ISSN: 2393-8528

Contents lists available at www.ijicse.in

International Journal of Innovative Computer Science & Engineering

Volume 4 Issue 3; May-June-2017; Page No. 102-109

Modern C++ Design Patterns and Idioms with New Features

Saqib Mamoon[1], Irfan Yaqoob[2], Nouman Naseer[3], Jazeb Akram[4], Raheel Osman[5], Sidra Zulfiqar[6]

[1]University of the Punjab, Jhelum Campus, Jhelum, Pakistan

bcs.12.06@pujc.edu.pk

[2]University of the Punjab, Jhelum Campus, Jhelum, Pakistan

bcs.s12.33@pujc.edu.pk

[3]University of the Punjab, Jhelum Campus, Jhelum, Pakistan

bscs.s12.41@pujc.edu.pk

[4]University of the Punjab, Jhelum Campus, Jhelum, Pakistan

jazebakram@gmail.com

[5]NETCIA, Nanjing Agriculture University, China

raheel_osman@yahoo.com

[6]Government Post Graduate College Okara, Punjab Pakistan

duasidra127@gmail.com

Received 01 May 2017; Accepted 23 May 2017

ABSTRACT

The object oriented paradigm plays a vital role not only in programming languages but also in the field of database, operating system and other field of computer science. This paper defines the overview of the concept of Design Patterns, Idioms and functional programming and defines how these concepts are implemented in C++.It describes how New features in C++ change the way of the programming, and integrate the new algorithms, libraries and the way to program in Object Oriented fashion in C++.We illustrate how much efficient and effective programming practices are been done through these concepts in the industry. We have briefly discuss the major and prime new features, libraries, algorithms added to C++11 and improved in C++14.

Keywords: *Design Pattern, Idioms, Modern Features, pointers and functional Programming*

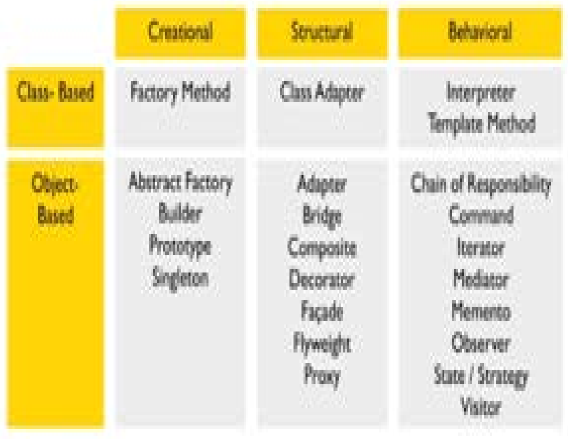
1. Introduction

Design patterns and idioms are important concepts helping programmers to produce good code and avoid pitfalls.So what is a design pattern? Design pattern is commonly defined as being tested solution to a recurring problem in a particular context.so let's take this apart see what it implies toqualify as a design pattern, a solution has to be tested and it's pros and cons well understood, design patterns document solutions that have been discovered and refine over time by experienced programmers. It always seems a little strange to see design patterns describe the technologies that are only a few months old so can’t have test that much. Secondly the problem solved by pattern should be widely applicable, it isn't really worth taking trouble to document the solution to a problem if it isn't likely to recur or if no one else is going to encounter it. [2,3] All design pattern have a context and may not be applicable or maybe even actively harmful if used outside that context, For example a pattern used to solve problems with wear application may not will be suitable for real-time applications. Patterns are designed to present best practices in a practical way building on the knowledge of experienced developers, there's a saying patterns are discovered not invented, if something really is a best practice you'd expect experienced practitioners to discover independently because it is the best way to do the job in fact the patterns are best practices you may have well discovered something by yourself and it is quite common for a good developer to get a slight feeling of anticlimax when hearing about the patterns.

Design patterns have several advantages for the developer, first they provide shared language that improves communication making it easier to communicate insight and experience about problems and how they can be solved for instance when someone says that use the visitor is a concise way of saying that they want to add functionality to a class without changing it, separating the operation from the class been operated on and of course using design patterns helps you build on the experience of others not onlydoes this help you to avoid problems and pitfalls that may not be obvious, but it also guards against reinventing the wheel thinking up your own solution to a problem where a perfectly good one already exist.[4,5]A good programmer is humble recognising that he or she doesn't necessarily know everything that's a home-grown solution is unlikely to match established best practice

2. Literature Review

Where did design patterns originate? An architect called Christopher Alexander discovered patterns in architectural designs and he published book called “A Pattern language towns buildings” construction in the 1970s, this defined 253 patterns that formed what he called a pattern language,[7] Kent Beck and Ward Cunningham began studying patterns and software in the early 1980sErich Gamma became intrusted while doing his PhD. Gamma Richard Helm began cataloguing patterns in 1991,they were joined by Ralph Johnson and John Vlissidesand they all became known as the gang of four their work was published as a book design patterns elements of reusable object oriented software, this now universally known as gand of four book and describe 23 basic patterns.[13]



	Creational	Structural	Behavioral
Class- Based	Factory Method	Class Adapter	Interpreter Template Method
Object-Based	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State / Strategy Visitor

Figure 1:

So what were the original patterns the 23 patterns described by gang of four book can be divided into three groups by function *Creational*,*Structural* and *Behavioural*. Creational patterns describe ways of creating objects, structuralpatterns deal with how objects are built into larger structures and Behavioural patterns relate to run time behaviour. They can further be divided in to twofamilies, consisting of those that are class based and those that are object based and here we can see the 23 divided into their groupas shown in fig.

Let's briefly reviews some common patterns, first one we discuss is *Model View Controller*better known as MVC, this was perhaps the original design pattern it was discovered by*TrygveReenskaug,* when he was a visiting science conference in the smalltalk*Rupert XeroxPalo Alto* research Lab and he published the first paper on his in 1978, [1] the idea behind MVC is to decouple a model which Reenskaug originally described as knowledge from the way in which it is viewed. Controllers provided interface between the user and the rest of the system MVC was discovered at the very birth of *Graphical User Interfaces* and is now the basis for countless applications and frameworks in both desktop and web world's there are many variations on MVC’s such as *Model View Presenter*and *Model View Viewmodel* all of which built with the basic idea separating a model from its representation so model holds and manipulate data which is displayed by one or more views controllers pass interaction information to the model and the user interacts with controllers and views.

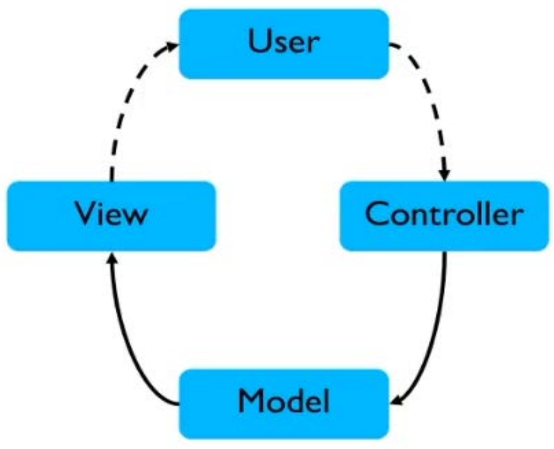


Figure 2:

*State Pattern*is used when an object needs to change its behaviour when its state changes, the classic example for this might be bank account whose behaviour changes depending on whether itsstate is in credit or over drawn. Objectstates are presented by classes and this removes the need for multiple if and switch statements to checkthe state. an object known as a context refers to two or more States objects which implement a common interface it's interesting to note that States is related to its strategy both of them has the same UML class diagram which could be sort of the general breach design the difference between them is being there intent.

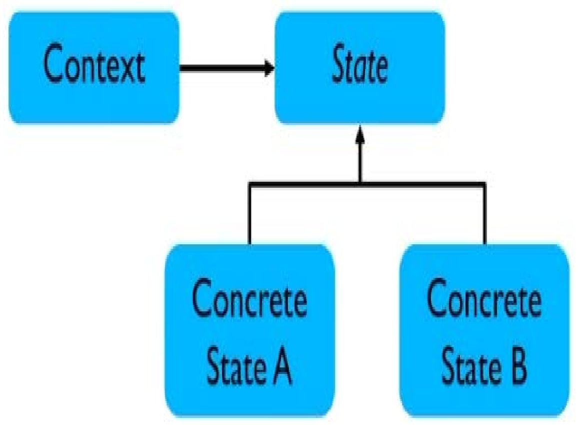


Figure 3:

Singleton is designed to allow creation of only one instance of a class and provide a global point of access to it but that rather depends on your definition of one you mean one instance per app, one per machine or one in entire worldand do you mean one physical object or one set of states so that you could have multiple instances that look identical, it's also rather harderto implement than you might think in C++ due to life time it is use and also initialization. Despite looking simple *Singleton* can be tricky to handle.

Observer which is designed to let a component provide notification of state changes to interested parties, observer is the good example of pattern has been incorporated to many libraries and frameworks, so it is often not necessary for the developer to implement themselves. so we have an object that may change state and observer can register a call back with the object and when a state change occurs the object send notifications to each call back in its list

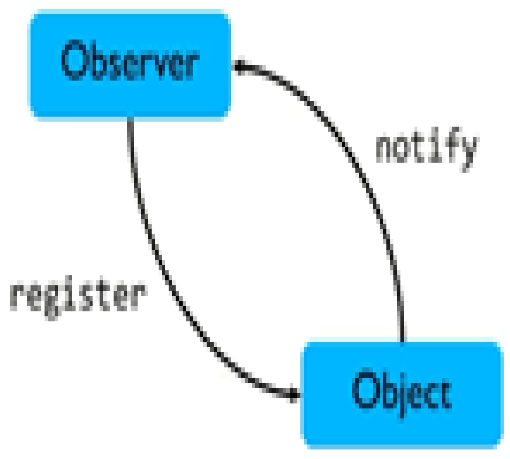


Figure 4:

3. Idioms

As well as design patterns we have idioms so what is an idiom idioms? Idioms differs from patterns in that they are specific to a language or platform as such idiots are rather lower level compact intend to apply at the code level. good examples of these would include pointers and templates in C++, perhaps they are especially templates is there is nothing quite like them in any other languages I've come across in languages that use garbage collection such as Java you will see finally use with try blocks there's a way to introduce deterministic clean up while avoiding codeduplication. C++ offcourse doesn't need this because destructors give us real control over object life time.

4. Implementation (Modern Patterns and Idioms)

Implementing patterns, it's important to understand that there is no one correct way to implement pattern, many people seem to get hung up on the UML diagrams that they seem books on the web and think that they show the one true way to implement the design pattern in truth the intent is far more important than the actual structure.[8,10,11]If that part of your Design is expressing the intent of the pattern then it is a valid implementation. Reification means make some real or concrete and in context of design patterns it means producing something that expresses the intent of the pattern whether or not it exactly matcher the UML

Off course patterns are ideas and have to be realised in code in order to be useful, let's look at how several common patterns and idioms can be implemented in C++. We're going to consider six.

1) RAII
2) States
3) smart pointer
4) PIMPL
5) CRTP
6) Singleton

of these six States and Singleton are Design patterns and the rest are idioms. we are going to discuss how these common patterns and idioms can be implemented in C++ 98 before the changes made to C++ 11 and 14 release, we will note that these are very simplisticimplementations, designed to illustrate the principles and not suited for real world use a used to illustrate how the patterns already can be implemented cater for possible corner cases for handle errors

4.1 RAII (Resource Acquisition Is Initialisation)

RAII (resource acquisition is initialization) essentially means doing something in the constructor of a class and undoing it in the destructor although it is widely used in the C++ world as you might expect it is also used in other languages that have a concept of a destructor including Rust & Ada, locking provides a good example so let's see how that works in practicewhen using the mutex, you need to remember to unlock it, at the end of the section of the code. It can be done by using only destructor but if the exception is thrown the mutex may not be unlocked, so we can use a simple class that uses RAII to unlock the mutex after its been used, which will even work correctly if exception gets thrown, particle example given below. All of this should work in any development environment that supports C++ 11 or 14.

```
Class Lock {
        lock () { // Do something here  }
        ~lock() { //Undo here  }
};
```

4.2 STATE

As name implies the state pattern is designed to represent changing objects states representing individual States by subclass is a good sign that this pattern may be applicable is when you find similar switch statements or changes of if in the classes methods, all checking for the same objects state a good example might be the traditional bank account class with deposit withdraw another method check whether the account is in credit or not, using this pattern removes the need for these Repeater checks and code is more modular because a logic for a particular state is not spread over several functions, so we do this by representing States as subclasses all of which implement a particular interface inheriting from an abstract base class.[9]as an example let's consider membership for example subscribing to website, theremay will be several classes of memberships such as basic free access and ordinary subscriber and a premium subscriber the type of membership for user has will determine their access to feature of the site you might think about using inheritance the model of various membership types with free member types inherited from base class but this will work until we need to handle upgrading users membership how do we convert basic member to premium member, an object in C++ can't changes its type and client code may have a pointer or reference this object, so we mustn't do anything that would in validate data. Thinkingdeeply about problem we realised there's this property of the member class to change over time then heritance isn't the right way to model it, membership type is actually the role members playing so we decide to have a data member that holds the membership.

4.3 PIMPL (Pointer to Implementation)

The PIMPL or pointer to implementation idioms provides a way to hide implementation details of a class so that the implementations details and dependencies don't pollute the class interface. This has number of uses one of which is improving compilation speed since the compiler doesn't have to recompile when implementation details changed as long as the class interface remains the same. Typically implementation is placed in a separate class with the main interface class attaining only a pointer to the implementing class

```
classPim {
PimImp *Pi;
//…
};
classPimImp {};
```

4.4 Smart Pointers

Smart pointers provide a way to manage resources but the concert what the resources is, what management means is very broad then out very widely used in cplusplus you rarely see wrote pointers in modern c plus plus code, aclasses that implement a smart pointer overload the arrow or crows foot operator giving indirect access to an internal point of that it manages. Simple model of the code how its implemented given below.

```
Class Ptr<T>{
T* p:
public:
//….
T* operator ->() const{
return p:
}
};
```

4.5 CRTP (Curiously Recurring Template Pattern)

The CRTP the *Curiously Recurring Template Pattern*provides a way to implement static polymorphism and remove the need for Virtual functions it does this by finding a templated base class that uses a derived class as its template parameter, it also generator a considerable small amount of assembly code as compared to the virtual functions dose during polymorphism, as shown below in the below.

```
class B<T>{ };

class D: public B<D> { };
```

4.6 Singleton

Singleton is used when only one instance of the class is required by adhering truly object oriented fashion. The Singleton Pattern comes under that classification of Creational Pattern, which deals with the best ways to create objects. These are used where only one instance of an object is needed throughout the lifetime of an application. The Singleton class is instantiated at the time of first access and same instance is used thereafter till the application quits.

```
class Singleton
{
private:
static Singleton* instance;
public:
static Singleton* getInstance();
};
```

5. Modern Features in C++

Let briefly discuss the version of C++, then see what they added to the language. Although C++ is been around for about some time, the first release was in 1998, this defined why we called this thing as traditional C++,[6] which included classes and templates. In 2003 a minor update, mostly considering the bug fixes, and this was followed by the major development program, which took 8 years and added a large number of significant new features to the language, we will be looking at these and how they affect the way we write code. Now the standard committee set schedule for release every 3 years, that's why we saw the release of C++ 14, this mainly provided the bug fixes and improvements. So 3 year release cycle which means we see the new release in 2017.

Let's run through the new features that's were introduced by the C++ 11 standard, we see that they have significantly updated the language.

a. **Constructors**: Two changes were made the way in whichobjects are constructed, delegating constructor like one constructor call another, while inheriting constructor let the derived class initialize itself using constructors form the base class. Both these can help reduce code duplication.

b. **Members**: Two keywords have been added to control how class member's functions are handled. The default keywords tell the compiler to generate the default implementations, such as default copy constructors. The Delete keyword tells the compiler not to generate the default version and now provide the more efficient and better documented way to inhibit the default creation and copy.

c. **Initialization**: There are major changes in the area of initialization, Uniform initialization means just about any data structure can be initialization using the same syntax. For example a vector can be initialized as the same way as array. This also apply to object construction so the same initialization can be used when calling object constructor.

d. **R values**: The area of Rvalues has perhaps a greatest effect on how the developer write in C++. Particularly who are writing the libraries? C++ now let you take a reference to a temporaryRvalue which hasn't been previously possible. Function overloading let you distinguished between normal and Rvalue references and this is very useful because if you know you are dealing with the temporary you can steal its state rather than have to copy it, this is called Move semantics. And has hugely increase the efficiency of code and standard library.

e. **Explicit Keyword**: The explicit keyword can now be applied to conversion operators as well as constructors and that helps avoid unexpected conversions.

f. **Inference**: Type inference is now supported through the alter keyword, we also have decltype which deduces the type of the variable or expression, which can be used to declare a variable that for example may has same type as x*y, this is particularly useful in template code, when the types are only known at the compile time.

g. **Lamdas**: Lamdas or anonymous functions are particularly useful addition to C++.not only the made code more concisely which is easy to understand but they are are also necessary if one is going to program in the functional style.

h. **Range For**: Range for is relatively simple addition providing a high level way to iterate over collection of all types, this idea is supported by many languages, it is also possible to write custom classes that will work for the range for, which make code more useable and maintainable because you no longer have to wonder just how you going to iterate over custom collection.

i. **Nullptr**: C++ now has a value to represent Null pointer. This is nullptr which is the proper pointer of type null pointer T. prior to this developers had to use either the null macro inherited from c, which is implementation dependence in assassinating including header files or zero, which make null pointer an int, that can cause problems

j. **Constexpr**: constexpr keyword provides support for generalized constant expressions, which means computation at runtime rather than compile time.

k. **Static_assert**: static assertion is the one that is checked at the compile time, rather than runtime, this can be used to get away with type traits which enables the compilers to check property of types. These are especially useful when using templates as they give the developer a control over the types used to extensiate the templates.

l. **Templates**: there are two main changes in this area, veriadic templates are perhaps the most advanced features introduced by C++ 11 and they provide a way to construct the templates having variable numbers of arguments. Templates aliases which not only allow to provide aliases for template types but more usefully to provide aliases with parameters types already bound.

6. New Libraries Features:

Number of new features have been added to standard libraries as well.
a. **Containers**: A number of new containers including hashtables, tuples, singly-linked lists and fixed arrays, that last one is interesting because it a refer for built-in array that allows the arrays to support all the standard containers operations.
b. **Algorithms**: A number of new algorithms have been added as well. Such as ALL OFF, ANY OFF, COPY IF,MOVE is sorted and various others.
c. **Pointers**: standard small pointers are been completely upgraded. Replacing alter pointer which were never very satisfactory. Std::unique_ptr provides a single owner of small pointer, while std:shared_ptr allow shared use, a std::week_ptr holds a reference but has no concept of owner ship, and that is useful in scenario such as cashes
d. **Functional**: Although C++ is not widely known as functional programming language, but C++ 11 added new features that make it easier to make it in the functional programming style. std::function is used to represent a callable target which can be a function pointer, function object or lamda and this make it much simpler to manipulate functions. std::bind provides a mechanism for partial function application. Taking a function binding one or more of its arguments than returning a new function.
e. **Concurrency**: prior to C++11 all concurrency functionality were platform dependent making it hard to write a portable code, but this no longer the case in C++11.

7. Functional Programming (FP)

In this section we will discuss some unique and powerful feature of functional programing, before this, it is important to know about what is FP? Why it is so important to develop to in C++? It is a type of programming that focusing on functional execution as its main computational mechanism. It is a concise and efficient way to write algorithmic code. And this is the new style of programming as it introduced many new patterns. [15] The best thing about functional programming is, it is supported by functional language. You can program in a functional style and one thing C++ make FP much easier. Why we need of functional programming?
Followings are some reasons why we need functional programming

- Object oriented is not enough.
- Object oriented is not good at composing algorithms.
- Object oriented data structure is often slow.
- Object oriented does not handle concurrency.

Principal of function programming:

7.1 Higher order Function

Function as a data, so functions are consider to be object and passed around and operated on, they can pass the function as arguments and used them as a result value. Function that operates on other function is called higher order functions. Function is used as building blocks and simple function composed of larger constructs. Now we have functions as binds in standard library, std::function &std:: bind, which is power full and expressive.

7.2 Immutable data

Once object is created, data is not changed, if it is required to be changed, new object is created. Mutation may create copy, string in java and C# also works on it. It has several advantages, first it helps with concurrency. Because variable is not going to change, while using it. And reasoning about code, once you create them you know while this code is for? If we create a copy every time variable change his state in that fashion, but that is not necessary is the case. Let's consider a simple example

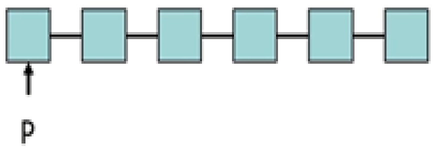


Figure: 5

We have link list and a pointer p, point's first element, if we add a new element at head then it is simple,

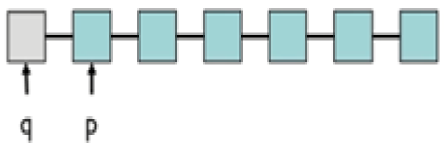


Figure: 6

New element link to the rest and no copying is required most importantly, point p is still valid pointing to the original values, point q , points to the new sequence.
What if we want to add new element to the tail of the list then, in this case we have to copy the list, if point p still going to valid.

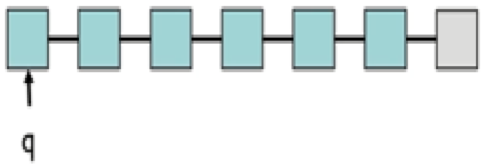


Figure: 7

Although working with data structure, immutable will efficient and we still understand what's going on.

7.3 No Side Effect

It means just what we access, execution of the function does not affect anything else itself. Function with two integer is classic example for this, giving you result not changing the arguments, what everything in the environment function with no side effect are called pure function, unfortunately exception and I/O are not pure function. So it is important to write code with pure core operation and other important function can be handled.

7.4 Always the same result

Calling the function with same arguments always give the same result, for example adding 2+2 perfectly replaced with 4. Consider another expression in more detail.
a=b+C and d=e+f , both does not contain anything similar , as a result they can be evaluated in many orders, further the expression is pure 'a' can be used anywhere else the entire expression is removed, don't affecting the rest of the code. Both are independent than they don't affect and execution independently.

7.5 Lazy Evaluation

This is another feature of functional programming, it means don't evaluate something until you need it, for example when you want to take square of all the terms in a list , first it adds one term by other and then take the square. Lazy evaluation is useful with chained transmission. Functional programming also support recursion, which is natural way to show some algorithms.
Beside in functional programming we have higher order operation, which we discuss below.

7.6 Filter (Selection)

In this method we select element from sequence using a predicate, for example

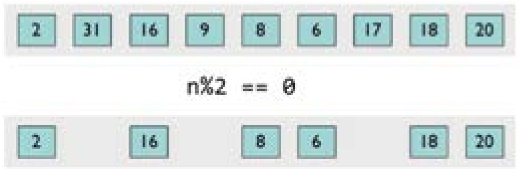


Figure: 8

We apply this operation and get even values from the sequence. This filter helps to reduce the amount of data by proceeding different object.

7.7 MAP (Transformation)

In this method we apply some technique on each element , and resulting new sequence is generated, for example in a diagram below we see that, when we apply operation n=>n+2 then it will give us new sequence , It transforms to a new values.

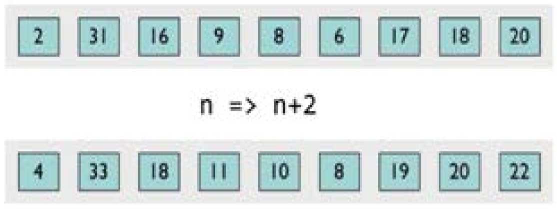
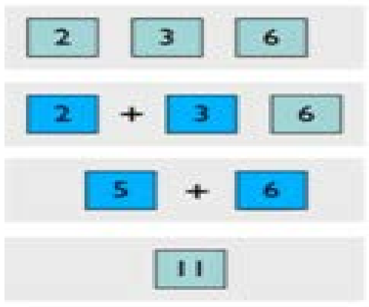


Figure: 9

7.8 Reduce (Concatenate)

In this method our sequence is reduced to single value, like in the diagram below we will see its working
We apply this operation (a,b)=>a+b, and we find this result.
FP provide us some standard library and operation to perform filtration, MAP, and reduce operation we see these function in the given below,
Std::remove copy_if will do for filter
Std::transform does a map
Std::accumalte for reduce Boost. Range and Range –V3 provides concept for PF.



8. Conclusion

In this paper we have represented the mostly used Design Patterns and Idioms used in the C++ programming in both object oriented and functional ways, This papers shows how much improvements and changes had been made to ease and promote the reusability, portability, accessibility and compatibility. We showed addition's in C++11 and improvements in C++14, and expecting the great new features in C++17 according to the standard rules set by the committee. At the last we also look for functional programming concepts and how they can be used in C++. We have gone through the details of different concepts of FP like filter, MAP and reduce as well. Experience, continuous practices have made the C++ to work in a completely new environment.

References

1. E. Casais, *"An Object-Oriented System Implementing KNOs*, "Proceedings of the Conference on Office Information Systems (COIS), pp. 284-290, Palo Alto, March 1988.
2. Esterie, P., Falcou, J., Gaunard, M., Lapresté, J. T., &Lacassagne, L. (2014). The numerical template toolbox: A modern C++ design for scientific computing. *Journal of Parallel and Distributed Computing*, *74*(12), 3240-3253.
3. Schmidt, D. C., Stal, M., Rohnert, H., & Buschmann, F. (2013). *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects* (Vol. 2). John Wiley & Sons.
4. García Sánchez, J. D., & Stroustrup, B. (2015). Improving performance and maintainability through refactoring in C++ 11.
5. Aragón, A. M. (2014). A C++ 11 implementation of arbitrary-rank tensors for high-performance computing. *Computer Physics Communications*, *185*(6), 1681-1696.
6. B. Stroustrup, *the C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.
7. *A Pattern Language:* Towns, Buildings, Construction (Center for Environmental Structure) (17 August 1977) by Christopher Alexander, Sara Ishikawa, Murray Silverstein
8. H. Albin-Amiot, P. Cointe, Y. G.Gueheneuc, and N. Jussien. Instantiating and detecting design patterns: putting bits and pieces together. In *16th AnnuInternational Conference on Automated Software Engineering (ASE 2001)*, pages 26–29, San Diego, CA, USA, 2001. Ecole des Mines Nantes France.
9. G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181–196, 2001.
10. R.B. France, D.-K. Kim, Sudipto Ghosh, and E. Song. Auml-based pattern specification technique. *Software Engineering, IEEET ransactions on*, 30(3): 193–206, 2004.
11. Cargill, Tom. Localized Ownership: Managing Dynamic Objects in C++. Pattern Languages of Program Design – 2. John M. Vlissides et al., eds. Reading, MA: Addison-Wesley, 1996
12. Martin, R. Design Patterns for Dealing with Dual Inheritance Hierarchies in C++.SIGS Publications: C++ Report, April, 1997.
13. Gamma, E. (1995). Design patterns: *Elements Of Reusable Object-Oriented Software*. Pearson Education India.
14. Orlov, S., & Melnikova, N. (2015). Compound Object Model for Scalable System Development in C++. *Procedia Computer Science*, *66*, 651-660.
15. Oliveira, C. (2016). Functional Programming Techniques. *In Options and Derivatives Programming in C++* (pp. 127-142). Apress.