Why should one not derive from c++ std string class?

Asked 13 years, 7 months ago Modified 6 years, 5 months ago

Viewed 25k times

73

I wanted to ask about a specific point made in Effective C++.

It says:

> A destructor should be made virtual if a class needs to act like a polymorphic class. It further adds that since `std::string` does not have a virtual destructor, one should never derive from it. Also `std::string` is not even designed to be a base class, forget polymorphic base class.

I do not understand what specifically is required in a class to be eligible for being a base class (not a polymorphic one)?

Is the only reason that I should not derive from `std::string` class is it does not have a virtual destructor? For reusability purpose a base class can be defined and multiple derived class can inherit from it. So what makes `std::string` not even eligible as a base class?

Also, if there is a base class purely defined for reusability purpose and there are many derived types, is there any way to prevent client from doing `Base* p = new Derived()` because the classes are not meant to be used polymorphically?

c++ string inheritance stl

Share

Improve this question

Follow

edited Aug 2, 2011 at 17:56

Viktor Dahl

1,982 ● 3 ● 25 ● 36

asked May 15, 2011 at 6:26

Sriram Subramanian

2,743 ● 7 ● 34 ● 37

8 Answers

Sorted by: Highest score (default)

66

I think this statement reflects the confusion here (emphasis mine):

> I do not understand what specifically is required in a class to be eligible for being a base clas (**not a polymorphic one**)?

In idiomatic C++, there are two uses for deriving from a class:

- **private** inheritance, used for mixins and aspect oriented programming using templates.
- **public** inheritance, used for *polymorphic situations only*. **EDIT**: Okay, I guess this could be used in a few mixin scenarios too -- such as `boost::iterator_facade` -- which show up when the CRTP is in use.

There is absolutely no reason to publicly derive a class in C++ if you're not trying to do something polymorphic. The language comes with free functions as a standard feature of the language, and free functions are what you should be using here.

Think of it this way -- do you really want to force clients of your code to convert to using some proprietary string class simply because you want to tack on a few methods? Because unlike in Java or C# (or most similar object oriented languages), when you derive a class in C++ most users of the base class need to know about that kind of a change. In Java/C#, classes are usually accessed through references, which are similar to C++'s pointers. Therefore, there's a level of indirection involved which decouples the clients of your class, allowing you to substitute a derived class without other clients knowing.

However, in C++, classes are **value types** -- unlike in most other OO languages. The easiest way to see this is what's known as the slicing problem. Basically, consider:

```
int StringToNumber(std::string copyMeByValue)
{
    std::istringstream converter(copyMeByValue);
    int result;
    if (converter >> result)
    {
        return result;
    }
    throw std::logic_error("That is not a number.");
}
```

If you pass your own string to this method, the copy constructor for `std::string` will be called to make a copy, **not the copy constructor for your derived object** -- no matter what child class of `std::string` is passed. This can lead to inconsistency between your methods and anything attached to the string. The function `StringToNumber` cannot simply take whatever your derived object is and copy that, simply because your derived object probably has a different size than a `std::string` -- but this function was compiled to reserve only the space for a `std::string` in automatic storage. In Java and C# this is not a problem because the only thing like automatic storage involved are reference types, and the references are always the same size. Not so in C++.

Long story short -- don't use inheritance to tack on methods in C++. That's not idiomatic and results in problems with the language. Use non-friend, non-member functions where possible, followed by composition. Don't use inheritance unless you're template metaprogramming or want polymorphic behavior. For more information, see

Scott Meyers' Effective C++ Item 23: Prefer non-member non-friend functions to member functions.

EDIT: Here's a more complete example showing the slicing problem. You can see it's output on codepad.org

```
#include <ostream>
#include <iomanip>

struct Base
{
    int aMemberForASize;
    Base() { std::cout << "Constructing a base." << st
    Base(const Base&) { std::cout << "Copying a base."
    ~Base() { std::cout << "Destroying a base." << std
};

struct Derived : public Base
{
    int aMemberThatMakesMeBiggerThanBase;
    Derived() { std::cout << "Constructing a derived."
    Derived(const Derived&) : Base() { std::cout << "C
std::endl; }
    ~Derived() { std::cout << "Destroying a derived."
};

int SomeThirdPartyMethod(Base /* SomeBase */)
{
    return 42;
}

int main()
{
    Derived derivedObject;
    {
        //Scope to show the copy behavior of copying a
        Derived aCopy(derivedObject);
    }
    SomeThirdPartyMethod(derivedObject);
}
```

Share Improve this answer

Follow

edited May 23, 2017 at 11:54

Community Bot

1 ● 1

answered May 15, 2011 at 7:12

Billy ONeal

106k ● 61 ● 326 ● 561

3 @davka: That should be done with composition instead. The bits that are common should be in a class or classes of their own. – Billy ONeal May 15, 2011 at 8:13

2 @davka: Two reasons: 1. because you will have the slicing problem on a regular basis. There's no reason to add this kind of pain unless you have to. 2. Because if you're trying to do this, you class likely is in violation of SRP, and should be split anyway. If you need to do something requiring internal access to the class, then add a member function to the original class. If you don't, then use composition or non-member functions. (For this reason I recommend doing this in C# and Java as well, even though slicing isn't an issue) – Billy ONeal May 15, 2011 at 8:37

1 @davka: Basically, a mixin is an "implemented-in-terms-of" relationship. Your `UsesLogging` class is not part of the interface of `A`, so therefore it shouldn't be `public` ly inherited from. Public inheritance is for taking the interface of a base class. If you were to modify `A`'s interface using `UsesLogging`, (e.g. by exposing functions of `UsesLogging` in `A`) then you would be in a polymorphic situation. Rule of thumb == `public` inheritance means "is-a", `private` inheritance means "implemented-in-terms-of". – Billy ONeal May 17, 2011 at 12:29

1 @Tony: Yes, that "very specific corner case" is called an "example". Those are generally used to demonstrate programming ideas to beginners. I state that the slicing

problem causes unexpected client behavior, and then show a case where it causes unexpected behavior. As far as the base being copied cleanly, I believe I said that; it says "using the base class' copy constructor". However, a typical programmer's look at this code would suggest that the derived class was actually copied instead. The size argument is there to show why the derived copy constructor cannot be called instead. – Billy ONeal Mar 19, 2012 at 13:33

1 There is at least one reason to inherit from std::string: creating classes for entities which are semantically different but are represented by string. Inherited class allows to never accidentally mix one with another - which is an immense help when refactoring. The class definition itself may even be empty, I.e. no problems with destructors – DarkWanderer Oct 6, 2014 at 15:37

31

To offer the counter side to the general advice (which is sound when there are no particular verbosity/productivity issues evident)...

Scenario for reasonable use

There is at least one scenario where public derivation from bases without virtual destructors can be a good decision:

- you want some of the type-safety and code-readability benefits provided by dedicated user-defined types (classes)
- an existing base is ideal for storing the data, and allows low-level operations that client code would

also want to use

- you want the convenience of reusing functions supporting that base class
- you understand that any any additional invariants your data logically needs can only be enforced in code explicitly accessing the data as the derived type, and depending on the extent to which that will "naturally" happen in your design, and how much you can trust client code to understand and cooperate with the logically-ideal invariants, you may want members functions of the derived class to reverify expectations (and throw or whatever)
- the derived class adds some highly type-specific convenience functions operating over the data, such as custom searches, data filtering / modifications, streaming, statistical analysis, (alternative) iterators
- coupling of client code to the base is more appropriate than coupling to the derived class (as the base is either stable or changes to it reflect improvements to functionality also core to the derived class)
    - put another way: you want the derived class to continue to expose the same API as the base class, even if that means the client code is forced to change, rather than insulating it in some way that allows the base and derived APIs to grow out of sync

- you're not going to be mixing pointers to base and derived objects in parts of the code responsible for deleting them

This may sound quite restrictive, but there are plenty of cases in real world programs matching this scenario.

Background discussion: relative merits

Programming is about compromises. Before you write a more conceptually "correct" program:

- consider whether it requires added complexity and code that obfuscates the real program logic, and is therefore more error prone overall despite handling one specific issue more robustly,
- weigh the practical costs against the probability and consequences of issues, and
- consider "return on investment" and what else you could be doing with your time.

If the potential problems involve usage of the objects that you just can't imagine anyone attempting given your insights into their *accessibility, scope and nature of usage in the program*, or you can generate compile-time errors for dangerous use (e.g. an assertion that derived class size matches the base's, which would prevent adding new data members), then anything else may be

premature over-engineering. Take the easy win in clean, intuitive, concise design and code.

Reasons to consider derivation sans virtual destructor

Say you have a class D publicly derived from B. With no effort, the operations on B are possible on D (with the exception of construction, but even if there are a lot of constructors you can often provide effective forwarding by having one template for each distinct number of constructor arguments: e.g. `template <typename T1, typename T2> D(const T1& x1, const T2& t2) : B(t1, t2) { }`. Better generalised solution in C++0x variadic templates.)

Further, if B changes then by default D exposes those changes - staying in sync - but someone may need to review extended functionality introduced in D to see if it remains valid, *and* the client usage.

Rephrasing this: there is *reduced explicit coupling between base and derived class, but increased coupling between base and client*.

This is often NOT what you want, but sometimes it is ideal, and other times a non issue (see next paragraph). Changes to the base force more client code changes in places distributed throughout the code base, and sometimes the people changing the base may not even have access to the client code to review or update it

correspondingly. Sometimes it is better though: if you as the derived class provider - the "man in the middle" - want base class changes to feed through to clients, and you generally want clients to be able - sometimes forced - to update their code when the base class changes without you needing to be constantly involved, then public derivation may be ideal. This is common when your class is not so much an independent entity in its own right, but a thin value-add to the base.

Other times the base class interface is so stable that the coupling may be deemed a non issue. This is especially true of classes like Standard containers.

Summarily, public derivation is a quick way to get or approximate the ideal, familiar base class interface for the derived class - in a way that's concise and self-evidently correct to both the maintainer and client coder - with additional functionality available as member functions (which IMHO - which obviously differs with Sutter, Alexandrescu etc - can aid usability, readability and assist productivity-enhancing tools including IDEs)

C++ Coding Standards - Sutter & Alexandrescu - cons examined

Item 35 of *C++ Coding Standards* lists issues with the scenario of deriving from `std::string`. As scenarios go, it's good that it illustrates the burden of exposing a large but useful API, but both good and bad as the base API is remarkably stable - being part of the Standard Library. A

stable base is a common situation, but no more common than a volatile one and a good analysis should relate to both cases. While considering the book's list of issues, I'll specifically contrast the issues' applicability to the cases of say:

a) `class Issue_Id : public std::string { ...handy stuff... };` <-- public derivation, our controversial usage
b) `class Issue_Id : public string_with_virtual_destructor { ...handy stuff... };` <- safer OO derivation
c) `class Issue_Id { public: ...handy stuff... private: std::string id_; };` <-- a compositional approach
d) using `std::string` everywhere, with freestanding support functions

(Hopefully we can agree the composition is acceptable practice, as it provides encapsulation, type safety as well as a potentially enriched API over and above that of `std::string`.)

So, say you're writing some new code and start thinking about the conceptual entities in an OO sense. Maybe in a bug tracking system (I'm thinking of JIRA), one of them is say an Issue_Id. Data content is textual - consisting of an alphabetic project id, a hyphen, and an incrementing issue number: e.g. "MYAPP-1234". Issue ids can be stored in a `std::string`, and there will be lots of fiddly little text searches and manipulation operations needed on issue ids - a large subset of those already provided on

`std::string` and a few more for good measure (e.g. getting the project id component, providing the next possible issue id (MYAPP-1235)).

On to Sutter and Alexandrescu's list of issues...

> Nonmember functions work well within existing code that already manipulates `string`s. If instead you supply a `super_string`, you force changes through your code base to change types and function signatures to `super_string`.

The fundamental mistake with this claim (and most of the ones below) is that it promotes the convenience of using only a few types, ignoring the benefits of type safety. It's expressing a preference for d) above, rather than insight into c) or b) as alternatives to a). The art of programming involves balancing the pros and cons of distinct types to achieve reasonable reuse, performance, convenience and safety. The paragraphs below elaborate on this.

Using public derivation, the existing code can implicitly access the base class `string` as a `string`, and continue to behave as it always has. There's no specific reason to think that the existing code would want to use any additional functionality from `super_string` (in our case Issue_Id)... in fact it's often lower-level support code pre-existing the application for which you're creating the `super_string`, and therefore oblivious to the needs provided for by the extended functions. For example, say there's a non-member function `to_upper(std::string&,`

`std::string::size_type from, std::string::size_type to)` - it could still be applied to an `Issue_Id`.

So, unless the non-member support function is being cleaned up or extended at the deliberate cost of tightly coupling it to the new code, then it needn't be touched. If it *is* being overhauled to support issue ids (for example, using the insight into the data content format to upper-case only leading alpha characters), then it's probably a good thing to ensure it really is being passed an `Issue_Id` by creating an overload ala `to_upper(Issue_Id&)` and sticking to either the derivation or compositional approaches allowing type safety. Whether `super_string` or composition is used makes no difference to effort or maintainability. A `to_upper_leading_alpha_only(std::string&)` reusable free-standing support function isn't likely to be of much use - I can't recall the last time I wanted such a function.

The impulse to use `std::string` everywhere isn't qualitatively different to accepting all your arguments as containers of variants or `void*`s so you don't have to change your interfaces to accept arbitrary data, but it makes for error prone implementation and less self-documenting and compiler-verifiable code.

> Interface functions that take a string now need to: a) stay away from `super_string`'s added functionality (unuseful); b) copy their argument to a super_string (wasteful); or c) cast the string

> reference to a super_string reference (awkward and potentially illegal).

This seems to be revisiting the first point - old code that needs to be refactored to use the new functionality, albeit this time client code rather than support code. If the function wants to start treating its argument as an *entity* for which the new operations are relevant, then it *should* start taking its arguments as that type and the clients should generate them and accept them using that type. The exact same issues exists for composition. Otherwise, `c)` can be practical and safe if the guidelines I list below are followed, though it is ugly.

> super_string's member functions don't have any more access to string's internals than nonmember functions because string probably doesn't have protected members (remember, it wasn't meant to be derived from in the first place)

True, but sometimes that's a good thing. A lot of base classes have no protected data. The public `string` interface is all that's needed to manipulate the contents, and useful functionality (e.g. `get_project_id()` postulated above) can be elegantly expressed in terms of those operations. Conceptually, many times I've derived from Standard containers, I've wanted not to extend or customise their functionality along the existing lines - they're already "perfect" containers - rather I've wanted to add another dimension of behaviour that's specific to my

application, and requires no private access. It's because they're already good containers that they're good to reuse.

> If `super_string` hides some of `string`'s functions (and redefining a nonvirtual function in a derived class is not overriding, it's just hiding), that could cause widespread confusion in code that manipulates `string`s that started their life converted automatically from `super_string`s.

True for composition too - and more likely to happen as the code doesn't default to passing things through and hence staying in sync, and also true in some situations with run-time polymorphic hierarchies as well. Samed named functions that behave differently in classes that initial appear interchangeable - just nasty. This is effectively the usual caution for correct OO programming, and again not a sufficient reason to abandon the benefits in type safety etc..

> What if `super_string` wants to inherit from `string` to add more *state* [explanation of slicing]

Agreed - not a good situation, and somewhere I personally tend to draw the line as it often moves the problems of deletion through a pointer to base from the realm of theory to the very practical - destructors aren't invoked for additional members. Still, slicing can often do what's wanted - given the approach of deriving

`super_string` not to change its inherited functionality, but to add another "dimension" of application-specific functionality....

> Admittedly, it's tedious to have to write passthrough functions for the member functions you want to keep, but such an implementation is vastly better and safer than using public or nonpublic inheritance.

Well, certainly agree about the tedium....

Guidelines for successful derivation sans virtual destructor

- ideally, avoid adding data members in derived class: variants of slicing can accidentally remove data members, corrupt them, fail to initialise them...
- even more so - avoid non-POD data members: deletion via base-class pointer is technically undefined behaviour anyway, but with non-POD types failing to run their destructors is more likely to have non-theoretical problems with resource leaks, bad reference counts etc.
- honour the Liskov Substitution Principal / you can't robustly maintain new invariants
    - for example, in deriving from `std::string` you can't intercept a few functions and expect your

objects to remain uppercase: any code that accesses them via a `std::string&` or `...*` can use `std::string`'s original function implementations to change the value)

    - derive to model a higher level entity in your application, to extend the inherited functionality with some functionality that uses but doesn't conflict with the base; do not expect or try to change the basic operations - and access to those operations - granted by the base type

- be aware of the coupling: base class can't be removed without affecting client code even if the base class evolves to have inappropriate functionality, i.e. your derived class's usability depends on the ongoing appropriateness of the base
    - sometimes even if you use composition you'll need to expose the data member due to performance, thread safety issues or lack of value semantics - so the loss of encapsulation from public derivation isn't tangibly worse
- the more likely people using the potentially-derived class will be unaware of its implementation compromises, the less you can afford to make them dangerous
    - therefore, low-level widely deployed libraries with many ad-hoc casual users should be more wary of dangerous derivation than localised use by programmers routinely using the functionality

at application level and/or in "private" implementation / libraries

Summary

Such derivation is not without issues so don't consider it unless the end result justifies the means. That said, I flatly reject any claim that this can't be used safely and appropriately in particular cases - it's just a matter of where to draw the line.

Personal experience

I do sometimes derive from `std::map<>`, `std::vector<>`, `std::string` etc - I've never been burnt by the slicing or delete-via-base-class-pointer issues, and I've saved a lot of time and energy for more important things. I don't store such objects in heterogeneous polymorphic containers. But, you need to consider whether all the programmers using the object are aware of the issues and likely to program accordingly. I personally like to write my code to use heap and run-time polymorphism only when needed, while some people (due to Java backgrounds, their prefered approach to managing recompilation dependencies or switching between runtime behaviours, testing facilities etc.) use them habitually and therefore need to be more concerned about safe operations via base class pointers.

 edited May 25, 2012 at 2:58

Follow

community wiki
12 revs
Tony D

11

If you *really* want to derive from it (not discussing why you want to do it) I think you can prevent `Derived` class direct heap instantiation by making it's `operator new` private:

```
class StringDerived : public std::string {
//...
private:
  static void* operator new(size_t size);
  static void operator delete(void *ptr);
};
```

But this way you restrict yourself from any dynamic `StringDerived` objects.

Share Improve this answer

Follow

answered May 15, 2011 at 6:43

beduin
8,243 ● 3 ● 28 ● 24

You can, but this really doesn't answer the OP's question. – Billy ONeal May 15, 2011 at 6:59

6 @Billy: it does answer the OP **second** question, see "Also..." – davka May 15, 2011 at 8:07

2 +1 Now I know how to prevent usage of new/delete operators on a specific class. – Viet Jul 19, 2012 at 1:24

Alternatively, a public `static void* operator new(std::size_t) = delete;` gives a better compiler diagnostic message. – Eljay Oct 2, 2023 at 11:53

10

Not only is the destructor not virtual, std::string contains no virtual functions **at all**, and no protected members. That makes it very hard for the derived class to modify its functionality.

Then why would you derive from it?

Another problem with being non-polymorphic is that if you pass your derived class to a function expecting a string parameter, your extra functionality will just be sliced off and the object will be seen as a plain string again.

Share Improve this answer Follow

edited May 15, 2011 at 6:54

answered May 15, 2011 at 6:40

Bo Persson
92.1k ● 31 ● 151 ● 208

7 a base class need not have virtual functions to derive from it. For code reusability it is common to derive from another class. The only restriction is that you should not use the class as a polymorphic one. – Sriram Subramanian May 15, 2011 at 6:44

@Siriam - True, but it is hard to see any reason for code reusability from a string. It already has too many member functions, so extending it further doesn't seem like the best

idea. You should also consider that when textbooks say "never do this" they actually mean "you should hardly ever do this". – Bo Persson May 15, 2011 at 6:49

9 @Sriram: There's no reason to derive from a class in C++ unless you need to modify internal state of the base class. Any extension you'd do to `std::basic_string<CharT, Allocator<CharT>>` would never be touching internal members. Therefore there's no need for any deriving to occur. C++ has free functions just for adding this type of functionality. Use a free function for your extensions (e.g. as `boost::algorithm::string` does) -- that's how this is done in ideomatic C++. – Billy ONeal May 15, 2011 at 6:59

you could want to derive from `std::string` just to make some strings to be considered as a separate type. Like in OpenGL: the vertex shader source code VS the fragment shader source code. I think there is another way to do it in that case. But cannot find it. – xealits Jul 14 at 23:35

4

> Why should one not derive from c++ std string class?

Because it is **not necessary**. If you want to use `DerivedString` for functionality extension; I don't see any problem in deriving `std::string`. The only thing is, you should not interact between both classes (i.e. don't use `string` as a receiver for `DerivedString`).

> Is there any way to prevent client from doing `Base* p = new Derived()`

Yes. Make sure that you provide `inline` wrappers around `Base` methods inside `Derived` class. e.g.

```
class Derived : protected Base { // 'protected' to avo
  const char* c_str () const { return Base::c_str(); }
//...
};
```

Share Improve this answer

edited May 15, 2011 at 8:36

Follow

answered May 15, 2011 at 7:30

iammilind

69.8k • 38 • 182 • 354

This works until you have to turn around and pass your derived string to a method or something of that nature. What's wrong with non-member functions? – Billy ONeal May 15, 2011 at 7:35

1 @Billy: non-member functions are workable, but it's painful for programmers to deal with two interface styles for operations on the type. IDEs tend to provide better productivity-enhancing completion features for member functions. C++03 doesn't offer any good solutions to this pain as composition is also painful. – Tony Delroy May 16, 2011 at 3:56

@Tony: I strongly disagree on both fronts. C++ is not designed to be a syntacticly pretty language. If one cannot deal with nonmember functions then one shouldn't be using C++. – Billy ONeal May 16, 2011 at 4:51

@Tony: For an example, I don't think most C++ programmers have problems using the STL -- but the STL is full of free functions which operate on types. Want to sort a `vector`?

You're going to be calling the free function `std::sort`.
– Billy ONeal May 16, 2011 at 4:55

1 @Billy: consistency and tooling productivity dismissed with "not designed to be pretty"? Is there a connection? ;-P. It's not what one can deal with, but making it simple, elegant and intuitive to do what's often wanted and useful. C++ beats every alternative I know but can improve. C++0x will - but a lot of the proposals incorporated therein were initially resisted with the same kind of passion to protect the status quo that you display above. And your example falls afoul of "Learnt how to sort a `vector`, now trying `std::sort(my_list. ... )`" ;-) – Tony Delroy May 16, 2011 at 6:08

2

There are two simple reasons for not deriving from a non-polymorphic class:

- **Technical**: it introduces slicing bugs (because in C++ we pass by value unless otherwise specified)
- **Functional**: if it is non-polymorphic, you can achieve the same effect with composition and some function forwarding

If you wish to add new functionalities to `std::string`, then first consider using free functions (possibly templates), like the Boost String Algorithm library does.

If you wish to add new data members, then properly wrap the class access by embedding it (Composition) inside a class of your own design.

EDIT:

@Tony noticed rightly that the *Functional* reason I cited was probably meaningless to most people. There is a simple rule of thumb, in good design, that says that when you can pick a solution among several, you should consider the one with the weaker coupling. Composition has weaker coupling that Inheritance, and thus should be preferred, when possible.

Also, composition gives you the opportunity to nicely wrap the original's class method. This is not possible if you pick inheritance (public) and the methods are not virtual (which is the case here).

edited May 16, 2011 at 6:21

answered May 15, 2011 at 7:48

Matthieu M.
299k ● 54 ● 479 ● 747

1 "Functional" as presented is not a reason for not doing it, but a comment that it's not necessary. That said, while C++'s forwarding "capabilities" make composition possible, it's a right pain initially and a maintainence burden thereafter which is why so many people want to consider, and do use (with varying degrees of safety) a derivation approach....
– Tony Delroy May 16, 2011 at 3:50

@Tony: Right, I keep forgetting that not everyone will remember than when you can choose between derivation and composition, you should pick composition because it's a weaker relation-ship. I'll expand the answer. – Matthieu M. May 16, 2011 at 6:18

1

The C++ standard states that If Base class destructor is not virtual and you delete an object of Base class that points to the object of an derived class then it causes an undefined Behavior.

C++ standard section 5.3.5/3:

if the static type of the operand is different from its dynamic type, the static type shall be a base class of the operand’s dynamic type and the static type shall have a virtual destructor or the behavior is undefined.

To be clear on the Non-polymorphic class & need of virtual destructor

The purpose of making a destructor virtual is to facilitate the polymorphic deletion of objects through delete-expression. If there is no polymorphic deletion of objects, then you don't need virtual destructor's.

Why not to derive from String Class?

One should generally avoid deriving from any standard container class because of the very reason that they don' have virtual destructors, which make it impossible to delete objects polymorphically.

As for the string class, the string class doesn't have any virtual functions so there is nothing that you can possibly override. The best you can do is hide something.

If at all you want to have a string like functionality you should write a class of your own rather than inherit from std::string.

Share Improve this answer Follow

edited May 15, 2011 at 6:51

answered May 15, 2011 at 6:37

Alok Save

206k ● 54 ● 441 ● 536

2 i am talking about non polymorphic – Sriram Subramanian May 15, 2011 at 6:38

0

As soon as you add any member (variable) into your derived std::string class, will you systematically screw the stack if you attempt to use the std goodies with an instance of your derived std::string class? Because the stdc++ functions/members have their stack pointers[indexes] fixed [and adjusted] to the size/boundary of the (base std::string) instance size.

Right?

Please, correct me if I am wrong.

Share Improve this answer

Follow

answered Jul 15, 2018 at 13:53

Bretzelus

345 ● 2 ● 8