

We changed our privacy policy. [Read more.](#)



# What is object slicing?

Asked 12 years, 11 months ago   Active 5 months ago   Viewed 213k times



Someone mentioned it in the IRC as the slicing problem.

825

c++

inheritance

c++-faq

object-slicing



372



Share Edit Follow

edited Nov 13 '18 at 10:37



Cœur

33.3k ● 22 ● 176 ● 236

asked Nov 8 '08 at 11:10



Frankomania

8,307 ● 3 ● 15 ● 3

18 Answers

Active

Oldest

Votes



"Slicing" is where you assign an object of a derived class to an instance of a base class, thereby losing part of the information - some of it is "sliced" away.

675



For example,



```
class A {
    int foo;
};

class B : public A {
    int bar;
};
```

So an object of type `B` has two data members, `foo` and `bar`.

Then if you were to write this:

```
B b;

A a = b;
```

Then the information in `b` about member `bar` is lost in `a`.

Share Edit Follow

edited May 11 '16 at 7:47



ForceBru


38.5k ● 10 ● 55 ● 83

answered Nov 8 '08 at 11:22



David Dibben

17.1k ● 6 ● 40 ● 41

- 69 Very informative, but see [stackoverflow.com/questions/274626#274636](https://stackoverflow.com/questions/274626#274636) for an example of how slicing occurs during method calls (which underscores the danger a little better than the plain assignment example). – [Blair Conrad](#) Nov 8 '08 at 13:53
- 
- 59 Interesting. I've been programming in C++ for 15 years and this issue never occurred to me, as I've always passed objects by reference as a matter of efficiency and personal style. Goes to show how good habits can help you. – [Karl Bielefeldt](#) Feb 2 '11 at 3:48
- 
- 12 @Felix Thanks but I don't think casting back (since not a pointer arithmetic) will work , `A a = b;` `a` is now object of type `A` which has copy of `B::foo` . It will be mistake to cast it back now i think. – [user72424](#) Aug 12 '11 at 12:27
- 
- 42 This isn't "slicing", or at least a benign variant of it. The real problem occurs if you do `B b1; B b2; A& b2_ref = b2; b2 = b1` . You might think you have copied `b1` to `b2` , but you haven't! You have copied a *part* of `b1` to `b2` (the part of `b1` that `B` inherited from `A` ), and left the other parts of `b2` unchanged. `b2` is now a frankensteinian creature consisting of a few bits of `b1` followed by some chunks of `b2` . Ugh! Downvoting because I think the answer is very misleading. – [fgp](#) Jan 22 '13 at 14:07 
- 
- 27 @fgp Your comment should read `B b1; B b2; A& b2_ref = b2; b2_ref = b1` "*The real problem occurs if you*" ... derive from a class with a non-virtual assignment operator. Is `A` even intended for derivation? It has no virtual functions. If you derive from a type, you have to deal with the fact that its member functions can be called! – [curiousguy](#) Jun 29 '13 at 14:54
- 



563



Most answers here fail to explain what the actual problem with slicing is. They only explain the benign cases of slicing, not the treacherous ones. Assume, like the other answers, that you're dealing with two classes `A` and `B` , where `B` derives (publicly) from `A` .

In this situation, C++ lets you pass an instance of `B` to `A`'s assignment operator (and also to the copy constructor). This works because an instance of `B` can be converted to a `const A&` , which is what assignment operators and copy-constructors expect their arguments to be.

## The benign case

```
B b;
A a = b;
```

Nothing bad happens there - you asked for an instance of `A` which is a copy of `B` , and that's exactly what you get. Sure, `a` won't contain some of `b`'s members, but how should it? It's an `A` , after all, not a `B` , so it hasn't even *heard* about these members, let alone would be able to store them.

## The treacherous case

```

B b1;
B b2;
A& a_ref = b2;
a_ref = b1;
//b2 now contains a mixture of b1 and b2!

```

You might think that `b2` will be a copy of `b1` afterward. But, alas, it's **not**! If you inspect it, you'll discover that `b2` is a Frankensteinian creature, made from some chunks of `b1` (the chunks that `B` inherits from `A`), and some chunks of `b2` (the chunks that only `B` contains). Ouch!

What happened? Well, C++ by default doesn't treat assignment operators as `virtual`. Thus, the line `a_ref = b1` will call the assignment operator of `A`, not that of `B`. This is because, for non-virtual functions, the **declared** (formally: *static*) type (which is `A&`) determines which function is called, as opposed to the **actual** (formally: *dynamic*) type (which would be `B`, since `a_ref` references an instance of `B`). Now, `A`'s assignment operator obviously knows only about the members declared in `A`, so it will copy only those, leaving the members added in `B` unchanged.

## A solution

Assigning only to parts of an object usually makes little sense, yet C++, unfortunately, provides no built-in way to forbid this. You can, however, roll your own. The first step is making the assignment operator *virtual*. This will guarantee that it's always the **actual** type's assignment operator which is called, not the **declared** type's. The second step is to use `dynamic_cast` to verify that the assigned object has a compatible type. The third step is to do the actual assignment in a (protected!) member `assign()`, since `B`'s `assign()` will probably want to use `A`'s `assign()` to copy `A`'s members.

```

class A {
public:
    virtual A& operator= (const A& a) {
        assign(a);
        return *this;
    }

protected:
    void assign(const A& a) {
        // copy members of A from a to this
    }
};

class B : public A {
public:
    virtual B& operator= (const A& a) {
        if (const B* b = dynamic_cast<const B*>(&a))
            assign(*b);
        else
            throw bad_assignment();
    }
};

```

```

    return *this;
}

protected:
    void assign(const B& b) {
        A::assign(b); // Let A's assign() copy members of A from b to this
        // copy members of B from b to this
    }
};

```

Note that, for pure convenience, `B's operator=` covariantly overrides the return type, since it **knows** that it's returning an instance of `B`.

Share Edit Follow

edited Feb 26 '20 at 14:33



YSC

35k ● 8 ● 85 ● 134

answered Jan 22 '13 at 15:00



fgp

7,408 ● 1 ● 15 ● 17

- 13 IMHO, the problem is that there are two different kinds of substitutability that may be implied by inheritance: either any `derived` value may be given to code expecting a `base` value, or any derived reference may be used as a base reference. I would like to see a language with a type system which addresses both concepts separately. There are many cases where a derived reference should be substitutable for a base reference, but derived instances should not be substitutable for base ones; there are also many cases where instances should be convertible but references should not substitute. – [supercat](#) Aug 12 '13 at 16:11 ✎
- 20 I don't understand what is so bad in your "treacherous" case. You stated that you want to: 1) get a reference to an object of class A and 2) cast the object b1 to class A and copy its stuff to a reference of the class A. What is actually wrong here is the proper logic behind the given code. In other words, you took a small image frame (A), placed it over a bigger image (B) and you painted through that frame, complaining later that your bigger image now looks ugly :) But if we just consider that framed area, it looks pretty good, just as the painter wanted, right? :) – [Mladen B.](#) Nov 14 '13 at 13:05
- 13 The problem is, differently put, that C++ by default assumes a very strong kind of *substitutability* - it requires the base class's operations to workly correctly on subclass instances. And that even for operations which the compiler autogenerated like assignment. So it's not enough to not screw up your own operations in this regard, you also have to explicitly disable the wrong ones generated by the compiler. Or of course, stay away from public inheritance, which usually is a good suggestion anyway ;) – [fgp](#) Nov 16 '13 at 16:31
- 15 Another common approach is to simply disable the copy and assignment operator. For classes within inheritance hierarchy, usually there is no reason to use value instead of reference or pointer. – [Siyuan Ren](#) Aug 22 '14 at 10:48
- 17 What the? I had no idea operators could be marked virtual – [paulm](#) Mar 2 '15 at 15:23

If You have a base class `A` and a derived class `B`, then You can do the following.

```

void wantAnA(A myA)
{

```

161

```
// work with myA
}

B derived;
// work with the object "derived"
wantAnA(derived);
```

Now the method `wantAnA` needs a copy of `derived`. However, the object `derived` cannot be copied completely, as the class `B` could invent additional member variables which are not in its base class `A`.

Therefore, to call `wantAnA`, the compiler will "slice off" all additional members of the derived class. The result might be an object you did not want to create, because

- it may be incomplete,
- it behaves like an `A`-object (all special behaviour of the class `B` is lost).

Share Edit Follow

edited Mar 4 '18 at 15:35



Will Vousden

30.4k ● 9 ● 78 ● 92

answered Nov 8 '08 at 11:28



Black

4,770 ● 1 ● 20 ● 32

47 C++ is **not** Java! If `wantAnA` (as its name implies!) wants an `A`, then that's what it gets. And an instance of `A`, will, uh, behave like an `A`. How is that surprising? – [fgp](#) Jan 22 '13 at 16:39

89 @fgp: It's surprising, because you **don't pass an A** to the function. – [Black](#) Mar 3 '13 at 7:03

11 @fgp: The behaviour is similar. However, to the average C++ programmer it might be less obvious. As far as I understood the question, nobody is "complaining". It's just about how the compiler handles the situation. Imho, it is better to avoid slicing at all by passing (const) references. – [Black](#) Apr 7 '13 at 12:13

9 @ThomasW No, I would not throw out inheritance, but use references. If the signature of `wantAnA` would be **`void wantAnA(const A & myA)`**, then there had been not slicing. Instead, a read-only reference to the caller's object is passed. – [Black](#) May 28 '13 at 7:44

15 the problem is mostly on the automatic casting that the compiler performs from `derived` to the type `A`. Implicit casting is always a source of unexpected behavior in C++, because it is often hard to understand from looking at the code locally that a cast took place. – [pqnet](#) Aug 6 '14 at 23:15

These are all good answers. I would just like to add an execution example when passing objects by value vs by reference:

```
#include <iostream>

using namespace std;
```

```
// Base class
class A {
public:
    A() {}
    A(const A& a) {
        cout << "'A' copy constructor" << endl;
    }
    virtual void run() const { cout << "I am an 'A'" << endl; }
};

// Derived class
class B: public A {
public:
    B():A() {}
    B(const B& a):A(a) {
        cout << "'B' copy constructor" << endl;
    }
    virtual void run() const { cout << "I am a 'B'" << endl; }
};

void g(const A & a) {
    a.run();
}

void h(const A a) {
    a.run();
}

int main() {
    cout << "Call by reference" << endl;
    g(B());
    cout << endl << "Call by copy" << endl;
    h(B());
}
```

The output is:

```
Call by reference
I am a 'B'

Call by copy
'A' copy constructor
I am an 'A'
```

Share Edit Follow

answered Aug 22 '14 at 18:33



geh

696 ● 5 ● 9

Hello. Great answer but I have one question. If I do something like this `** dev d; base* b = &d;**` The slicing also takes place? – Adrian Jul 10 '18 at 6:58

@Adrian If you introduce some new member functions or member variables in the derived class then those are not accessible from the base class pointer directly. However you can still access them from inside the overloaded base class virtual functions. See this:

[godbolt.org/z/LABx33](https://godbolt.org/z/LABx33) – Vishal Sharma Dec 25 '19 at 14:20



Third match in google for "C++ slicing" gives me this Wikipedia article [http://en.wikipedia.org/wiki/Object\\_slicing](http://en.wikipedia.org/wiki/Object_slicing) and this (heated, but the first few posts define the problem) : <http://bytes.com/forum/thread163565.html>



So it's when you assign an object of a subclass to the super class. The superclass knows nothing of the additional information in the subclass, and hasn't got room to store it, so the additional information gets "sliced off".

If those links don't give enough info for a "good answer" please edit your question to let us know what more you're looking for.

Share Edit Follow

edited Nov 8 '08 at 11:20

answered Nov 8 '08 at 11:14



The Archetypal Paul

39.8k ● 18 ● 99 ● 128



The slicing problem is serious because it can result in memory corruption, and it is very difficult to guarantee a program does not suffer from it. To design it out of the language, classes that support inheritance should be accessible by reference only (not by value). The D programming language has this property.



Consider class A, and class B derived from A. Memory corruption can happen if the A part has a pointer p, and a B instance that points p to B's additional data. Then, when the additional data gets sliced off, p is pointing to garbage.

Share Edit Follow

edited Nov 12 '08 at 4:20

answered Nov 8 '08 at 11:56



Walter Bright

4,151 ● 1 ● 21 ● 28

3 Please explain how the memory corruption can occur. – [foraidt](#) Nov 8 '08 at 12:48

4 I forgot that the copy ctor will reset the vptr, my mistake. But you can still get corruption if A has a pointer, and B sets that to point into B's section that gets sliced off. – [Walter Bright](#) Nov 11 '08 at 2:21

18 This problem isn't just limited to slicing. Any classes that contain pointers are going to have dubious behaviour with a default assignment operator and copy-constructor. – [Weeble](#) Feb 11 '09 at 11:54

2 @Weeble - Which is why you override the default destructor, assignment operator and copy-constructor in these cases. – [Bjarke Freund-Hansen](#) Jul 24 '09 at 19:55

7 @Weeble: What makes object slicing worse than general pointer fixups is that to be certain you have prevented slicing from happening, a base class must provide converting constructors *for every derived class*. (Why? Any derived classes that are missed are susceptible to being picked up by the base class's copy ctor, since `Derived` is implicitly

convertible to `Base`.) This is obviously counter to the Open-Closed Principle, and a big maintenance burden. – [j\\_random\\_hacker](#) Oct 24 '12 at 12:30

In C++, a derived class object can be assigned to a base class object, but the other way is not possible.

12

```
class Base { int x, y; };  
  
class Derived : public Base { int z, w; };  
  
int main()  
{  
    Derived d;  
    Base b = d; // Object Slicing, z and w of d are sliced off  
}
```

Object slicing happens when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off to form the base class object.

[Share](#) [Edit](#) [Follow](#)

answered Mar 7 '18 at 9:35

[Kartik Maheshwari](#)

329 ● 2 ● 9

The slicing problem in C++ arises from the value semantics of its objects, which remained mostly due to compatibility with C structs. You need to use explicit reference or pointer syntax to achieve "normal" object behavior found in most other languages that do objects, i.e., objects are always passed around by reference.

The short answer is that you slice the object by assigning a derived object to a base object *by value*, i.e. the remaining object is only a part of the derived object. In order to preserve value semantics, slicing is a reasonable behavior and has its relatively rare uses, which doesn't exist in most other languages. Some people consider it a feature of C++, while many considered it one of the quirks/misfeatures of C++.

[Share](#) [Edit](#) [Follow](#)

answered Nov 9 '08 at 0:31

[idadak](#)

5,584 ● 1 ● 18 ● 21

6 ""normal" object behavior" that's not "normal object behaviour", that's **reference semantic**. And it relates **in no way** with C `struct`, compatibility, or other non-sense the any random OOP priest told you. – [curiousguy](#) Nov 27 '11 at 11:27

4 @curiousguy Amen, brother. It's sad to see how often C++ get bashed from not being Java, when value semantics is one of the things that makes C++ so insanely powerfull. – [fgp](#) Jan



22 '13 at 16:42

This is not a feature, not a quirk/misfeature. It is normal on-stack-copying behavior, since calling a function with an arg or (same) allocating stack variable of type `Base` must take exactly `sizeof(Base)` bytes in memory, with possible alignment, maybe, that's why "assignment" (on-stack-copy) will not copy derived class members, their offsets are outside `sizeof`. To avoid "losing data", just use pointer, like anyone else, since pointer memory is fixed in place and size, whereas stack is very volatile – [Croll](#) Nov 16 '18 at 20:32

Definitely a misfeature of C++. Assigning a derived object to a base object should be banned, while binding a derived object to a reference or a pointer of the base class should be OK. – [John Z. Li](#) May 2 '19 at 9:11



7



So ... Why is losing the derived information bad? ... because the author of the derived class may have changed the representation such that slicing off the extra information changes the value being represented by the object. This can happen if the derived class is used to cache a representation that is more efficient for certain operations, but expensive to transform back to the base representation.

Also thought someone should also mention what you should do to avoid slicing... Get a copy of C++ Coding Standards, 101 rules guidelines, and best practices. Dealing with slicing is #54.

It suggests a somewhat sophisticated pattern to fully deal with the issue: have a protected copy constructor, a protected pure virtual `DoClone`, and a public `Clone` with an assert which will tell you if a (further) derived class failed to implement `DoClone` correctly. (The `Clone` method makes a proper deep copy of the polymorphic object.)

You can also mark the copy constructor on the base explicit which allows for explicit slicing if it is desired.

[Share](#) [Edit](#) [Follow](#)

answered Nov 8 '08 at 17:38

**Steve Steiner**

5,219 ● 3 ● 30 ● 43

3 "You can also mark the copy constructor on the base explicit" which does **not** help at all.  
– [curiousguy](#) Aug 4 '12 at 22:25



7



I see all the answers mention when object slicing happens when data members are sliced. Here I give an example that the methods are not overridden:

```
class A{
public:
    virtual void Say(){
        std::cout<<"I am A"<<std::endl;
    }
}
```

```
};

class B: public A{
public:
    void Say() override{
        std::cout<<"I am B"<<std::endl;
    }
};

int main(){
    B b;
    A a1;
    A a2=b;

    b.Say(); // I am B
    a1.Say(); // I am A
    a2.Say(); // I am A    why???
}
```

B (object b) is derived from A (object a1 and a2). b and a1, as we expect, call their member function. But from polymorphism viewpoint we don't expect a2, which is assigned by b, to not be overridden. Basically, a2 only saves A-class part of b and that is object slicing in C++.

To solve this problem, a reference or pointer should be used

```
A& a2=b;
a2.Say(); // I am B
```

or

```
A* a2 = &b;
a2->Say(); // I am B
```

Share Edit Follow

edited Apr 22 at 8:34



Alan Birtles

24.6k ● 4 ● 22 ● 45

answered Sep 12 '20 at 11:27



Soroush

1,760 ● 4 ● 18 ● 30

## 1. THE DEFINITION OF SLICING PROBLEM

5

If D is a derived class of the base class B, then you can assign an object of type Derived to a variable (or parameter) of type Base.

### EXAMPLE

```
class Pet
{
public:
    string name;
```

```
};
class Dog : public Pet
{
public:
    string breed;
};

int main()
{
    Dog dog;
    Pet pet;

    dog.name = "Tommy";
    dog.breed = "Kangal Dog";
    pet = dog;
    cout << pet.breed; //ERROR
```

Although the above assignment is allowed, the value that is assigned to the variable `pet` loses its `breed` field. This is called the **slicing problem**.

## 2. HOW TO FIX THE SLICING PROBLEM

To defeat the problem, we use pointers to dynamic variables.

### EXAMPLE

```
Pet *ptrP;
Dog *ptrD;
ptrD = new Dog;
ptrD->name = "Tommy";
ptrD->breed = "Kangal Dog";
ptrP = ptrD;
cout << ((Dog *)ptrP)->breed;
```

In this case, none of the data members or member functions of the dynamic variable being pointed to by `ptrD` (descendant class object) will be lost. In addition, if you need to use functions, the function must be a virtual function.

Share Edit Follow

edited Jan 28 '12 at 18:16

answered Jan 28 '12 at 18:00




haberdar

501 ● 5 ● 6


8 I understand the "slicing" part, but I don't understand "problem". How is it a problem that some state of `dog` that isn't part of class `Pet` (the `breed` data member) isn't copied in the variable `pet` ? The code is only interested in the `Pet` data members - apparently. Slicing is definitely a "problem" if it is unwanted, but I don't see that here. – [curiousguy](#) Feb 18 '12 at 4:18

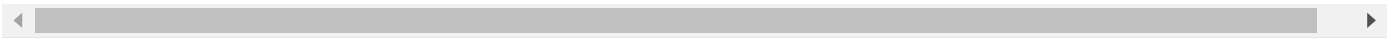
4 " ((Dog \*)ptrP) " I suggest using `static_cast<Dog*>(ptrP)` – [curiousguy](#) Feb 18 '12 at 4:20

I suggest pointing out that you will make the string 'breed' eventually leak memory without a virtual destructor (the destructor of 'string' will not be called) when deleting through 'ptrP'...

Why is what you show problematic? The fix is mostly proper class design. The problem in this case is that writing down constructors to control visibility when inheriting is tedious and easily forgotten. You won't get anywhere near the danger zone with your code as there is no polymorphism involved or even mentioned (slicing will truncate your object but not make your program crash, here). – [Dude](#) Oct 18 '12 at 2:58 

- 25 -1 This completely fails to explain the actual problem. C++ has value semantics, **not** reference semantics like Java, so this is all entirely to be expected. And the "fix" really is an example of truly *horrible* C++ code. "Fixing" non-existing problems like this type of slicing by resorting to dynamic allocation is a recipe for buggy code, leaked memory and horrible performance. Note that there *are* cases where slicing is bad, but this answer fails to point them out. Hint: the trouble starts if you assign through *references*. – [fgp](#) Jan 22 '13 at 16:35

Do you even understand that trying to access member of type that is not defined ( `Dog::breed` ) is no way an ERROR related to SLICING? – [Croll](#) Nov 16 '18 at 20:22 



It seems to me, that slicing isn't so much a problem other than when your own classes and program are poorly architected/designed.

If I pass a subclass object in as a parameter to a method, which takes a parameter of type superclass, I should certainly be aware of that and know the internally, the called method will be working with the superclass (aka baseclass) object only.

It seems to me only the unreasonable expectation that providing a subclass where a baseclass is requested, would somehow result in subclass specific results, would cause slicing to be a problem. Its either poor design in the use of the method or a poor subclass implementation. I'm guessing its usually the result of sacrificing good OOP design in favor of expediency or performance gains.

Share Edit Follow

answered Jul 24 '09 at 19:45




**Minok**

522 ● 4 ● 9

- 4 But remember, Minok, that you're NOT passing in a reference of that object. You're passing a NEW copy of that object, but using the base class to copy it in the process. – [Arafangion](#) Dec 22 '10 at 11:06

protected copy/assignment on the base class and this problem is solved. – [Dude](#) Oct 18 '12 at 2:51

- 1 You're right. Good practice is to use abstract base classes or to restrict the access to copy/assignment. However, it's not so easy to spot once it's there and easy to forget to take care of. Calling virtual methods with sliced `*this` can make mysterious things happen if you get away without an access violation. – [Dude](#) Oct 18 '12 at 3:06 

- 1 I recall from my C++ programming courses in university that there were standing best practices that for every class we created, we were required to write default constructors, copy constructors and assignment operators, as well as a destructor. This way you made

sure that copy construction and the like happened the way you needed it to, while writing the class... rather than later on some odd behavior showing up. – [Minok](#) Jul 25 '14 at 17:24

OK, I'll give it a try after reading many posts explaining object slicing but not how it becomes problematic.

4

The vicious scenario that can result in memory corruption is the following:

- Class provides (accidentally, possibly compiler-generated) assignment on a polymorphic base class.
- Client copies and slices an instance of a derived class.
- Client calls a virtual member function that accesses the sliced-off state.

Share Edit Follow

answered Oct 18 '12 at 3:22



[Dude](#)

583 ● 2 ● 9

Slicing means that the data added by a subclass are discarded when an object of the subclass is passed or returned by value or from a function expecting a base class object.

3

**Explanation:** Consider the following class declaration:

```
class baseclass
{
    ...
    baseclass & operator =(const baseclass&);
    baseclass(const baseclass&);
}
void function( )
{
    baseclass obj1=m;
    obj1=m;
}
```

As baseclass copy functions don't know anything about the derived only the base part of the derived is copied. This is commonly referred to as slicing.

Share Edit Follow

edited Jun 9 '19 at 14:28



[Jean-François Fabre](#) ♦

129k ● 22 ● 109 ● 175

answered Mar 12 '14 at 18:08



[Santosh](#)

1,184 ● 2 ● 14 ● 30

class A

0

```
{
    int x;
};

class B
{
    B( ) : x(1), c('a') { }
    int x;
    char c;
};

int main( )
{
    A a;
    B b;
    a = b;    // b.c == 'a' is "sliced" off
    return 0;
}
```

Share Edit Follow

edited Nov 29 '12 at 12:55



Tim Cooper

147k ● 36 ● 309 ● 266

answered Nov 29 '12 at 12:32



quidkid

17 ● 1

5 Would you mind giving some extra details? How does your answer differ from the already posted ones? – [Alexis Pigeon](#) Nov 29 '12 at 12:55

3 I guess that more explanation wouldn't be bad. – [looper](#) Nov 29 '12 at 12:55

when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off (discard) from the base class object.

-1

```
class Base {
    int x;
};

class Derived : public Base {
    int z;
};

int main()
{
    Derived d;
    Base b = d; // Object Slicing, z of d is sliced off
}
```

Share Edit Follow

answered May 11 '16 at 7:39



Varun Kumar

166 ● 1 ● 8

-1

When a Derived class Object is assigned to Base class Object, all the members of derived class object is copied to base class object except the members which are not present in the base class. These members are Sliced away by the compiler. This is called Object Slicing.



**Here is an Example:**

```
#include<bits/stdc++.h>
using namespace std;
class Base
{
    public:
        int a;
        int b;
        int c;
        Base()
        {
            a=10;
            b=20;
            c=30;
        }
};
class Derived : public Base
{
    public:
        int d;
        int e;
        Derived()
        {
            d=40;
            e=50;
        }
};
int main()
{
    Derived d;
    cout<<d.a<<"\n";
    cout<<d.b<<"\n";
    cout<<d.c<<"\n";
    cout<<d.d<<"\n";
    cout<<d.e<<"\n";

    Base b = d;
    cout<<b.a<<"\n";
    cout<<b.b<<"\n";
    cout<<b.c<<"\n";
    cout<<b.d<<"\n";
    cout<<b.e<<"\n";
    return 0;
}
```

**It will generate:**

```
[Error] 'class Base' has no member named 'd'
[Error] 'class Base' has no member named 'e'
```

Share Edit Follow

answered Jul 19 '17 at 8:02



Ghulam Moinul Quadir

1,488 ● 1 ● 9 ● 16

- 1 Downvoted because that's not a good example. It wouldn't work either if instead of copying d to b, you would use a pointer in which case d and e would still exist but Base doesn't have those members. Your example only shows that you can't access members that the class doesn't have. – [Stefan Fabian](#) Mar 29 '19 at 12:33

[Why should I not #include <bits/stdc++.h>?](#), [Why is "using namespace std;" considered bad practice?](#) – [phuclv](#) Apr 18 at 11:06



I just ran across the slicing problem and promptly landed here. So let me add my two cents to this.

-2



Let's have an example from "production code" (or something that comes kind of close):



Let's say we have something that dispatches actions. A control center UI for example.

This UI needs to get a list of things that are currently able to be dispatched. So we define a class that contains the dispatch-information. Let's call it `Action`. So an `Action` has some member variables. For simplicity we just have 2, being a `std::string name` and a `std::function<void()> f`. Then it has an `void activate()` which just executes the `f` member.

So the UI gets a `std::vector<Action>` supplied. Imagine some functions like:

```
void push_back(Action toAdd);
```

Now we have established how it looks from the UI's perspective. No problem so far. But some other guy who works on this project suddenly decides that there are specialized actions that need more information in the `Action` object. For what reason ever. That could also be solved with lambda captures. This example is not taken 1-1 from the code.

So the guy derives from `Action` to add his own flavour. He passes an instance of his home-brewed class to the `push_back` but then the program goes haywire.

So what happened?

As you *might* have guessed: the object has been sliced.



The extra information from the instance has been lost, and `f` is now prone to undefined behaviour.

---

I hope this example brings light about for those people who can't really imagine things when talking about `A` s and `B` s being derived in some manner.

Share Edit Follow

answered Oct 13 '17 at 8:17



Martin B.

1,366 ● 9 ● 23



**Highly active question.** Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.