7.4a — Returning values by value, reference, and address

BY ALEX ON FEBRUARY 25TH, 2008 | LAST MODIFIED BY ALEX ON FEBRUARY 20TH, 2016

In the three previous lessons, you learned about passing arguments to functions by value, reference, and address. In this section, we'll consider the issue of returning values back to the caller via all three methods.

As it turns out, returning values from a function to its caller by value, address, or reference works almost exactly the same way as passing parameters to a function does. All of the same upsides and downsides for each method are present. The primary difference between the two is simply that the direction of data flow is reversed. However, there is one more added bit of complexity -- because local variables in a function go out of scope when the function returns, we need to consider the effect of this on each return type.

Return by value

Return by value is the simplest and safest return type to use. When a value is returned by value, a copy of that value is returned to the caller. As with pass by value, you can return by value literals (eg. 5), variables (eg. x), or expressions (eg. x+1), which makes return by value very flexible.

Another advantage of return by value is that you can return variables (or expressions) that involve local variables declared within the function without having to worry about scoping issues. Because the variables are evaluated before the function goes out of scope, and a copy of the value is returned to the caller, there are no problems when the function's variable goes out of scope at the end of the function.

```
int doubleValue(int x)
{
    int value = x * 2;
    return value ; // A copy of value will be returned here
} // value goes out of scope here
```

Return by value is the most appropriate when returning variables that were declared inside the function, or for returning function arguments that were passed by value. However, like pass by value, return by value is slow for structs and large classes.

When to use return by value:

- When returning variables that were declared inside the function
- When returning function arguments that were passed by value

When not to use return by value:

- When returning an built-in array or pointer (use return by address)
- When returning a large struct or class (use return by reference)

Return by address

Returning by address involves returning the address of a variable to the caller. Similar to pass by address, return by address can only return the address of a variable, not a literal or an expression (which don't have addresses). Because return by address just copies an address from the function to the caller, return by address is fast.

However, returning by address has one additional downside that pass by reference doesn't -- you can not return local variables to the function by address. Consider the following example:

```
int* doubleValue(int x)
{
    int value = x * 2;
    return &value; // return nValue by address here
} // value destroyed here
```

As you can see here, value is destroyed just after its address is returned to the caller. The end result is that the caller ends up with the address of non-allocated memory (a dangling pointer), which will cause problems if used. This is a common mistakes that new programmers make. Many newer compilers will give a warning (not an error) if the programmer tries to return a local variable by address -- however, there are quite a few ways to trick the compiler into letting you do something illegal without generating a warning, so the burden is on the programmer to ensure the address they are returning will be to a valid variable after the function returns.

Return by address is often used to return dynamically allocated memory to the caller:

```
int* allocateArray(int size)
{
    return new int[size];
}

int main()
{
    int *array = allocateArray(25);

    // do stuff with array

    delete[] array;
    return 0;
}
```

When to use return by address:

- When returning dynamically allocated memory
- When returning function arguments that were passed by address

When not to use return by address:

- When returning variables that were declared inside the function (use return by value)
- When returning a large struct or class (use return by reference)

Return by reference

Similar to pass by reference, values returned by reference must be variables (you can not return a reference to a literal or an expression). When a variable is returned by reference, a reference to the variable is passed back to the caller. The caller can then use this reference to continue modifying the variable, which can be useful at times. Return by reference is also fast, which can be useful when returning structs and classes.

However, just like return by address, you should not return local variables to the function by reference. Consider the following example:

```
int& doubleValue(int x)
{
    int value = x * 2;
    return value; // return a reference to value here
} // value is destroyed here
```

In the above program, the program is returning a reference to a value that will be destroyed when the function returns. This would mean the caller receives a reference to garbage. Fortunately, your compiler will probably give you a warning or error if you try to do this.

Return by reference is typically used to return arguments passed by reference to the function back to the caller. In the following example, we return (by reference) an element of an array that was passed to our function by reference:

```
#include <array>
#include <iostream>

// Returns a reference to the index element of array
int& getElement(std::array<int, 25> &array, int index)
{
    // we know that array[index] will not be destroyed when the function ends
```

```
    // so it's okay to return it by reference
    return array[index];
}

int main()
{
    std::array<int, 25> array;

    // Set the element of array with index 10 to the value 5
    getElement(array, 10) = 5;

    std::cout << array[10] << '\n';

    return 0;
}
```

This prints:

5

When we call `getElement(array, 10)`, getElement() returns a reference to the element of the array inside array that has the index 10. main() then uses this reference to assign that element the value 5.

Although this is somewhat of a contrived example (because you can access array[10] directly), once you learn about classes you will find a lot more uses for returning values by reference.

When to use return by reference:

- When returning a reference parameter
- When returning a large struct or class that has scope outside of the function

When not to use return by reference:

- When returning variables that were declared inside the function (use return by value)
- When returning a built-in array or pointer value (use return by address)

Mixing return references and values

Although a function may return a value or a reference, the caller may or may not assign the result to a value or reference accordingly. Let's look at what happens when we mix value and reference types.

```
int returnByValue()
{
    return 5;
}

int& returnByReference()
{
     static int x = 5; // static ensures x doesn't go out of scope when we return it by refere
nce
     return x;
}

int main()
{
    int value = returnByReference(); // case A -- ok, treated as return by value
    int &ref = returnByValue(); // case B -- compile error
    const int &cref = returnByValue(); // case C -- ok, the lifetime of return value is extend
ed to the lifetime of cref
}
```

In case A, we're assigning a reference return value to a non-reference variable. Because value isn't a reference, the return value is copied into value, as if returnByReference() had returned by value. In case B, we're trying to initialize reference ref

with the copy of the return value returned by returnByValue(). However, because the value being returned doesn't have an address (it's an rvalue), this will cause a compile error.
In case C, we're trying to initialize const reference ref with the copy of the return value returned by returnByValue(). Because const references can be assigned to rvalues, perhaps surprisingly, this actually works! The lifetime of the return value is extended to match the lifetime of cref so that it doesn't die until cref does.

Conclusion

Most of the time, return by value will be sufficient for your needs. It's also the most flexible and safest way to return information to the caller. However, return by reference or address can also be useful, particularly when working with dynamically allocated classes or structs. When using return by reference or address, make sure you are not returning a reference to, or the address of, a variable that will go out of scope when the function returns!

Quiz time

Write function prototypes for each of the following functions. Use the most appropriate parameter and return types (by value, by address, or by reference), including use of const where appropriate.

1) A function named sumTo() that takes an integer parameter and returns the sum of all the numbers between 1 and the input number.

Show Solution

2) A function named printEmployeeName() that takes an Employee struct as input.

Show Solution

3) A function named minmax() that takes two integers as input and returns the smaller and larger number as separate parameters.

Show Solution

4) A function named getIndexOfLargestValue() that takes an integer array (as a pointer) and an array size, and returns the index of the largest element in the array.

Show Solution

5) A function named getElement() that takes an integer array (as a pointer) and an index and returns the array element at that index (not a copy). Assume the index is valid, and the return value is const.

Show Solution

 **7.5 -- Inline functions**

 **Index**

 **7.4 -- Passing arguments by address**

Share this:

Email Facebook 8 Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

52 comments to 7.4a — Returning values by value, reference, and address

PRABHAKAR
February 26, 2008 at 7:51 am · Reply

DEAR ALEX, your extremely fast responce to my query, by adding section 7.4a on 26th feb itself -i am overjoyed that my query deserved attention. thanks. i will persue c++ more vigorously. please extend your best wishes for me to have patience PRABHAKAR

saç bak?m?
June 14, 2008 at 1:52 am · Reply

thanks

INDIAN
August 11, 2008 at 8:01 pm · Reply

I am glad to see a number of indian names here such as Abhishek and Prabhakar using Internet to their advantage.

I have been quietly reading through your tutorials on c++ and I must admit it is one of the best!!

Pathik
November 25, 2008 at 6:50 pm · Reply

There's also me but i name myself afds. 😀

Pathik
November 25, 2008 at 6:53 pm · Reply

great tutorials by the way.

Alex
November 25, 2008 at 8:49 pm · Reply

According to Google Analytics, 11% of the visitors of this site are Indian.

Sylvio
March 1, 2012 at 10:28 am · Reply

And one brazilian here, also!

Alex, thanks for this excellent material. I've already said thanks before, but now once again. In my opinion, the best free material on c++ in the internet. Just amazing.

Greetings from Rio.

vipul
January 26, 2014 at 10:14 am · Reply

i never touch c++ book of college subject and i only use for exam and only see question and solve myself due to this website and lot of information available here and i really appreciate for alex thanks a lot !!!

Ben
November 5, 2008 at 6:01 pm · Reply

Alex,

Is there a way to pass something back to the main function without returning it.

For example, I want to create a function that reads data from the user: arguments are number of data to be entered and type of those data (int, double, char…). So I thought I would create a dynamic array of the required size and type (by using "switch"). But how can I pass that array back to the main function? Actually the way I want to define my function there's no "return", it would have to be a "void".

And we can't overload the returning type so this doesn't help me much.

Maybe a better way to do it would be to pass the dynamic array as an argument with the appropriate type so I could use function overloading. It's kind of avoiding the issue without solving it though. Plus how can you find the length of a dynamic array? (something equivalent to "strlen" for example).

Thanks,

Ben

Alex
November 8, 2008 at 11:38 am · Reply

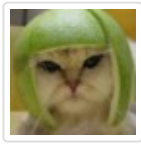
If you need to pass back an object from a function with a void return type, pass it as a reference parameter.

There's no way to find the length of a dynamic array that I know of. This is always one of the things that's bugged me about C++ (doubly so because the C++ runtime has to know the length of the array in order to delete it).

Ben
November 10, 2008 at 12:17 pm · Reply

Sorry I'm not clear with your answer. Could you refer me to the appropriate chapter or show me an exemple?

I understand "Return by Reference" but what is "Pass back as a Reference parameter"?



Alex
November 10, 2008 at 6:17 pm · Reply

I'm referring to the concepts in the lesson on **reference parameters**.

For example:

```
1  void foo(int &nX)
2  {
3      nX = 5;
4  }
5
6  int main()
7  {
8      int nX = 4;
9      foo(nX); // nX will now equal 5
10 }
11 <!--formatted-->
```

Reference parameters give you the ability to change the value of the variable passed in, so you can use them to essentially return things from your function without using a return value. For example, the above function foo() returns 5 in nX.

dimonic
May 18, 2011 at 5:17 am · Reply

Excuse the late reply - I found this from a google search.

Isn't it more likely the c++ runtime that deletes arrays? So the compiler cannot tell how big an allocated array will be (becuase it can be allocated by a variable amount). So if you want an array you can tell the siz of, you should use an stl vector.

Ben
December 30, 2008 at 2:04 pm · Reply

Alex,

I tried to define a dynamic array in the main() function

```
1  int *anTest
```

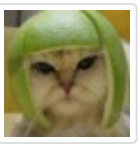
then initialize it in another function (called after the definition):

```
1  int *anTest
2  ReadInput(anTest)
3
4  void ReadInput(int *pn)
5  {
6     nbpts = 10;
7     pn = new int[nbpts];
8  }
```

It compiles fine, but when I run it, it tells me that "anTest was not initialized".

So how can I define or initialize a dynamic array in a function then send it back to my main function? I can't really use the Return by address because I have more than one dynamic array to define in the same function.

I need to do that because the size of my arrays will be read from a file (i.e "ReadInput()") but the arrays will be used throughout my program and therefore should belong to the main() function. Oh and except the size, all the information in those 2 arrays will also be read from the file, so I don't know how I could define them afterwards.



Alex
January 3, 2009 at 5:17 pm · Reply

Great question. The problem here is that when you pass anTest to ReadInput(), the address that anTest is holding is COPIED into pn. This means that when you allocate a dynamic array into pn, you're essentially overwriting the copied address, and the value of anTest never gets changed.

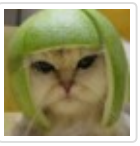
What you need to do is pass the pointer by reference, so that when you assign your dynamic array to pn, you'll actually be assigning the value to anTest, and not a copy of anTest.

You can do that by changing your declaration of ReadInput to the following:

```
1 void ReadInput(int *&pn)
```

Tony
January 18, 2009 at 9:17 pm · Reply

Functions can be lvalues? Did you just use a function as an lvalue in the return by reference example?



Alex
February 5, 2009 at 11:08 pm · Reply

I didn't use the function itself as an lvalue, I used the reference that is returned by the function as an lvalue.

Kerrick
February 15, 2009 at 8:00 am · Reply

Can you use the static keyword with a local variable in order to return it by reference or address? Thanks for these tutorials by the way.

Alex
October 26, 2015 at 12:25 pm · Reply

Yes. I've updated the tutorial to include an example that does just that.

LOLY
April 6, 2009 at 7:27 am · Reply

Thaaaaank u Mr,Alex ..
useful tutorial

five
June 17, 2009 at 10:59 am · Reply

Hey Alex and also everyone on this site. This is an extremely great site, I've "favorite'd' hhaha many c++

sites around the web, but goodness, this is some good stuff here..Even though this section just whizzes right over my head, I havnt seen a site that explains c++ this well. Anyway, 7-7.4 thus far is smoking my brain….is there another angle to look at this section from, haha, seriously though.

Five aka Mersinarii

Michel
July 8, 2009 at 4:58 pm · Reply

How does a varible passed by reference, change the original variable's data while being modified at once? (I know this sounds confusing but please try to understand

Kurt
July 14, 2009 at 8:06 pm · Reply

A reference works like an alias for the original data. You're not changing the value of the reference itself, because references are in a sense immutable. You're changing the value of the data the reference aliases. It's like modifying the dereferenced value of a pointer.

pankaj
January 2, 2010 at 9:36 am · Reply

I try to return the value of the local variable by using return by reference. It is possible to do that.
I am getting the correct values.
Compiler is only giving waring…warning C4172: returning address of local variable or temporary

plese see the code below and kindly give your comments..

```
result of below code is

Hello World!
res = 7
res1 = 12

int add (int x, int y)
{
int result = x+y;
return result;
}

// return by reference
int& multiply(int x, int y)
{
 int result = x*y;
 return result;
}

int main(int argc, char* argv[])
{
    printf("Hello World!\n");
    int res = add(3,4);
    std::cout<< "res = " << res << std::endl;

    int res1 = multiply(3,4);
    std::cout << "res1 = " << res1 << std::endl;
    return 0;
}
```

Alex
October 26, 2015 at 12:27 pm · Reply

Returning a reference to a local variable is something you shouldn't do, as you'd be returning a reference to a value that is being destroyed when the function ends.

ellankavi
July 27, 2010 at 6:20 am · Reply

Hello!
In the following code, is the dynamic array deleted properly?

```
#include<iostream>

int* CreateArray(int nSize)
{
    int* const npPtr = new int[nSize];

    //initialize the array
    for (int iii=0;iii<nSize;iii++)
        npPtr[iii]=iii;
    return npPtr;
}

int main()
{
    int nSize=5;
    int *apArray=CreateArray(nSize);
    delete[] apArray;
    system("pause");
    return 0;
}
```

Thanks 🙂

Alex
October 26, 2015 at 12:29 pm · Reply

Yes.

Sunny Chouhan
June 20, 2012 at 3:46 am · Reply

Hi Alex,
As Mention in your article 'Return by reference' local variable cannot be return because they go out of scope when function returns.
But I did not receive any compile error
program is running fine
I am using visual studio 2010 (vc++)

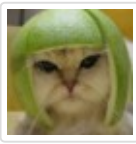
#include
using namespace std;

int main()
{
int &DoubleValue(int x);
int number;
number = DoubleValue(10);
cout << "Number = " << number << endl;

```
cin.clear();
cin.ignore(255, '\n');
cin.get();
return 0;
}
int& DoubleValue(int nX)
{
int nValue = nX * 2;
return nValue; // return a reference to nValue here
}
```

vipul
January 26, 2014 at 10:50 am · Reply

what is the diffrence betn int &DoubleValue(int x); and int& DoubleValue(int nX) ?i am stucked in int& and int &!!!!please understand this



Alex
October 26, 2015 at 12:33 pm · Reply

They are the same thing.

Alex
October 26, 2015 at 12:33 pm · Reply

It's possible that since you're printing the value of number immediately, the value of nValue in the function hasn't actually been destroyed/overwritten yet.

Trying to access a reference to a variable that has been destroyed will lead to unpredictable results, and should be avoided.

MrFinn
November 12, 2012 at 11:07 pm · Reply

I found this sentence (from MSVC++ documentation) to be very clarifying:
"A reference holds the address of an object, but behaves syntactically like an object."

Therefore, the object which is passed back to the caller must keep on existing (and it's address as well) when the function goes out of scope. This is why you can not pass reference to a local variable which ceases to exist. If you change the orginal sample code so that nValue is declared in global scope it will work because nValue object stays alive.

int nValue; // declared in global scope.

```
int& DoubleValue(int nX) {
//int nValue = nX * 2;
nValue = nX * 2;
return nValue;
}
```

BTW. Sunnys program does return the same error message here (Qt & MinGW) and the math is wrong:
"warning: reference to local variable 'nValue' returned [enabled by default]"

MrFinn
November 12, 2012 at 11:16 pm · Reply

PS. The latter example in the text works for this same reason, the struct is declared in gobal scope and therefore still keeps on existing after the function call.

Reinstein
February 3, 2015 at 11:22 am · Reply

HI Alex. Many thanks to you. You are crafting programmers out of many

lt1
July 13, 2015 at 6:11 pm · Reply

Hi Alex,

I have a question about the following program:

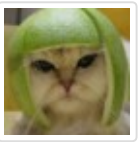
```
1
```

```
int& Value(FixedArray25 &rArray, int nIndex)
{
    return rArray.anValue[nIndex];
}
```

```
1
```

Can we consider a reference as a dereferenced pointer? If we can consider so, when we call the function above, the Value can be consider as a pointer to rArray.anValue[nIndex], but when the function is finished, the rArray.anValue[nIndex] will go out of scope, and the Value will pointing to a garbage. Is there something wrong?

Thank you for your good tutorial!!!



Alex
July 14, 2015 at 9:07 am · Reply

Yes, one way to think about references is to consider them as implicitly-dereferenced pointers. Another way is to consider them as aliases to the underlying data.

I think your function should work fine. When the function terminates, the rArray reference goes out of scope, but the underlying data (the value you passed in for rArray) does not. Consequently, the reference returned by Value is still valid after the function terminates.

Mr D
September 25, 2015 at 1:23 pm · Reply

Hi Alex,

In this tutorial you seem to start doing something that you earlier (6.7) said we shouldn't do:

```
1 int* iPtr2; // also valid syntax (but don't do this)
```

But in this lesson:

```
1 int* DoubleValue(int nX)
```

…..so, the * next to int. And several other places in this tutorial.
So me is getting confused, is there some special reason for this or…….

Alex
September 27, 2015 at 3:16 pm · Reply

Yes, there is a slight difference.

In the case where we're declaring a variable, putting the asterisk next to the variable name helps us avoid this mistake:

```
1 int* ptr1, ptr2; // ptr1 is a pointer to an int, but ptr2 is a plain int!
```

(IMO, C++ made a syntactical mistake here, they really should both be pointers to int, but that's not the case)

However, with a function return value where there's only one return value:

```
1 int* foo(); // declaring a function that returns some value of type int*
```

Makes it clear that the function is returning a value of type int*.

If you do this instead:

```
1 int *foo(); // declaring a function that returns some value of type int*
```

It's easy to misinterpret this as a pointer to a function that returns an plain int.

For that reason, I find using attaching the * to the type for return values clearer.

Make sense?

I've updated lesson 6.7 to differentiate these two cases.

eli
September 30, 2015 at 10:12 pm · Reply

Hi Alex,

I still have a small missing part, that is related to assignments.
On regular assignments a = b, b is copied and assigned to a.
When we declare a as reference (SomeStruct &a = b), b i not copied.
Now, what happens when instead of b we have a function?

```
1 SomeStruct& a = ByReferenceFunction(); // no copy is made
2 SomeStruct& b = ByValueFunction(); // a copy is made before function returns, and b now poin
3 ts to it
4 SomeStruct c = ByReferenceFunction(); // UNCLEAR: no copy is made by the function, but is th
  e returned value copied to c?
  SomeStruct d = ByValueFunction(); // UNCLEAR: a copy is made before function returns, but is
  it copied again before assigning it to d?
```

Thanks,
Eli

Alex
October 1, 2015 at 12:12 pm · Reply

Good question.

> SomeStruct& a = ByReferenceFunction(); // no copy is made

a references the return value, no copy is made.

> SomeStruct& b = ByValueFunction(); // a copy is made before function returns, and b now points to it

This won't compile. You can't set a reference to a rvalue because the return value has no memory address. However, you could do this:

```
const SomeStruct& b = ByValueFunction();
```

In this case, the lifetime of the copied return value is extended to the lifetime of the const reference.

> SomeStruct c = ByReferenceFunction(); // UNCLEAR: no copy is made by the function, but is the returned value copied to c?

This returns a reference to some value, but because c is not a reference, a copy is made and placed in c. So this is essentially the same thing as not returning a reference.

> SomeStruct d = ByValueFunction(); // UNCLEAR: a copy is made before function returns, but is it copied again before assigning it to d?

No, the return value is copied into d.

Mario
November 20, 2015 at 8:04 am · Reply

Hi Alex,
Congrats for the great tutorial. By far the best of C++ I've ever come across!

One question, in the following code, that you have used when explaining 'return by address', after using the delete operator, shouldn't we set 'array' to be a null pointer? (array = 0)
Otherwise wouldn't the pointer 'array' end up as a dangling pointer?:

```
int* allocateArray(int size)
{
    return new int[size];
}

int main()
{
    int *array = allocateArray(25);

    // do stuff with array

    delete[] array;
    return 0;
}
```

Thanks
Mario

Alex
November 20, 2015 at 1:26 pm · Reply

We could, but since the program ends immediately after, it doesn't really matter. Dangling pointers won't do any harm when the program ends.

Mario
November 22, 2015 at 3:26 am · Reply

Good point!
Thank you.

Mike
December 5, 2015 at 10:46 pm · Reply

Hi Alex!

In quiz 1, I think it should be "…and returns the sum of all the number_S_ between 1 and the input number".

In quiz 4 and 5, isn't it safer to declare int *array as const?

Thank you!

Alex
December 7, 2015 at 12:38 pm · Reply

Correct on all counts. I've updated the article. Thanks!

Bede
February 19, 2016 at 12:50 pm · Reply

Hello Alex,
In the solution of quiz 5, I think, there's a mistake now:

```
1 int& getElement(const int *array, int index);
```

It is supposed to return an array element (no copy), so IMHO it should be either:

```
1 const int& getElement(const int *array, int index); // for read-only access of the returned element
```

or

```
1 int& getElement(int *array, int index); // to be able to modify the returned element
```

Alex
February 20, 2016 at 11:32 am · Reply

Yep, thanks for pointing this out. I've updated the quiz to address this.

Leonardo
February 24, 2016 at 10:21 am · Reply

Hi Alex
Greetings for your excellent blog. I have a question and i need help .

I have a source code with three files. There is a file called MEDICAO.c . In this file there is a function called LER_VRMS ( ). In this file, there is a function called LER_VRMS ( ) , within this function, is called other function called LER_REG_24_BITS ( ) that is located inside another file.(DRIVER_ADE.c) After being executed, the return value should be assigned to the array [ ] .
But that is not happening , I get a completely different value.

Note: Array is declarared as global in file MEDIÇÃO.c

Regards
Leonardo

Alex
February 28, 2016 at 12:42 pm · Reply

Hi Leonardo. I can't even begin to guess what the problem might be from the information you've given me. My advice is use the debugger to step through your program and watch it execute. Put watches on the variables you're interested in and see where their values are getting changed.

Xhevo
February 26, 2016 at 3:32 am · Reply

Dear Alex,
besides greetings for your excellent explanations in this tutorial, can you explain, please, where is stored the value of "value" returned by the function, in the case of returning by value (and it works when the function is called(examp.1)), and where is stored the value of "&value" in the case of returning by address, and it does'n work when the function is called(examp.2). In both cases "value" is a local variable.
Thanks a lot.

[examp. 1: this one works

```
int doubleValue(int x)
{
    int value = x * 2;
    return value ; // A copy of value will be returned here
} // value goes out of scope here
```

examp.2: this one does'n works

```
int* doubleValue(int x)
{
    int value = x * 2;
    return &value; // return nValue by address here
} // value destroyed here
]
```

Alex
February 28, 2016 at 2:29 pm · Reply

Both of these functions work similarly. The difference is that in the top example, the value of variable value is copied to the caller. Because this is a copy, it doesn't matter what happens to local variable value after that point.

In the bottom example, the address of local variable value is returned to the caller. Because local variable value then goes out of scope and is destroyed, this address becomes invalid.

Xhevo
February 29, 2016 at 4:18 am · Reply

Thank you!