How do I create and use a class arrow operator?

Asked 11 years ago Active 3 years, 9 months ago Viewed 48k times

38

16

So, after researching everywhere for it, I cannot seem to find how to create a class arrow operator, i.e.,

```
class Someclass
{
  operator-> ()  /* ? */
  {
  }
};
```

I just need to know how to work with it and use it appropriately. - what are its inputs? - what does it return? - how do I properly declare/prototype it?

c++ class operator-keyword

Share Edit Follow Flag

asked Feb 8, 2011 at 1:02
Codesmith
4,970 ● 4 ● 34 ● 44

6 no inputs. Return type should be a pointer. This is usually used to create "smart" pointers, so it returns a pointer to the wrapped object. – Tim Feb 8, 2011 at 1:18

4 Answers

Sorted by: Highest Score (default)

41

The operator -> is used to overload member access. A small example:

```
#include <iostream>
struct A
{
    void foo() {std::cout << "Hi" << std::endl;}
};

struct B
{
    A a;
    A* operator->() {
        return &a;
    }
};

int main() {
    B b;
    b->foo();
}
```

This outputs:

```
Hi
```

Share Edit Follow Flag

edited Feb 8, 2011 at 1:40
user405725

answered Feb 8, 2011 at 1:15
gr0v3r
588 ● 4 ● 5

does the arrow operator also work to return structure members as it does class members? – Codesmith Feb 8, 2011 at 1:25

It can return struct members just as easily as it can return class members. What happens when you use the arrow is determined by the body of the operator->() member function. The line "return &a;" can be changed to return whatever you decide is appropriate. – Darryl Feb 8, 2011 at 1:31

The arrow operator has no inputs. Technically, it can return whatever you want, but it should return something that either is a pointer or can become a pointer [through chained `->` operators](#).

The `->` operator automatically dereferences its return value before calling its argument *using the built-in pointer dereference, not `operator*`*, so you could have the following class:

```
class PointerToString
{
    string a;

public:
    class PtPtS
    {
    public:
        PtPtS(PointerToString &s) : r(s) {}
        string* operator->()
        {
            std::cout << "indirect arrow\n";
            return &*r;
        }
    private:
        PointerToString & r;
    };

    PointerToString(const string &s) : a(s) {}
    PtPtS operator->()
    {
        std::cout << "arrow dereference\n";
        return *this;
    }
    string &operator*()
    {
        std::cout << "dereference\n";
        return a;
    }
};
```

Use it like:

```
PointerToString ptr(string("hello"));
string::size_type size = ptr->size();
```

which is converted by the compiler into:

```
string::size_type size = (*ptr.operator->().operator->()).size();
```

(with as many `.operator->()` as necessary to return a real pointer) and should output

```
arrow dereference
indirect dereference
dereference
```

Note, however, that you can do the following:

```
PointerToString::PtPtS ptr2 = ptr.operator->();
```

run online: https://wandbox.org/permlink/Is5kPamEMUCA9nvE

From Stroupstrup:

> The transformation of the object *p* into the pointer `p.operator->()` does not depend on the member *m* pointed to. That is the sense in which `operator->()` is a unary postfix operator. However, there is no new syntax introduced, so a member name is still required after the `->`

Share Edit Follow Flag

edited Jun 7, 2018 at 5:49

cztchoice
39 ● 8

answered Feb 8, 2011 at 1:30

Daniel Gallagher
6,712 ● 24 ● 30

There is no way this code compiles on a decent compiler. – Sebastian Redl Mar 30, 2015 at 17:01

@SebastianRedl, you are correct. I've updated the explanation and example so that it does compile. – Daniel Gallagher Apr 3, 2015 at 14:09

Interesting, but what's the point of unneeded indirections? why not just returning reference of underlying object? (reference of `string a` in your case). Why making simple things complicated? I would downvote you for complicating, but first there must be a reason for your approach. – metablaster Apr 24, 2020 at 15:02

1

```
class T {
    public:
        const memberFunction() const;
};

// forward declaration
class DullSmartReference;

class DullSmartPointer {
    private:
        T *m_ptr;
    public:
        DullSmartPointer(T *rhs) : m_ptr(rhs) {};
        DullSmartReference operator*() const {
            return DullSmartReference(*m_ptr);
        }
        T *operator->() const {
            return m_ptr;
        }
};
```

http://en.wikibooks.org/wiki/C++_Programming/Operators/Operator_Overloading#Address_of.2C_Reference.2C_and_Pointer_operators

Share Edit Follow Flag

answered Feb 8, 2011 at 1:05

Anycorn

48.3k ●42 ●160 ●257

-1

The "arrow" operator can be overloaded by:

```
a->b
```

will be translated to

```
return_type my_class::operator->()
```

Share Edit Follow Flag

answered Feb 8, 2011 at 1:09

Foo Bah

24.4k ●5 ●52 ●79

1 It is not that easy, `return_type{}->()->()->() ... ->()` has to be a pointer for this to be a valid code. – alfC May 13, 2017 at 1:37