13. The answers are found in the following two FAQs at https://isocpp.org/wiki/faq/virtual-functions

### What is a “virtual constructor”?

An idiom that allows you to do something that C++ doesn’t directly support. You can get the effect of a virtual constructor by a virtual clone() member function (for copy constructing), or a virtual create() member function (for the [default constructor](https://isocpp.org/wiki/faq/ctors#default-ctor)).

1. class Shape {
2. public:
3. virtual ~Shape() { } *// A virtual destructor*
4. virtual void draw() = 0; *// A pure virtual function*
5. virtual void move() = 0;
6. *// ...*
7. virtual Shape\* clone() const = 0; *// Uses the copy constructor*
8. virtual Shape\* create() const = 0; *// Uses the default constructor*
9. };
10. class Circle : public Shape {
11. public:
12. Circle\* clone() const; *// Covariant Return Types; see below*
13. Circle\* create() const; *// Covariant Return Types; see below*
14. *// ...*
15. };
16. Circle\* Circle::clone() const { return new Circle(\*this); }
17. Circle\* Circle::create() const { return new Circle(); }

In the clone() member function, the new Circle(\*this) code calls Circle’s copy constructor to copy the state of this into the newly created Circle object. (Note: unless Circle is known to be [final (AKA a leaf)](https://isocpp.org/wiki/faq/strange-inheritance#final-classes), you can reduce the chance of [slicing](https://isocpp.org/wiki/faq/value-vs-ref-semantics#pass-by-value) by making its copy constructor protected.) In the create() member function, the new Circle() code calls Circle’s [default constructor](https://isocpp.org/wiki/faq/ctors#default-ctor).

Users use these as if they were “virtual constructors”:

1. void userCode(Shape& s)
2. {
3. Shape\* s2 = s.clone();
4. Shape\* s3 = s.create();
5. *// ...*
6. delete s2; *// You need a virtual destructor here*
7. delete s3;
8. }

This function will work correctly regardless of whether the Shape is a Circle, Square, or some other kind-of Shape that doesn’t even exist yet.

Note: The return type of Circle’s clone() member function is intentionally different from the return type of Shape’s clone() member function. This is called Covariant Return Types, a feature that was not originally part of the language. If your compiler complains at the declaration of Circle\* clone() const within class Circle (e.g., saying “The return type is different” or “The member function’s type differs from the base class virtual function by return type alone”), you have an old compiler and you’ll have to change the return type to Shape\*.

**Why don’t we have virtual constructors?**

A virtual call is a mechanism to get work done given partial information. In particular, virtual allows us to call a function knowing only an interfaces and not the exact type of the object. To create an object you need complete information. In particular, you need to know the exact type of what you want to create. Consequently, a “call to a constructor” cannot be virtual.

Techniques for using an indirection when you ask to create an object are often referred to as “Virtual constructors”. For example, see [TC++PL3](http://stroustrup.com/3rd.html) 15.6.2.

For example, here is a technique for generating an object of an appropriate type using an abstract class:

1. struct F { *// interface to object creation functions*
2. virtual A\* make\_an\_A() const = 0;
3. virtual B\* make\_a\_B() const = 0;
4. };
5. void user(const F& fac)
6. {
7. A\* p = fac.make\_an\_A(); *// make an A of the appropriate type*
8. B\* q = fac.make\_a\_B(); *// make a B of the appropriate type*
9. *// ...*
10. }
11. struct FX : F {
12. A\* make\_an\_A() const { return new AX(); } *// AX is derived from A*
13. B\* make\_a\_B() const { return new BX(); } *// BX is derived from B*
14. };
15. struct FY : F {
16. A\* make\_an\_A() const { return new AY(); } *// AY is derived from A*
17. B\* make\_a\_B() const { return new BY(); } *// BY is derived from B*
18. };
19. int main()
20. {
21. FX x;
22. FY y;
23. user(x); *// this user makes AXs and BXs*
24. user(y); *// this user makes AYs and BYs*
25. user(FX()); *// this user makes AXs and BXs*
26. user(FY()); *// this user makes AYs and BYs*
27. *// ...*
28. }

This is a variant of what is often called “the factory pattern”. The point is that user() is completely isolated from knowledge of classes such as AX and AY.

14. The answer is found as a FAQ here : https://isocpp.org/wiki/faq/strange-inheritance#calling-virtuals-from-base

### Is it okay for a non-virtual function of the base class to call a virtual function?

Yes. It’s sometimes (not always!) a great idea. For example, suppose all Shape objects have a common algorithm for printing, but this algorithm depends on their area and they all have a potentially different way to compute their area. In this case Shape’s area() method would necessarily have to be virtual (probably pure virtual) but Shape::print() could, [if we were guaranteed no derived class wanted a different algorithm for printing](https://isocpp.org/wiki/faq/strange-inheritance#redefining-nonvirtuals), be a non-virtual defined in the base class Shape.

1. *#include* "Shape.h"
2. void Shape::print() const
3. {
4. float a = this->area(); *// area() is pure virtual*
5. *// ...*
6. }