Programming in C++
Prof. Partha Pratim Das
Department Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 30
Const-ness (Contd.)

Welcome back to Module 15 of Programming in C++. We have been discussing Const-ness, we have introduced constant objects; we have seen how this pointer of a constant object changes in its type because now it becomes a constant pointer pointing to a constant object.

We have seen with a constant object we cannot invoke a normal or non constant member function and we introduced the notion of constant member function which has a this pointer of the same type, that is a constant pointer to a constant object and we have seen with the use of constant member function, how we can protect the constant object and still invoke member function to read or axis the value of the data members and we have learnt that whenever in a class, a member function is not changing any object then it should be made has const.

Of course, if a member function wants to change a part of the object, any one of the data members also then it cannot be a const member function and complier will give you an error; obviously, those member function cannot be invoked from const objects.

We have also seen that part of the object can be selectively made constant by using const data member and we have seen the const data members cannot be changed by any of the member functions and they have to be initialized, they must be initialized at the time of construction and with the credit card example we have shown how the constant data members can be used to create class designs where the known editable data, whether they are normal data or they are pointed type data can be protected from the any changes to be done from the application.

Now we will proceed and talk about another feature, another selective const-ness feature known as Mutable Data Members.

(Refer Slide Time: 02:35)

Mutable - concept of being mutable is closely related to the concept of const-ness. As we know that a constant data member is not changeable, even when the object is non-constant, this is what we have learnt. On the other hand, a mutable data member is changeable in a constant object. So, a constant object as such is not changeable, nothing in that constant object can be changed, but if I define a data member of a class as mutable then even when an object of that class is defined to be constant even in that case that particular data member can be changed and that is what is the genesis of this name mutable as you know mutation means change, so mutable means changeable data member.

Now, there is a significant reason and specific purpose of why mutable data members exist. Mutable key word is provided or the mutable feature is provided to support logical const-ness against the bit wise const-ness of C++, what I mean is if I declare an object to be const; I say this object is constant then I am saying that nothing can be changed in that. If nothing can be changed in that; that means, whatever bit pattern that particular object has got at the time of initialization, no changes to that bit pattern ever is possible, but that often becomes little bit difficult to work with, I may have a concept which is constant, but its implementation may need additional fields, additional parameters, additional data members which are not exactly constant, so we will look into that.

Meanwhile you note, please note that mutable is applicable only in the case of data members, if you just have another a sample variable you cannot declare it to mutable and any reference status static data member of course you have not done static data member yet, we will talk about next module or constant data members can be declared as mutable. The most important part is if a data member is declared as mutable then it is legal to change it in a constant member function. We have seen that a constant member function cannot change any data member, but if a data member is mutable then it can still change it, because you are saying mutable means that it can be changed even when things are constant. So, let us look at to an example.

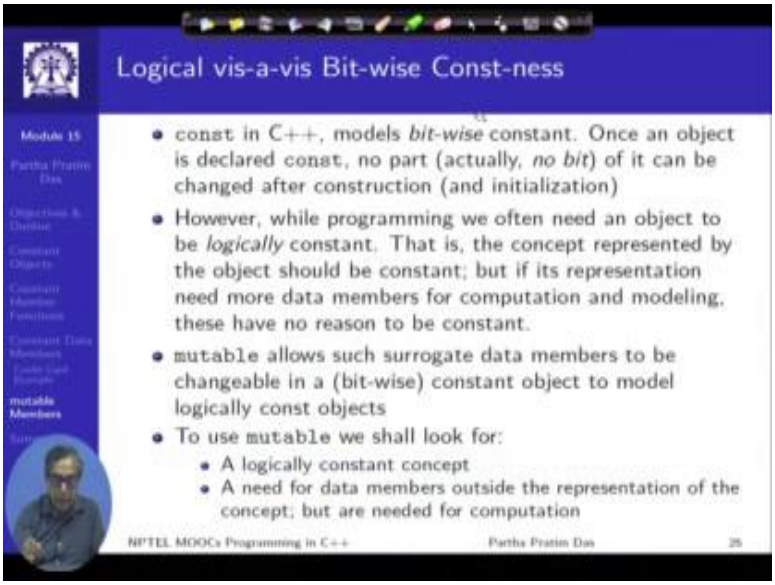
(Refer Slide Time: 05:37)

So, here I have introduced one data member which is mutable, so you write it syntax similar to writing const impress of const your just writing mutable and then we have declared a constant object, now we have a get method, we have a set method. Now we are; the get set on the non mutable one are different, the get is a const member function, set is a non const member function. So, which what means that if I have my const object, then I can do get on it, I cannot do set on it because constant objects cannot invoke non constant member.

Now look at the mutable part, here both these member functions had been declared to be const, so both of them can be invoked by the constant object. Now this function; set function is trying to change the member, if this particular member were not mutable then

this would have been a complication error, this would not have been possible, but since this member, data member is mutable it is allowed to change it. Therefore, even though this is a const member function, it can make changes to the data member, that is the basic consequence of having a mutable data member, so this is the basic behavior of the feature. Now let use try to see how this can be used, how this should be used.

(Refer Slide Time: 07:18)



So, as I was explaining const in C++ models bit wise constants that nothing in the bits of the constant object can change, but what is often important is I have a concept which is logically constant, that I am trying to model some constant, concept. So, that is described in terms of a few data members, but when I want to do something computation; in that same class I need some more data members which I just supporting the computation. Now if I make the object constant since all bits are constant nothing can change certainly all those I mean so called mutable surrogate data members which are just supporting the computation they also cannot change and therefore I cannot write the code.

Mutable has been provided to work around this problem, so to be able to use or to effectively use mutable, we will basically; please try to remember these two point, we will try to use; look for a logically constant concept, it is just not for a programming it there must be a concept which is logically constant and to implement that concept, we should need some data members which are outside of that concept, which are just written

required for the computation, I am sure this is not sounding very clear, so let me take an example.

(Refer Slide Time: 09:01)

We will read the top notes later on; first let us understand what we are doing. We are defining a class math object and my intention of doing this is, in this sample case my intention is math object should give me a constant object which is pi, we all know pi 3.14159 and so on. So, what I want is it should whenever I will construct object construct a constant object of this class, I will not construct non constant object.

So, I will construct this and on this if I invoke this method by, then it should give me the value of pi. Now why do we need get the value of pi, this is the algorithm say as you know pi is a transcendental number, so I cannot just put down, it is a long chain of approximation, long tail of approximation. So, I provide an algorithm which as it happens, there are better algorithms to compute pi, but this one happens to a pretty slow algorithm. So, if I invoke this mo dot pi; mo is the object, so if I do mo dot pi then this member function will be called which is a constant member function it can be called on constant object, this algorithm will execute and give me the value of pi.

Now, since this is potentially very slow, this can be potentially very slow, I want to do some cashing which means see I know the pi is a constant; concept is a constant. So, if I compute it once, the second time as I compute it will give me the same value, so why should I repeatedly compute, so I want to remember that whether I have computed it or I

have not computed it. So, the first time I compute it, next time onwards I should use that same computed value, so I use two data members in this, one is the value of pi; initially that value I do not know, in the constructor I cannot put this because I do not know what the value is, it has to be computed.

I will not do that computation unless somebody wants to use the value pi because this takes a lot of time as I said and when first time pi is to be used then this member function will get called. Now, I am maintaining another data member pi cashed which is a bull flag, which remembers whether pi has been computed or it has not been computed. So, that flag is initially set to false in the constructor of the object, so it says that it has not been computed.

So, now if I call pi, then this check will say the pi has been not computed, this whole computation will happen, the value will get updated in the data member pi and as I complete the iteration, I come out and put pi cashed to be true. So, in future when ever this data member pi is; a member function pi is called again, this will turn out to true therefore, whole of this will bypassed and I will simply be able to; this becomes, this will behave like a get function, this will just become get one pi, so this ensures that it will just be executed once and will be used number of times.

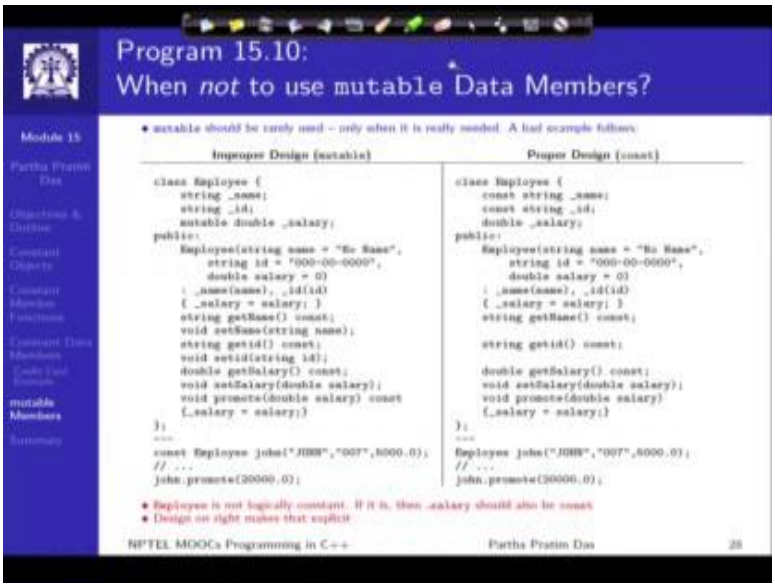
So let us see how you could have done this, the first thing that we require, we need to say that this object has to be constant because pi cannot be a non-constant. So the concept that is constant is pi, so that this is my basic requirement, it has to be a constant. Now if this is constant then these data members cannot be changed, now of these two data members; of these data members for example, let us say the cash data member, this cashed data member is not a part of the concept.

Cashing the value of pi is not a constant concept the constant-ness of the concept is in the value of pi. So, the pi cashed is been required because I want to do the computation, this pi; the value itself is required to be modified because I do not know what that value is when I am initializing. So, these data members are basically supporting the implementation of the concept so if you just take that; I will have a constant object then mo; mo will have false and whatever value becomes a bit wise constant nothing can be changed, so this whole strategy will not work. So, I make these two data members as mutable (Refer Time: 14:12) even though the object is constant, the concept is constant,

but in the implementation I need some data members which need not be constant and this sense of const-ness is called the logical const-ness.

So, if you look into the bit pattern; bit pattern is changing, but what you wanted represent is a basic concept of pi being constant that is preserved. So, mutable has a very specific purpose in this manner and that is a typical case to the best of my knowledge possibly the only case where mutable should be used. So, now, you can read the initial comments, this typically use when a class represent a constant concept and it computes a value first time and cash is a result for future use.

(Refer Slide Time: 15:06)



So, just to complete the discussion on mutable, I do present an example to show when not to use mutable, because you know it is a very symmetric concept. I can make data members constant in a non constant object and I can make data members non constant in a constant object or very symmetric concept. So, I have often seen that students tend to make the mistake of using them as equivalent or complementary concept, but that is not true.

So, to show I just show an example of an employee class, the employee has name, ID and salary. So, I say that okay, employees once created let us say this is an organization where no changes of employees are possible, so employees once created are constant objects, they cannot be changed. So, in the application we write employee is constant, but the salary of the employees change, employees need to be promoted, they need to be

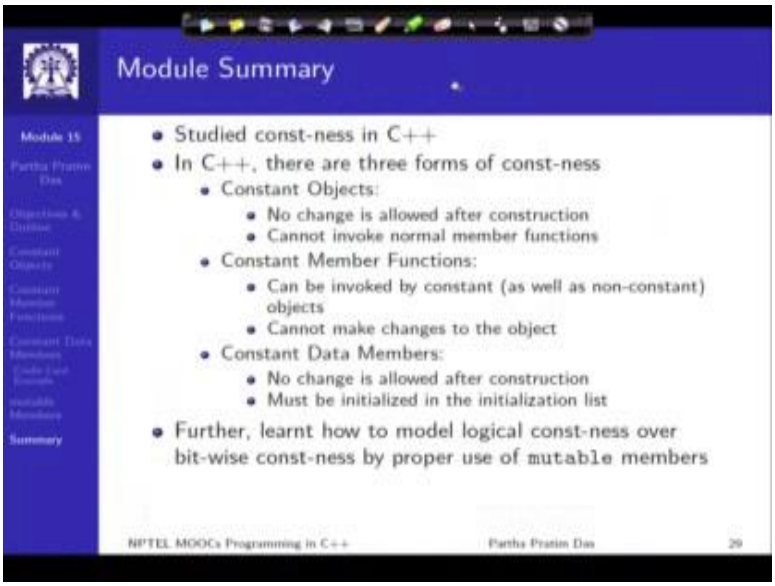
promoted then they will get a better salary and if the employee is constant, if John is constant then this is not possible, so I make it a constant member function.

If this is a constant member function then this itself is an error because constant member function cannot change data member; so to work around I put a mutable, this code will be compile, this code will run, this code will give result, there is nothing wrong, so far as the syntax of C++ is concerned but so far as the design, the concept is involved or so far as what others will understand from that code is involved, this is a disaster. This is a disaster because the mutable has been introduced to particularly model constant concepts, employees cannot be constant concepts, if they are constant concepts then their salary will also be constant, the salary should also not be changeable because that is a part and part salary of the employee.

So, rather this should be model in the way we had shown earlier that make the employee objects non constant because they do not need to be constant; they are not constant by concept but just make those data members constant which cannot change for an employee that is the name and the ID and make the changeable data member as a non-constant data member so that you can normally invoke promote on this, promote now does not need to be a constant member function and therefore can make changes to the salary as required.

So, if you carefully study the design of these two parts; the associated part and the associated use, code wise both of them will work, but concept wise this is an improper design, this should not be done and this is the right way of doing a normal class design, only when you have constant concepts that you want to represent whether be it natural constants or other kinds of constant concepts then you should use mutable for making your implementation type data members as editable.

(Refer Slide Time: 18:40)



So, with this we come to the end of the current module. Here we have studied const-ness in C++, in C++ we have seen three forms of const-ness, the object as a whole can be constant and if the object is constant, then it can only invoke constant member functions.

So, we have seen that a constant member functions cannot change the object, but non constant objects can also invoke constant member functions and if we want to selectively make a part of the object constant like the ID of an employee, roll number of a student; then we can make the corresponding data member constant, then constant member function or non constant member function, none of them can change the constant data member. We have also seen that C++ by default supports bit wise const-ness in the const use, but it is possible to use mutable data member to achieve logical const-ness that we can have a logically constant const-ness concept which we can code in C++ using the mutable data member.