2.7 — Forward declarations and definitions

BY ALEX ON JUNE 2ND, 2007 | LAST MODIFIED BY ALEX ON JANUARY 5TH, 2020

Take a look at this seemingly innocent sample program:

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
6      return 0;
7  }
8
9  int add(int x, int y)
10 {
11     return x + y;
12 }
```

You would expect this program to produce the result:

```
The sum of 3 and 4 is: 7
```

But in fact, it doesn't compile at all! Visual Studio produces the following compile error:

```
add.cpp(5) : error C3861: 'add': identifier not found
```

The reason this program doesn't compile is because the compiler compiles the contents of code files sequentially. When the compiler reaches the function call to *add* on line 5 of *main*, it doesn't know what *add* is, because we haven't defined *add* until line 9! That produces the error, *identifier not found*.

Older versions of Visual Studio would produce an additional error:

```
add.cpp(9) : error C2365: 'add' : redefinition; previous definition was 'formerly unknown
```

This is somewhat misleading, given that *add* wasn't ever defined in the first place. Despite this, it's useful to generally note that it is fairly common for a single error to produce many redundant or related errors or warnings.

Best practice

When addressing compile errors in your programs, always resolve the first error produced first and then compile again.

To fix this problem, we need to address the fact that the compiler doesn't know what add is. There are two common ways to address the issue.

Option 1: Reorder the function calls

One way to address the issue is to reorder the function calls so *add* is defined before *main*:

```
1  #include <iostream>
```

```
2
3  int add(int x, int y)
4  {
5      return x + y;
6  }
7
8  int main()
9  {
10     std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
11     return 0;
12 }
```

That way, by the time *main* calls *add*, the compiler will already know what *add* is. Because this is such a simple program, this change is relatively easy to do. However, in a larger program, it can be tedious trying to figure out which functions call which other functions (and in what order) so they can be declared sequentially.

Furthermore, this option is not always possible. Let's say we're writing a program that has two functions *A* and *B*. If function *A* calls function *B*, and function *B* calls function *A*, then there's no way to order the functions in a way that will make the compiler happy. If you define *A* first, the compiler will complain it doesn't know what *B* is. If you define *B* first, the compiler will complain that it doesn't know what *A* is.

Option 2: Use a forward declaration

We can also fix this by using a forward declaration.

A **forward declaration** allows us to tell the compiler about the existence of an identifier *before* actually defining the identifier.

In the case of functions, this allows us to tell the compiler about the existence of a function before we define the function's body. This way, when the compiler encounters a call to the function, it'll understand that we're making a function call, and can check to ensure we're calling the function correctly, even if it doesn't yet know how or where the function is defined.

To write a forward declaration for a function, we use a declaration statement called a **function prototype**. The function prototype consists of the function's return type, name, parameters, but no function body (the curly braces and everything in between them), terminated with a semicolon.

Here's a function prototype for the *add* function:

```
1  int add(int x, int y); // function prototype includes return type, name, parameters, and semico
```

Now, here's our original program that didn't compile, using a function prototype as a forward declaration for function *add*:

```
1  #include <iostream>
2
3  int add(int x, int y); // forward declaration of add() (using a function prototype)
4
5  int main()
6  {
7      std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n'; // this works because we forw
8      return 0;
9  }
10
11 int add(int x, int y) // even though the body of add() isn't defined until here
12 {
13     return x + y;
14 }
```

Now when the compiler reaches the call to *add* in main, it will know what *add* looks like (a function that takes two integer parameters and returns an integer), and it won't complain.

It is worth noting that function prototypes do not need to specify the names of the parameters. In the above code, you can also forward declare your function like this:

```
int add(int, int); // valid function prototype
```

However, we prefer to name our parameters (using the same names as the actual function), because it allows you to understand what the function parameters are just by looking at the prototype. Otherwise, you'll have to locate the function definition.

Best practice

When defining function prototypes, keep the parameter names. You can easily create forward declarations by using copy/paste on your function declaration. Don't forget the semicolon on the end.

Forgetting the function body

New programmers often wonder what happens if they forward declare a function but do not define it.

The answer is: it depends. If a forward declaration is made, but the function is never called, the program will compile and run fine. However, if a forward declaration is made and the function is called, but the program never defines the function, the program will compile okay, but the linker will complain that it can't resolve the function call.

Consider the following program:

```
#include <iostream>

int add(int x, int y); // forward declaration of add() using function prototype

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
    return 0;
}

// note: No definition for function add
```

In this program, we forward declare *add*, and we call *add*, but we never define *add* anywhere. When we try and compile this program, Visual Studio produces the following message:

```
Compiling...
add.cpp
Linking...
add.obj : error LNK2001: unresolved external symbol "int __cdecl add(int,int)" (?add@@YAH
add.exe : fatal error LNK1120: 1 unresolved externals
```

As you can see, the program compiled okay, but it failed at the link stage because *int add(int, int)* was never defined.

Other types of forward declarations

Forward declarations are most often used with functions. However, forward declarations can also be used with other identifiers in C++, such as variables and user-defined types. Variables and user-defined types have a different syntax for forward declaration, so we'll cover these in future lessons.

Declarations vs. definitions

In C++, you'll often hear the words "declaration" and "definition" used, often interchangeably. What do they mean? You now have enough of a framework to understand the difference between the two.

A **definition** actually implements (for functions or types) or instantiates (for variables) the identifier. Here are some examples of definitions:

```
int add(int x, int y) // implements function add()
{
    int z{ x + y }; // instantiates variable z

    return z;
}
```

A definition is needed to satisfy the *linker*. If you use an identifier without providing a definition, the linker will error.

The **one definition rule** (or ODR for short) is a well-known rule in C++. The ODR has three parts:

1. Within a given *file*, a function, object, type, or template can only have one definition.
2. Within a given *program*, an object or normal function can only have one definition. This distinction is made because programs can have more than one file (we'll cover this in the next lesson).
3. Types, templates, and inline functions and variables are allowed to have identical definitions in different files. We haven't covered what most of these things are yet, so don't worry about this for now -- we'll bring it back up when it's relevant.

Violating part 1 of the ODR will cause the compiler to issue a redefinition error. Violating ODR part 2 will likely cause the linker to issue a redefinition error. Violating ODR part 3 will cause undefined behavior.

Here's an example of a violation of part 1:

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

int add(int x, int y) // violation of ODR, we've already defined function add
{
     return x + y;
}

int main()
{
    int x;
    int x; // violation of ODR, we've already defined x
}
```

Because the above program violates ODR part 1, this causes the Visual Studio compiler to issue the following compile errors:

```
project3.cpp(9): error C2084: function 'int add(int,int)' already has a body
project3.cpp(3): note: see previous definition of 'add'
project3.cpp(16): error C2086: 'int x': redefinition
project3.cpp(15): note: see declaration of 'x'
```

For advanced readers

Functions that share an identifier but have different parameters are considered to be distinct functions. We discuss this further in lesson **7.6 -- Function overloading**

A **declaration** is a statement that tells the *compiler* about the existence of an identifier and its type information. Here are some examples of declarations:

```
int add(int x, int y); // tells the compiler about a function named "add" that takes two int pa
int x; // tells the compiler about an integer variable named x
```

A declaration is all that is needed to satisfy the compiler. This is why we can use a forward declaration to tell the compiler about an identifier that isn't actually defined until later.

In C++, all definitions also serve as declarations. This is why *int x* appears in our examples for both definitions and declarations. Since *int x* is a definition, it's a declaration too. In most cases, a definition serves our purposes, as it satisfies both the compiler and linker. We only need to provide an explicit declaration when we want to use an identifier before it has been defined.

While it is true that all definitions are declarations, the converse is not true: all declarations are not definitions. An example of this is the function prototype -- it satisfies the compiler, but not the linker. These declarations that aren't definitions are called **pure declarations**. Other types of pure declarations include forward declarations for variables and type declarations (you will encounter these in future lessons, no need to worry about them now).

The ODR doesn't apply to pure declarations (it's the *one definition rule*, not the *one declaration rule*), so you can have as many pure declarations for an identifier as you desire (although having more than one is redundant).

Author's note

In common language, the term "declaration" is typically used to mean "a pure declaration", and "definition" is used to mean "a definition that also serves as a declaration". Thus, we'd typically call *int x;* a definition, even though it is both a definition and a declaration.

Quiz time

Question #1

What is a function prototype?

Show Solution

Question #2

What is a forward declaration?

Show Solution

Question #3

How do we declare a forward declaration for functions?

Show Solution

Question #4

Write the function prototype for this function (use the preferred form with names):

```
int doMath(int first, int second, int third, int fourth)
{
	return first + second * third / fourth;
}
```

Show Solution

Question #5

For each of the following programs, state whether they fail to compile, fail to link, or compile and link. If you are not sure, try compiling them!

a)

```
#include <iostream>
int add(int x, int y);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << '\n';
    return 0;
}

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

Show Solution

b)

```
#include <iostream>
int add(int x, int y);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << '\n';
    return 0;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

Show Solution

c)

```
#include <iostream>
int add(int x, int y);

int main()
```

```
{
    std::cout << "3 + 4 + 5 = " << add(3, 4) << '\n';
    return 0;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

Show Solution

d)

```
#include <iostream>
int add(int x, int y, int z);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << '\n';
    return 0;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

Show Solution

2.8 -- Programs with multiple code files

Index

2.6 -- Whitespace and basic formatting

C++ TUTORIAL | PRINT THIS POST

264 comments to 2.7 — Forward declarations and definitions

« Older Comments 1 2 3 4

Lord Voldemort
January 17, 2020 at 12:01 pm · Reply

Don't you think it is redundant to have identical definition in different files as mentioned in ODR.

Alex
January 21, 2020 at 1:36 pm · Reply

Identical definitions of what?

Remember that the compiler compiles file-by-file. This means each file must include all of the declarations and definitions required for that file to compile. In some cases, (e.g. for variables) declarations are enough to use them in multiple files. But for other things, full definitions are needed. For example, with program-defined types (types that you've created for use in your own programs, e.g. enums, structs, classes, etc...), it would be impossible to use those types in more than one file if type definitions were limited to a single file.

arntj
January 2, 2020 at 7:29 am · Reply

Seems like there is an unclosed <i> tag on this page. Even this comment is in italic...

nascardriver
January 2, 2020 at 7:33 am · Reply

I misspelled em...
Thanks! I can't believe I didn't notice it :D

arntj

January 2, 2020 at 7:34 am · Reply

No problem :)

koe

December 8, 2019 at 3:35 pm · Reply

I think this chapter could use some added clarity, since multiple functions with the same identifier but different parameters can work.

```
#include <iostream>

int add(int x, int y);
int add(int x, int y, int z);
void print_int(int an_int);

int main()
{
    int a{1};
    int b{2};
    int c{add(a, b)};

    print_int(add(a,b));
    print_int(add(a,b,c));

    return 0;
}

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

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

//prints an integer to console
void print_int(int an_int)
{
    std::cout << an_int << '\n';
}
```

nascardriver

December 9, 2019 at 4:43 am · Reply

Hi,

could you explain where exactly you think this chapter needs clarification?
Function overloads (Multiple functions with the same name) are covered later.

koe

December 9, 2019 at 2:32 pm · Reply

"A definition actually implements (for functions or types) or instantiates (for variables) the identifier."

This isn't entirely accurate, since a function identifier is sub-related to its parameters. If just the identifier is relevant to a definition the ODR would seem to preclude function overloads.

Possible improvement:

"A definition actually implements (for functions or types) or instantiates (for variables) an identifier with its related details (type, parameters, etc.)."

nascardriver
December 11, 2019 at 4:10 am · Reply

Wording updated to mention function overloading.

Humam
December 5, 2019 at 12:42 pm · Reply

How would you say that int x; is a definition?

int x; has not been initialized, if I did std::cout << x; it will give me a compiler error because we are using uninitialized memory x (and using uninitialized local variable x)

Wouldn't the definition of x make more sense if it was initialized (Given a value) at the same time? Instead I'm looking at int x; more as a forward declaration for a variable, even though you stated that forward declarations of variables require a different syntax (Supposedly not the same as int x;)

I hope you understand my point, this downright confused me right now, maybe I need missed some simple point, clarification would be great, thanks.

nascardriver
December 6, 2019 at 3:04 am · Reply

`int x;` is a definition. `x` exists, it has a location in memory, the value doesn't matter. Things can be defined without being initialized.
A declaration only declares that something exists, but it doesn't create anything.

You can think of a declaration as me saying "I have a bucket". You now know that I have a bucket. It might exist, but I might have lied to you.
A definition is me showing you how I create the bucket. You no longer need an explicit declaration, because it's obvious that I have a bucket, you saw it.
In neither of these cases do you know what's inside of my bucket. You only know what's inside if you try to find and use my bucket after I declared it, or you saw how I defined it and I poured something into it (I initialized it). If I didn't fill it with anything, it exists anyway.

Humam
December 6, 2019 at 8:40 am · Reply

Alright so it can be like that with variables!
That makes more sense now, I appreciate your answer, thank you for your help.

BooGDaaN
November 4, 2019 at 1:01 pm · Reply

Hello! I created this code:

```
#include <iostream>

int getValueFromUser();
int getMultiplicatorFromUser();
void printValueMultiplicated(int value);
void printValueMultiplicated(int value, int multiplicator);

int main()
{
    printValueMultiplicated(getValueFromUser());
    std::cout << '\n';
    printValueMultiplicated(getValueFromUser(), getMultiplicatorFromUser());

    std::cout << '\n';
    return 0;
}

int getValueFromUser()
{
    std::cout << "Enter integer value: ";
    int input{};
    std::cin >> input;
    return input;
}

int getMultiplicatorFromUser()
{
    std::cout << "Enter multiplicator value: ";
    int input{};
    std::cin >> input;
    return input;
}

void printValueMultiplicated(int value)
{
    std::cout << value << " multiplicated with 2 is " << value * 2 << '\n';
}

void printValueMultiplicated(int value, int multiplicator)
{
    std::cout << value << " multiplicated with " << multiplicator << " is " << value * mult
}
```

I don't understand why when i call function:

```
printValueMultiplicated(getValueFromUser(), getMultiplicatorFromUser())
```

In console firstly appears to introduce the multiplicator value, not the integer value.

5irKawfycups
November 4, 2019 at 5:49 pm · Reply

See this from Lesson 2.3:

"A warning about function argument order of evaluation

The C++ specification does not define whether arguments are matched with parameters in left to right order or right to left order. When copying values, is of no consequence. However, if the arguments are function calls, then this can be problematic"

In your case, I would separate out the function to take the integer input from user and the function to take the multiplicator from user.

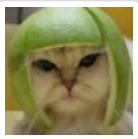
```
int main()
{
    int value{ getValueFromUser() };
    int multiplicator{ getMultiplicatorFromUser() };
    printValueMultiplicated(value, multiplicator);

    std::cout << '\n';
    return 0;
}
```

Jonas
June 11, 2019 at 3:36 am · Reply

Found an error under the part about the 'one definition rule' (error marked with asterisks):

Violating part 1 of the ODR will cause *a compile* to issue a redefinition error. Violating ODR parts 2 or 3 will cause the linker to issue a redefinition error.

I assume this should say 'the compiler' instead of 'a compile'.



Alex
June 14, 2019 at 10:48 am · Reply

Fixed. Thanks!

Nikola
May 15, 2019 at 10:32 pm · Reply

Hello Alex,

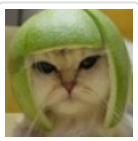
Just a side note. In the section "Declaration vs. definition", line three (of the text) is written as "A definition is needed to satisfy the linker. If you use an identifier without providing a definition, the linker will error."
I know what you wanted to say but the ending of the second sentence is wrong, I think - ", the linker will throw an (perhaps) error."

Regards.

alfonso
May 5, 2019 at 12:34 am · Reply

b) Doesn’t compile. The compiler will complain that the add() called in main() does not have the same number of parameters as the one that was forward declared.

Is what will say true?
The compiler gives us more information telling we have a declared function with the same identifier but less parameters. But in fact, the program won't compile because the add function used in main (with three parameters) was not forward declared. Because we can have functions with the same identifier but different types or number of parameters. add function with three parameters is not the same function as add function with two parameters.



Alex
May 6, 2019 at 3:15 pm · Reply

Correct, so the compiler will complain that the user is trying to call an add() function with 3 parameters but it can't find a declaration for such a function.

alfonso
May 4, 2019 at 1:59 am · Reply

Which is the difference between an instantiated variable and an initialized variable?

Alex
May 6, 2019 at 2:00 pm · Reply

Instantiated = memory was allocated for the object.
Initialized = an initial value for the object was provided.

Eva
April 9, 2019 at 2:25 am · Reply

Hey Alex,

why is it necessary to state the type of the function again if we are defining the function body, when we have already given the type in the forward declaration?
Since I cannot define two functions with the same name but different types anyway this seems unnecessary. But the compiler complains if I do this saying that I can have no declaration without a type, which would be fine...but I thought we declared the function in the line above main() and are now simply giving the definition...
Also, if I would deal with variables and I defined variable int x without initialising it and wanted to initialise it later on I would simply say x = 5
without explicitly providing the type of the variable, because it is already known that x is an integer. Why does this not work for functions? I think I am missing something here...

Alex
April 9, 2019 at 11:46 am · Reply

What do you mean by "type of the function"?

The purpose of the forward declaration is to provide the compiler with a way to pre-check whether a call to the function is typed correctly before seeing the actual definition of the function.

It's not redundant to include return type and parameter information in both the forward declaration and the definition. When the compiler compiles, it does so file by file. The function definition may serve as the declaration in the file the function is defined in, whereas in another file, the forward declaration is used. Omitting information from either would not give the compiler enough information to do type checking.

I think this will make more sense once you've gotten through the lessons on multiple files.

Eva
April 10, 2019 at 3:50 am · Reply

Thank you so much for your reply!
I have gone through the lessons on multiple files, but it still, kind of, does not make sense to me...maybe because I am missing an example where things could go wrong if it were any different.

I do get that if I have my main.cpp program (with main()) and I am using a function, say int add(int x, int y), that is defined in an add.cpp and has a forward declaration in main.cpp I have to provide the return type of the function both in main.cpp and add.cpp because both files will be compiled

separately and the compiler needs to know the return type of add(). (The definition of add() in add.cpp serves also as the declaration of the function in this file.) That is completely logical to me.

What I do not understand is the following case:
Let's say I write a header file add.h where I declare function int add(int x, int y). I include add.h in the add.cpp and main.cpp (where I have now deleted the forward declaration of add()). Now, when compiling add.cpp the compiler will first see the declaration of add() from the header file add.h and then the definition of add() in the add.cpp. In the declaration I have already stated that the return type of add() is integer. I cannot define another function with the same identifier but a different return type (in this file) because the compiler would throw an error.
So why do I need to provide the return type for add() in the definition in this case? Is this 'just' a fail safe? Or why can't the compiler match the function add() from the add.cpp with the one in add.h?

Alex
April 11, 2019 at 8:41 am · Reply

I'm not sure there's any particular reason. But a couple of speculations:
1) Definitions act as declarations, and it would be weird to have a definition that was missing information
2) Having each declaration be a full declaration helps prevent code from breaking when modified (e.g. if you no longer included that header)

blazk
March 6, 2019 at 1:14 am · Reply

```
1  int add(int x, int y);
2
3  int main()
4  {
5      std::cout << "3 + 4 + 5 = " << add(3, 4) << '\n';
6      return 0;
7  }
8
9  int add(int x, int y, int z)
10 {
11     return x + y + z;
12 }
13
14 int add(int x, int y)
15 {
16     return x + y;
17 }
```

this code has two definition with same name but different arity. It compiles fine.
why?
is two function with same name but different arity considered as different function?

and I tried using both function

```
1  std::cout << "3 + 4 + 5 = " << add(3, 4) << add(3, 4, 5) << '\n';
```

and it runs fine as expected. so it seems like same name function with different arity is treated like different function

nascardriver

March 6, 2019 at 7:16 am · Reply

Hi!

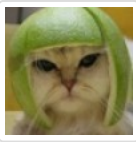
They are different functions. This is called overloading. It's covered in lesson 7.6.

Juan
March 4, 2019 at 3:56 pm · Reply

Thank you very much for explaining the difference in an understandable way. I looked for it in Stack Overflow but those who "explained" gave few truths and a lot of theater. Thanks!

Levi
February 19, 2019 at 9:56 am · Reply

"New programmers often wonder what happens if forward declare a function but do not define it." idk if this is complete english haha



Alex
February 19, 2019 at 8:35 pm · Reply

Added a missing "they" so the sentence reads.

Ethan Smith
January 5, 2019 at 9:46 pm · Reply

What is the standard for placing in Forward declarations, should they be outside of any functions and at the start of the code outside of functions entirely?

nascardriver
January 6, 2019 at 6:28 am · Reply

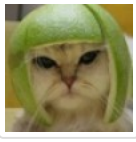
Forward declarations are usually in a header file (Lesson 1.9), ie. outside of functions.

Derick
November 7, 2018 at 1:19 pm · Reply

I find the section the one definition rule a bit confusing.

The first part states: "Within a given file, an identifier can only have one definition.", but I can have two variables with the same name in the same file, as long as they're in different scopes.

I guess the question is, what constitutes an identifier?



Alex
November 10, 2018 at 8:42 am · Reply

I updated the section on the ODR to make it more accurate, as you are correct -- it's not the identifier that must be unique, it's the underlying things themselves (objects, functions, types, etc...) that must be unique.

An identifier is just a name for something, and as you correctly note, identifiers don't need to be unique so long as they're in different scopes.

Ove
November 6, 2018 at 6:15 am · Reply

Hello!
In the section Function prototypes and forward declaration of functions you write this tip:
Tip: You can easily create function prototypes by using copy/paste on your function declaration. Don't forget the semicolon on the end.

My understanding is that you meant to write the following:
Tip: You can easily create function prototypes by using copy/paste on your function definition. Don't forget the semicolon on the end.

Is this correct or have I misunderstood something?
Thanks for these awesome tutorials!

nascardriver
November 6, 2018 at 6:53 am · Reply

The declaration is the top of the function

```
1 void myFunction(int i, float fl) // <- This part
2 {
3
4 }
```

The definition includes the function body (the part inside the curly brackets)

Ove
November 8, 2018 at 12:43 am · Reply

Aha! Ok, thanks.
My understanding was that the function prototype and forward declaration was the same thing and that the actual function is the definition.

Have I understood it rightly that there are two kinds of declarations?
Forward declaration where you have the function prototype, and function declaration where you have the definition.

nascardriver
November 8, 2018 at 6:04 am · Reply

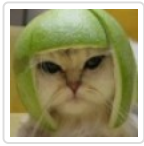
> My understanding was that the function prototype and forward declaration was the same thing and that the actual function is the definition.
That's correct.

> Have I understood it rightly that there are two kinds of declarations?
You can either have the declaration alone (forward declaration/prototype) or the declaration along with the definition. I wouldn't call that two kinds of declaration since it's the same thing. Only what comes after the declaration is different (semicolon or body).

Aditya
October 2, 2018 at 12:53 am · Reply

Hey Alex!
Thanks for the correction.
Now, I want this code to calculate in decimals. That is,if i say 35/45,i want to get the answer in decimals.
How do i do that?

```
#include <iostream>
int add(int x, int y)
{
    return x + y;
}
int subtract(int x, int y)
{
    return x - y;
}
int multiply(int x, int y)
{
    return x * y;
}
int divide(int x, int y)
{
    return x / y;
}
int main()
{
    int x;
    int y;
    int z;
    std::cout << "Enter the first number";
    std::cin >> x;
    std::cout << "Enter the second number";
    std::cin >> y;
    std::cout << "Now,If you want to add,Enter 1" << std::endl;
    std::cout << "If you want to subtract,Enter 2" << std::endl;
    std::cout << "If you want to multiply,Enter 3" << std::endl;
    std::cout << "If you want to divide,Enter 4" << std::endl;
    std::cin >> z;
    if (z == 1)
    {
        std::cout<< add(x,y);
    }
    if (z == 2)
    {
        std::cout << subtract(x, y);
    }
    if (z == 3)
    {
        std::cout << multiply(x, y);
    }
    if (z == 4)
    {
        std::cout << divide(x, y);
    }
}
```



Alex
October 2, 2018 at 4:12 pm · Reply

If you use type "double" instead of "int", you can use numbers with decimals (e.g. 1.45). Handling fractions (e.g. 35/45) is a lot more complicated and far beyond the scope of these

introductory lessons.

I do show examples of a Fraction class in chapter 8 or 9.

aditya
September 30, 2018 at 6:37 am · Reply

What is wrong with this code?

```
#include <iostream>
int add(int x, int y)
{
    return x + y;
}
int subtract(int x, int y)
{
    return x - y;
}
int multiply(int x, int y)
{
    return x * y;
}
int divide(int x, int y)
{
    return x / y;
}
int main()
{
    int x;
    int y;
    int z;
    std::cout << "Enter the first number";
    std::cin >> x;
    std::cout << "Enter the second number";
    std::cin >> y;
    std::cout << "Okay,now comes the difficult part.If you want to add,enter 1.";
    std::cout << "If you want to subtract,enter 2.If you want to multiply,enter 3.If you wa
    std::cin >> z;
    if (int z = 1)
    {
        std::cout<<  add(x,y);
    }
    if (int z = 2)
    {
        std::cout<< subtract(x,y);
    }
    if (int z = 3)
    {
        std::cout<< multiply(x,y);
    }
    if (int z = 4)
    {
        std::cout<< divide(x,y);
    }


}
```

nascardriver

September 30, 2018 at 8:10 am · Reply

You're using = instead of ==.
= is used for assignment
== is used for comparison

Diego Sandoval
July 4, 2018 at 7:27 pm · Reply

You say that a function can have only one definition, yet this code compiles and executes correctly. both functions are called (depending on the arguments provided). I didn't understand the rule correctly? right?

```
#include <iostream>

int fn(int, int);
int fn(int*, char);

using namespace std;

int main(){

    int* ptr;
    int num;

    fn(num, num);
    fn(ptr, num);

    return 0;
}

int fn(int, int){
    cout << "called with 2 ints" << endl;
    return 1;
}
int fn(int*, char){
    cout << "called with one pointer to int and one char" << endl;
    return 2;
}
```

nascardriver
July 5, 2018 at 3:55 am · Reply

Hi Diego!
Those are two different functions which share the same name (Function overloading).
The one definition rule is only valid for the same function header (Same return-type, same name, same parameters).

References
* Lesson 7.6 - Function overloading

Nigel Booth
June 29, 2018 at 6:11 am · Reply

Hi, couldn't all the forward definitions just be put together in a header file such as <<forward.h>> and precompiled with a #include call at the top such as

```
#include "forward.h"
```

?

nascardriver
June 29, 2018 at 6:16 am · Reply

Hi Nigel!

Yes, that's how it's usually done, because it allows usage of the functions from any file. Nothing is being compiled though, only the contents will be copied.

Nigel Booth
June 29, 2018 at 6:25 am · Reply

thanks, just getting my head around why we were looking at "global" scoped definitions in the program that just make the listing look untidy. Are headers covered later?

nascardriver
June 29, 2018 at 6:27 am · Reply

Yep, starting with the next lesson.
The code on learncpp is single-file in most cases, because it's easier to understand that way. In your projects, you should use multiple files.

jft
May 2, 2018 at 3:30 am · Reply

```
int x;
```

This is always a definition as a memory location is allocated for x (which is uninitialized).

```
extern int x;
```

This is a declaration as no memory is allocated for x and x here refers to x defined in another translation unit.

Zane
March 19, 2018 at 10:10 pm · Reply

Why does this work?

```
#include "stdafx.h"
#include <iostream>

int add(int x, int y); // forward declaration of add() (using a function prototype)

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl; // this works because
    return 0;
}

int add(int x, int y) // even though the body of add() isn't defined until here
```

```
{
    return x + y;
}
```

I ask, because the definition of add() is after that of the body of main(). If the the program is read/ran sequentially, when would it ever encounter the definition? Shouldn't the definition come before main()? If not, I am missing something. As I understand it, at the return of a value for main() the program stops, never having made it to/read add().

nascardriver
March 20, 2018 at 5:19 am · Reply

Hi Zane!

Your compiler compiles all functions, no matter where they are. To call a function, all your compiler needs to know, is the function declaration, not the definition. After compilation the order of functions doesn't matter, because your computer can call functions above and below the calling function.

Zane
March 20, 2018 at 5:25 am · Reply

Thank you for explaining that. Makes perfect sense, now.

Deepesh Choudhary
March 13, 2018 at 1:03 am · Reply

Okay, so a declaration is used just to satisfy the compiler.
Does that mean that the declaration statement does not get converted to any machine code?
Does that also mean that if main.o is reversed, there won't be any code/statements indicating such declaration?

nascardriver
March 13, 2018 at 4:48 am · Reply

Hi Deepesh!

Yes to both.
The compiler will generate the function at address Y. Your computer isn't restricted to calling functions that have been declared before the current point of execution. It can just from address X to address Y without worries.

Deepesh Choudhary
March 13, 2018 at 5:08 am · Reply

Thanks nascardriver. :)

Pork and Beans
March 1, 2018 at 3:55 am · Reply

Hello Alex!

I hope you are gearing up for a stellar weekend.

I also hope that you do, in fact, want me to point out any mistakes I see.

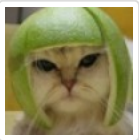
In the paragraph above "Quiz", there is a mistake.

"Others types of pure declarations..." should be "Other".

I thought you would want to know.

Thank you as always for your incredible work.

Have a nice day.

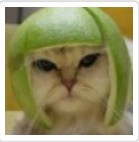


Alex
March 1, 2018 at 8:39 pm · Reply

Indeed. Typo fixed. Thanks for pointing it out!

Farzad.S
February 27, 2018 at 1:54 pm · Reply

I see, by Forward declarations and definitions, we can call the functions in any order we want. Am I right?



Alex
February 27, 2018 at 2:21 pm · Reply

Yes, and you can even call functions that may be defined in other files.

Farzad.S
February 26, 2018 at 8:52 am · Reply

Hello, I did understand the Forward declarations and prototype. But I am curious why do we need it, we can just use function and keep calling it. I believe it makes the program more complicated.

By the way, for ages, I was trying to learn C++ but couldn't find the right book or courses to clearly teaching me from scratch. Great job. thanks a lot

nascardriver
February 27, 2018 at 2:08 am · Reply

Hi Farzad!

> I believe it makes the program more complicated
It sure does, but there are situations where it cannot be avoided.

```
1  void fn2()
2  {
3    // Cannot call fn1, because fn1 hasn't been declared yet.
4    fn1();
5  }
6
7  void fn1()
8  {
9    // fn1 cannot be moved above fn2, because fn1 needs fn2 to be declared already.
10   fn2();
11 }
12
13 int main()
14 {
```

```
15    fn1();
16
17    return 0;
18  }
```

This program will just run an infinite loop, but there are cases in which a construction like this is required.

vbot
September 25, 2018 at 3:09 pm · Reply

My preferred programing-flow is linear from top to bottom, because I'm used to languages where the order of function definitions and calls doesn't matter:

```
1   int main() {
2      doFirstThing();
3      doSecondThing();
4      doThirdThing();
5   }
6
7   void doFirstThing() {
8     doFirstThingA();
9     doFirstThingB();
10    doFirstThingC();
11  }
12
13  void doFirstThingA() {
14  //something
15  }
16
17  void doFirstThingB() {
18  //something
19  }
20
21  void doFirstThingC() {
22  //something
23  }
24
25  void doSecondThing() {
26  //something
27  }
28
29  void doThirdThing() {
30  //something
31  }
```

so this is where I see the need for forward declarations.

« Older Comments 1 2 3 4