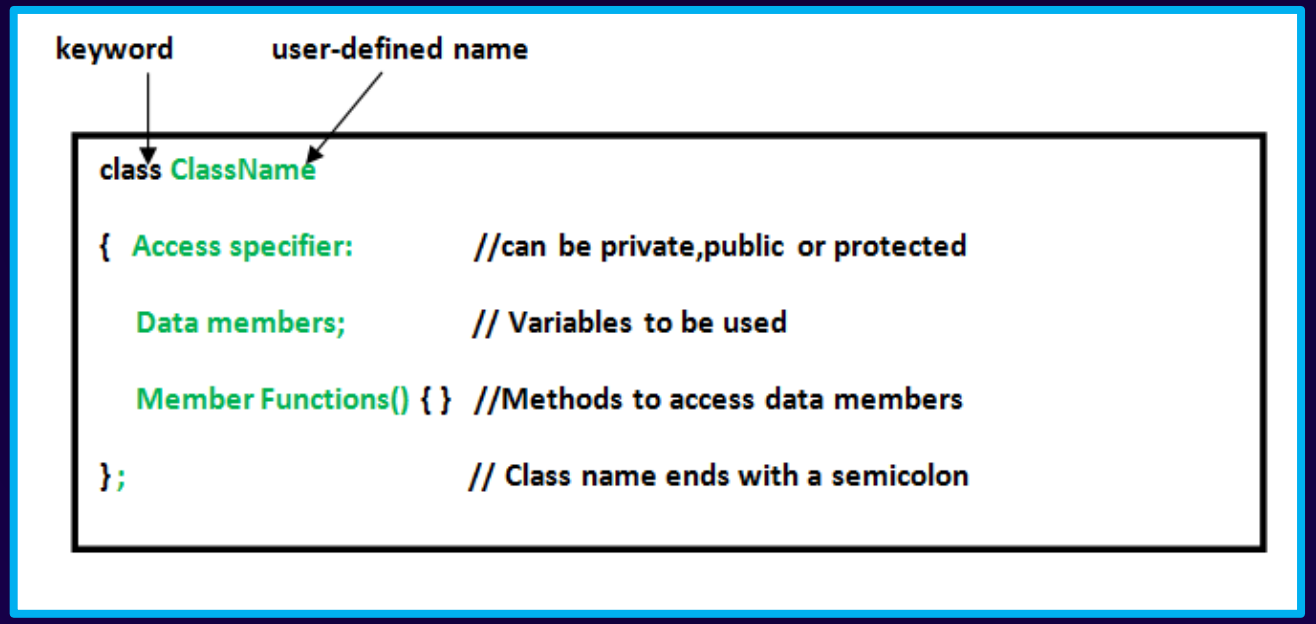
- Objects communicate with one another by sending and receiving information. A message for an object is a request for the execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object,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



Object Oriented Programming in C++

Classes and Objects

- Defining Class and Declaring Objects
  - A class is defined in C++ using the keyword class followed by the name of the class. The body of the class is defined inside the curly brackets and terminated by a semicolon at the end. Default modifier is private.



Object Oriented Programming in C++

Classes and Objects

- Declaring Objects
  - When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

```
ClassName ObjectName;
```

- Accessing data members and member functions
  - The data members and member functions of the class can be accessed using the dot('.') operator with the object. For example, if the name of the object is obj and you want to access the member function with the name printName() then you will have to write obj.printName().
  - The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by Access modifiers in C++. There are three access modifiers: public, private, and protected.

Object Oriented Programming in C++

Classes and Objects

- Member Functions in Classes
  - There are 2 ways to define a member function:
    - ✓ Inside class definition
    - ✓ Outside class definition
  - To define a member function outside the class definition we have to use the scope resolution:: operator along with the class name and function name.
- Constructors
  - Constructors are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition. There are 3 types of constructors:
    - ✓ Default Constructors
    - ✓ Parameterized Constructors
    - ✓ Copy Constructors

    Note: A Copy Constructor creates a new object, which is an exact copy of the existing object. The compiler provides a default Copy Constructor to all the classes.

```
class-name (class-name &){}
```

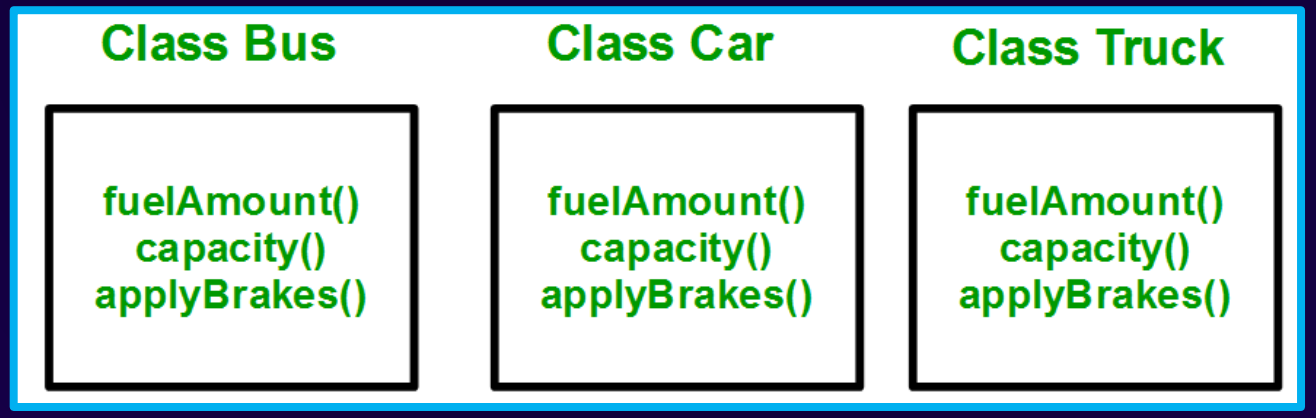
Object Oriented Programming in C++

Classes and Objects

- Destructors
  - Destructor is another special member function that is called by the compiler when the scope of the object ends.
- Interesting Fact (Rare Known Concept)
  - Many people might say that it's a basic syntax and we should give a semicolon at the end of the class as its rule defines in cpp. But the main reason why semi-colons are there at the end of the class is compiler checks if the user is trying to create an instance of the class at the end of it.
  - Yes just like structure and union, we can also create the instance of a class at the end just before the semicolon. As a result, once execution reaches at that line, it creates a class and allocates memory to your instance.

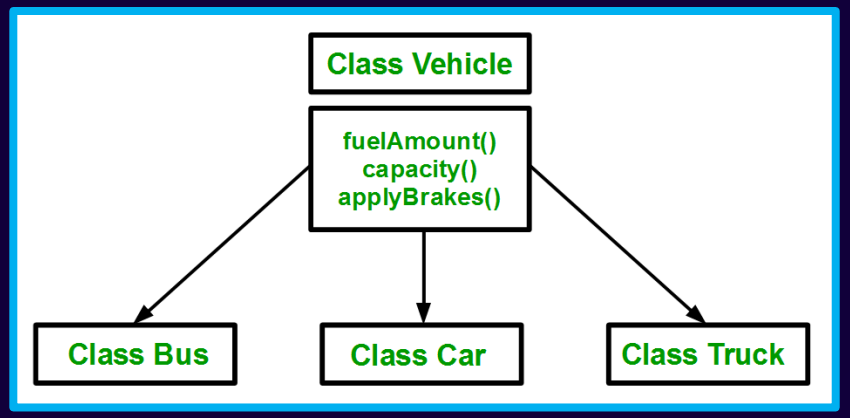
Inheritance in C++

- Why and when to use inheritance?
  - Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown below figure:



Inheritance in C++

- Why and when to use inheritance?
  - You can clearly see that the above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used.



  - Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle).

Object Oriented Programming in C++

Inheritance in C++

- Implementing inheritance in C++
  - access-specifier either of private, public or protected. If neither is specified, PRIVATE is taken as default.

```
class  <derived_class_name> : <access-specifier> <base_class_name>
{
        //body
}
```

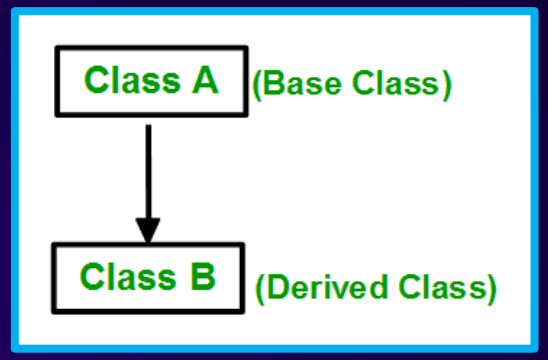
  - Note: When a base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class.
- Modes of Inheritance

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Object Oriented Programming in C++

Inheritance in C++

- Types Of Inheritance:
  1. Single inheritance
  2. Multilevel inheritance
  3. Multiple inheritance
  4. Hierarchical inheritance
  5. Hybrid inheritance
  6. Multipath inheritance
- Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.

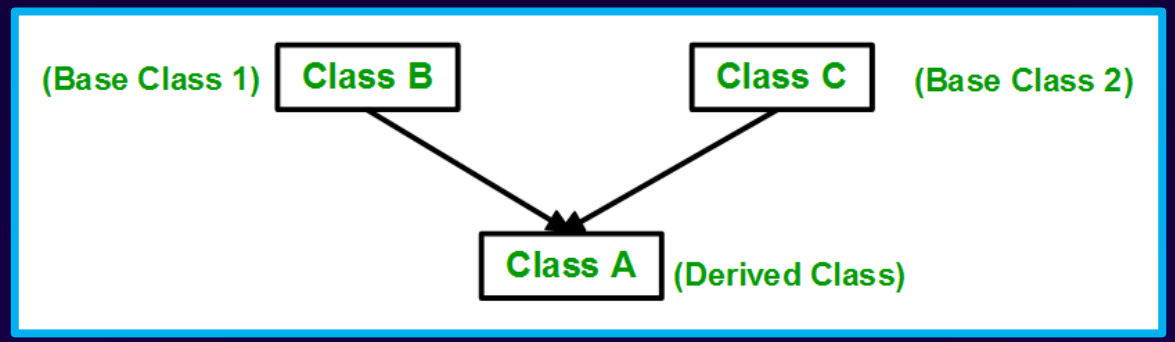



```
class subclass_name : access_mode base_class
{
  // body of subclass
};
```

Inheritance in C++

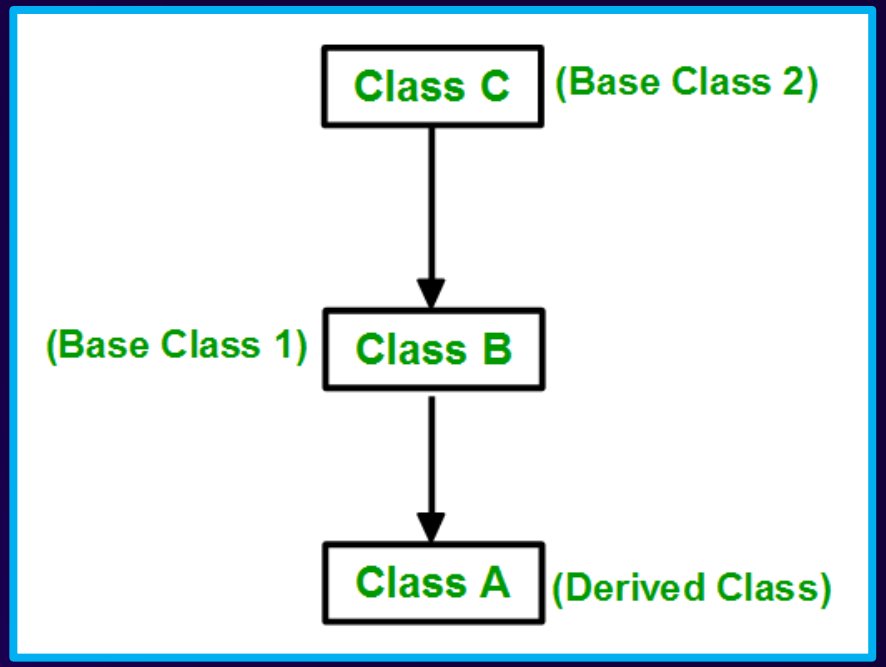
- Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one subclass is inherited from more than one base class.

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  // body of subclass
};
```



Inheritance in C++

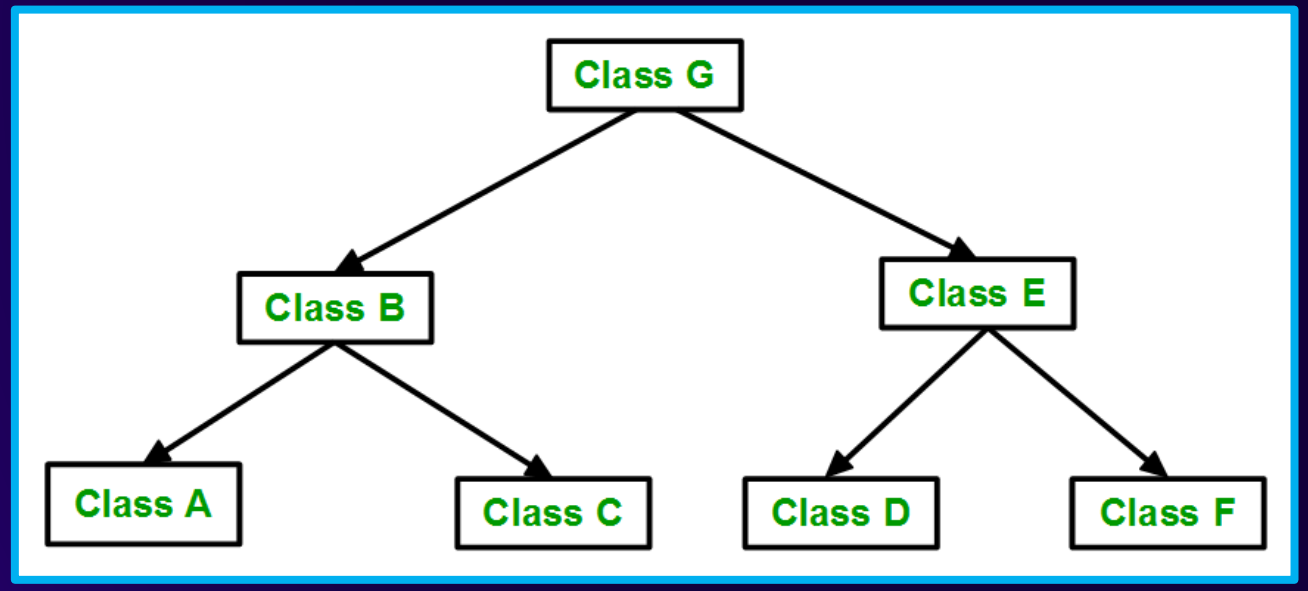
- Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.




```
class C
{
... .. ...
};
class B:public C
{
... .. ...
};
class A: public B
{
... ... ...
};
```

Inheritance in C++

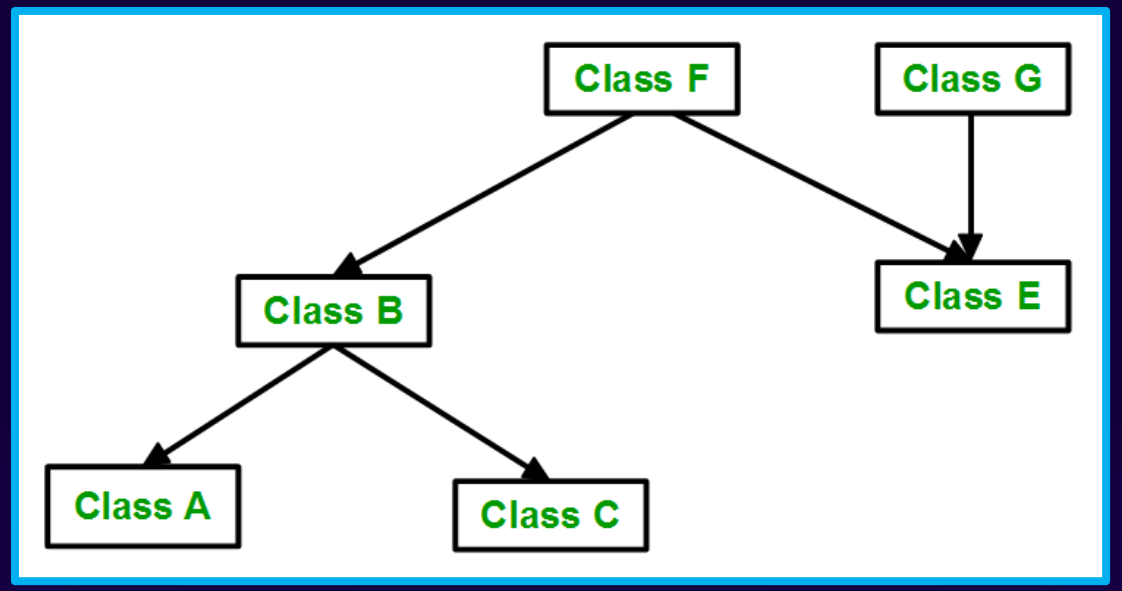
- Hierarchical Inheritance: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.




```
class A
{
    // body of the class A.
}
class B : public A
{
    // body of class B.
}
class C : public A
{
    // body of class C.
}
class D : public A
{
    // body of class D.
}
```

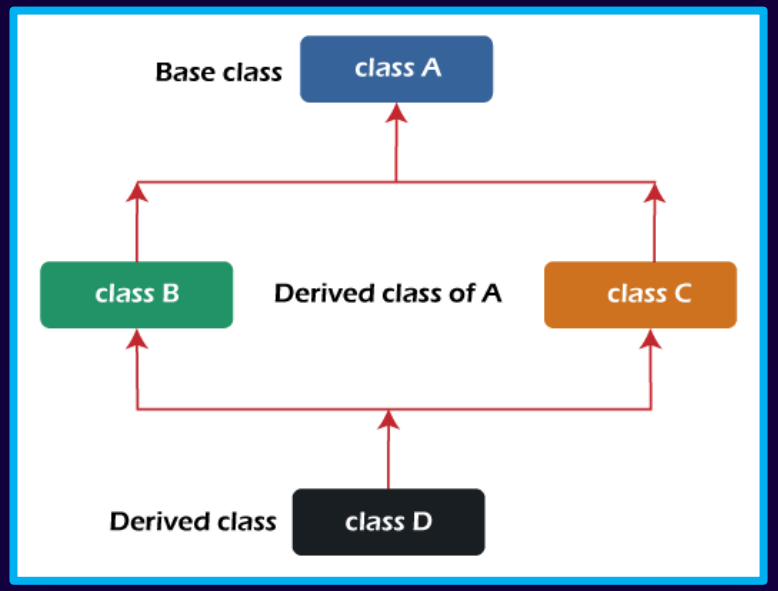
Inheritance in C++

- Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritances:



Inheritance in C++

- Multipath inheritance (A special case of hybrid inheritance): A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. Ambiguity can arise in this type of inheritance.



Inheritance in C++

- Multipath inheritance (A special case of hybrid inheritance)
  - There are 2 Ways to Avoid this Ambiguity:
    1. Avoiding ambiguity using the scope resolution operator: In this case there are multiple copies of the parent class
    2. Avoiding ambiguity using the virtual base class: In this case there is a copy of the parent class.

```
class A
{
    member;
}
class B : public A {}
class C : public A {}
class D : public B, public C {}

D d;
d.B::member = x;
d.C::member = x;
```

```
class A
{
    member;
}
class B : virtual public A {}
class C : virtual public A {}
class D : public B, public C {}

D d;
d.member = x;
```

Object Oriented Programming in C++

Compile-Time Polymorphism

- Function Overloading
  - When there are multiple functions with the same name but different parameters, then the functions are said to be overloaded, hence this is known as Function Overloading. Functions can be overloaded by changing the number of arguments or/and changing the type of arguments.
- Operator Overloading
  - C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

Runtime Polymorphism

- Function Overriding or Virtual Function
  - A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.
    - Virtual functions are Dynamic in nature.
    - They are defined by inserting the keyword “virtual” inside a base class and are always declared with a base class and overridden in a child class
    - A virtual function is called during Runtime
  - The function in the parent class must be defined as virtual.
- Runtime Polymorphism with Data Members
  - Runtime Polymorphism can be achieved by data members in C++.

Object Oriented Programming in C++

Constructor

- A constructor is a member function of a class that has the same name as the class name. It helps to initialize the object of a class. It can either accept the arguments or not.

```
ClassName()
{
    //Constructor's Body
}
```

Destructor

- Like a constructor, Destructor is also a member function of a class that has the same name as the class name preceded by a tilde(~) operator. It helps to deallocate the memory of an object. It is called while the object of the class is freed or deleted.

```
~ClassName()
{
      //Destructor's Body
}
```

Object Oriented Programming in C++

Virtual Destructor

- Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.

Pure Virtual Destructor

- The inclusion of a pure virtual destructor in a C++ program has no negative consequences. Pure virtual destructors must have a function body because their destructors are called before those of base classes; if one is absent, object destruction will fail since there won't be anything to call when the object is destroyed. Making a pure virtual destructor with its definition allows us to create an abstract class easily.

  Note: Only Destructors can be Virtual. Constructors cannot be declared as virtual.

Object Oriented Programming in C++

Pure Virtual Functions and Abstract Classes in C++

- Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.
- A pure virtual function (or abstract function) in C++ is a virtual function for which we can have implementation, But we must override that function in the derived class, otherwise the derived class will also become abstract class.
    - Some Interesting Facts:
        1. A class is abstract if it has at least one pure virtual function.
        2. We can have pointers and references of abstract class type.
        3. If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.
        4. An abstract class can have constructors.
        5. An abstract class in C++ can also be defined using struct keyword.

Object Oriented Programming in C++

static_cast

- static_cast is used for ordinary typecasting. It is responsible for the implicit type of coercion and is also called explicitly. We should use it in cases like converting the int to float, int to char, etc.

dynamic_cast

- In C++, we can treat the derived class's reference or pointer as the base class's pointer. This method is known as upcasting in C++. But its opposite process is known as downcasting, which is not allowed in C++. So, the dynamic_cast in C++ promotes safe downcasting. We can only perform this in polymorphic classes, which must have at least one virtual function.
- dynamic_cast is useful when you don't know what the dynamic type of the object is. It returns a null pointer if the object referred to doesn't contain the type casted to as a base class (when you cast to a reference, a bad_cast exception is thrown in that case).

Object Oriented Programming in C++

Copy Constructor

- A copy constructor is a member function that initializes an object using another object of the same class. In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a copy constructor.

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

- Characteristics of Copy Constructor
  1. The copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.
  2. Copy constructor takes a reference to an object of the same class as an argument.
  3. The process of initializing members of an object through a copy constructor is known as copy initialization.
  4. It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member-by-member copy basis.
  5. The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

Object Oriented Programming in C++

Copy Constructor

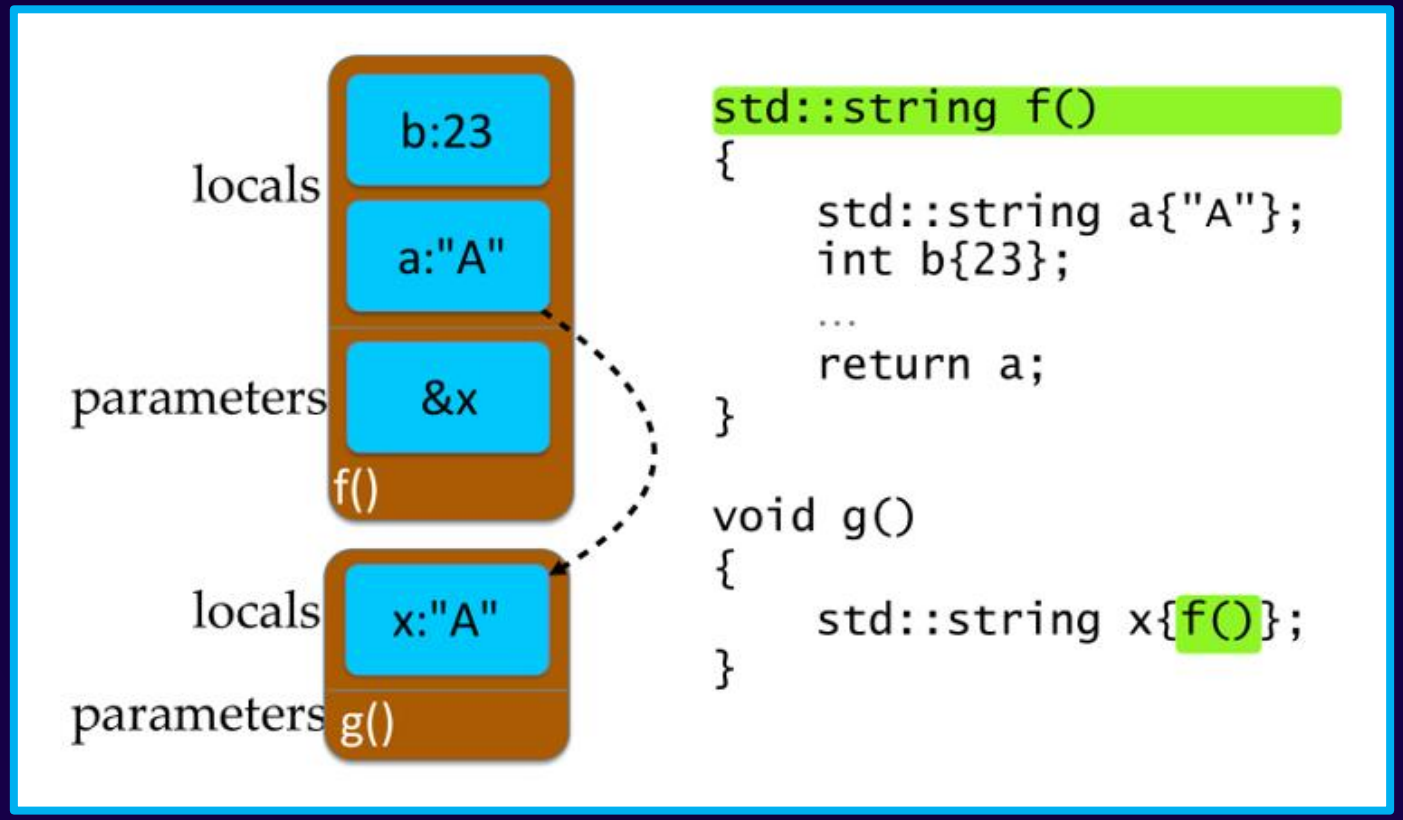
- Types of Copy Constructors
  1. Default Copy Constructor
  2. User Defined Copy Constructor
- When is the copy constructor called? In C++, a Copy Constructor may be called in the following cases:
  1. When an object of the class is returned by value.
  2. When an object of the class is passed (to a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

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

  In copy elision, the compiler prevents the making of extra copies which results in saving space and better the program complexity(both time and space); Hence making the code more optimized.

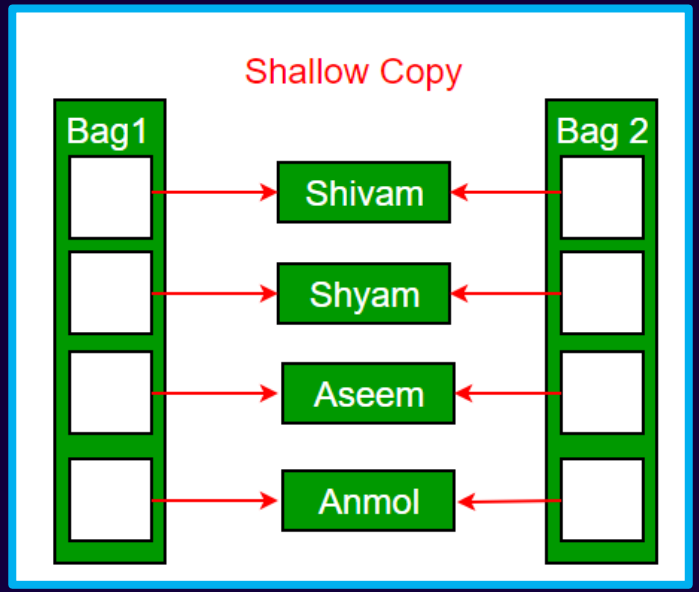
Copy Constructor

- RVO



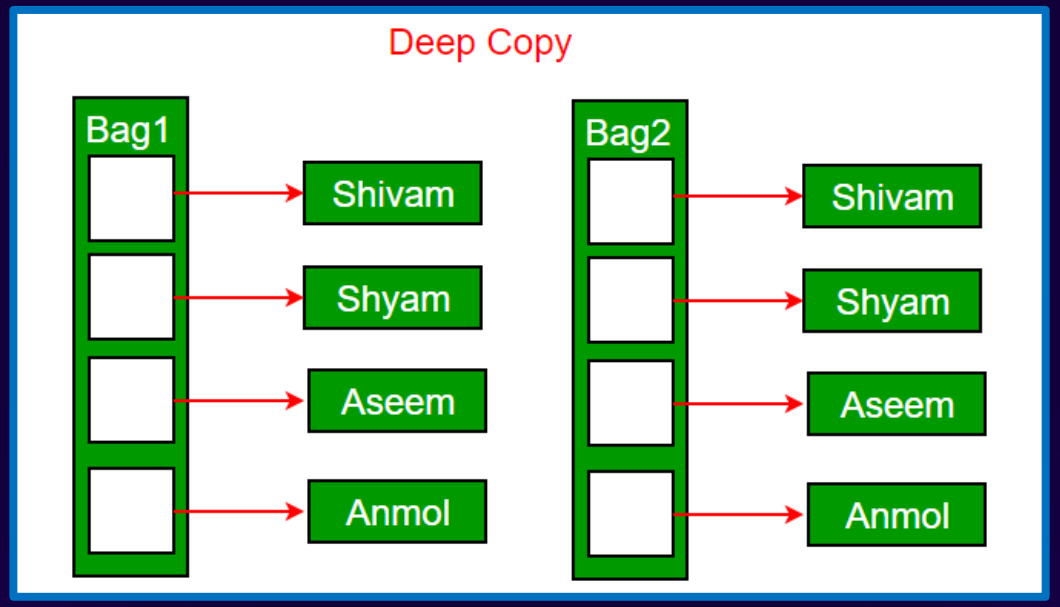
Copy Constructor

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



Copy Constructor

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



Copy Constructor

- Copy constructor vs Assignment Operator
  - The main difference between Copy Constructor and Assignment Operator is that the Copy constructor makes a new memory storage every time it is called while the assignment operator does not make new memory storage.
  - A copy constructor is called when a new object is created from an existing object, as a copy of the existing object. The assignment operator is called when an already initialized object is assigned a new value from another existing object. In the bellow example (1) calls the copy constructor and (2) calls the assignment operator. See this for more details.

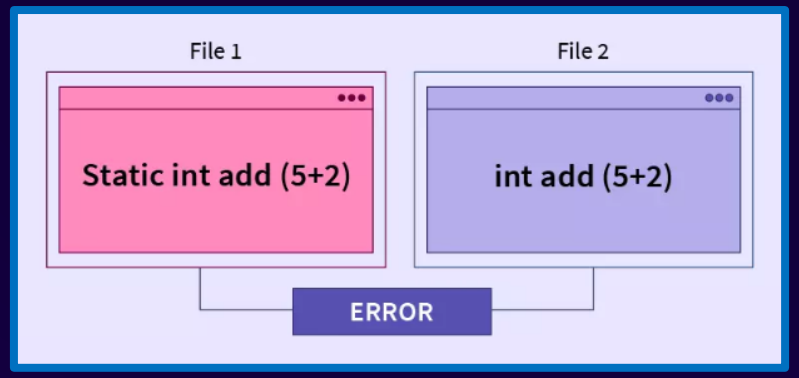
```
MyClass t1, t2;
MyClass t3 = t1;  // ----> (1)
t2 = t1;          // -----> (2)
```

Advanced C++ Programming

static

- The static keyword has different meanings when used with different types. We can use static keywords with:
  - Static Functions in C: The “static” keyword before a function name makes it static.
  - Static Variables: Variables in a function, Variables in a class.
  - Static Members of Class: It is used for functions inside the class.

- Static Functions in C
  - Unlike global functions in C, access to static functions is restricted to the file where they are declared. Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files.

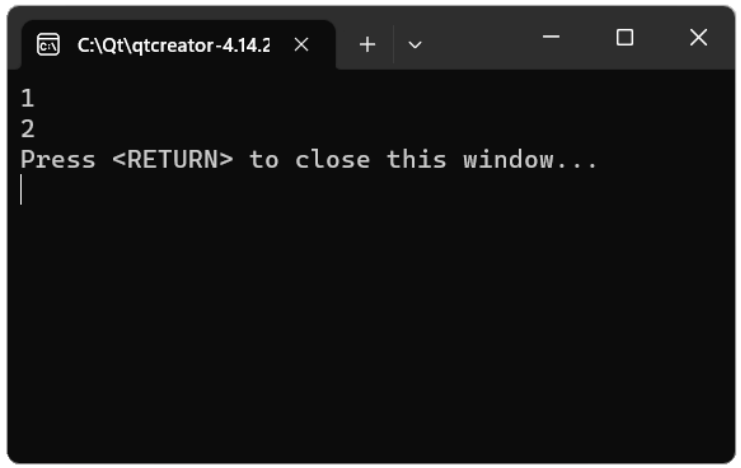


Advanced C++ Programming

static

- Static Variables
  - When a variable is declared as static, space for it gets allocated for the ***lifetime*** of the program. Even if the function is called multiple times, space for the static variable is allocated only once and the value of the variable in the previous call gets carried through the next function call.
  - Static variables must be initialized. Initialization is done only once.

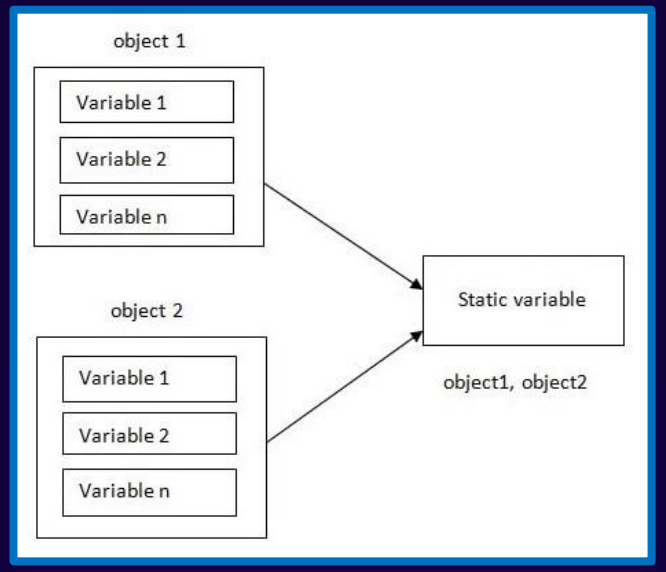
```
5  int fun()
6  {
7      static int count = 0;
8      count++;
9      return count;
10 }
11
12 int main()
13 {
14     cout << fun() << endl;
15     cout << fun() << endl;
16     return 0;
17 }
```



static

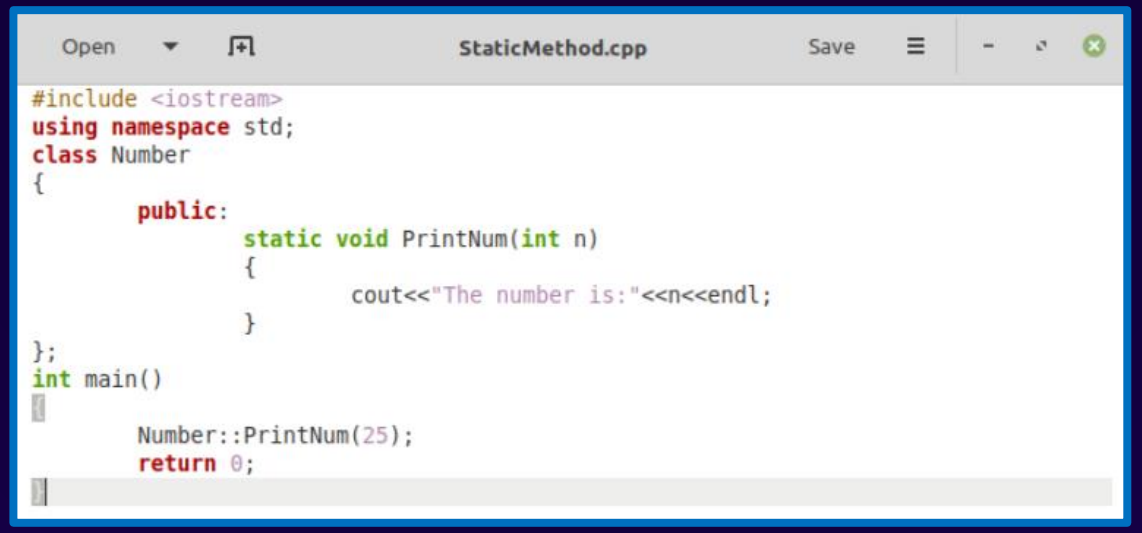
- Static Member Variables (Static variables in a class)
  - The static variables in a class are ***shared*** by the objects. There can not be multiple copies of the same static variables for different objects. Also because of this reason static variables ***can not be initialized*** using constructors.
  - The static variable must be initialized as follows:

```
type Class::variable = value;
```



static

- Static Member Functions (Static functions in a class)
  - Static member functions also do not depend on the object of the class. We are allowed to invoke a static member function using the object and the '.' operator but it is recommended to invoke the static members using the class name and the scope resolution operator.
  - Static member functions are allowed to access only the ***static data members or other static member functions***, they can not access the ***non-static data members or member functions*** of the class.

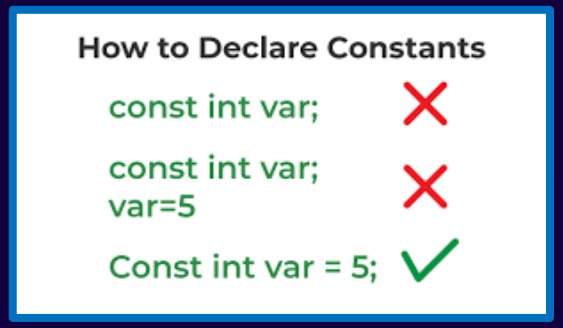


Advanced C++ Programming

const

- Whenever const keyword is attached with any method(), variable, pointer variable, and with the object of a class it prevents that specific object/method()/variable to modify its data items value.
  - Constant Variables
  - Constant Pointer
  - Constant Methods

- Constant Variables
  - There are a certain set of rules for the declaration and initialization of the constant variables:
    - The const variable cannot be left un-initialized at the time of the definition.
    - It cannot be assigned value anywhere in the program.
    - Explicit value needed to be provided to the constant variable at the time of declaration of the constant variable.



const

- Constant Pointer
  - Pointers can be declared with a const keyword.
    - When the *pointer variable point to a const value*:

```
const data_type* var_name;
```

    - When *const pointer variable point to the value*:

```
data_type* const var_name;
```

    - When *const pointer pointing to a const variable*:

```
const data_type* const var_name;
```

Advanced C++ Programming

Qt

const

- Constant Pointer
  - Note: Pass const-argument value to a non-const parameter of a function cause error.

```
1   #include <iostream>
2   using namespace std;
3
4   int foo(int* y)
5   {
6     return *y;
7   }
8
9   // Driver code
10  int main()
11  {
12    int z = 8;
13    const int* x = &z;
14    cout << foo(x);
15    return 0;
16  }
```

  - Output: The compile-time error that will appear as if const value is passed to any non-const argument of the function then the following compile-time error will appear.

Advanced C++ Programming

const

- Constant Methods (A const member function of the class)
  - Constant member functions are those functions which are denied permission to change the values of the data members of their class. To make a member function constant, the keyword “const” is appended to the function prototype and also to the function definition header.

```
class
{
   void foo() const
   {
       //.....
   }
}
```

  - When a function is declared as const, it can be called on any type of object, const object as well as non-const objects.
  - Whenever an object is declared as const, it needs to be initialized at the time of declaration. however, the object initialization while declaring is possible only with the help of constructors.

const

- Constant Objects
  - An object declared as const cannot be modified and hence, can invoke only const member functions as these functions ensure not to modify the object.

```
const Class_Name Object_name;
```

  - When a function is declared as const, it can be called on any type of object, const object as well as non-const objects.
  - Whenever an object is declared as const, it needs to be initialized at the time of declaration. However, the object initialization while declaring is possible only with the help of constructors.

const

- Constant Function Parameters
  - A function() parameters and return type of function() can be declared as constant. Constant values cannot be changed as any such attempt will generate a compile-time error.

- Constant Return Type
  - The value of a return type that is declared const cannot be changed. This is especially useful when giving a reference to a class's internals.

mutable

- In Class Member
  - Sometimes there is requirement to modify one or more data members of class/struct through const function even though you don’t want the function to update other members of class/struct. This task can be easily performed by using mutable keyword.

- In Lambda Expression
  - Since ***C++11*** mutable can be used on a lambda to denote that things captured ***by value*** are ***modifiable*** (they aren't by default).

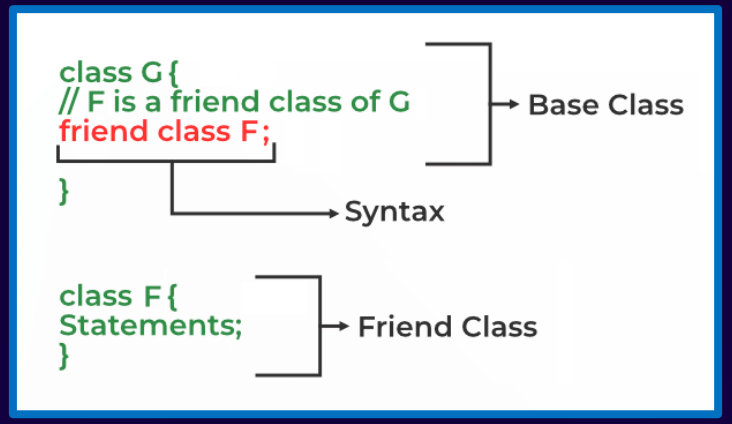
Advanced C++ Programming

friend

- Friend Class
  - A friend class can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes. For example, a LinkedList class may be allowed to access private members of Node.

```
friend class class_name;    // declared in the base class
```

  - Note: We can declare friend class or function anywhere in the base class body whether its private, protected or public block. It works all the same.



friend

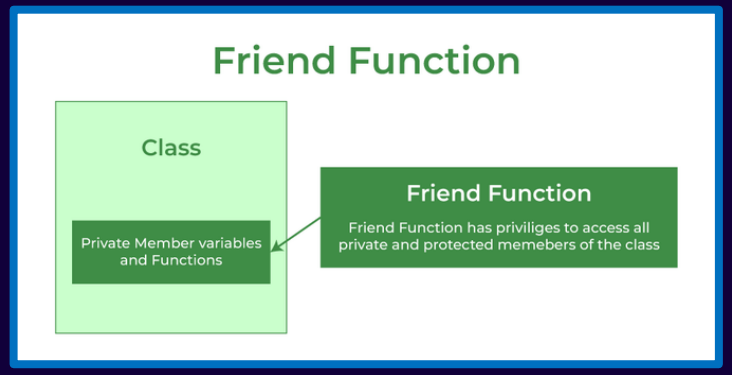
- Friend Function
  - A friend function can be granted special access to private and protected members of a class in C++. They are the non-member functions that can access and manipulate the private and protected members of the class for they are declared as friends. A friend function can be:
    - A global function
    - A member function of another class

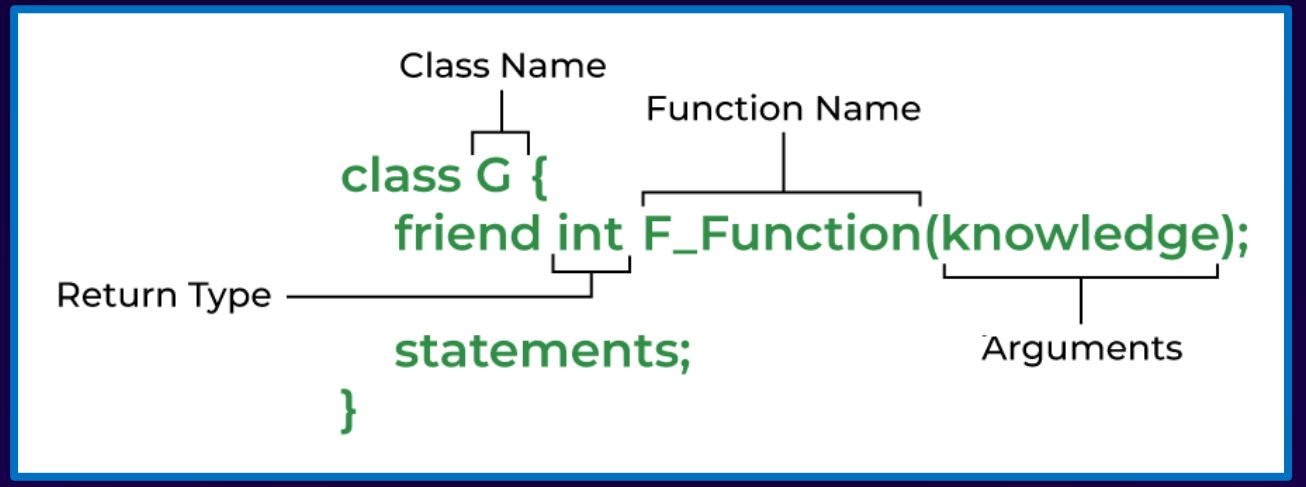
```
// for a global function
friend return_type function_name (arguments);


// for a member function of another class
friend return_type class_name::function_name (arguments);
```

friend

- Friend Function





Advanced C++ Programming

friend

- Friend Function
  - Advantages of Friend Functions
    - A friend function is able to access members without the need of inheriting the class.
    - The friend function acts as a bridge between two classes by accessing their private data.
    - It can be used to increase the versatility of overloaded operators.
    - It can be declared either in the public or private or protected part of the class.
  - Disadvantages of Friend Functions
    - Friend functions have access to private members of a class from outside the class which violates the law of data hiding.
    - Friend functions cannot do any run-time polymorphism in their members.
  - Important Points About Friend Functions and Classes
    - Friends should be used only for limited purposes. Too many functions or external classes are declared as friends of a class with protected or private data access lessens the value of encapsulation of separate classes in object-oriented programming.
    - Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.
    - Friendship is not inherited.
    - The concept of friends is not in Java.

Advanced C++ Programming

References

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

```
data_type &ref = variable;
```

- Applications of Reference in C++
  1. Modify the passed parameters in a function
  2. Avoiding a copy of large structures
  3. In For Each Loop to modify all objects
  4. For Each Loop to avoid the copy of objects

References vs Pointers

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

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

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

2. The pointer variable has n-levels/multiple levels of indirection i.e. single-pointer, double-pointer, triple-pointer. Whereas, the reference variable has only one/single level of indirection.
3. Reference variables cannot be updated.
4. Reference variable is an internal pointer.
5. Declaration of a Reference variable is preceded with the '&' symbol (but do not read it as "address of").

Advanced C++ Programming

Qt

References vs Pointers

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

- Advantages of using References
  1. Safer: Since references must be initialized, wild references like wild pointers are unlikely to exist.
  2. Easier to use: References don't need a dereferencing operator to access the value. They can be used like normal variables. The '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with the dot operator ('.'), unlike pointers where the arrow operator (->) is needed to access members.

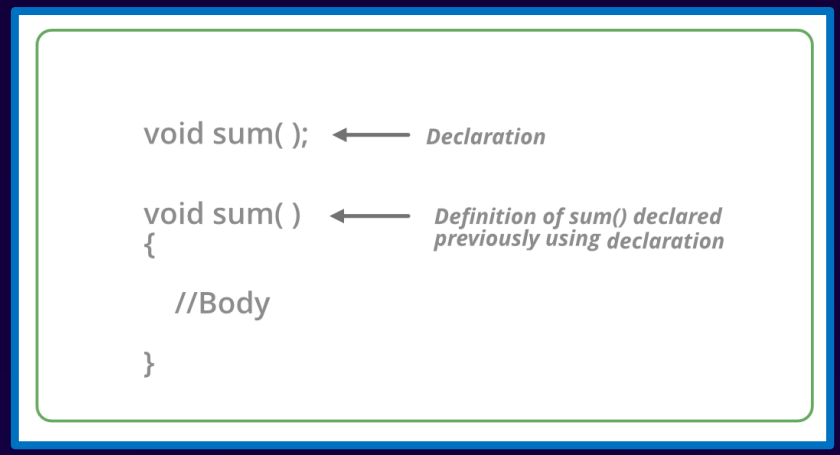
Advanced C++ Programming

References vs Pointers

	References	Pointers
Reassignment	The variable cannot be reassigned in Reference.	The variable can be reassigned in Pointers.
Memory Address	It shares the same address as the original variable.	Pointers have their own memory address.
Work	It is referring to another variable.	It is storing the address of the variable.
Null Value	It does not have null value.	It can have value assigned as null.
Arguments	This variable is referenced by the method pass by value.	The pointer does it work by the method known as pass by reference.

Declaration vs Definition

- Declaration: Declaration of a ***variable*** or ***function*** or ***class*** simply declares that the variable or function exists somewhere in the program, but the memory is ***not allocated*** for them. The declaration of a variable or function serves an important role–it tells the program what its type is going to be. In the case of function declarations, it also tells the program the ***arguments***, their ***data types***, the ***order of those arguments***, and the ***return type*** of the function. So that's all about the declaration.
- Definition: When we define a variable or function, in addition to everything that a declaration does, it also ***allocates memory*** for that variable or function or class. Therefore, we can think of the definition as a superset of the declaration (or declaration as a subset of the definition).



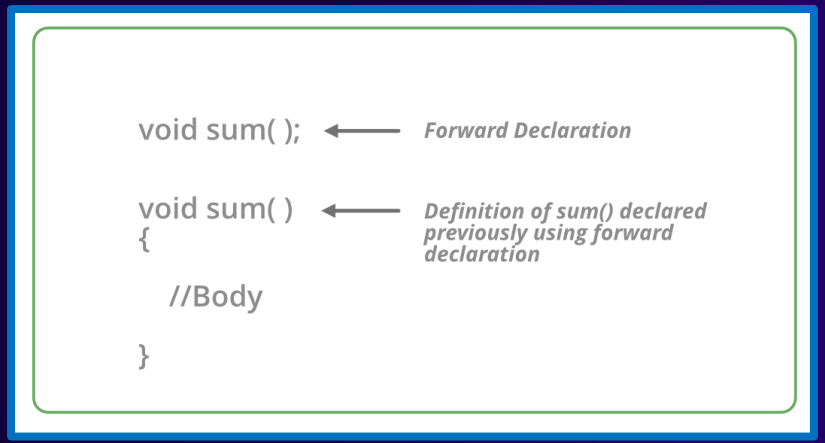
Forward Declaration

- Forward Declaration refers to the beforehand declaration of the syntax or signature of an identifier, variable, function, class, etc. prior to its usage (done later in the program).

Note: Forward declarations in C++ are useful to save in compile time as the compiler does not need to check for translation units in the included header. Also it has other benefits such as preventing namespace pollution, allowing to use PImpl idiom and it may even reduce the binary size in some cases.

```
// Forward Declaration class A
class A;


// Definition of class A
class A{
    // Body
};
```



Advanced C++ Programming

Qt

extern

- Basically, the extern keyword extends the visibility of the C variables and C functions.
  - Extern is a short name for external.
  - The extern variable is used when a particular files need to access a variable from another file.

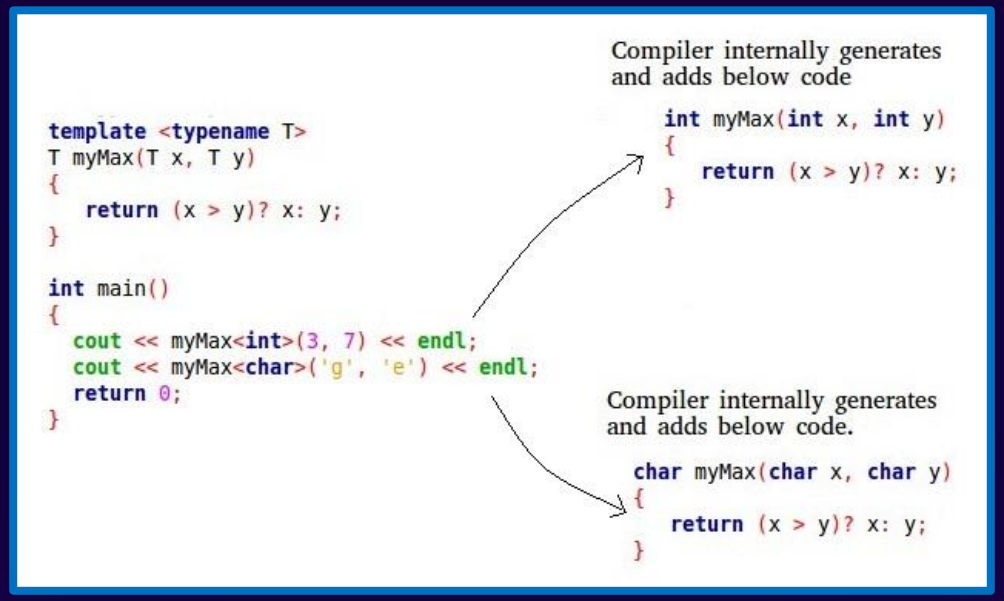
```
extern data_type variable_name;
```

- Properties of extern Variable in C
  - When we write extern some_data_type some_variable_name; *no memory is allocated*. Only the property of the variable is announced.
  - *Multiple declarations* of extern variable is allowed within the file. This is not the case with automatic variables.
  - The extern variable says to the compiler "Go *outside my scope* and you will find the definition of the variable that I declared."
  - The *compiler* believes that whatever that extern variable said is true and produces no error. *Linker* throws an error when it finds no such variable exists.
  - When an extern variable is initialized, then memory for this is allocated and it will be considered defined.

Advanced C++ Programming

Templates

- A template is a simple yet very powerful tool in C++. The simple idea is to pass the ***data type*** as a ***parameter*** so that we don't need to write the same code for different data types.
- How Do Templates Work?
  - Templates are ***expanded at compiler time***. This is like macros. The difference is, that the compiler does type-checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain ***multiple copies*** of the same function/class.



Advanced C++ Programming

Templates

- Type of Templates:
  - Function Templates
    - We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().
  - Class Templates
    - Class templates like function templates, class templates are useful when a class defines something that is independent of the data type. Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.
  - Notes:
    - We can specify default arguments to templates.
    - Both function overloading and templates are examples of polymorphism features of OOP. Function overloading is used when multiple functions do quite similar (not identical) operations, templates are used when multiple functions do identical operations.
    - Each instance of a template contains its own static variable. See Templates and Static variables for more details.
    - Template specialization allows us to have different codes for a particular data type.
    - We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of a template.

Templates

- Source code organization
  - When defining a class template, you must organize the source code in such a way that the member definitions are visible to the compiler when it needs them. You have the choice of using the inclusion model or the explicit instantiation model. In the inclusion model, you include the member definitions in every file that uses a template. This approach is simplest and provides maximum flexibility in terms of what concrete types can be used with your template. Its disadvantage is that it can increase compilation times. The times can be significant if a project or the included files themselves are large. With the explicit instantiation approach, the template itself instantiates concrete classes or class members for specific types. This approach can speed up compilation times, but it limits usage to only those classes that the template implementer has enabled ahead of time. In general, we recommend that you use the inclusion model unless the compilation times become a problem.

Advanced C++ Programming

Templates

- The inclusion model
  - The simplest and most common way to make template definitions visible throughout a translation unit, is to put the definitions in the header file itself. Any .cpp file that uses the template simply has to #include the header. This approach is used in the Standard Library.

```
#ifndef MYARRAY
#define MYARRAY
#include <iostream>

template<typename T, size_t N>
class MyArray
{
    T arr[N];
public:
    // Full definitions:
    MyArray(){}
    void Print()
    {
        for (const auto v : arr)
        {
            std::cout << v << " , ";
        }
    }

    T& operator[](int i)
   {
       return arr[i];
   }
};
#endif
```

Templates

- The explicit instantiation model
  - If the inclusion model isn't viable for your project, and you know definitively the set of types that will be used to instantiate a template, then you can separate out the template code into an .h and .cpp file, and in the .cpp file explicitly instantiate the templates. This approach generates object code that the compiler will see when it encounters user instantiations. In the example, the explicit instantiations are at the bottom of the .cpp file. A MyArray may be used only for double or String types.

```
//MyArray.h
#ifndef MYARRAY
#define MYARRAY

template<typename T, size_t N>
class MyArray
{
    T arr[N];
public:
    MyArray();
    void Print();
    T& operator[](int i);
};
#endif
```

```
//MyArray.cpp
#include <iostream>
#include "MyArray.h"

using namespace std;

template<typename T, size_t N>
MyArray<T,N>::MyArray(){}

template<typename T, size_t N>
void MyArray<T,N>::Print()
{
    for (const auto v : arr)
    {
        cout << v << "'";
    }
    cout << endl;
}

template MyArray<double, 5>;
template MyArray<string, 5>;
```

Exception Handling

- One of the advantages of C++ over C is Exception Handling. Exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution.
- There are two types of exceptions:
  - Synchronous: i.e., exceptions which are beyond the program's control, such as disc failure, keyboard interrupts etc.
  - Asynchronous: Out of range, over flow, ...

- Why Exception Handling?
  - Separation of Error Handling code from Normal Code (In traditional error handling codes, there are always if-else conditions to handle errors).
  - Functions/Methods can handle only the exceptions they choose.
  - Grouping of Error Types.

Exception Handling

- Synchronous

  There is a way to prevent the use of try, catch blocks.

  - try: Represents a block of code that can throw an exception.
  - catch: Represents a block of code that is executed when a particular exception is thrown.
  - throw: Used to throw an exception. Also used to list the exceptions that a function throws but doesn't handle itself.

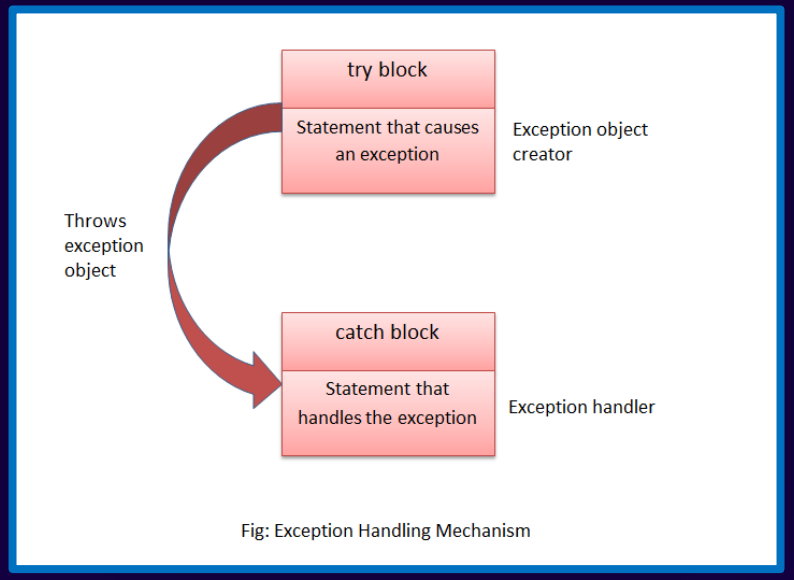


Fig: Exception Handling Mechanism

Advanced C++ Programming

Qt

Exception Handling

- Synchronous

  Notes:

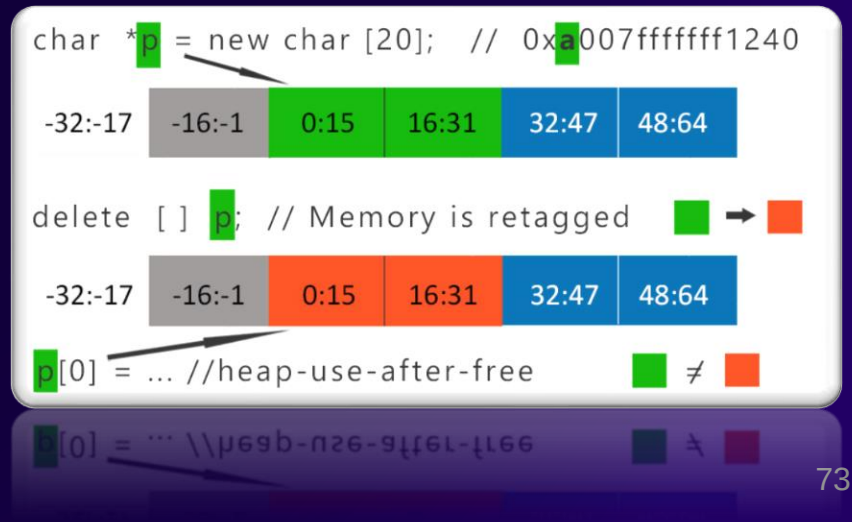
  - There is a special catch block called the ‘catch all’ block, written as catch(…), that can be used to catch all types of exceptions.
  - Implicit type conversion doesn’t happen for primitive types (Example: throw 'a' not catch by catch (int x)).
  - If an exception is thrown and not caught anywhere, the program terminates abnormally.
  - A derived class exception should be caught before a base class exception.
  - Like Java, the C++ library has a standard exception class which is the base class for all standard exceptions.
  - Unlike Java, in C++, all exceptions are unchecked (In checked, all classes have error handling).
  - In C++, try/catch blocks can be nested. Also, an exception can be re-thrown using “throw;“.
  - When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block.
  - Note : The use of Dynamic Exception Specification has been deprecated since C++11 (Place the throw after the function signature).

Advanced C++ Programming

Memory Corruption

- Memory corruption occurs in a computer program when the contents of a memory location are modified due to programmatic behavior that exceeds the intention of the original programmer or program/language constructs; this is termed as violation of memory safety.

- The main causes of memory corruption:
  - Casting to the wrong type
  - Uninitialized pointers
  - Typo error for -> and .
  - Typo error when using * and & (or multiple of either)
  - Mixing new [] and new with delete [] and delete
  - Missing or incorrect copy-constructors
  - Pointer pointing to garbage
  - Calling delete multiple times on the same data
  - Polymorphic baseclasses without virtual destructors

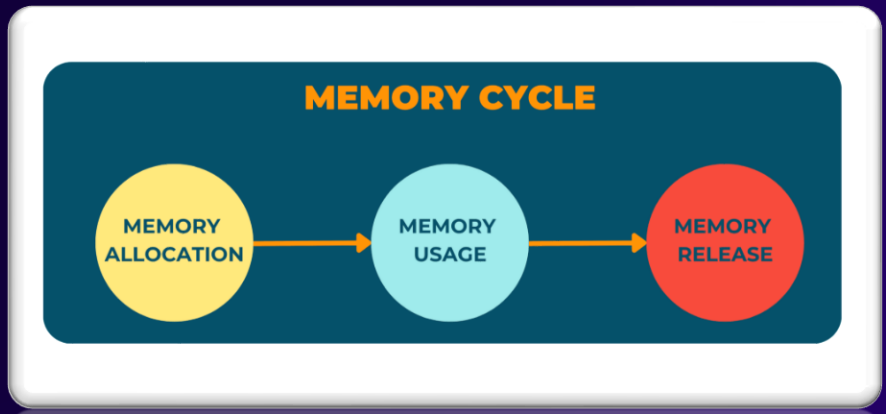


Advanced C++ Programming

Memory Leak

- A memory leak occurs when a programmer does not allocate a previously allocated memory, resulting in deallocation, which thus causes a memory leak. This is because the program does not require this memory but it is still present in the program.

- There are many forms of leaks:
  - Unmanaged leaks (code that allocates unmanaged code)
  - Resource leaks (code that allocates and uses unmanaged resources, like files, sockets)
  - Extended lifetime of objects
  - Incorrect understanding of how GC and .NET memory management works
  - Bugs in the .NET runtime



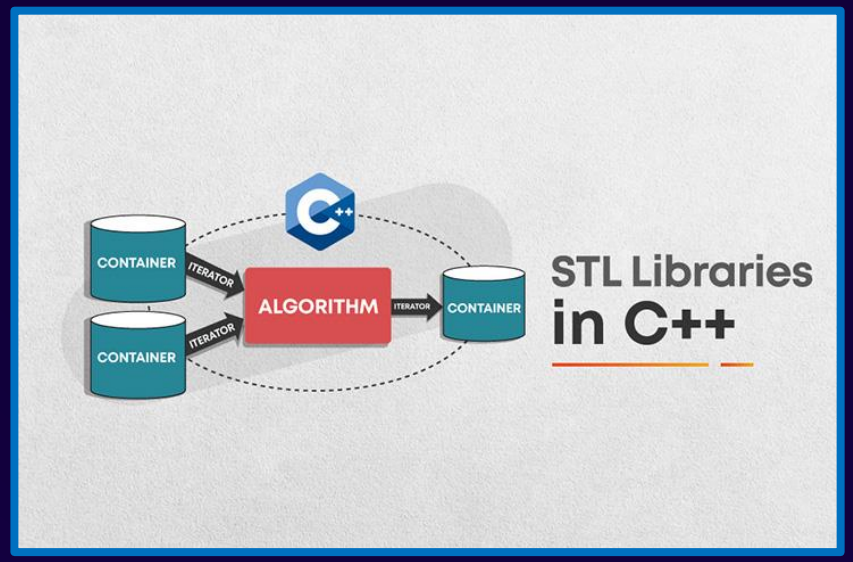
Differences in Stack and Heap Memory

Feature	Stack	Heap
Access Speed	Fast	Slow
Memory Allocation	Handled automatically by runtime	Only automatically handled in high level languages
Performance Cost	Less	More
Size	Fixed Size	Dynamic Size
Variable Access	Local variables only	Global variable access
Data Structure	Linear data structure (stack)	Hierarchical Data Structure (array/tree)
Main Issue	Small fixed amount of memory (stack overflow risk)	Memory fragmentation over time

The C++ Standard Template Library (STL)

STL

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized.
- By using the STL, you can simplify your code, reduce the likelihood of errors, and improve the performance of your programs.



The C++ Standard Template Library (STL)

STL

- Some of the key components of the STL include:
  - Containers: The STL provides a range of containers, such as vector, list, map, set, and stack, which can be used to store and manipulate data.
  - Algorithms: The STL provides a range of algorithms, such as sort, find, and binary_search, which can be used to manipulate data stored in containers.
  - Iterators: Iterators are objects that provide a way to traverse the elements of a container. The STL provides a range of iterators, such as forward_iterator, bidirectional_iterator, and random_access_iterator, that can be used with different types of containers.
  - Function Objects: Function objects, also known as functors, are objects that can be used as function arguments to algorithms. They provide a way to pass a function to an algorithm, allowing you to customize its behavior.
  - Adapters: Adapters are components that modify the behavior of other components in the STL. For example, the reverse_iterator adapter can be used to reverse the order of elements in a container.

The C++ Standard Template Library (STL)

Algorithms

- sort(first_iterator, last_iterator) – To sort the given vector.
- sort(first_iterator, last_iterator, greater<int>()) – To sort the given container/vector in descending order
- reverse(first_iterator, last_iterator) – To reverse a vector. (if ascending -> descending OR if descending -> ascending)
- max_element (first_iterator, last_iterator) – To find the maximum element of a vector.
- min_element (first_iterator, last_iterator) – To find the minimum element of a vector.
- accumulate(first_iterator, last_iterator, initial value of sum) – Does the summation of vector elements
- count(first_iterator, last_iterator,x) – To count the occurrences of x in vector.
- find(first_iterator, last_iterator, x) – Returns an iterator to the first occurrence of x in vector and points to last address of vector ((name_of_vector).end()) if element is not present in vector.
- binary_search(first_iterator, last_iterator, x) – Tests whether x exists in sorted vector or not.
- lower_bound(first_iterator, last_iterator, x) – returns an iterator pointing to the first element in the range [first,last) which has a value not less than ‘x’.
- upper_bound(first_iterator, last_iterator, x) – returns an iterator pointing to the first element in the range [first,last) which has a value greater than ‘x’.

The C++ Standard Template Library (STL)

Algorithms

- arr.erase(position to be deleted) – This erases selected element in vector and shifts and resizes the vector elements accordingly.
- arr.erase(unique(arr.begin(),arr.end()),arr.end()) – This erases the duplicate occurrences in sorted vector in a single line.
- next_permutation(first_iterator, last_iterator) – This modified the vector to its next permutation.
- prev_permutation(first_iterator, last_iterator) – This modified the vector to its previous permutation.
- distance(first_iterator,desired_position) – It returns the distance of desired position from the first iterator.This function is very useful while finding the index.

The C++ Standard Template Library (STL)

Containers

- vector
- list
- deque
- arrays
- forward_list (Introduced in C++11)
- queue
- priority_queue
- stack
- set
- multiset
- map
- multimap

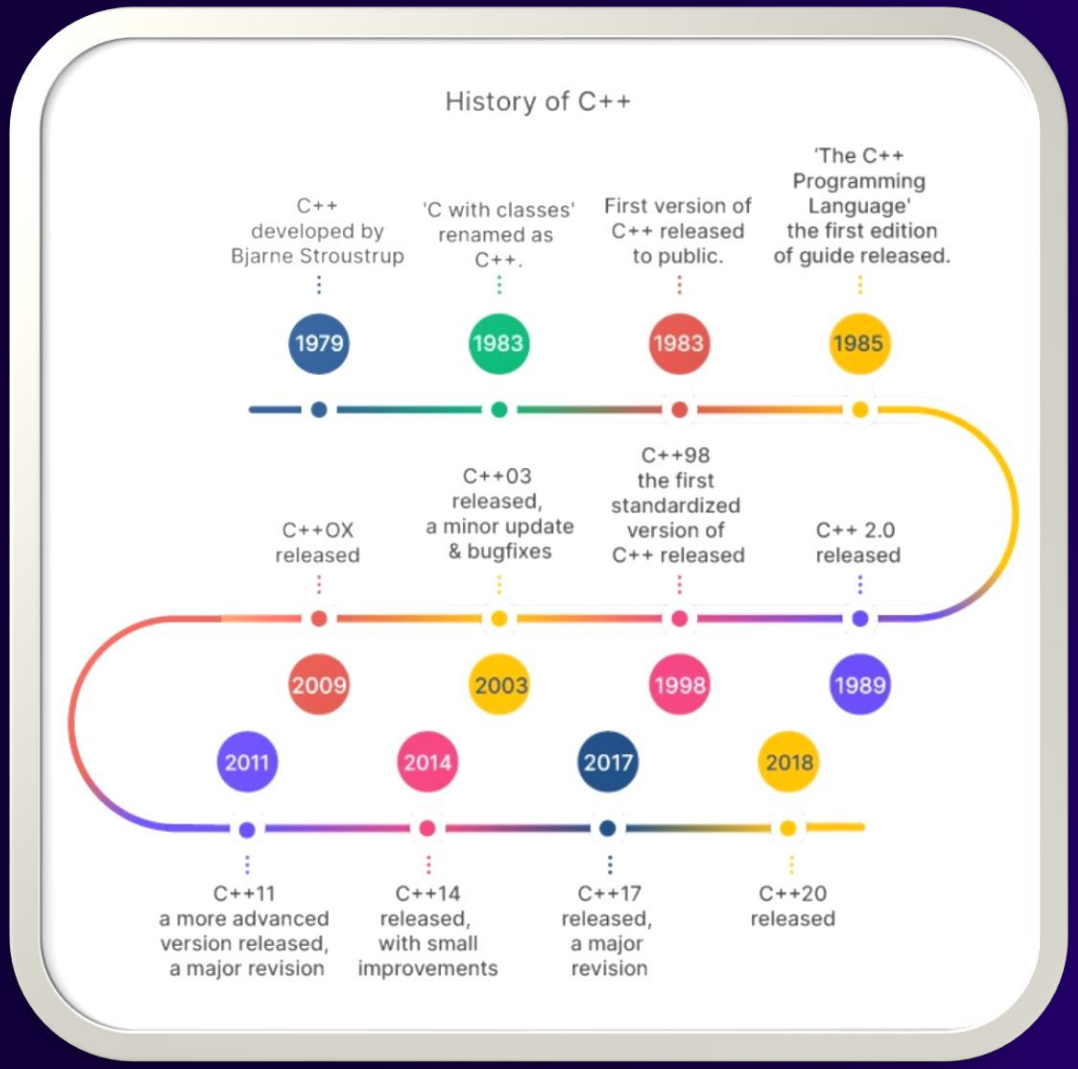
Containers

- unordered_set (Introduced in C++11)
- unordered_multiset (Introduced in C++11)
- unordered_map (Introduced in C++11)
- unordered_multimap (Introduced in C++11)

Advanced C++ Programming

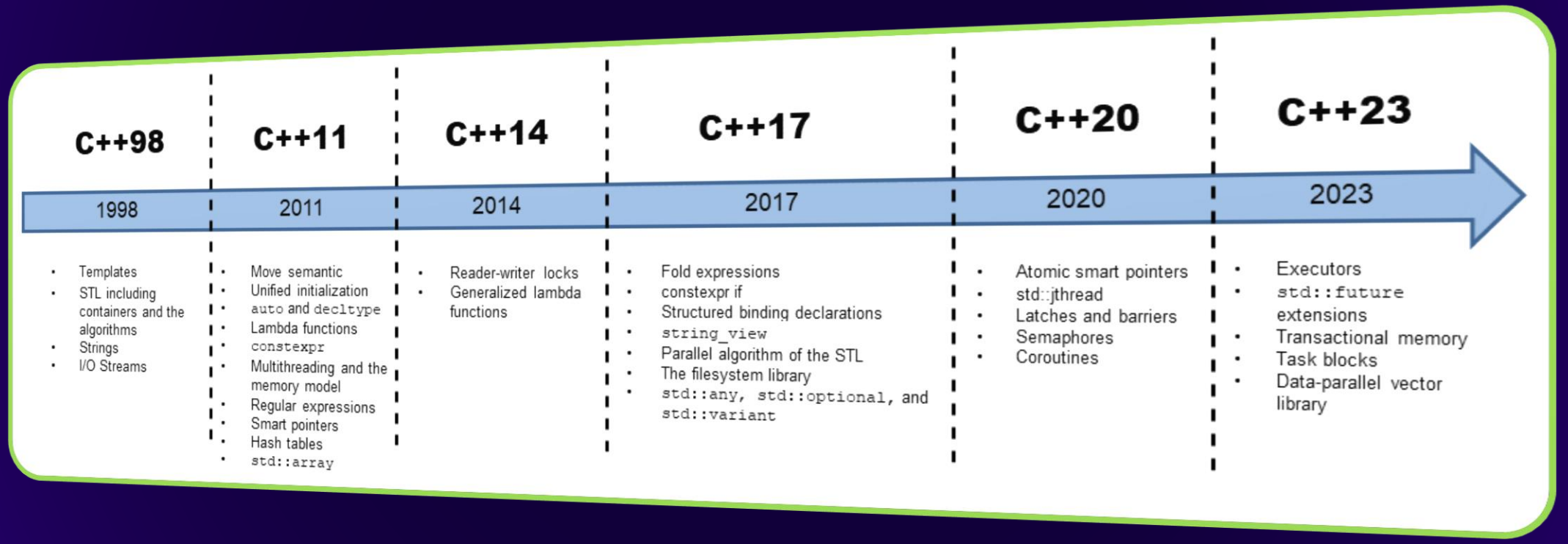
History Of C++

- Versions with major changes:
  - C++ 98
  - C++ 11
  - C++ 17



Advanced C++ Programming

Timeline of C++ Features



New features in C++11

C++11 was the largest change ever made to C++; and due to the changed release schedule, probably will remain the largest single change. It is a well thought out, mostly backward-compatible change that can completely change the way you write code in C++. It is best thought of as almost a new language, a sort of (C++)++ language.

- Types
  - Typing in C++11 is much simpler. If the type is deductible, auto allows you to avoid writing it out.

```
1   // Avoiding double writing on pointers
2   auto pointer = new SomeNamespace::MyLongType(arg1, arg2);
3
4   // Does anyone know the type of an iterator? It's ugly!
5   auto iterator = some_vector.begin();
6
7   // This is useful for prototyping, but probably should be explictly typed in real code to help coders
8   auto type = function_returns_something();
9
10  // Worst use of auto; don't do this
11  auto a = 1;
```

New features in C++11

- **Types**
  - The *NULL keyword*, which is equivalent to 0, causes typing issues. A new nullptr keyword was added, and is not equivalent to zero. This is significantly better type-safety, and should be used instead of NULL.
  - The definitions of *const* and *mutable* changed a little.

- **Containers and Iterators**
  - While iterators existed in previous versions, using them is now part of the language, with the iterating for statement (for each). It allows a unintuitive iteration loop to be written more cleanly and compactly.

```
1    for (vector<double>::iterator iter = vector.begin(); iter != vector.end(); ++iter)
2        *value = 0.0;
3
4    for (auto& value : vector)
5        value = 0.0;
```

New features in C++11

- **Containers and Iterators**
  - A related improvement is the addition of container constructors. C++ also has a variety of different initializers; C++11 added a uniform initializer syntax, so it has even more different initializers.

```
1    std::vector<int> values = { 1,2,3,4,5,6 };
```

- **Functional Programming**
  - Functions are now easier to refer to and create. The lambda function allows an inline function definition, with some perks. The syntax is [](){}, which looks like a normal function definition with the function name and type replaced by the square brackets.

```
1    auto square = [](double x) -> int { return x * x; };
2    double squared_five = square(5.0);
```

New features in C++11

- Functional Programming
  - The lambda function gets interesting when you add something to the square brackets; this is called "capture" and allows you to capture the surrounding variables.

```
1    int i = 0;
2    auto counter = [&i]() { return i++; };
3    counter(); // returns 0
4    counter(); // returns 1
```

  - You can capture by value, by reference, etc. If you use [=] or [&], the lambda function will automatically capture (by value or reference, respectively) any variables mentioned inside the function.
  - Syntax used for capturing variables :
    - [&] : capture all external variables by reference
    - [=] : capture all external variables by value
    - [a, &b] : capture a by value and b by reference

New features in C++11

- Class Improvements
  - Default values for members can be declared in the definition now (as I'm sure you've tried to do in older C++ at least once). You can also call a previous constructor in the initializer list (delegating constructors).

```
1   class A {
2       int x = 0, y, z;
3
4   public:
5       A()
6       {
7           x = 0;
8           y = 0;
9           z = 0;
10      }
11
12      // Constructor delegation
13      A(int z) : A()
14      {
15          this->z = z; // Only update z
16      }
17  };
```

New features in C++11

- Compile Time Improvements
  - The slow removal of the ugly, error-prone macro programming has started in C++11, with constexpr. constexpr is a feature added in C++ 11. The main idea is a performance improvement of programs by doing computations at compile time rather than run time. Note that once a program is compiled and finalized by the developer, it is run multiple times by users. The idea is to spend time in compilation and save time at run time (similar to template metaprogramming).

```
1   #include <iostream>
2
3   constexpr int product(int x, int y) { return (x * y); }
4
5   int main()
6   {
7       constexpr int x = product(10, 20);
8       std::cout << x;
9       return 0;
10  }
```

New features in C++11

- Compile Time Improvements
  - A function be declared as constexpr:
    1. In C++ 11, a constexpr function should contain only one return statement. C++ 14 allows more than one statement.
    2. constexpr function should refer only to constant global variables.
    3. constexpr function can call only other constexpr functions not simple functions.
    4. The function should not be of a void type.
    5. In C++11, prefix increment (++v) was not allowed in constexpr function but this restriction has been removed in C++14.

```cpp
// C++ program to demonstrate constexpr function to evaluate
// the size of array at compile time.
#include <iostream>

constexpr int product(int x, int y) { return (x * y); }

int main()
{
    int arr[product(2, 3)] = { 1, 2, 3, 4, 5, 6 };
    std::cout << arr[5];
    return 0;
}
```

Advanced C++ Programming

Qt

New features in C++11

- Compile Time Improvements
  - constexpr with constructors: A constructor that is declared with a constexpr specifier is a constexpr constructor also constexpr can be used in the making of constructors and objects. A constexpr constructor is implicitly inline.
  - Restrictions on constructors that can use constexpr:
    - No virtual base class
    - Each parameter should be literal
    - It is not a try block function

- Std Library Improvements
  - The powerful std::shared_ptr and std::unique_ptr remove most of the reasons to fear pointers.
  - A chrono library was added for consistent timekeeping on all platforms.
  - A threading library provides tools that work with the new functional tools and makes threading easy, and also a mutex and atomic library to support it.
  - A regular expression library was added.
  - Random number generation is finally properly supported, with a good set of algorithms and distributions.
  - Several container libraries were added.

New features in C++11

- Variadic Templates
  - Variadic templates allow a function or constructor to take an unlimited number of arguments of any type.

```
1  // To handle base case of below recursive
2  // Variadic function Template
3  void print()
4  {
5  }
6
7  // Variadic function Template that takes
8  // variable number of arguments and prints
9  // all of them.
10 template <typename T, typename... Types>
11 void print(T var1, Types... var2)
12 {
13   cout << var1 << endl;
14   print(var2...);
15 }
16
17 // Driver code
18 int main()
19 {
20   print(1, 2, 3.14, "Hello");
21
22   return 0;
23 }
```

New features in C++11

- Move Semantics
  - One of the more fundamental changes in the language was the promotion of move semantics to a language feature, as well as stronger guidelines on auto-optimization.

```
Object item = Object_returning_function();
```

  - Here, you create a Object inside the function, and then copy it to a new object item, then delete the old object. It's horribly wasteful in both time and memory; if you don't have enough memory for two separate copies of Object, you can crash your program. In C++11, not only is the idea of a move instead of a copy added, the compiler is generally recommended to do that for you if it can. So the value will simply be moved, with no changes to either the function or the line above, as long as there are no references retained to the object inside the function (though globals, members, parameters, etc).

constexpr vs inline Functions

Constexpr	Inline Functions
It removes the function calls as it evaluates the code/expressions in compile time.	It hardly removes any function call as it performs an action on expression in the run time.
It is possible to assess the value of the variable or function at compile time.	It is not possible to assess the value of the function or variable at compile time.
It does not imply external linkage.	It implies external linkage.

constexpr vs const

- constexpr is mainly for optimization while const is for practically const objects like the value of Pi.
- Both of them can be applied to member methods. Member methods are made const to make sure that there are no accidental changes in the method.
- On the other hand, the idea of using constexpr is to compute expressions at compile time so that time can be saved when the code is run.
- const can only be used with non-static member functions whereas constexpr can be used with member and non-member functions, even with constructors but with condition that argument and return type must be of literal types.

Advanced C++ Programming

Literals

- Literals are the Constant values that are assigned to the constant variables. Literals represent fixed values that cannot be modified. Literals contain memory but they do not have references as variables. For example, “const int = 5;”, is a constant expression and the value 5 is referred to as a constant integer literal.

- Literal Types
  - Integer literal
  - Float literal
  - Character literal
  - String literal
  - Boolean Literals

- Integer Literals
  - Integer literals are used to represent and store the integer values only. Integer literals are expressed in two types i.e.
    - Prefixes
    - Suffixes

Advanced C++ Programming

Qt

Literals

- Integer Literals
  - Prefixes
    a. Decimal-literal(base 10): A non-zero decimal digit followed by zero or more decimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

```
Example: 56, 78
```

    b. Octal-literal(base 8): a 0 followed by zero or more octal digits(0, 1, 2, 3, 4, 5, 6, 7).

```
Example: 045, 076, 06210
```

    c. Hex-literal(base 16): 0x or 0X followed by one or more hexadecimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F).

```
Example: 0x23A, 0Xb4C, 0xFEA
```

    d. Binary-literal(base 2): 0b or 0B followed by one or more binary digits(0, 1).

```
Example: 0b101, 0B111
```

Literals

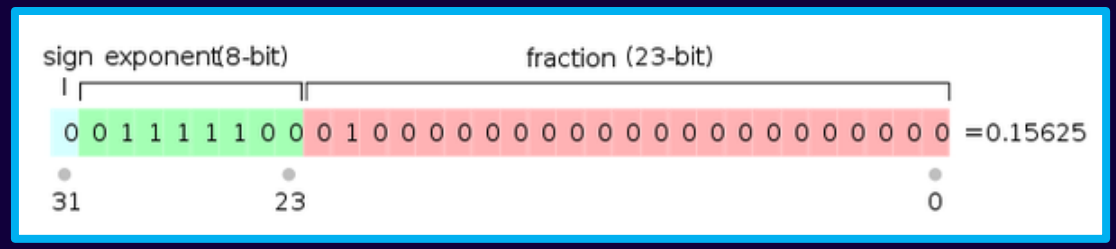
- Integer Literals
  - Suffixes: The Suffix of the integer literal indicates the type in which it is to be read. For example: 12345678901234LL indicates a long long integer value 12345678901234 because of the suffix LL.

```
Example: 12345678901234LL
```

a. int: No suffix is required because integer constant is by default assigned as an int data type.
b. unsigned int: character ***u*** or ***U*** at the end of an integer constant.
c. long int: character ***l*** or ***L*** at the end of an integer constant.
d. unsigned long int: character ***ul*** or ***UL*** at the end of an integer constant.
e. long long int: character ***ll*** or ***LL*** at the end of an integer constant.
f. unsigned long long int: character ***ull*** or ***ULL*** at the end of an integer constant.

Literals

- Floating-Point Literals
  - These are used to represent and store real numbers. The real number has an integer part, real part, fractional part, and exponential part.



  - Valid Floating Literals:
    a. 10.125
    b. 1.215-10L
    c. 10.5E-3
  - Invalid Floating Literals:
    a. 123E
    b. 1250f
    c. 0.e879

Literals

- Character Literal
  - This refers to the literal that is used to store a single character within a single quote. To store multiple characters, one needs to use a character array.
    a. char type: This is used to store normal character literal or narrow-character literals. This is supported by both C and C++.

```
Example: char chr = 'G';
```

    b. wchar_t type: This is used to store normal character literal or narrow-character literals. This is supported by both C and C++.

```
Example: wchar_t chr = L'G';
```

Literals

- Character Literal
  - This refers to the literal that is used to store a single character within a single quote. To store multiple characters, one needs to use a character array.
    a. char type: This is used to store normal character literal or narrow-character literals. This is supported by both C and C++.

```
Example: char chr = 'G';
```

    b. wchar_t type: This is used to store normal character literal or narrow-character literals. This is supported by both C and C++.

```
Example: wchar_t chr = L'G';
         wchar_t chr = L'\x3b1';
```

Literals

- String Literals
  - String literals are similar to that character literals, except that they can store multiple characters and uses a double quote to store the same.

```
Example: string stringVal = "GeeksforGeeks"
```

- Boolean Literals
  - This literal is provided only in C++ and not in C.

```
Example: bool var = true/false;
```

Literals

- User Defined Literals (UDL)
  - A literal is used for representing a fixed value in a program. A literal could be anything in a code like a, b, c2. , 'ACB', etc.

```
// Examples of classical literals for built-in types.
42       // int
2.4      // double
3.2F     // float
'w'      // char
32ULL    // Unsigned long long
0xD0     // Hexadecimal unsigned
"cd"     // C-style string(const char[3]")
```

  - UDLs are treated as a call to a literal operator. Only suffix form is supported. The name of the literal operator is operator "" followed by the suffix (operator "" suffix-identifier c++11).

Obfuscation

Obfuscation

- In software development, obfuscation is the act of creating source or machine code that is difficult for humans or computers to understand. Other approaches include stripping out potentially revealing metadata, replacing class and variable names with meaningless labels and adding unused or meaningless code to an application script.
- Obfuscation techniques:
  - *Renaming:* The obfuscator alters the methods and names of variables. The new names may include unprintable or invisible characters.
  - *Packing:* This compresses the entire program to make the code unreadable.
  - *Control flow:* The decompiled code is made to look like spaghetti logic, which is unstructured and hard to maintain code where the line of thought is obscured. Results from this code are not clear, and it's hard to tell what the point of the code is by looking at it.

Obfuscation

Obfuscation

- Obfuscation techniques:
  - *Instruction pattern transformation:* This approach takes common instructions created by the compiler and swaps them for more complex, less common instructions that effectively do the same thing.
  - *Dummy code insertion:* Dummy code can be added to a program to make it harder to read and reverse engineer, but it does not affect the program's logic or outcome.
  - *Metadata or unused code removal:* Unused code and metadata give the reader extra information about the program, much like annotations on a Word document, that can help them read and debug it. Removing metadata and unused code leaves the reader with less information about the program and its code.

Obfuscation

Obfuscation

- Obfuscation techniques:
  - *Opaque predicate insertion:* A predicate in code is a logical expression that is either true or false. Opaque predicates are conditional branches -- or if-then statements -- where the results cannot easily be determined with statistical analysis. Inserting an opaque predicate introduces unnecessary code that is never executed but is puzzling to the reader trying to understand the decompiled output.
  - *Anti-debug:* Legitimate software engineers and hackers use debug tools to examine code line by line. With these tools, software engineers can spot problems with the code, and hackers can use them to reverse engineer the code. IT security pros can use anti-debug tools to identify when a hacker is running a debug program as part of an attack. Hackers can run anti-debug tools to identify when a debug tool is being used to identify the changes they are making to the code.
  - *Anti-tamper:* These tools detect code that has been tampered with, and if it has been modified, it stops the program.

Obfuscation

Obfuscation

- Obfuscation techniques:
  - *Anti-tamper:* These tools detect code that has been tampered with, and if it has been modified, it stops the program.
  - *String encryption:* This method uses encryption to hide the strings in the executable and only restores the values when they are needed to run the program. This makes it difficult to go through a program and search for particular strings.
  - *Code transposition:* This is the reordering of routines and branches in the code without having a visible effect on its behavior.

Obfuscation

Obfuscator tools

- Stunnix C/C++ Obfuscator
- Mangle-It C++ Code Obfuscator
- CodeMorph
- Online Offuscator (https://picheta.me/obfuscator)
- TinyObfuscate
- Themida
  - Themida is a powerful software protection system designed for software developers who wish to protect their applications against advanced reverse engineering and software cracking. Themida uses the SecureEngine® protection system to achieve its goals, making it really difficult to break using the traditional and newest cracking tools.

Binary Executable File Formats

Windows

- Portable Executable (PE) Format

Linux

- Linkable (ELF) Format

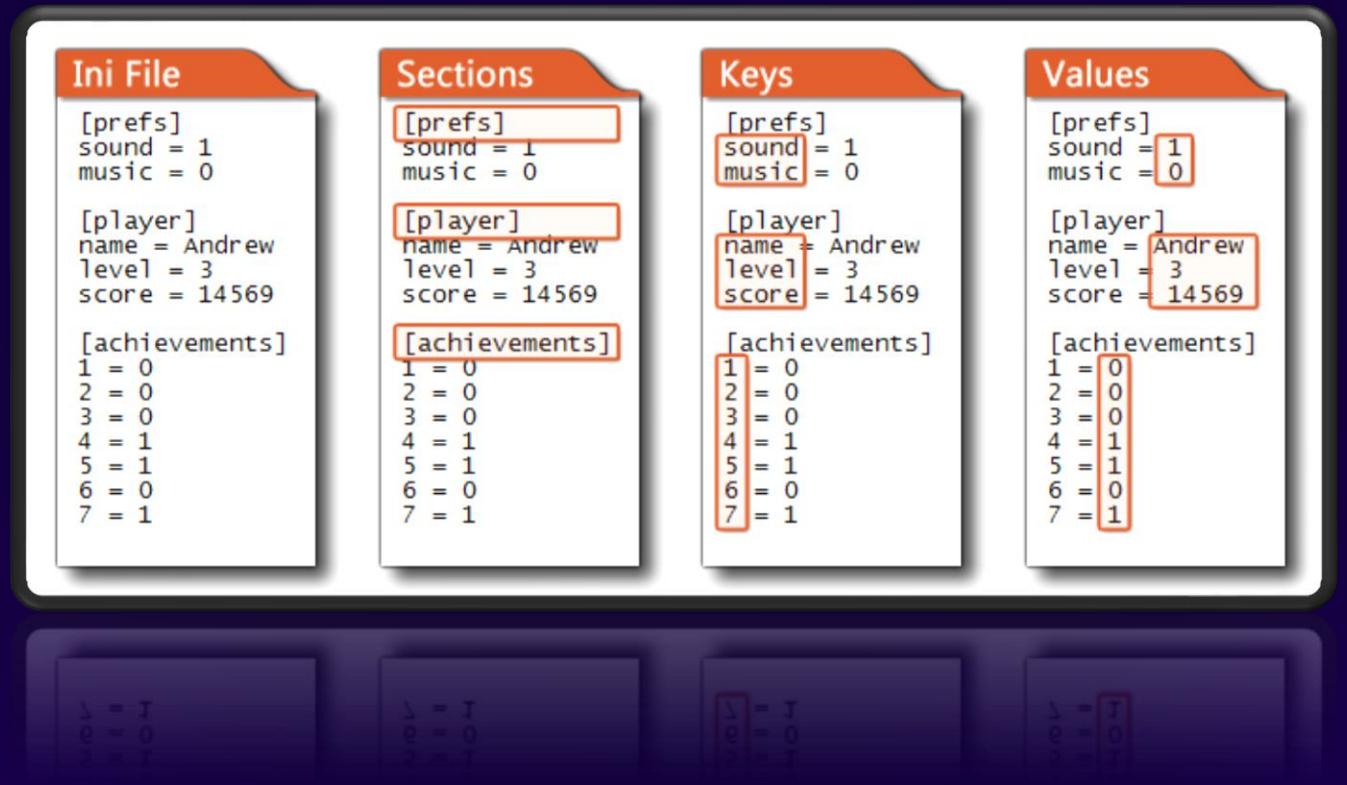
Binary file viewing and editing tools

- Resource Hacker
- ResEdit
- PE Explorer
- PE-bear
- PE Viewer
- XN Resource Editor

Qt Advanced Programming >> INI

INI file

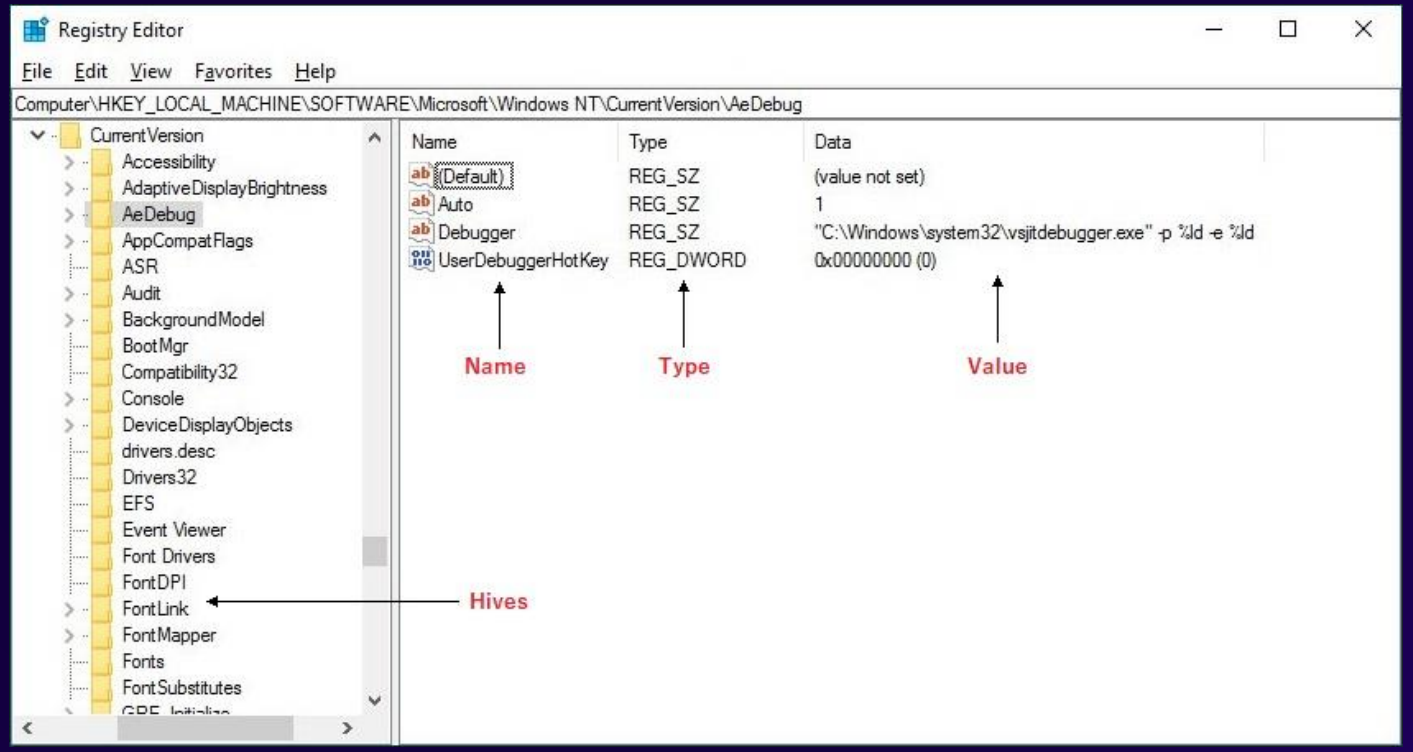
- An INI file (INItialization file) is a configuration file for computer software that consists of a text-based content with a structure and syntax comprising ***key–value*** pairs for properties, and sections that organize the properties.



Qt Advanced Programming >> Registry

Registry

- The Windows Registry is a hierarchical database that stores low-level settings for the Microsoft Windows operating system and for applications that opt to use the registry. The kernel, device drivers, services, Security Accounts Manager, and user interfaces can all use the registry.



Qt Advanced Programming >> QVariant

QVariant

- The QVariant class acts like a union for the most common Qt data types.
- A QVariant object holds a ***single value*** of a single typeId() at a time. (Some types are multi-valued, for example a string list.) You can find out what type, T, the variant holds, convert it to a different type using convert(), get its value using one of the toT() functions (e.g., toSize()), and check whether the type can be converted to a particular type using canConvert().
- The methods named toT() (e.g., toInt(), toString()) are const. If you ask for the stored type, they return a copy of the stored object. If you ask for a type that can be generated from the stored type, toT() copies and converts and leaves the object itself unchanged. If you ask for a type that cannot be generated from the stored type, the result depends on the type.

Header:	#include <QVariant>
qmake:	QT += core

QVariant

```
QDataStream out(...);
QVariant v(123);                // The variant now contains an int
int x = v.toInt();              // x = 123
out << v;                       // Writes a type tag and an int to out
v = QVariant(tr("hello"));      // The variant now contains a QString
int y = v.toInt();              // y = 0 since v cannot be converted to an int
QString s = v.toString();       // s = tr("hello")  (see QObject::tr())
out << v;                       // Writes a type tag and a QString to out
...
QDataStream in(...);            // (opening the previously written stream)
in >> v;                        // Reads an Int variant
int z = v.toInt();              // z = 123
qDebug("Type is %s",            // prints "Type is int"
       v.typeName());
v = v.toInt() + 100;            // The variant now holds the value 223
v = QVariant(QStringList());    // The variant now holds a QStringList
```

Qt Advanced Programming >> QSettings

QSettings

- The QSettings class provides persistent platform-independent application settings. Users normally expect an application to remember its settings (window sizes and positions, options, etc.) across sessions. This information is often stored in the system registry on Windows, and in property list files on macOS and iOS. On Unix systems, in the absence of a standard, many applications (including the KDE applications) use INI text files.

Methods

- QSettings(QSettings::Scope scope, QObject *parent = nullptr);
  - This enum specifies whether settings are user-specific or shared by all users of the same system.
- QSettings(const QString &fileName, QSettings::Format format, QObject *parent = nullptr);
  - If format is QSettings::NativeFormat, the meaning of fileName depends on the platform. On Unix, fileName is the name of an INI file. On macOS and iOS, fileName is the name of a .plist file. On Windows, fileName is a path in the system registry.
  - If format is QSettings::IniFormat, fileName is the name of an INI file.

    Warning: This function is provided for convenience. It works well for accessing INI or .plist files generated by Qt, but might fail on some syntaxes found in such files originated by other programs.

Header:	#include <QSettings>
qmake:	QT += core

Qt Advanced Programming >> QSettings

Methods

- QSettings(QSettings::Format format, QSettings::Scope scope, const QString &organization, const QString &application = QString(), QObject *parent = nullptr);
  - If scope is QSettings::UserScope, the QSettings object searches user-specific settings first, before it searches system-wide settings as a fallback. If scope is QSettings::SystemScope, the QSettings object ignores user-specific settings and provides access to system-wide settings.
  - If format is QSettings::NativeFormat, the native API is used for storing settings. If format is QSettings::IniFormat, the INI format is used.
  - If no application name is given, the QSettings object will only access the organization-wide locations.
- QSettings(QSettings::Scope scope, const QString &organization, const QString &application = QString(), QObject *parent = nullptr);
  - If scope is QSettings::UserScope, the QSettings object searches user-specific settings first, before it searches system-wide settings as a fallback. If scope is QSettings::SystemScope, the QSettings object ignores user-specific settings and provides access to system-wide settings.
  - The storage format is set to QSettings::NativeFormat (i.e. calling setDefaultFormat() before calling this constructor has no effect).
  - If no application name is given, the QSettings object will only access the organization-wide locations.

Qt Advanced Programming >> QSettings

Methods

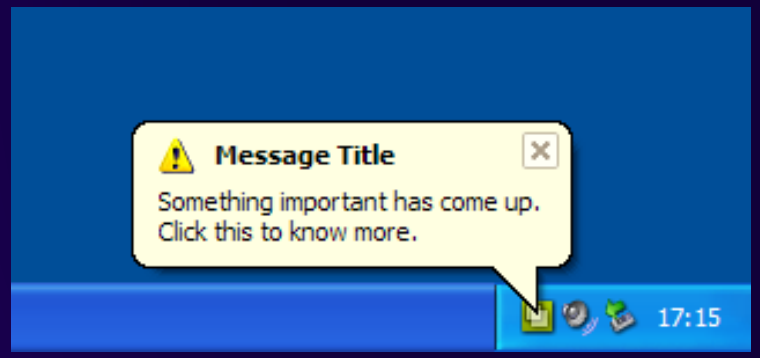
- void QSettings::setValue(const QString &key, const QVariant &value);
  - Sets the value of setting key to value. If the key already exists, the previous value is overwritten.

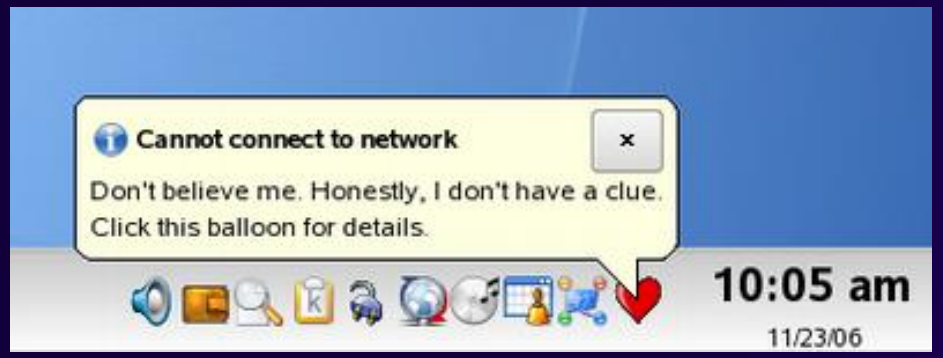
    Note that the Windows registry and INI files use case-insensitive keys, whereas the CFPreferences API on macOS and iOS uses case-sensitive keys.
- QVariant QSettings::value(const QString &key, const QVariant &defaultValue = QVariant()) const;
  - Returns the value for setting key. If the setting doesn't exist, returns defaultValue.
  - If no default value is specified, a default QVariant is returned.

    Note that the Windows registry and INI files use case-insensitive keys, whereas the CFPreferences API on macOS and iOS uses case-sensitive keys.
- void QSettings::beginGroup(const QString &prefix);
  - The current group is automatically prepended to all keys specified to QSettings. In addition, query functions such as childGroups(), childKeys(), and allKeys() are based on the group. By default, no group is set.
- void QSettings::endGroup();
  - Resets the group to what it was before the corresponding beginGroup() call.

QSystemTrayIcon

- The QSystemTrayIcon class provides an icon for an application in the system tray.
- Modern operating systems usually provide a special area on the desktop, called the system tray or notification area, where long-running applications can display icons and short messages.





Header:	`#include <QSystemTrayIcon>`
qmake:	`QT += widgets`

Qt Advanced Programming >> QSystemTrayIcon

Methods

- void setContextMenu(QMenu *menu);
  - Sets the specified menu to be the context menu for the system tray icon.
  - The menu will pop up when the user requests the context menu for the system tray icon by clicking the mouse button.

    Note: The system tray icon does not take ownership of the menu. You must ensure that it is deleted at the appropriate time by, for example, creating the menu with a suitable parent object.
- void setIcon(const QIcon &icon);
  - This property holds the system tray icon.
  - On Windows, the system tray icon size is 16x16; on X11, the preferred size is 22x22. The icon will be scaled to the appropriate size as necessary.
- void showMessage(const QString &title, const QString &message, QSystemTrayIcon::MessageIcon icon = QSystemTrayIcon::Information, int millisecondsTimeoutHint = 10000);
  - Shows a balloon message for the entry with the given title, message and icon for the time specified in millisecondsTimeoutHint. title and message must be plain text strings.

Qt Advanced Programming >> QSignalMapper

QSignalMapper

- This class collects a set of parameterless signals, and re-emits them with integer, string or widget parameters corresponding to the object that sent the signal. Note that in most cases you can use lambdas for passing custom parameters to slots.

Signals

- void mappedInt(int i);
- void mappedObject(QObject *object);
- void mappedString(const QString &text);

Methods

- void setMapping(QObject *sender, int id);
- void setMapping(QObject *sender, const QString &text);
- void setMapping(QObject *sender, QObject *object);

Header:	#include <QSignalMapper>
qmake:	QT += core

Qt Advanced Programming >> Sender

Sender (QObject::sender())

- Returns a pointer to the object that sent the signal, if called in a slot activated by a signal; otherwise it returns nullptr. The pointer is valid only during the execution of the slot that calls this function from this object's thread context.

  Note: The pointer returned by this function becomes invalid if the sender is destroyed, or if the slot is disconnected from the sender's signal.

  Warning: This function violates the object-oriented principle of modularity. However, getting access to the sender might be useful when many signals are connected to a single slot.

  Warning: As mentioned above, the return value of this function is not valid when the slot is called via a Qt::DirectConnection from a thread different from this object's thread. Do not use this function in this type of scenario.

Qt Advanced Programming >> Dynamic Properties

Dynamic Properties (in QObjects)

- From Qt 4.2, dynamic properties can be added to and removed from QObject instances at run-time. Dynamic properties do not need to be declared at compile-time, yet they provide the same advantages as static properties and are manipulated using the same API - using property() to read them and setProperty() to write them.
- From Qt 4.3, dynamic properties are supported by Qt Designer, and both standard Qt widgets and user-created forms can be given dynamic properties.

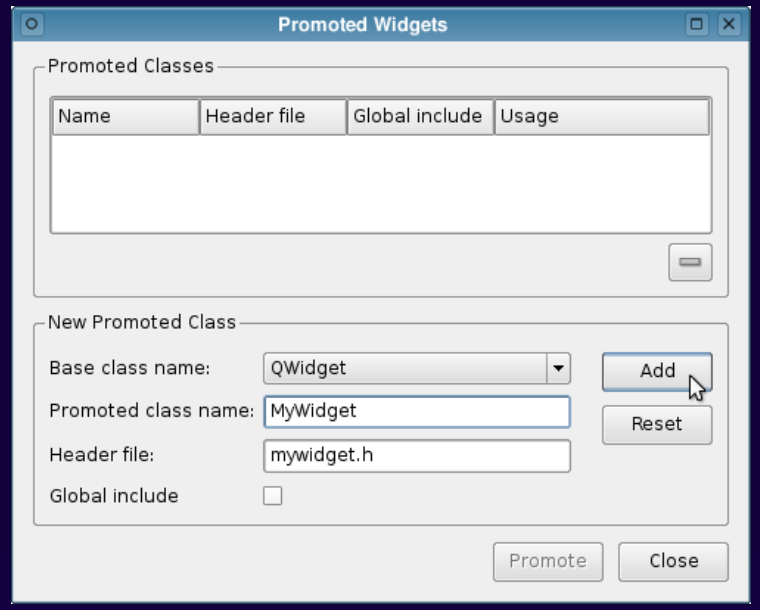
Methods

- bool QObject::setProperty(const char *name, const QVariant &value);
  - Sets the value of the object's name property to value.
- QVariant QObject::property(const char *name) const;
  - Returns the value of the object's name property.
  - If no such property exists, the returned variant is invalid.
- QList<QByteArray> QObject::dynamicPropertyNames() const;
  - Returns the names of all properties that were dynamically added to the object using setProperty().

Qt Advanced Programming >> Promote/Demote

Types of promotion

- Programmatically promote
- With Qt Designer
  - Qt Designer can display custom widgets through its extensible plugin mechanism, allowing the range of designable widgets to be extended by the user and third parties. Alternatively, it is possible to use existing widgets as placeholders for widget classes that provide similar APIs.



Qt Advanced Programming >> Third party libraries

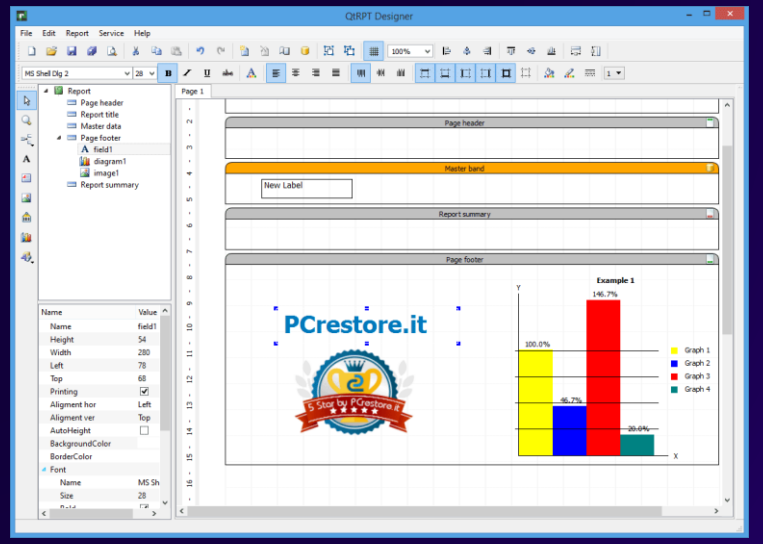
Third party libraries (External libraries)

- Source: https://github.com/qtproject/qt-solutions
  - QtSingleApplication
    - The QtSingleApplication class provides an API to detect and communicate with running instances of an application.
    - This class allows you to create applications where only one instance should be running at a time. I.e., if the user tries to launch another instance, the already running instance will be activated instead. Another usecase is a client-server system, where the first started instance will assume the role of server, and the later instances will act as clients of that server.
  - QtService
    - The QtService is a convenient template class that allows you to create a service for a particular application type.
    - A Windows service or Unix daemon (a "service"), is a program that runs "in the background" independently of whether a user is logged in or not. A service is often set up to start when the machine boots up, and will typically run continuously as long as the machine is on.

Qt Advanced Programming >> Third party libraries

Third party libraries (External libraries)

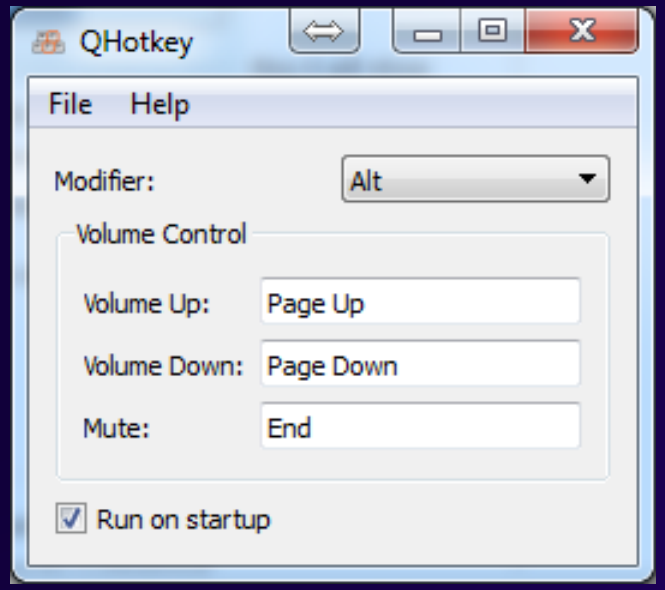
- Source: https://github.com/qt-project/qtrpt
  - QtRPT (Report)
    - QtRPT is the easy-to-use report engine written in C++ QtToolkit. It allows combining several reports in one XML file. For separately taken field, you can specify some condition depending on which this field will display in different font and background color, etc. The project consists of two parts: report library QtRPT and report designer application QtRptDesigner. Report file is a file in XML format. The report designer makes easy to create report XML file.



Qt Advanced Programming >> Third party libraries

Third party libraries (External libraries)

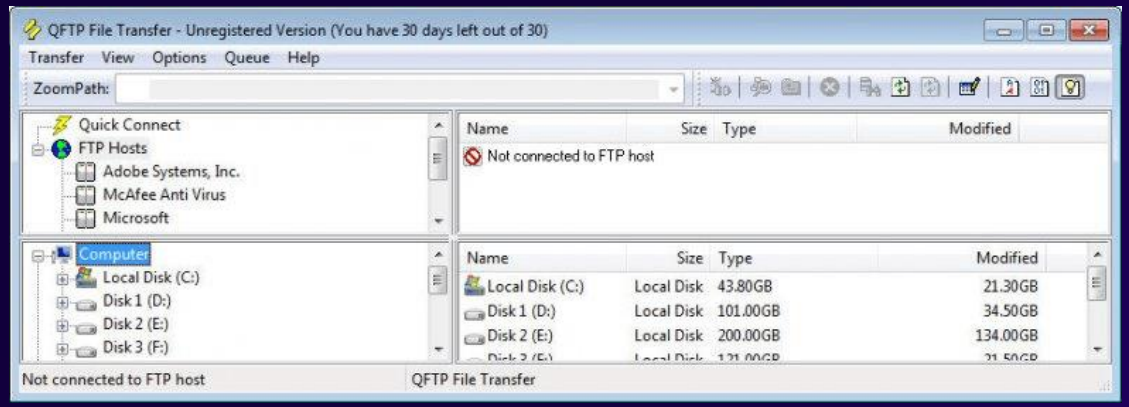
- Source: https://github.com/Skycoder42/QHotkey
  - QHotkey
    - A global shortcut/hotkey for Desktop Qt-Applications.
    - The QHotkey is a class that can be used to create hotkeys/global shortcuts, aka shortcuts that work everywhere, independent of the application state. This means your application can be active, inactive, minimized or not visible at all and still receive the shortcuts.



Qt Advanced Programming >> Third party libraries

Third party libraries (External libraries)

- Source: https://github.com/qt/qtftp
  - QtFtp
    - The QFtp class provides an implementation of the client side of FTP protocol.
    - This class provides a direct interface to FTP that allows you to have more control over the requests. However, for new applications, it is recommended to use QNetworkAccessManager and QNetworkReply, as those classes possess a simpler, yet more powerful API.
    - The class works asynchronously, so there are no blocking functions. If an operation cannot be executed immediately, the function will still return straight away and the operation will be scheduled for later execution. The results of scheduled operations are reported via signals. This approach depends on the event loop being in operation.



Thank You

By Ali Panahi