Back To Basics
Functions

MIKE SHAH

Cppcon
The C++ Conference

2023

October 01 - 06

Cppcon
The C++ Conference
2023
October 01-06
Aurora, Colorado, USA

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube:
www.youtube.com/c/MikeShah

Back To Basics
Functions

14:00 - 15:00 MDT
Mon, Oct. 2 2023

60 minutes
Introductory Audience

MIKE SHAH

Cppcon
The C++ Conference

20
23
October 01 - 06

Cppcon The C++ Conference | 2023 October 01-06 Aurora, Colorado, USA

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube:
www.youtube.com/c/MikeShah

Back To Basics Functions

14:00 - 15:00 MDT
Mon, Oct. 2 2023

60 minutes
Introductory Audience

If you've been programming C++ for many years -- please provide suggestions, analogies, and other useful ways to think about functions now or in the future!

MIKE SHAH

Cppcon The C++ Conference

2023 October 01 - 06

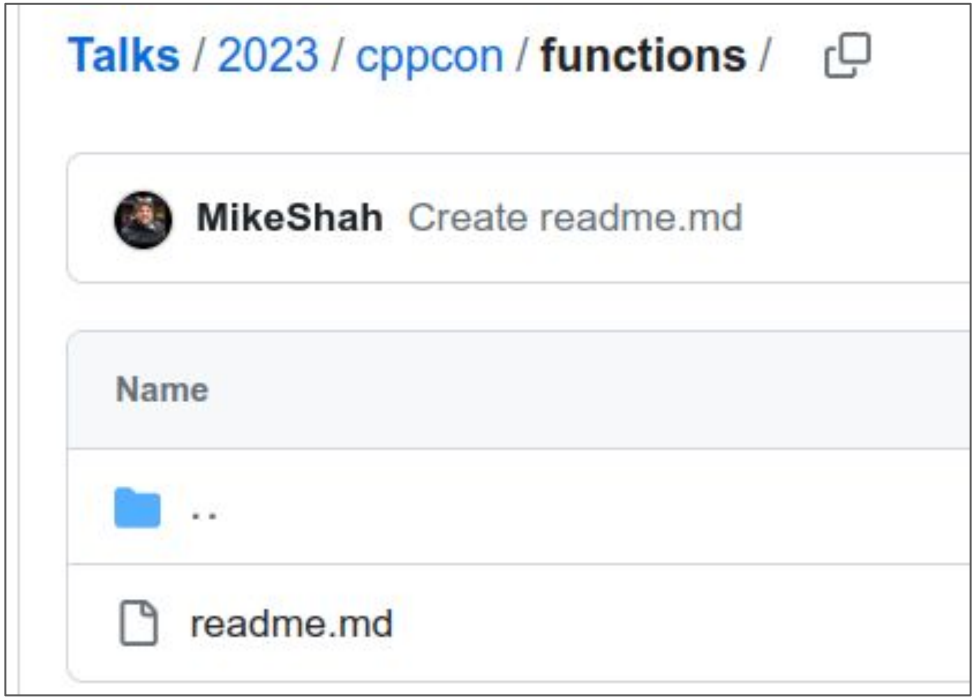
Your Tour Guide for Today

by Mike Shah

- Associate Teaching Professor at Northeastern University in Boston, Massachusetts.
  - I teach courses in computer systems, computer graphics, and game engine development.
  - My research in program analysis is related to performance building static/dynamic analysis and software visualization tools.
- I do consulting and technical training on modern C++, DLang, Concurrency, OpenGL, and Vulkan projects
  - (Usually graphics or games related)
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of computer graphics, visualization, concurrency, and parallelism.
- Contact information and more on: www.mshah.io
- More online training at courses.mshah.io and www.youtube.com/c/MikeShah

Code for the talk

- Located here: https://github.com/MikeShah/Talks/tree/main/2023/cppcon/functions



Talks / 2023 / cppcon / **functions** /

MikeShah Create readme.md

Name

..

readme.md

The abstract that you read and enticed you to join me is here!

Abstract

Functions are one of the first things programmers learn, granting you the ultimate power to 'reuse' code and build modular programs. In this talk, we are going to provide an overview of functions from the start to the end, on the various powers that are given to us from the ground up. Consider this talk your one stop for learning all of the great things about functions!

We'll start with a basic function example, identifying the function signature and basic abilities of a function. Then we are going to view this function again from the perspective of assembly (using compiler explorer) to show you how a function is structured. From the assembly view, we will then observe that functions have addresses (they must after all!) and that we can store functions in pointers. We'll take a brief aside to show you how modern C++ also gives us the convenient std::function. Functions need not always be 'global' building blocks of our programs, the next step in our journey will be to see how we can have functions at local scope (e.g. lambda's) and how they can be used (and oftentimes in handy ways in the STL). Ah, intrigued are you? We're not quite done! Now with building blocks such as lambda's (and related functors) we can utilize function composition to really unlock the power of functions. Towards the end of this talk, we will talk about grouping related functions (into namespaces) and as member functions in classes. Within our discussion of functions in classes, we'll touch on virtual functions, static functions, and operator overloading. We'll circle back to where we began on these topics, again showing you the assembly. At the end of this talk, you will have had FUN with functions (I couldn't resist...but you will see the complete C++ picture of functions).

Goal(s) for Today

Back to Basics: C++ Tour of Functions

- This talk part of the **Back to Basics** track in which we revisit fundamental ideas of programming and C++.
  - Today we will be talking about **Functions**
    - **We'll start from the basics** (what is a function) and **ramp up to more specific C++ usage of functions towards the end.**

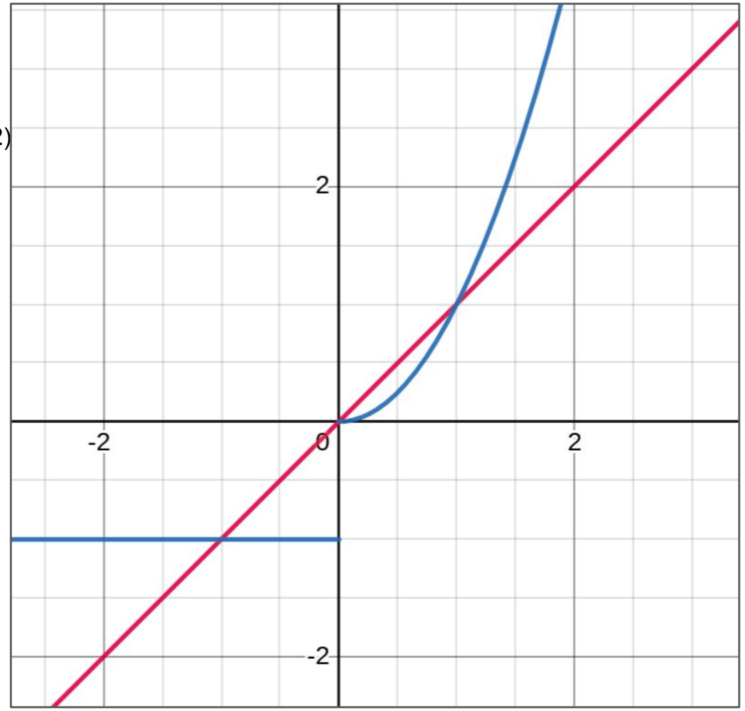
https://images.rawpixel.com/image_800/cHJpdmF0ZS9sci9pbWFnZXMvd2Vic2l0ZS8yMDIyLTA1L2pvYjcxMC0wNTMuanBn.jpg

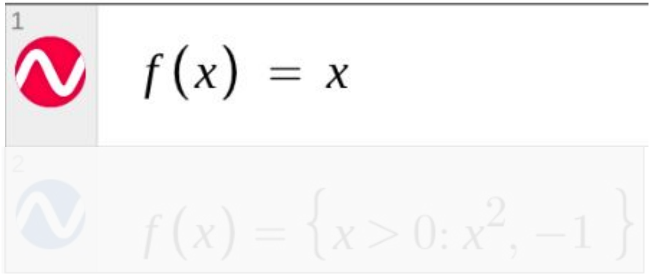
Functions

A familiar term -- perhaps from your math class?

Mathematical Functions (`f(x)=x`) (1/2)

- Functions on graphs are one domain many begin to think of the term **function**.
  - A function takes 0 or more inputs
    - f(x) means ‘x’ is the input representing a real number
  - Based on the inputs, an output is generated (dependent variable)
    - In the case of: f(x) =x we return ‘x’
- **Piecewise functions** get more interesting because we can add conditions
  - Observe `f(x)` is evaluated with x^2 when `x>0`.
  - Otherwise, we generate a value of -1
- In programming languages:
  - We have quite a lot of expressiveness in regards to how we express a functions operations!



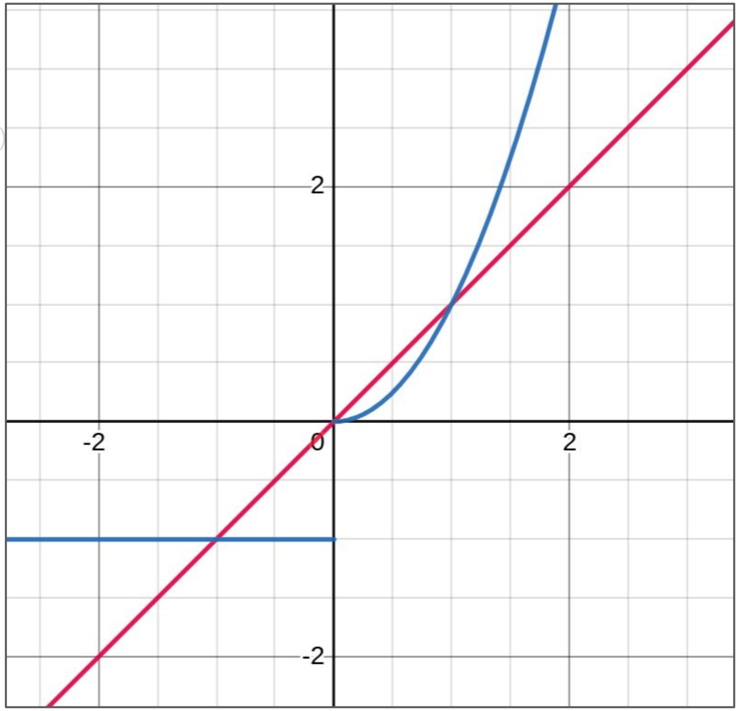


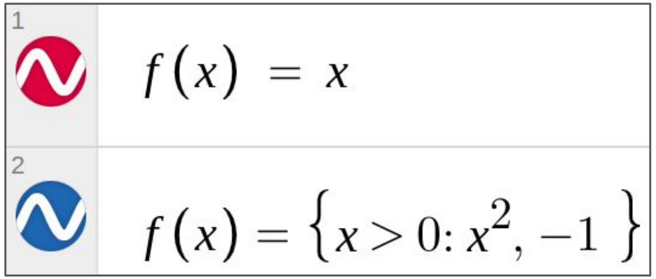
$f(x) = x$

$f(x) = \{x > 0 : x^2, -1\}$

Mathematical Functions (`f(x)=x`) (2/2)

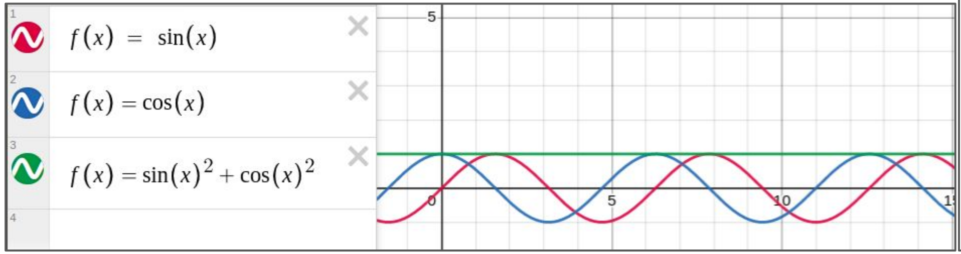
- Functions on graphs are one domain many begin to think of the term **function**.
  - A function takes 0 or more inputs
    - f(x) means 'x' is the input representing a real number
  - Based on the inputs, an output is generated (dependent variable)
    - In the case of: f(x) =x we return 'x'
- **Piecewise functions** get more interesting because we can add conditions
  - Observe `f(x)` is evaluated with x^2 when `x>0`.
  - Otherwise, we generate a value of -1
- In programming languages:
  - We have quite a lot of expressiveness in regards to how we express a functions operations!

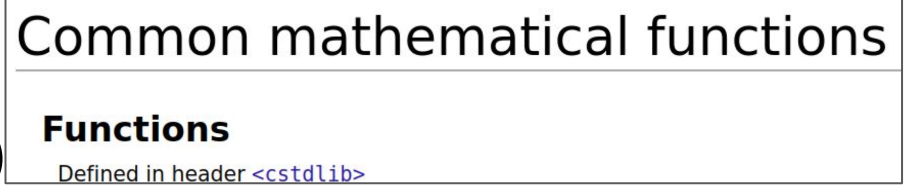



1
$$f(x) = x$$
2
$$f(x) = \{x > 0: x^2, -1\}$$

Common Math Functions (1/2)

- As a start, as you may expect, languages like C++ provide in the standard library many common functions for us.

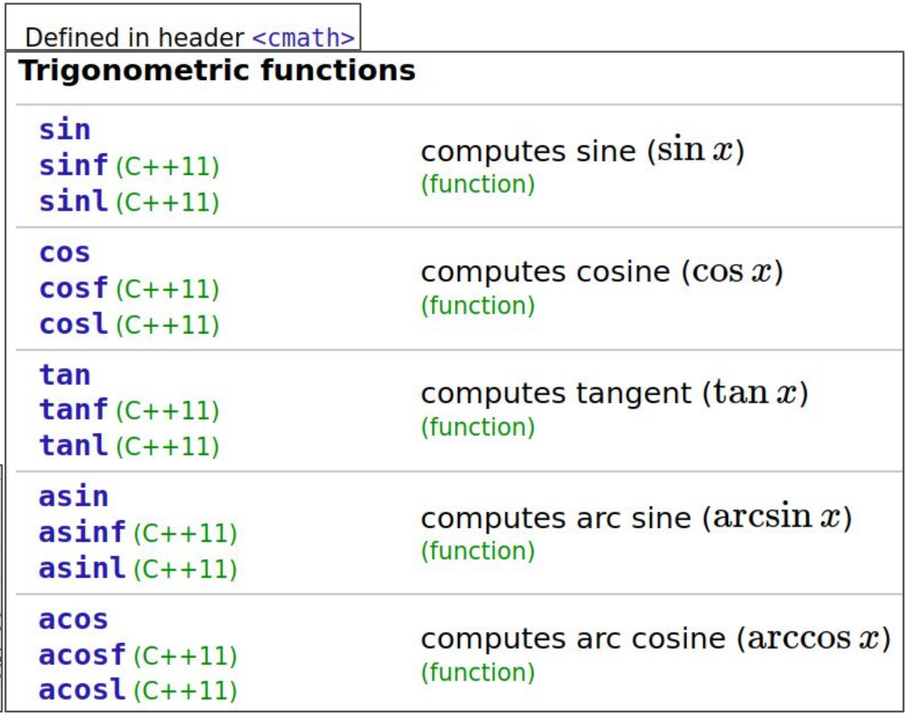




Common mathematical functions

Functions

Defined in header `<cstdlib>`



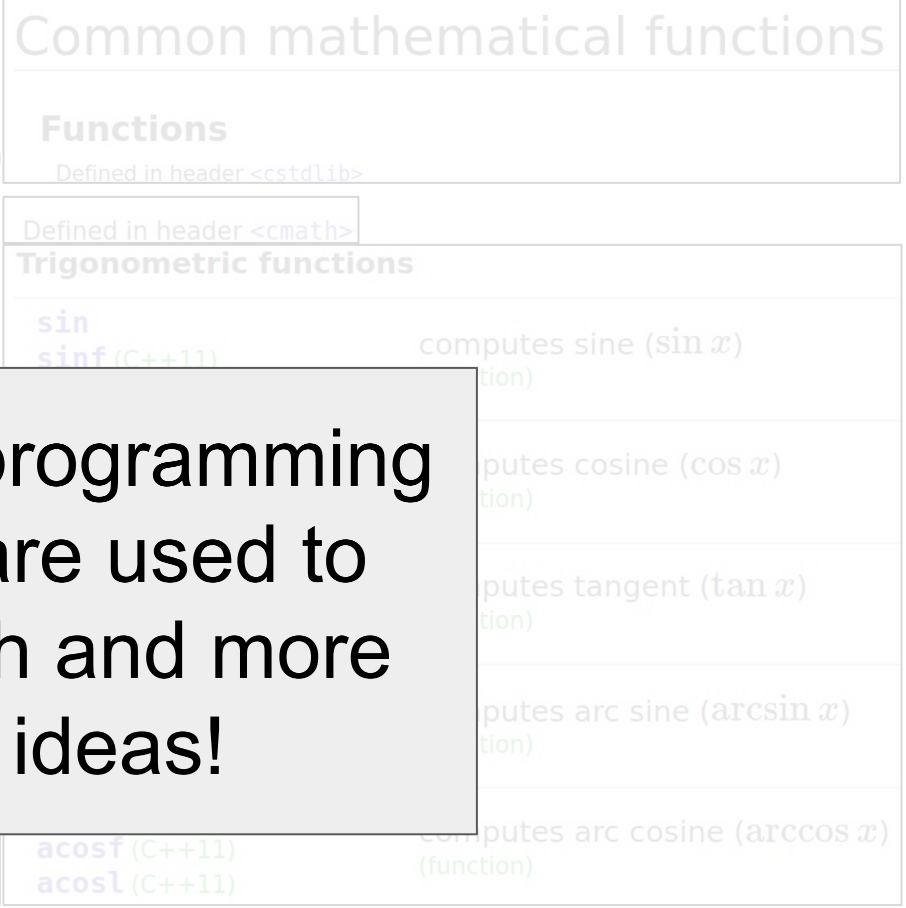
Defined in header `<cmath>`

Trigonometric functions	
`sin` `sinf` (C++11) `sinl` (C++11)	computes sine ($\sin x$) (function)
`cos` `cosf` (C++11) `cosl` (C++11)	computes cosine ($\cos x$) (function)
`tan` `tanf` (C++11) `tanl` (C++11)	computes tangent ($\tan x$) (function)
`asin` `asinf` (C++11) `asinl` (C++11)	computes arc sine ($\arcsin x$) (function)
`acos` `acosf` (C++11) `acosl` (C++11)	computes arc cosine ($\arccos x$) (function)

https://en.cppreference.com/w/cpp/numeric/math

Common Math Functions (2/2)

- As a start, as you may expect, languages like C++ provide in the standard libra… functions for …



Common mathematical functions

Functions

Defined in header `<cstdlib>`

Defined in header `<cmath>`

Trigonometric functions

`sin` `sinf` (C++11)	computes sine ($\sin x$) (function)
	…putes cosine ($\cos x$) (function)
	…putes tangent ($\tan x$) (function)
	…putes arc sine ($\arcsin x$) (function)
`acosf` (C++11) `acosl` (C++11)	…putes arc cosine ($\arccos x$) (function)

Functions in programming languages are used to express math and more exciting ideas!

https://en.cppreference.com/w/cpp/numeric/math

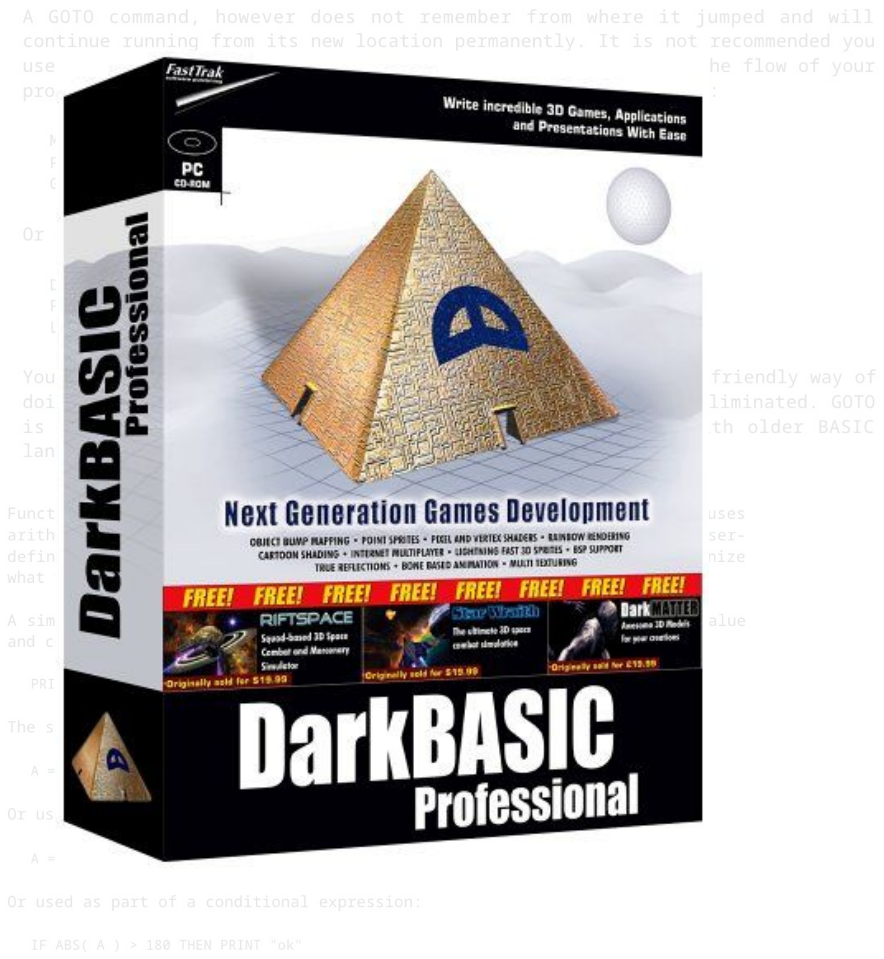
Origin Story: A Journey of Discovery

The magic and power of functions!

https://www.thegamecreators.com/product/dark-basic-pro-open-source

Origin Story:Let’s Give some Credit

- I started my journey in various BASIC programming languages
- **Dark Basic Pro (DBP)** (a game engine and BASIC programming language) was instrumental in my start in getting excited about programming
  - (And later informed my decision to study computer science)
- I can still remember reading the programming manuals that came with my installation CD


FastTrak
PC CD-ROM
Write incredible 3D Games, Applications and Presentations With Ease
DarkBASIC Professional
Next Generation Games Development
OBJECT BUMP MAPPING • POINT SPRITES • PIXEL AND VERTEX SHADERS • RAINBOW RENDERING
CARTOON SHADING • INTERNET MULTIPLAYER • LIGHTNING FAST 3D SPRITES • BSP SUPPORT
TRUE REFLECTIONS • BONE BASED ANIMATION • MULTI TEXTURING
FREE! FREE! FREE! FREE! FREE! FREE! FREE! FREE!
RIFTSPACE Squad-based 3D Space Combat and Mercenary Simulator
Originally sold for $19.99
Star Wraith The ultimate 3D space combat simulation
Originally sold for $19.99
DarkMATTER Awesome 3D Models for your creations
Originally sold for £19.99
DarkBASIC Professional

A GOTO command, however does not remember from where it jumped and will continue running from its new location permanently. It is not recommended you use [illegible] he flow of your pro[illegible]:

Or

You [illegible] friendly way of doi[illegible] liminated. GOTO is [illegible] th older BASIC lan[illegible]

Funct[illegible] uses arith[illegible] ser- defin[illegible] nize what

A sim[illegible] alue and c[illegible]

PRI[illegible]

The s[illegible]

A =

Or us[illegible]

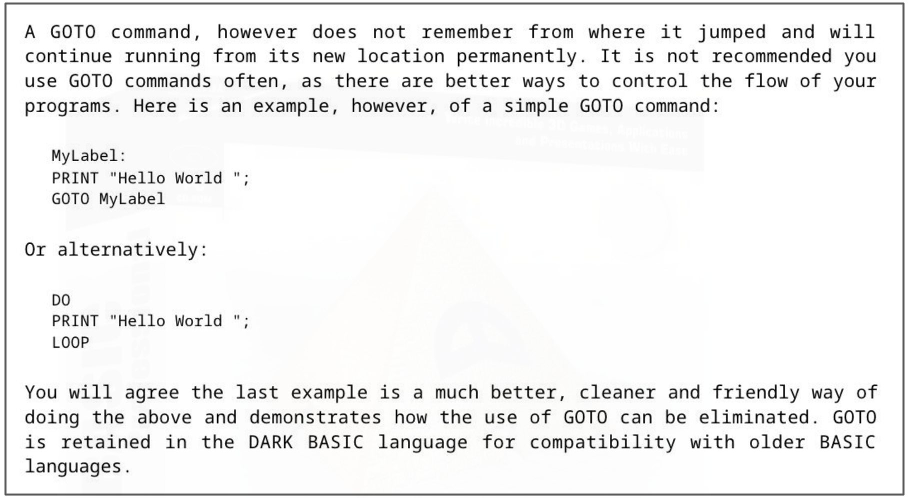
A =

Or used as part of a conditional expression:

```
IF ABS( A ) > 180 THEN PRINT "ok"
```

A Monumental Moment! (1/3)

- I remember learning about **loops**
  - (Using **goto** and **do/loop)**
    - Wow -- I can save myself repeating typing of code!



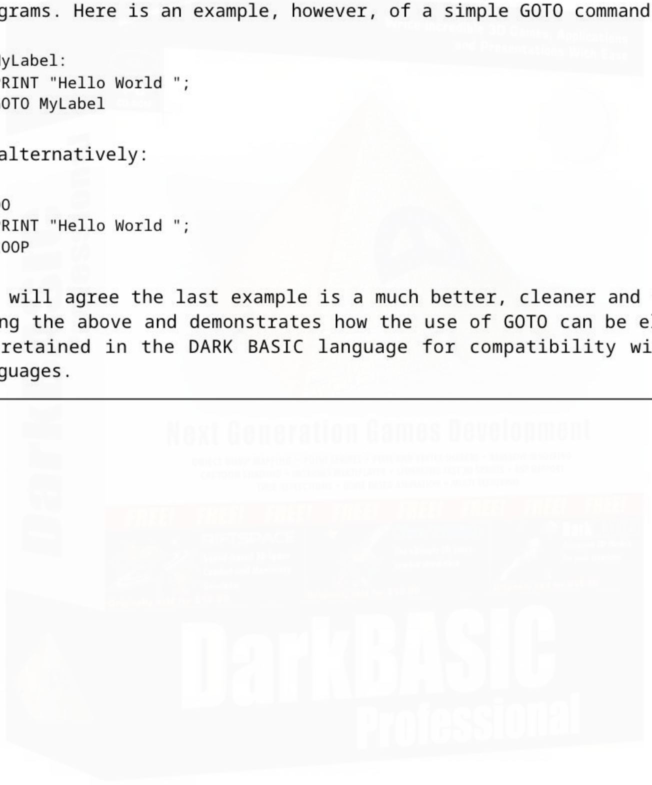
A GOTO command, however does not remember from where it jumped and will continue running from its new location permanently. It is not recommended you use GOTO commands often, as there are better ways to control the flow of your programs. Here is an example, however, of a simple GOTO command:

```
   MyLabel:
   PRINT "Hello World ";
   GOTO MyLabel
```

Or alternatively:

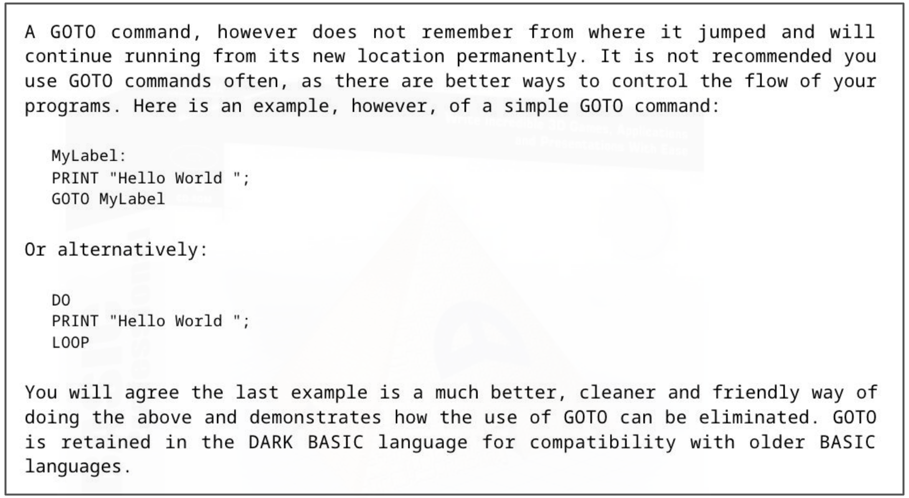
```
   DO
   PRINT "Hello World ";
   LOOP
```

You will agree the last example is a much better, cleaner and friendly way of doing the above and demonstrates how the use of GOTO can be eliminated. GOTO is retained in the DARK BASIC language for compatibility with older BASIC languages.



A Monumental Moment! (2/3)

- I remember learning about **loops**
  - (Using **goto** and **do/loop)**
    - Wow -- I can save myself repeating typing of code!
- I also remember finding a giant list of interesting graphics functions
  - Neat -- **something different than the math functions I'm learning in school**
    - Sounds like I can do some interesting stuff!



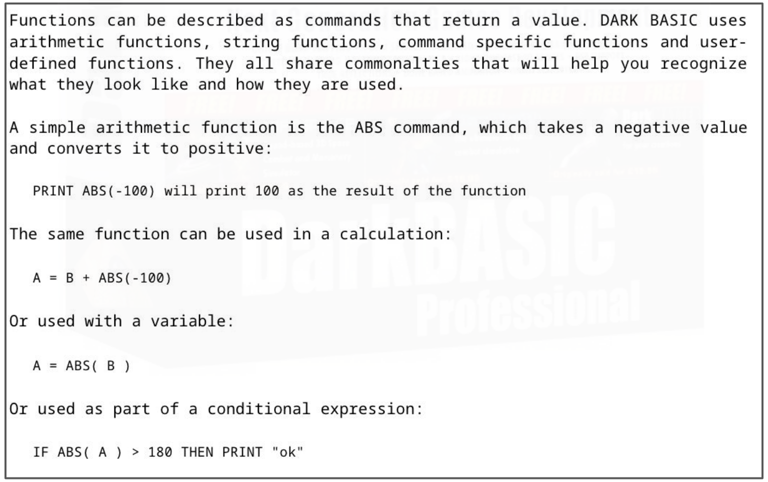
A GOTO command, however does not remember from where it jumped and will continue running from its new location permanently. It is not recommended you use GOTO commands often, as there are better ways to control the flow of your programs. Here is an example, however, of a simple GOTO command:

```
   MyLabel:
   PRINT "Hello World ";
   GOTO MyLabel
```

Or alternatively:

```
   DO
   PRINT "Hello World ";
   LOOP
```

You will agree the last example is a much better, cleaner and friendly way of doing the above and demonstrates how the use of GOTO can be eliminated. GOTO is retained in the DARK BASIC language for compatibility with older BASIC languages.



Functions can be described as commands that return a value. DARK BASIC uses arithmetic functions, string functions, command specific functions and user-defined functions. They all share commonalties that will help you recognize what they look like and how they are used.

A simple arithmetic function is the ABS command, which takes a negative value and converts it to positive:

```
  PRINT ABS(-100) will print 100 as the result of the function
```

The same function can be used in a calculation:

```
  A = B + ABS(-100)
```

Or used with a variable:

```
  A = ABS( B )
```

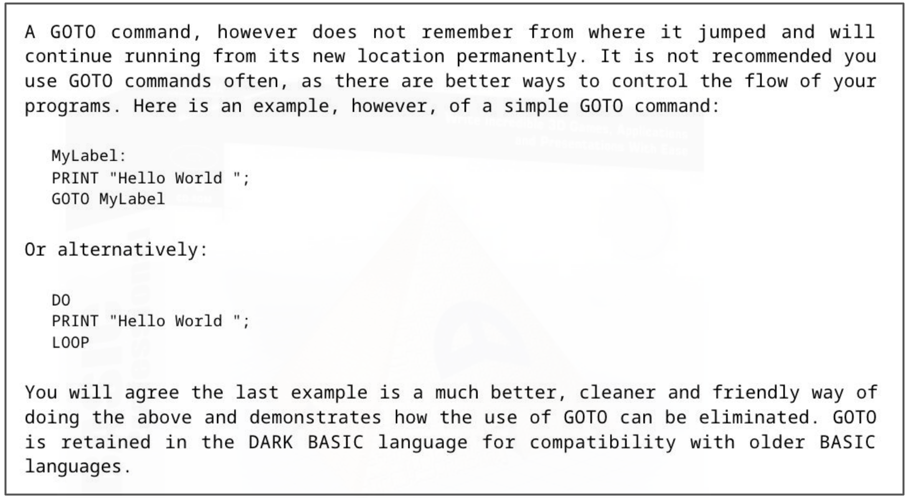
Or used as part of a conditional expression:

```
  IF ABS( A ) > 180 THEN PRINT "ok"
```

A Monumental Moment! (3/3)

- I remember learning about **loops**
  - (Using **goto** and **do/loop)**
    - Wow -- I can save myself repeating typing of code!
- I also remember finding a giant list of interesting graphics functions
  - Neat -- **something different than the math functions I'm learning in school**
    - Sounds like I can do some interesting stuff!

Note: Loops and functions are helping achieve code reuse, slightly different abstractions, but both serving as building blocks to implement algorithms.



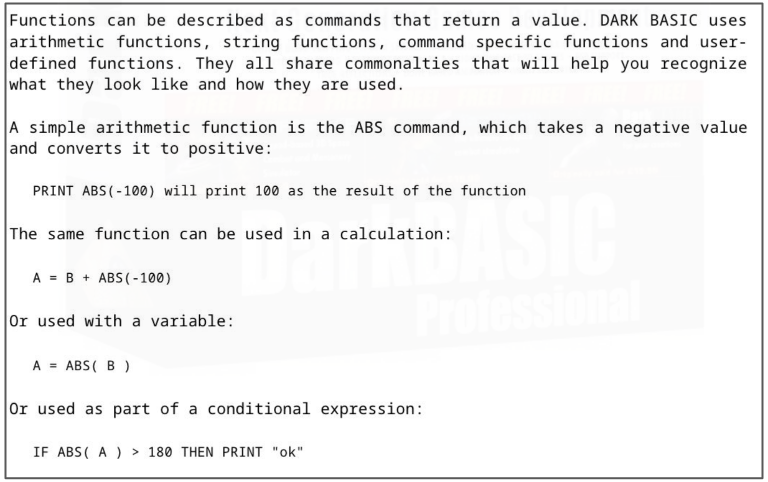
A GOTO command, however does not remember from where it jumped and will continue running from its new location permanently. It is not recommended you use GOTO commands often, as there are better ways to control the flow of your programs. Here is an example, however, of a simple GOTO command:

```
   MyLabel:
   PRINT "Hello World ";
   GOTO MyLabel
```

Or alternatively:

```
   DO
   PRINT "Hello World ";
   LOOP
```

You will agree the last example is a much better, cleaner and friendly way of doing the above and demonstrates how the use of GOTO can be eliminated. GOTO is retained in the DARK BASIC language for compatibility with older BASIC languages.



Functions can be described as commands that return a value. DARK BASIC uses arithmetic functions, string functions, command specific functions and user-defined functions. They all share commonalties that will help you recognize what they look like and how they are used.

A simple arithmetic function is the ABS command, which takes a negative value and converts it to positive:

```
  PRINT ABS(-100) will print 100 as the result of the function
```

The same function can be used in a calculation:

```
  A = B + ABS(-100)
```

Or used with a variable:

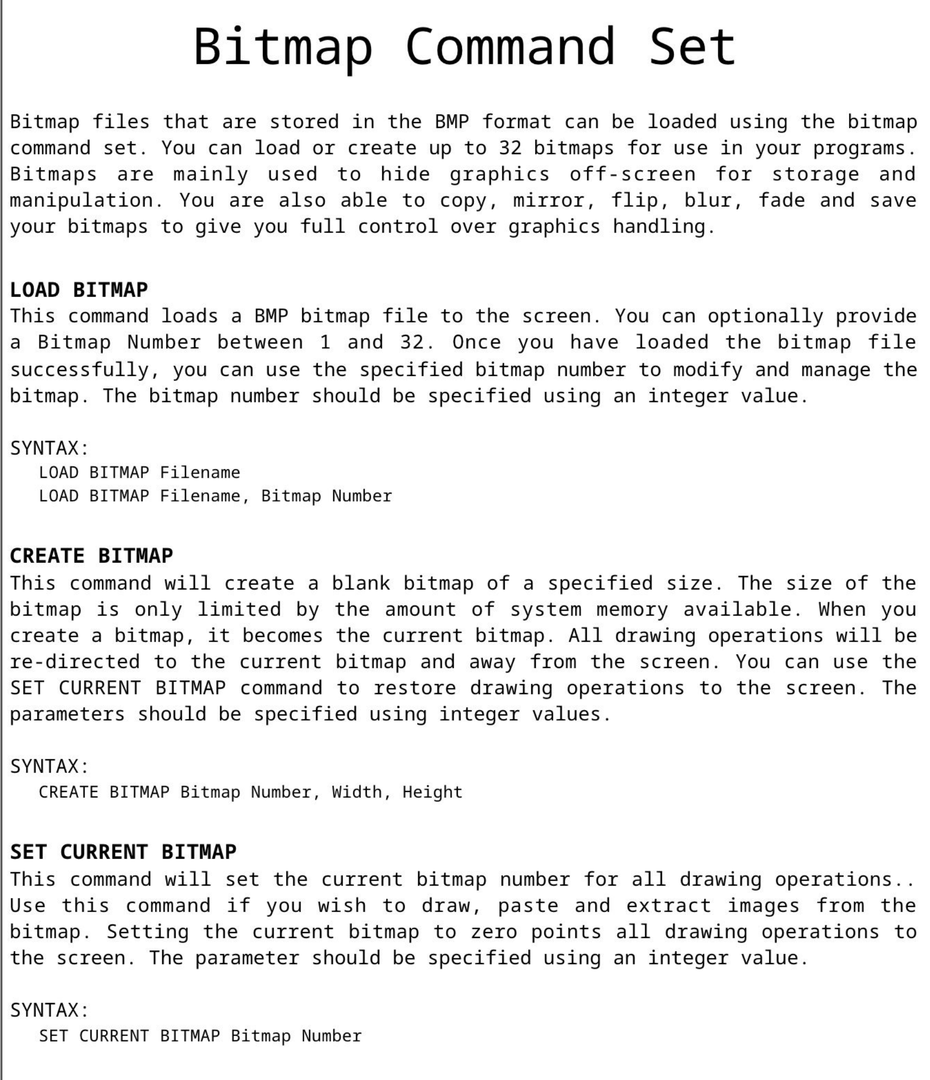
```
  A = ABS( B )
```

Or used as part of a conditional expression:

```
  IF ABS( A ) > 180 THEN PRINT "ok"
```

Example Bitmap Functions

- All of these functions are related to operating on ‘bitmap images’ -- they are a provided **common set of functions**
    - **Load Bitmap** Filename
    - **Load Bitmap** Filename, Bitmap Number
    - **Set Current Bitmap** Bitmap Number
        - etc.



Bitmap Command Set

Bitmap files that are stored in the BMP format can be loaded using the bitmap command set. You can load or create up to 32 bitmaps for use in your programs. Bitmaps are mainly used to hide graphics off-screen for storage and manipulation. You are also able to copy, mirror, flip, blur, fade and save your bitmaps to give you full control over graphics handling.

LOAD BITMAP
This command loads a BMP bitmap file to the screen. You can optionally provide a Bitmap Number between 1 and 32. Once you have loaded the bitmap file successfully, you can use the specified bitmap number to modify and manage the bitmap. The bitmap number should be specified using an integer value.

SYNTAX:
```
   LOAD BITMAP Filename
   LOAD BITMAP Filename, Bitmap Number
```

CREATE BITMAP
This command will create a blank bitmap of a specified size. The size of the bitmap is only limited by the amount of system memory available. When you create a bitmap, it becomes the current bitmap. All drawing operations will be re-directed to the current bitmap and away from the screen. You can use the SET CURRENT BITMAP command to restore drawing operations to the screen. The parameters should be specified using integer values.

SYNTAX:
```
   CREATE BITMAP Bitmap Number, Width, Height
```

SET CURRENT BITMAP
This command will set the current bitmap number for all drawing operations.. Use this command if you wish to draw, paste and extract images from the bitmap. Setting the current bitmap to zero points all drawing operations to the screen. The parameter should be specified using an integer value.

SYNTAX:
```
   SET CURRENT BITMAP Bitmap Number
```

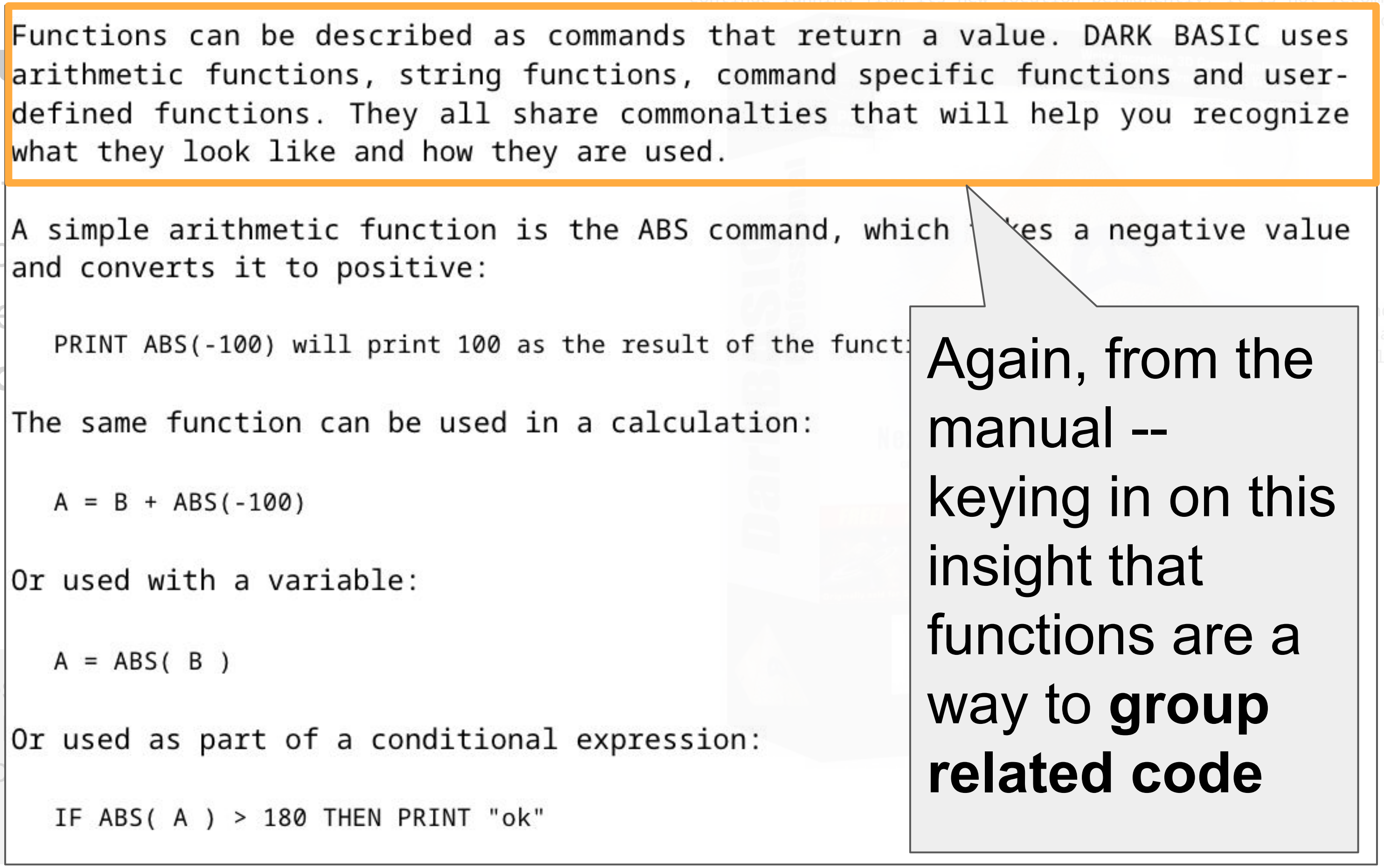
A GOTO command, however does not remember from where it jumped and will continue running from its new location permanently. It is not recommended you
ow of your

A Mon

- All o
to op
-- the
set c

dly way of
ated. GOTO
lder BASIC

- Loop
both
comp



Functions can be described as commands that return a value. DARK BASIC uses arithmetic functions, string functions, command specific functions and user-defined functions. They all share commonalties that will help you recognize what they look like and how they are used.

A simple arithmetic function is the ABS command, which takes a negative value and converts it to positive:

```
   PRINT ABS(-100) will print 100 as the result of the funct
```

The same function can be used in a calculation:

```
   A = B + ABS(-100)
```

Or used with a variable:

```
   A = ABS( B )
```

Or used as part of a conditional expression:

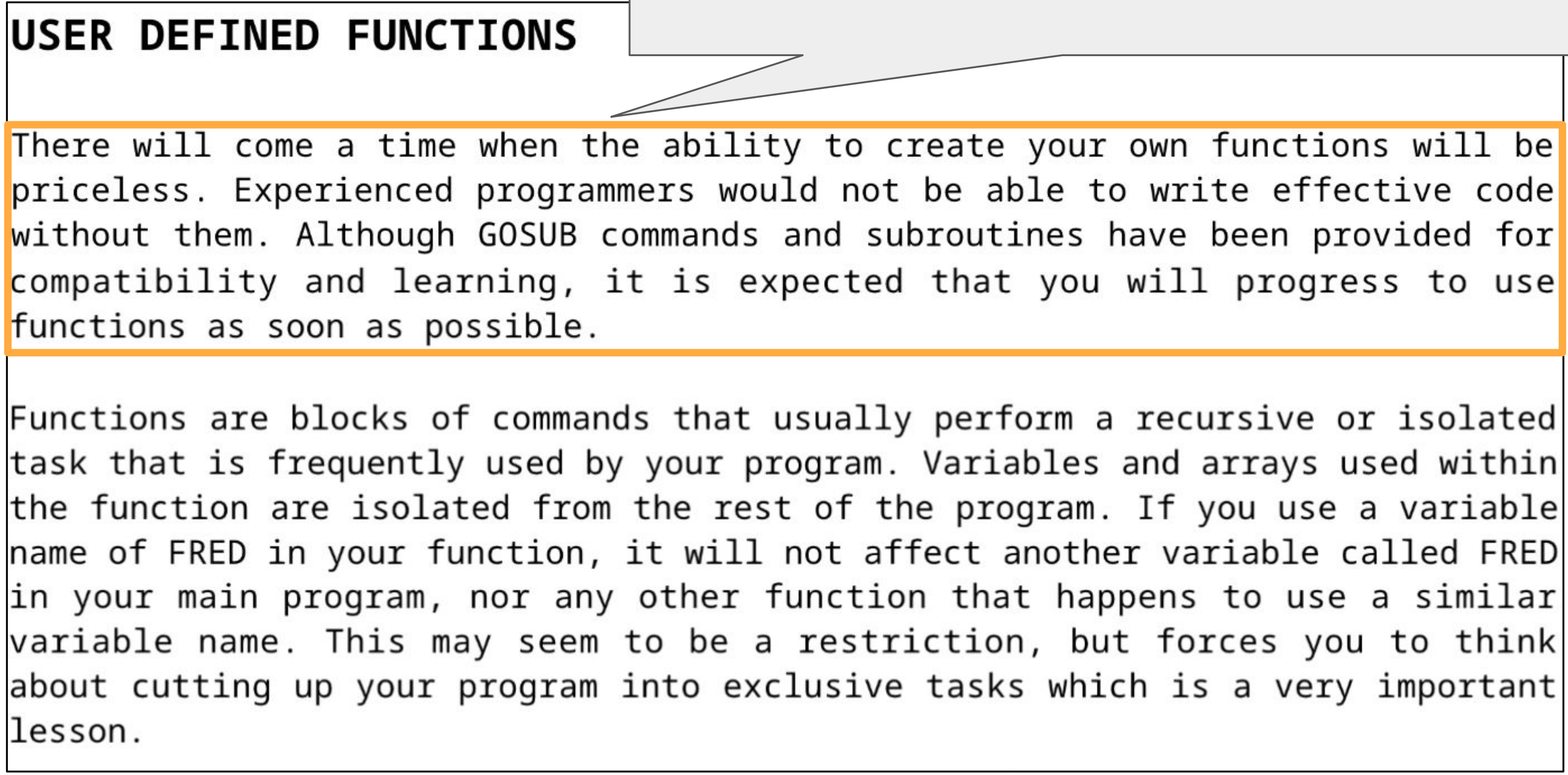
```
   IF ABS( A ) > 180 THEN PRINT "ok"
```

Again, from the manual -- keying in on this insight that functions are a way to **group related code**

More Insights

Wow! I can write my own functions?

The limit is our imagination (and perhaps RAM :))



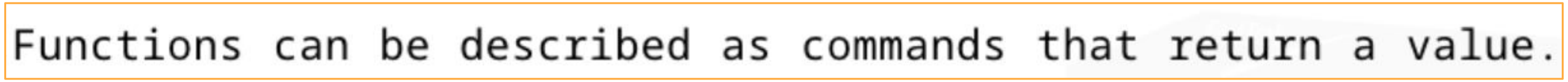
USER DEFINED FUNCTIONS

There will come a time when the ability to create your own functions will be priceless. Experienced programmers would not be able to write effective code without them. Although GOSUB commands and subroutines have been provided for compatibility and learning, it is expected that you will progress to use functions as soon as possible.

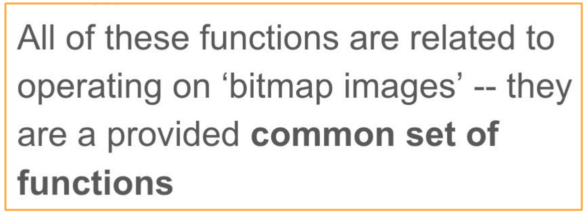
Functions are blocks of commands that usually perform a recursive or isolated task that is frequently used by your program. Variables and arrays used within the function are isolated from the rest of the program. If you use a variable name of FRED in your function, it will not affect another variable called FRED in your main program, nor any other function that happens to use a similar variable name. This may seem to be a restriction, but forces you to think about cutting up your program into exclusive tasks which is a very important lesson.

Capturing a Few Fundamental Ideas From my Origin Story

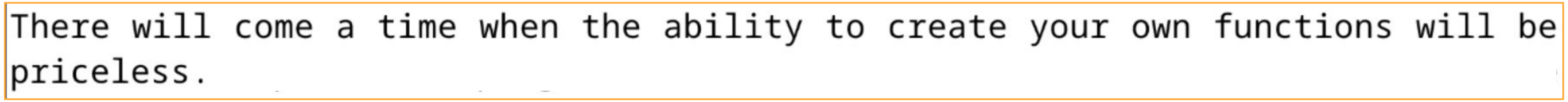
- From my initial discovery of functions -- I found a few interesting facts about functions and how to think about them.
    - Let’s explore further!



```
Functions can be described as commands that return a value.
```



All of these functions are related to operating on ‘bitmap images’ -- they are a provided **common set of functions**



```
There will come a time when the ability to create your own functions will be
priceless.
```

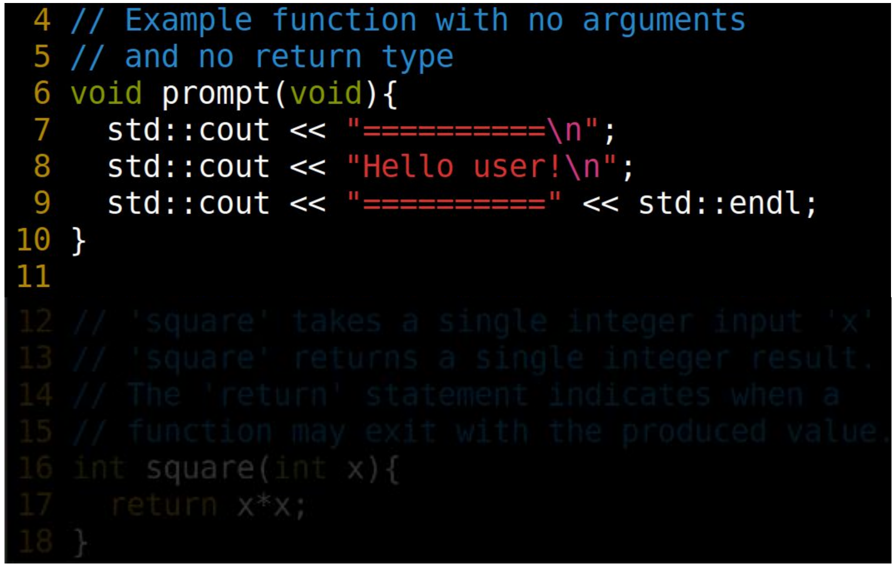
Functions

An Abstraction for Writing Reusable and Modular Code

At the very minimum -- “a command that returns a value”

General Purpose of a Function in a programming language (1/2)

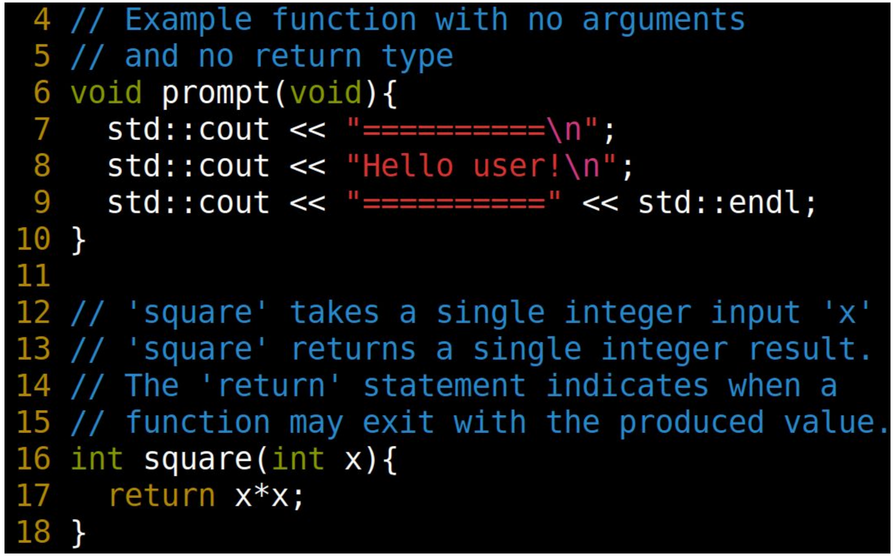
- Some functions purely run a routine of code -- no return value.
    - **`prompt()`** on the right is an example
        - **`void`** is the return type when nothing is returned.
    - Note: Other languages sometimes distinguish explicitly and name these procedures or subroutines
- Some functions compute a new value from 0 or more inputs.
    - **`int square(int x)`** on the right is an example function
    - `std::rand` is an example that takes no inputs and produces an output.
- Sometimes this means muting a given input and/or output
    - i.e. transforming data



```
// Example function with no arguments
// and no return type
void prompt(void){
  std::cout << "==========\n";
  std::cout << "Hello user!\n";
  std::cout << "==========" << std::endl;
}

// 'square' takes a single integer input 'x'
// 'square' returns a single integer result.
// The 'return' statement indicates when a
// function may exit with the produced value.
int square(int x){
  return x*x;
}
```

General Purpose of a Function in a programming language (2/2)

- Some functions purely run a routine of code -- no return value.
  - **`prompt()`** on the right is an example
    - **`void`** is the return type when nothing is returned.
  - Note: Other languages sometimes distinguish explicitly and name these procedures or subroutines
- Some functions compute a new value from 0 or more inputs.
  - **`int square(int x)`** on the right is an example function
  - `std::rand` is an example that takes no inputs and produces an output.
- Some functions mutate a given input and/or output
  - i.e. transforming data



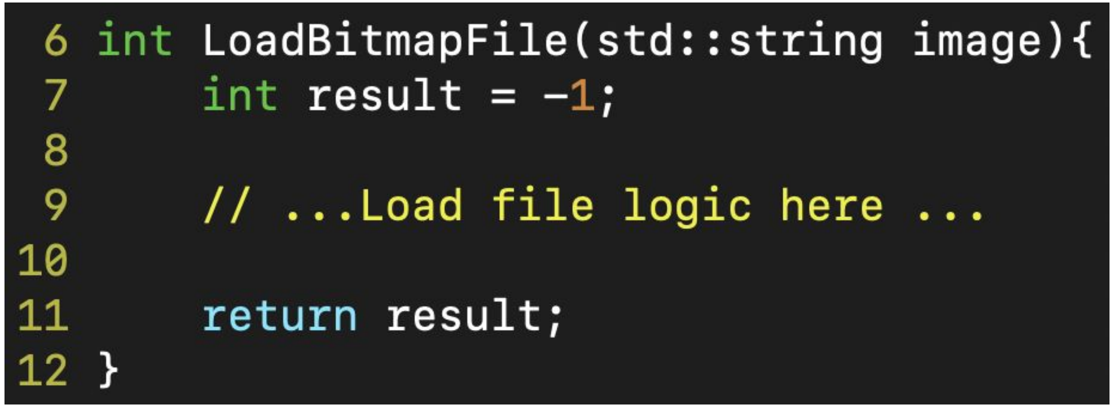
```
// Example function with no arguments
// and no return type
void prompt(void){
  std::cout << "==========\n";
  std::cout << "Hello user!\n";
  std::cout << "==========" << std::endl;
}

// 'square' takes a single integer input 'x'
// 'square' returns a single integer result.
// The 'return' statement indicates when a
// function may exit with the produced value.
int square(int x){
  return x*x;
}
```

Function Anatomy

The pieces that make up a function

Function Basics - Parts of a Function (1/6)

- Let's take a look at how to create a function, and the different components of a function.

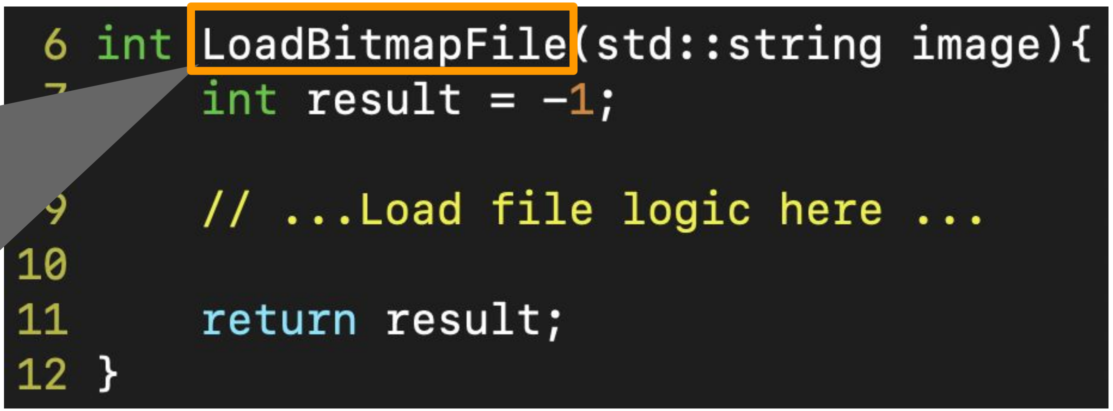


```
int LoadBitmapFile(std::string image){
    int result = -1;

    // ...Load file logic here ...

    return result;
}
```

Function Basics - Parts of a Function (2/6)

Functions must have a name:

The name should describe ‘what’ the function is doing at a minimum



```
int LoadBitmapFile(std::string image){
    int result = -1;

    // ...Load file logic here ...

    return result;
}
```

Some naming rules:

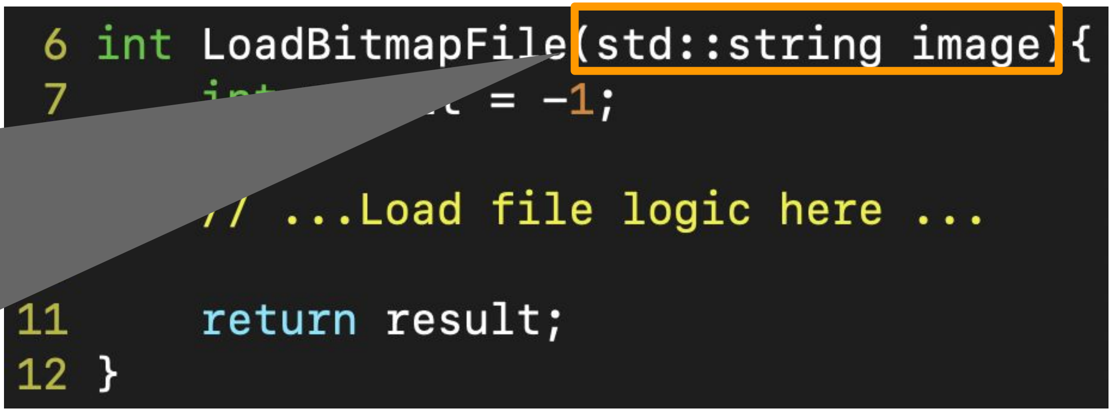
- Functions names must start with a letter or underscore.
- Note: ‘Usually’ names that begin with an underscore are reserved for something special -- the underscore intentionally making it harder to type.

Function Basics - Parts of a Function (3/6)

The next part of a function are the parameters of the function (i.e. the ‘input’)

Functions can have zero or more inputs.

This function has exactly one parameter of type std::string



```
int LoadBitmapFile(std::string image){
    int [illegible]t = -1;

    // ...Load file logic here ...

    return result;
}
```

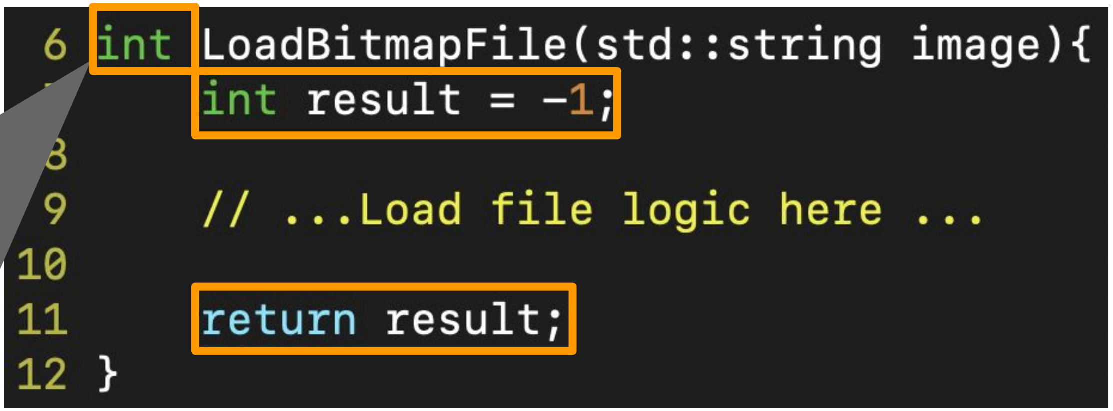
Notes on function parameters:

- Functions names must start with a letter or underscore.
- Note: Sometimes the term parameter and argument ge mixed up
    - parameters are part of the definition
    - arguments are the values we supply when we actually use the function.

Function Basics - Parts of a Function (4/6)

Next we have the ‘**return type**’ -- this tells us the type of the value returned.

Functions return at most 1 value (The type is ‘void’ if we return no values).



```cpp
int LoadBitmapFile(std::string image){
    int result = -1;

    // ...Load file logic here ...

    return result;
}
```

Notes on return values:

- There are a few ways to get more than 1 value returned from a function:
  - We could return an aggregate type (e.g. struct) containing multiple values
  - We could define parameters (very cautiously) that allow us to hold a result
- Another choice is to return `std::optional` -- this means 0 or 1 values are returned.

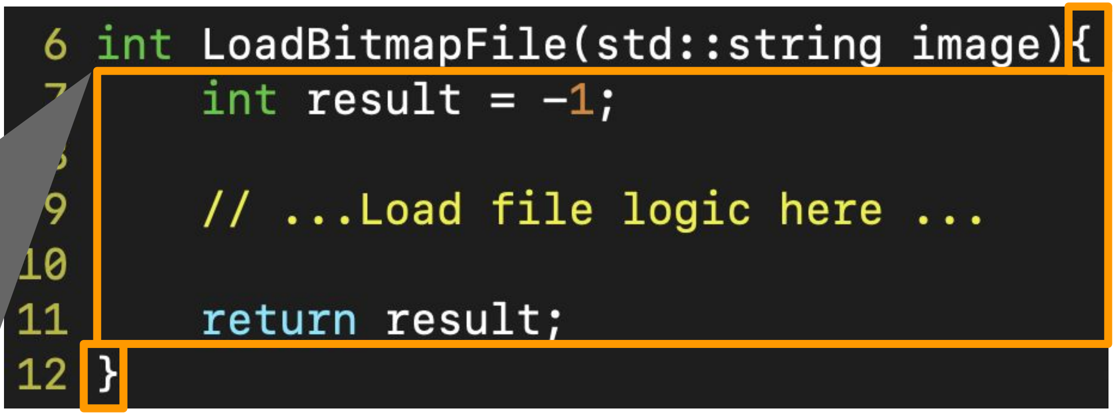
Function Basics - Parts of a Function (5/6)

The **function body** (between the { }'s) is where we do the actual work.

This is where we define the implementation of 'how' the function achieves its goal.

Where the 'goal' or 'action' of the function is well described by the function name.

Local variables declared in the function body follow normal scoping rules.



```
int LoadBitmapFile(std::string image){
    int result = -1;

    // ...Load file logic here ...

    return result;
}
```

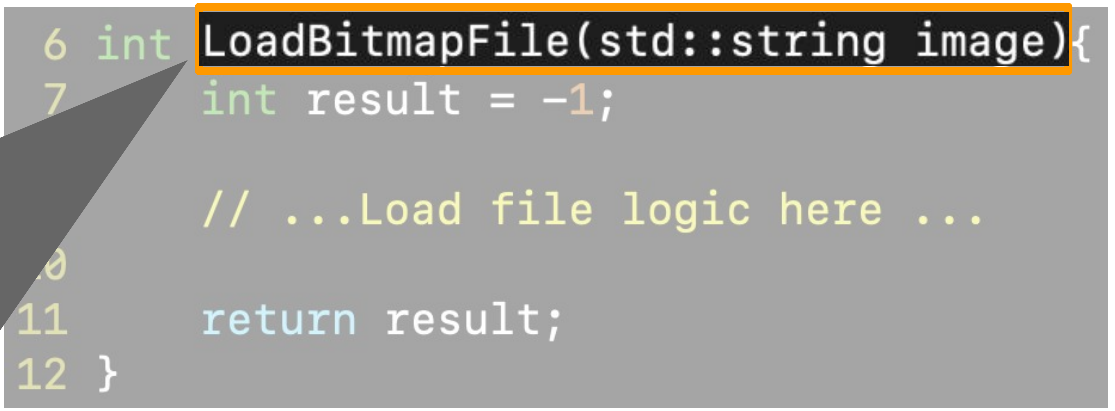
Notes on function body:

- Later on we'll see that the function body *usually* is defined in a source (.cpp) file.
  - We generally do not put the implementation of a function body in the header (.hpp) file.

Function Basics - Parts of a Function (6/6)

The combination of the function name and the parameters make up what is known as a ‘**function signature**’

When we use a function (a.k.a. ‘call a function’), the combination of the name and arguments we provide will call our function



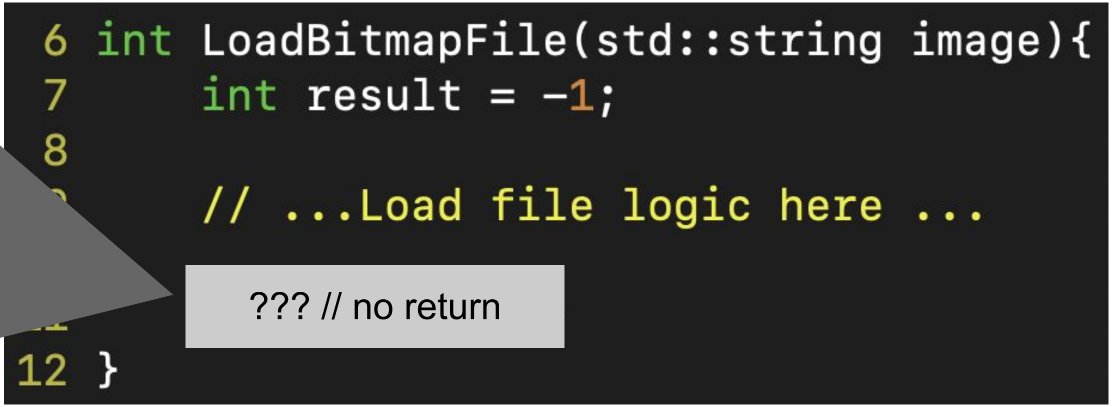
```
int LoadBitmapFile(std::string image){
    int result = -1;

    // ...Load file logic here ...

    return result;
}
```

Notes on Function Signature:

- The name and arguments in combination call a specific function.
    - For example:
        - `LoadBitmapFile(“./images/cpp.bmp”);`
        - This function call jumps to execute function with ‘some sort of valid’ string/char array version of our function (more on that later...)

(Aside on compiling in debug) Occasional - Gotcha!

- Depending on your compiler or IDE environment -- if you compile your source with a function that lacks a 'return' statement -- *it may* still work
    - That includes if there exist multiple return paths.
- Don't trust this however -- we need to have a return statement if we are expecting a result.



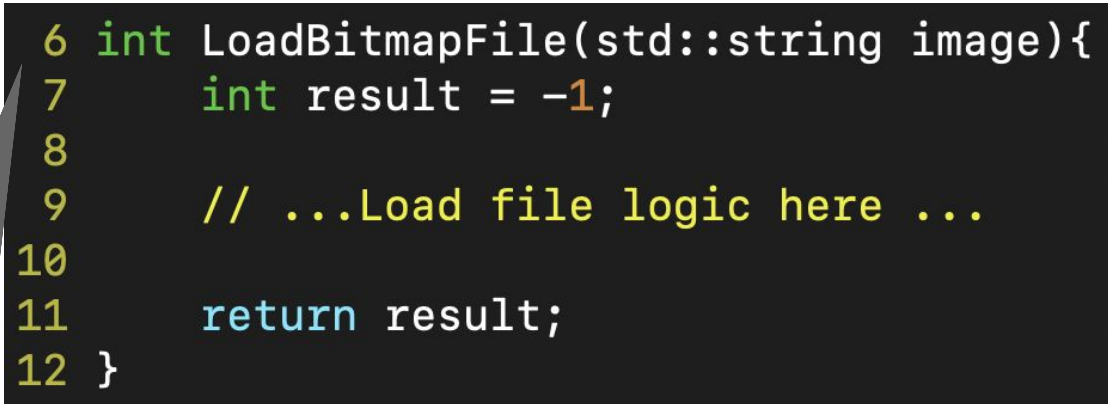
```
int LoadBitmapFile(std::string image){
    int result = -1;

    // ...Load file logic here ...

    ??? // no return
}
```

- Notes on Debugging:
    - Your compiler generally should issue a warning if there is a missing return
        - Listen to those warnings!

Congratulations -- We Understand the Pieces of a Function

That's really all there is to the basics!

- Return Type
- Function Signature
  - Descriptive Name
  - Parameters
- Function Body



```
int LoadBitmapFile(std::string image){
    int result = -1;

    // ...Load file logic here ...

    return result;
}
```

- Stay tuned for more!

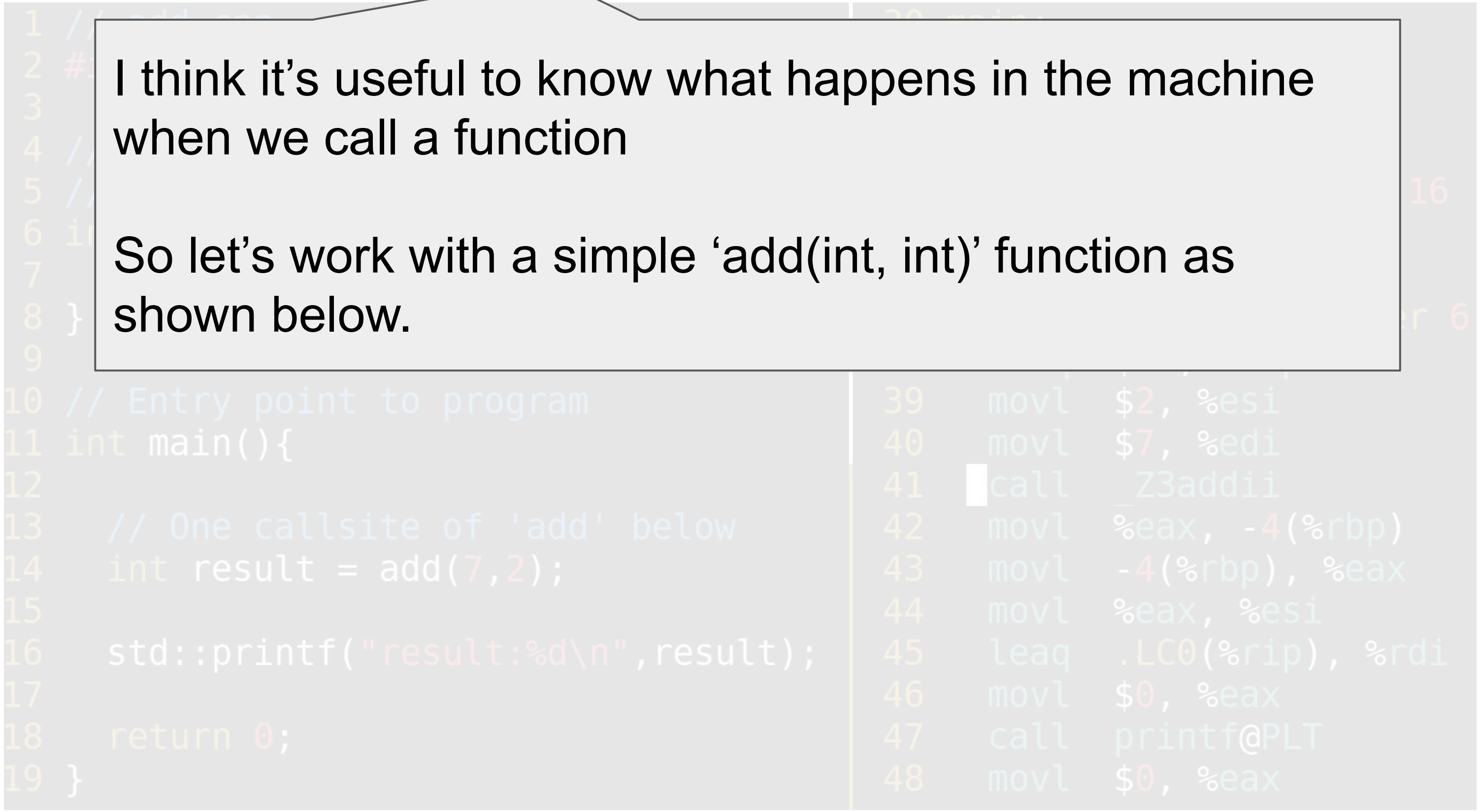
Function Call

What happens in the machine when we call a function?

Function Calls - From the Machine Viewpoint (1/5)

I think it’s useful to know what happens in the machine when we call a function

So let’s work with a simple ‘add(int, int)’ function as shown below.

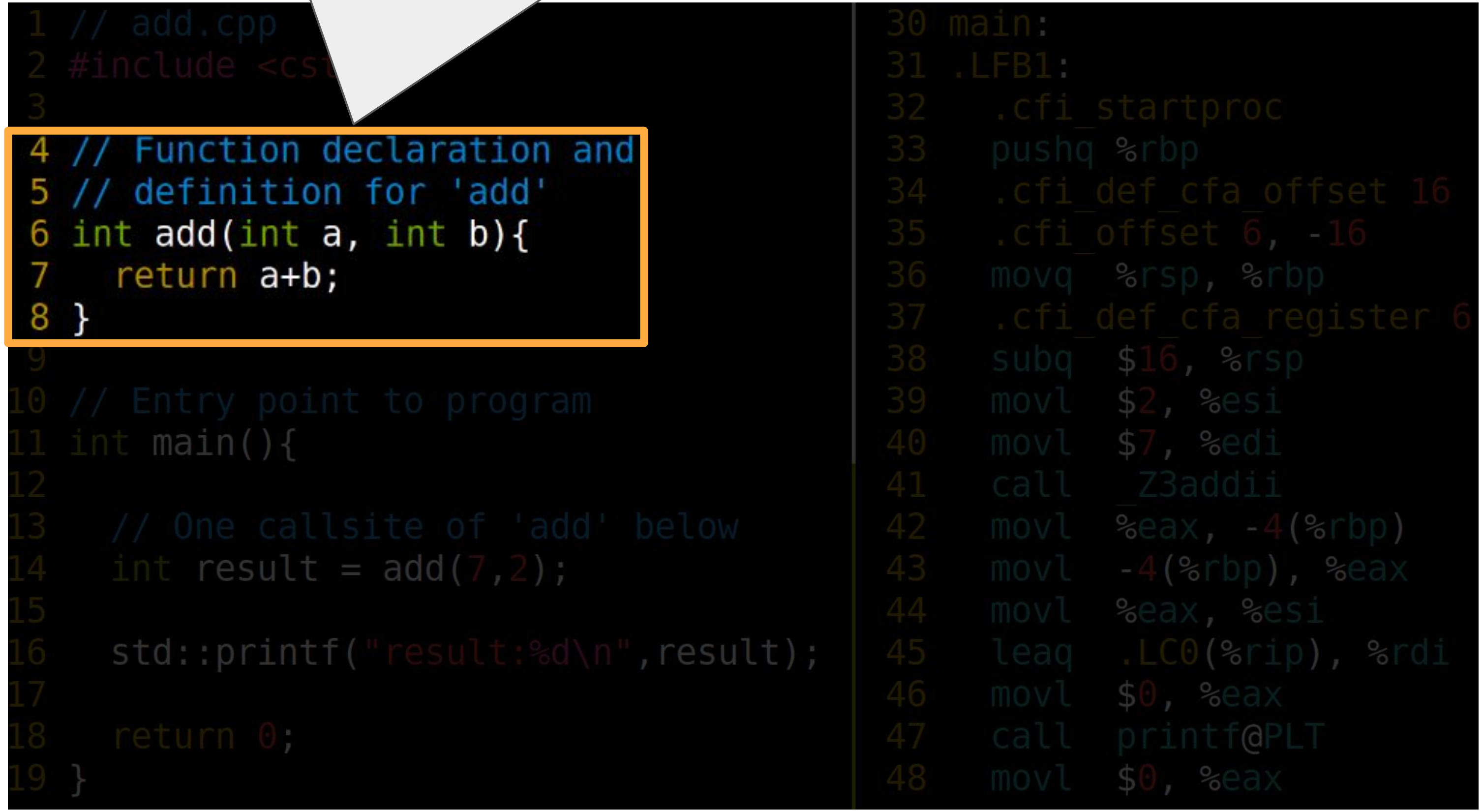


```
}

// Entry point to program
int main(){

  // One callsite of 'add' below
  int result = add(7,2);

  std::printf("result:%d\n",result);

  return 0;
}
```

```
  movl  $2, %esi
  movl  $7, %edi
  call  _Z3addii
  movl  %eax, -4(%rbp)
  movl  -4(%rbp), %eax
  movl  %eax, %esi
  leaq  .LC0(%rip), %rdi
  movl  $0, %eax
  call  printf@PLT
  movl  $0, %eax
```

Functi Calls Machine Viewpoint (2/5)

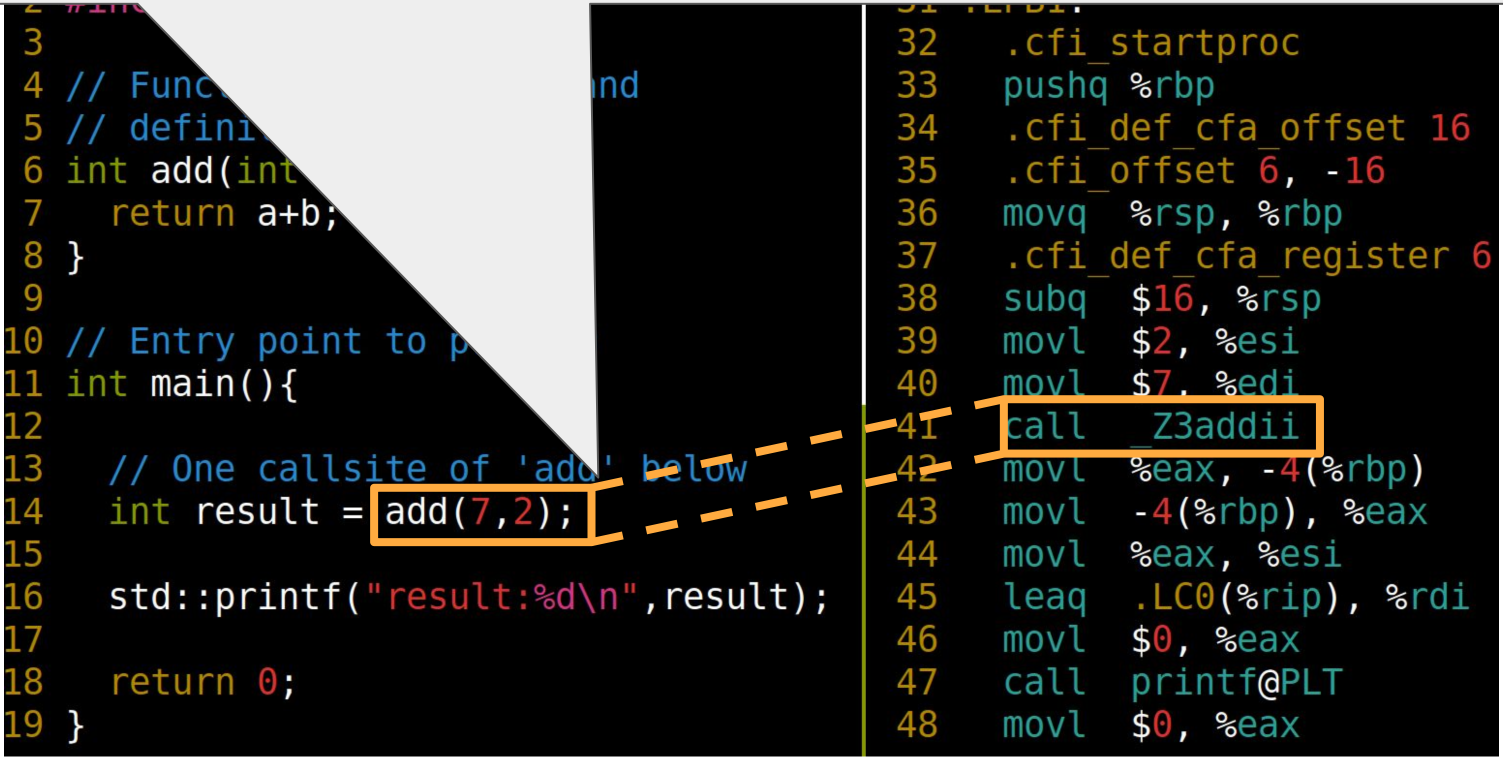
- Simple program focusing on an ‘add’ function.



```
// add.cpp
#include <cst

// Function declaration and
// definition for 'add'
int add(int a, int b){
  return a+b;
}

// Entry point to program
int main(){

  // One callsite of 'add' below
  int result = add(7,2);

  std::printf("result:%d\n",result);

  return 0;
}
```

```
main:
.LFB1:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	subq	$16, %rsp
	movl	$2, %esi
	movl	$7, %edi
	call	_Z3addii
	movl	%eax, -4(%rbp)
	movl	-4(%rbp), %eax
	movl	%eax, %esi
	leaq	.LC0(%rip), %rdi
	movl	$0, %eax
	call	printf@PLT
	movl	$0, %eax
```

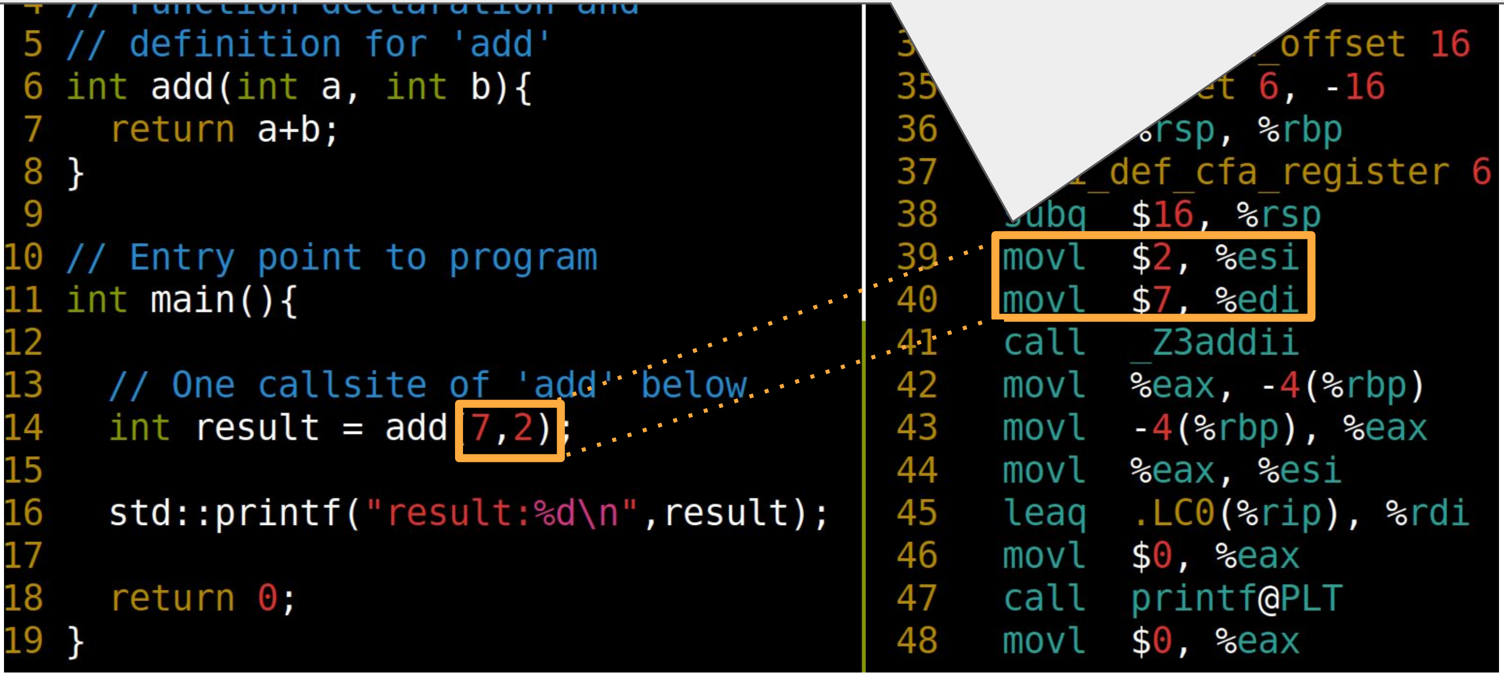
- When we call a function in C++
  - At the assembly level is is replaced with a ‘call’ instruction.
  - Note the ‘`addii`’ portion of the call you can kind of figure out the function signature (i.e. `add(int,int)`)



```
#inc
 
// Func                    nd
// defini
int add(int
  return a+b;
}

// Entry point to p
int main(){

  // One callsite of 'add' below
  int result = add(7,2);

  std::printf("result:%d\n",result);

  return 0;
}
```

```
.LFB1:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    $2, %esi
    movl    $7, %edi
    call    _Z3addii
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
```

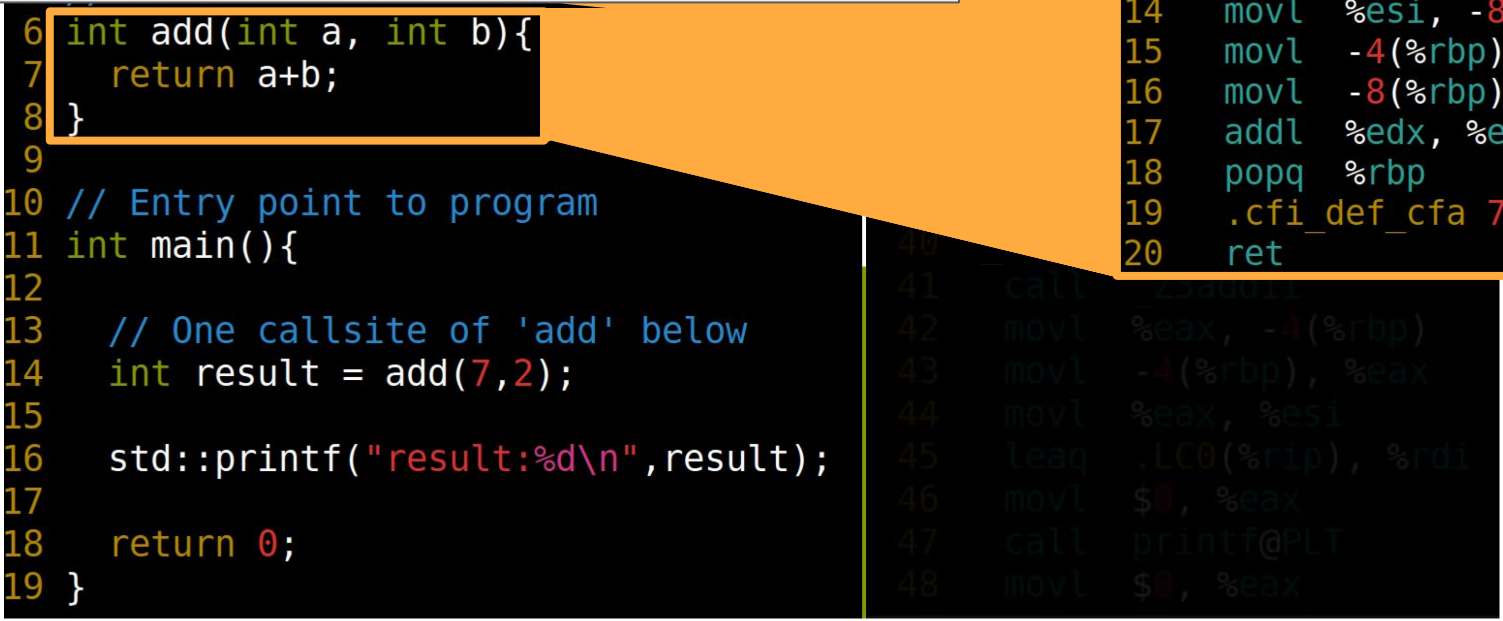
- Note that we also have to handle the arguments that we provide.
  - We either need to reference them for somewhere, or ‘copy’ (the movl instruction) data into registers.
  - Again, you can see corresponding $2 and $7



```
// Function declaration and
// definition for 'add'
int add(int a, int b){
  return a+b;
}

// Entry point to program
int main(){

  // One callsite of 'add' below
  int result = add(7,2);

  std::printf("result:%d\n",result);

  return 0;
}
```

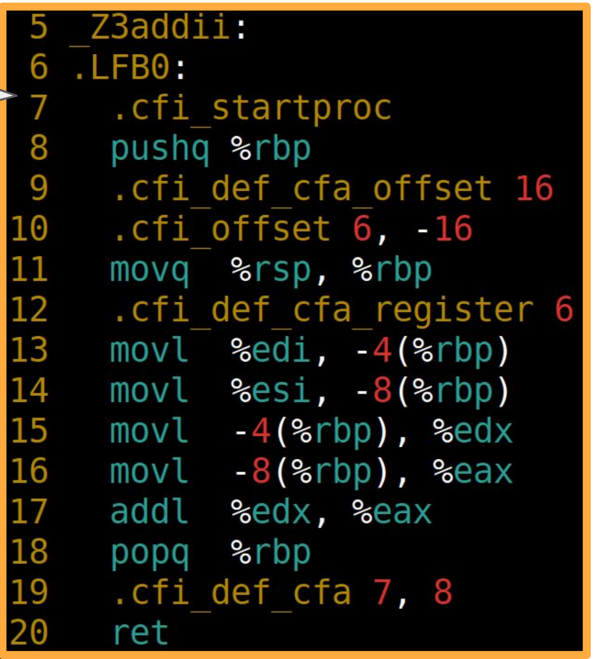
```
             offset 16
          et 6, -16
           rsp, %rbp
      def_cfa_register 6
 ubq    $16, %rsp
movl    $2, %esi
movl    $7, %edi
call    _Z3addii
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
movl    %eax, %esi
leaq    .LC0(%rip), %rdi
movl    $0, %eax
call    printf@PLT
movl    $0, %eax
```

- Exploring the assembly a bit more -- you’ll see the label for our add function.
  - The rest of our function body is then implemented.
  - Including copying the arguments
  - (and also a ‘ret’ to return to our callsite)



```
int add(int a, int b){
  return a+b;
}

// Entry point to program
int main(){

  // One callsite of 'add' below
  int result = add(7,2);

  std::printf("result:%d\n",result);

  return 0;
}
```

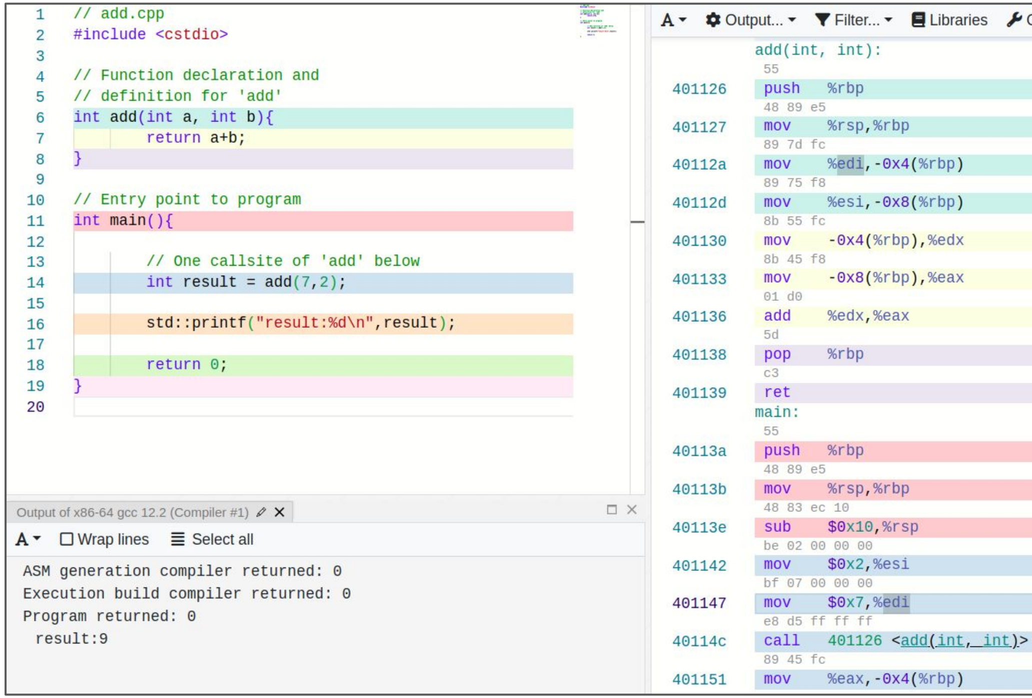
```
	call	_Z3addii
	movl	%eax, -4(%rbp)
	movl	-4(%rbp), %eax
	movl	%eax, %esi
	leaq	.LC0(%rip), %rdi
	movl	$0, %eax
	call	printf@PLT
	movl	$0, %eax
```



```
_Z3addii:
.LFB0:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movl	%edi, -4(%rbp)
	movl	%esi, -8(%rbp)
	movl	-4(%rbp), %edx
	movl	-8(%rbp), %eax
	addl	%edx, %eax
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
```

Recap: Machine Viewpoint

- The point of that exercise is for you to see when we call a function:
  - We *usually* jump somewhere in the code.
    - This at a minimum means we need to **store a return address**
    - We also may need to **copy or otherwise access arguments**.
    - The combination of the arguments and return address make up part of the **stack frame**
      - (Note: local variables in function body are also part of stack frame)



```cpp
// add.cpp
#include <cstdio>

// Function declaration and
// definition for 'add'
int add(int a, int b){
        return a+b;
}

// Entry point to program
int main(){

        // One callsite of 'add' below
        int result = add(7,2);

        std::printf("result:%d\n",result);

        return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 result:9
```

```
        add(int, int):
         55
401126   push   %rbp
         48 89 e5
401127   mov    %rsp,%rbp
         89 7d fc
40112a   mov    %edi,-0x4(%rbp)
         89 75 f8
40112d   mov    %esi,-0x8(%rbp)
         8b 55 fc
401130   mov    -0x4(%rbp),%edx
         8b 45 f8
401133   mov    -0x8(%rbp),%eax
         01 d0
401136   add    %edx,%eax
         5d
401138   pop    %rbp
         c3
401139   ret
        main:
         55
40113a   push   %rbp
         48 89 e5
40113b   mov    %rsp,%rbp
         48 83 ec 10
40113e   sub    $0x10,%rsp
         be 02 00 00 00
401142   mov    $0x2,%esi
         bf 07 00 00 00
401147   mov    $0x7,%edi
         e8 d5 ff ff ff
40114c   call   401126 <add(int, int)>
         89 45 fc
401151   mov    %eax,-0x4(%rbp)
```

Can generate assembly yourself from compiler:

```
g++ add.cpp -S
```

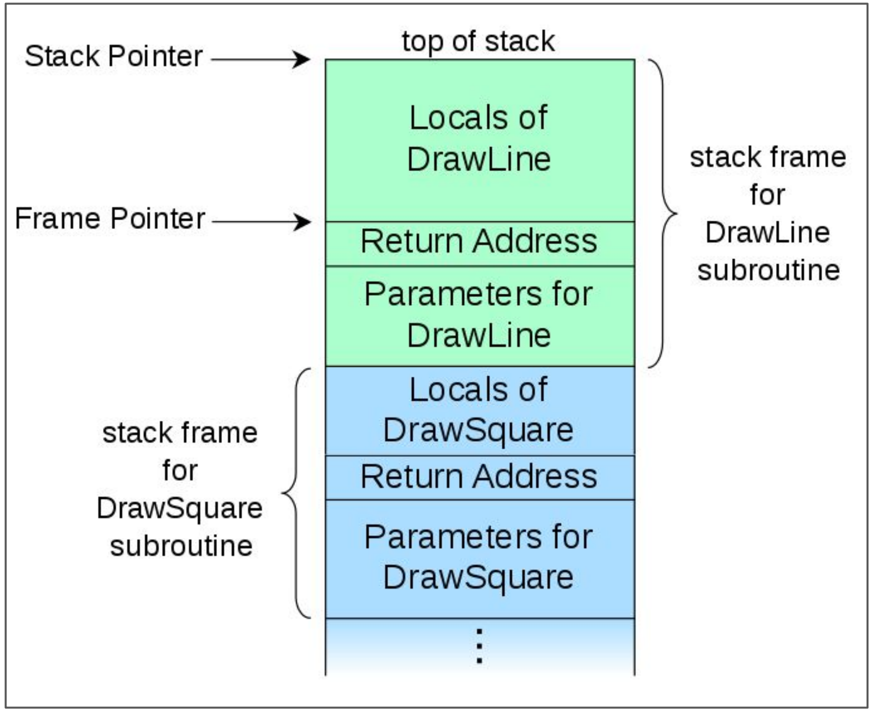
Or otherwise a nice interactive tool for exploring assembly:
https://godbolt.org/z/qdEc3G737

Recursive Function Call

Revisiting Functions with Recursion

(Review) Calling functions within functions (Call stack)

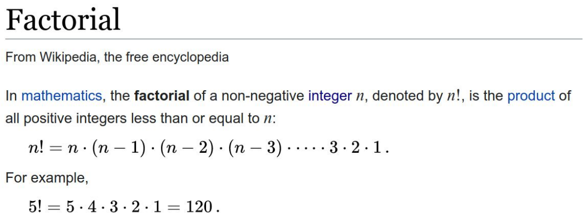
- When you call a function recall that the arguments are copied and the return address.
  - Any local variable are also stored on the call stack as well.
- If a function calls another function, yet again, more functions are placed on the call stack.
  - Understanding this can be useful for understanding how information moves through your C++ programs.
  - (It's also very useful for debugging!)



https://en.wikipedia.org/wiki/Call_stack

Recursion Example 1

- C++ supports recursive calls to functions
- Here's an example of computing factorial recursively
  - Note: We can also see that we have multiple returns paths in factorial
    - This is perfectly fine as long as every path the function may exit returns an integer.



Factorial

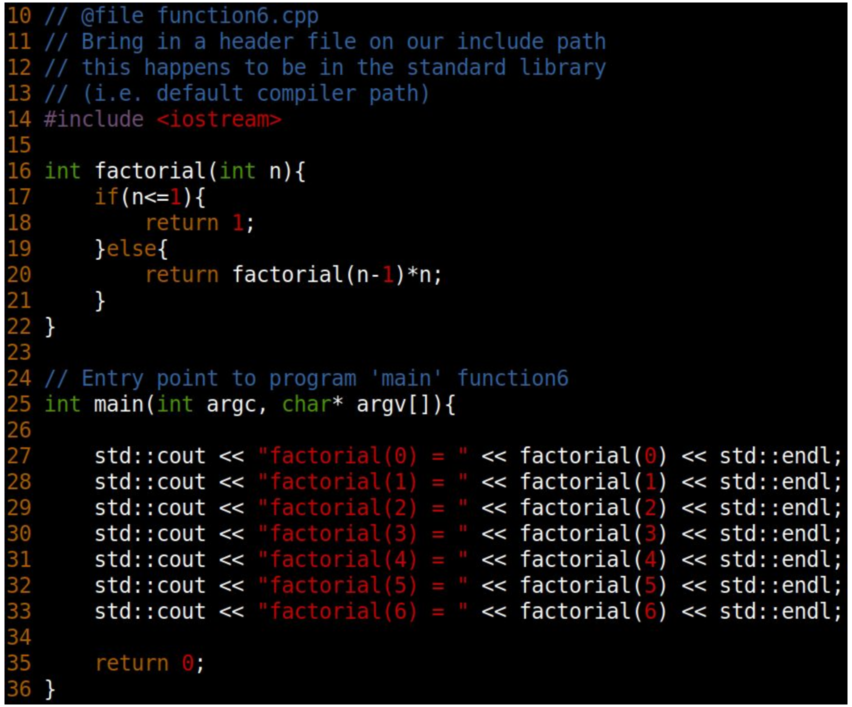
From Wikipedia, the free encyclopedia

In mathematics, the **factorial** of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n:

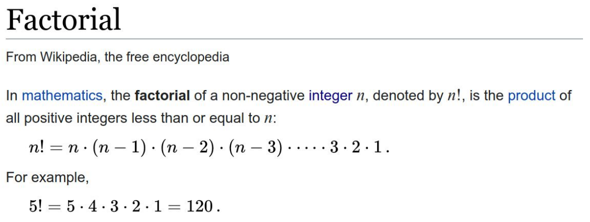
$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \cdots \cdot 3 \cdot 2 \cdot 1 \,.$$

For example,

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120 \,.$$



```cpp
// @file function6.cpp
// Bring in a header file on our include path
// this happens to be in the standard library
// (i.e. default compiler path)
#include <iostream>

int factorial(int n){
    if(n<=1){
        return 1;
    }else{
        return factorial(n-1)*n;
    }
}

// Entry point to program 'main' function6
int main(int argc, char* argv[]){

    std::cout << "factorial(0) = " << factorial(0) << std::endl;
    std::cout << "factorial(1) = " << factorial(1) << std::endl;
    std::cout << "factorial(2) = " << factorial(2) << std::endl;
    std::cout << "factorial(3) = " << factorial(3) << std::endl;
    std::cout << "factorial(4) = " << factorial(4) << std::endl;
    std::cout << "factorial(5) = " << factorial(5) << std::endl;
    std::cout << "factorial(6) = " << factorial(6) << std::endl;

    return 0;
}
```


Factorial

From Wikipedia, the free encyclopedia

In mathematics, the **factorial** of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n:

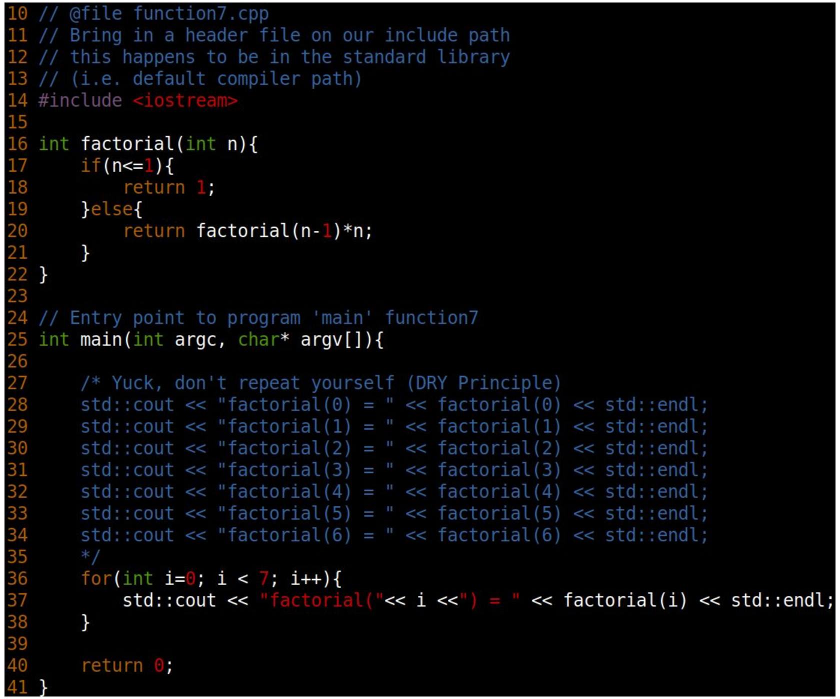
$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \cdots \cdot 3 \cdot 2 \cdot 1\,.$$

For example,

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120\,.$$

Recursion Example 1 - Refactored

- In that last example, I caught myself copying & pasting the ‘std::cout’ line several times.
  - There’s a general principle called ‘Don’t Repeat Yourself’ (DRY)
- So I couldn’t help myself but to refactor the code to make it a little cleaner.
  - This is our motivation for functions as well...modular pieces of code so we don’t have to repeat ourselves!

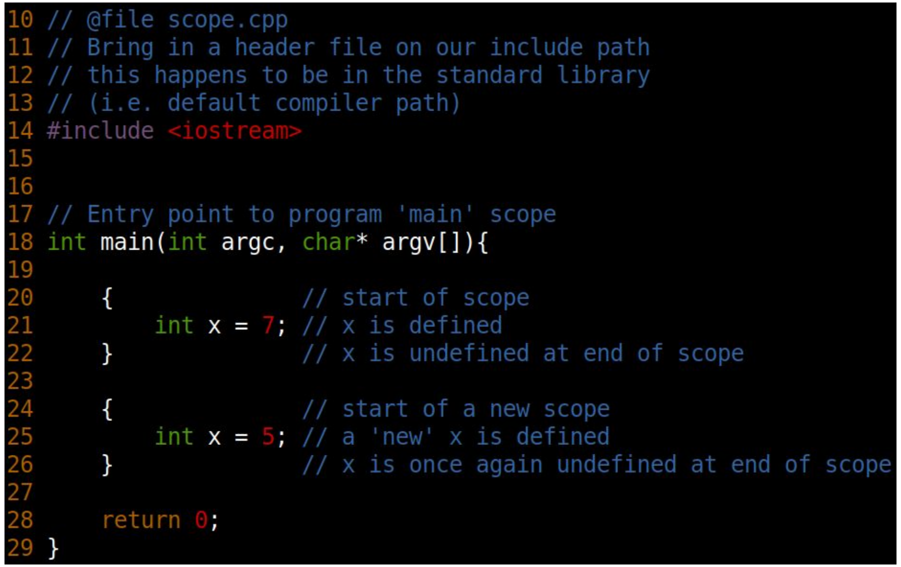


```
// @file function7.cpp
// Bring in a header file on our include path
// this happens to be in the standard library
// (i.e. default compiler path)
#include <iostream>

int factorial(int n){
    if(n<=1){
        return 1;
    }else{
        return factorial(n-1)*n;
    }
}

// Entry point to program 'main' function7
int main(int argc, char* argv[]){

    /* Yuck, don't repeat yourself (DRY Principle)
    std::cout << "factorial(0) = " << factorial(0) << std::endl;
    std::cout << "factorial(1) = " << factorial(1) << std::endl;
    std::cout << "factorial(2) = " << factorial(2) << std::endl;
    std::cout << "factorial(3) = " << factorial(3) << std::endl;
    std::cout << "factorial(4) = " << factorial(4) << std::endl;
    std::cout << "factorial(5) = " << factorial(5) << std::endl;
    std::cout << "factorial(6) = " << factorial(6) << std::endl;
    */
    for(int i=0; i < 7; i++){
        std::cout << "factorial("<< i <<") = " << factorial(i) << std::endl;
    }

    return 0;
}
```

Exercise: Try removing the loop and using iota or generate_n

Scope of Variables (1/2)

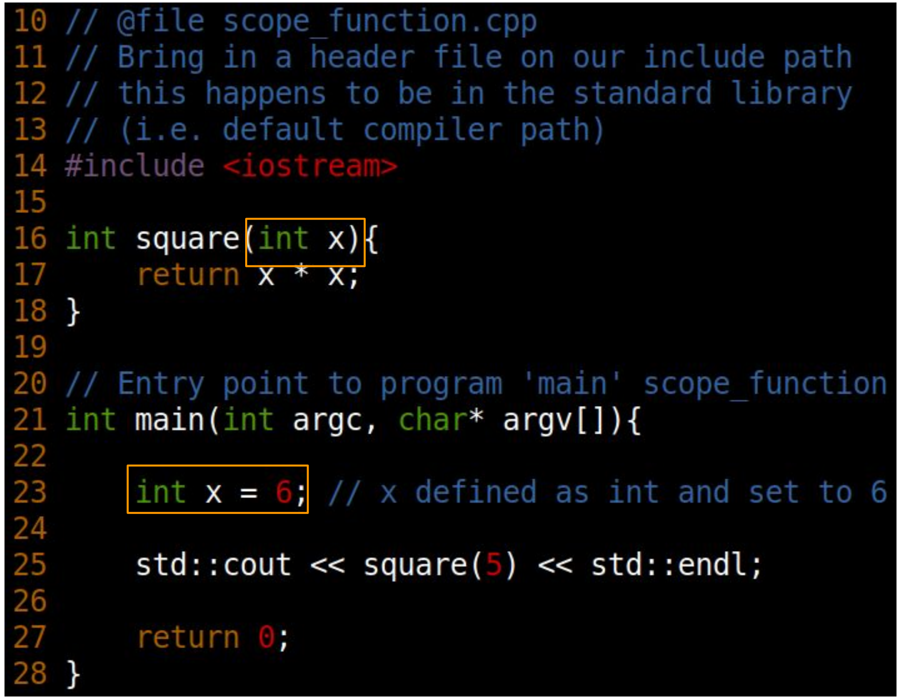
- Something else we want to keep in mind is the scope (or lifetime) of variables in functions.
  - Stack allocated variables scope is defined by the left and right curly braces{}
- See example on the right
  - Note: Sometimes we talk about this in terms of when the ‘variable’ is *alive* or ‘in scope’



```cpp
// @file scope.cpp
// Bring in a header file on our include path
// this happens to be in the standard library
// (i.e. default compiler path)
#include <iostream>


// Entry point to program 'main' scope
int main(int argc, char* argv[]){

    {                // start of scope
        int x = 7;   // x is defined
    }                // x is undefined at end of scope

    {                // start of a new scope
        int x = 5;   // a 'new' x is defined
    }                // x is once again undefined at end of scope

    return 0;
}
```

Scope of Variable Symbol Name is local to functions

- It’s worth explicitly pointing out that there are ‘different x variables’ in this code snippet
- Observe ‘square’ is using the parameter name ‘x’ at line 16.
  - This is fine because the scope of ‘x’ is local to each respective function.
- Thus ‘x’ is a local variable in each function.
  - (Thus we can reuse the name at line 23)



```
10 // @file scope_function.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15
16 int square(int x){
17     return x * x;
18 }
19
20 // Entry point to program 'main' scope_function
21 int main(int argc, char* argv[]){
22
23     int x = 6; // x defined as int and set to 6
24
25     std::cout << square(5) << std::endl;
26
27     return 0;
28 }
```

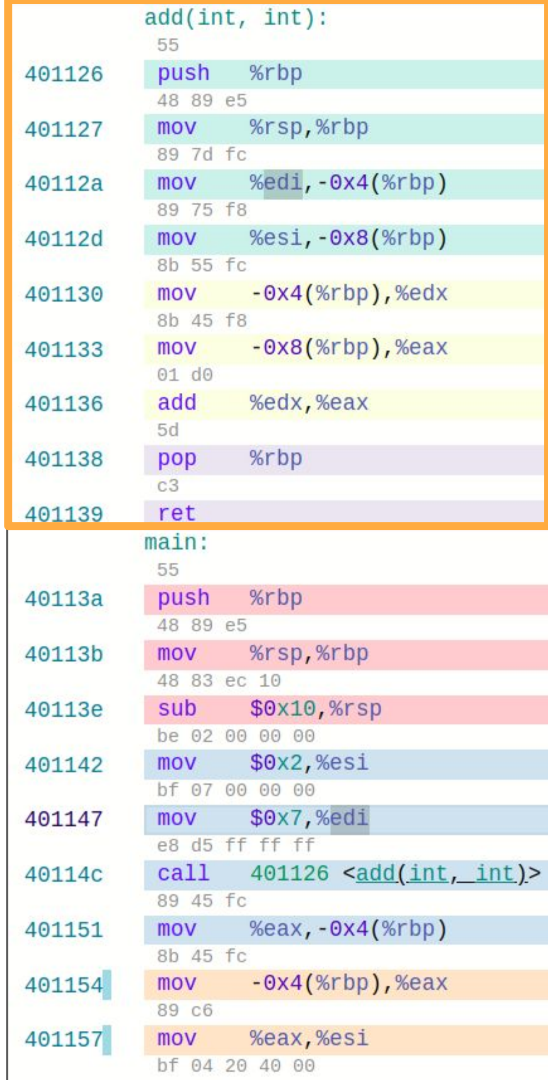
Creating Libraries with Functions

Where are they stored?

How are they organized?

Functions are part of our code

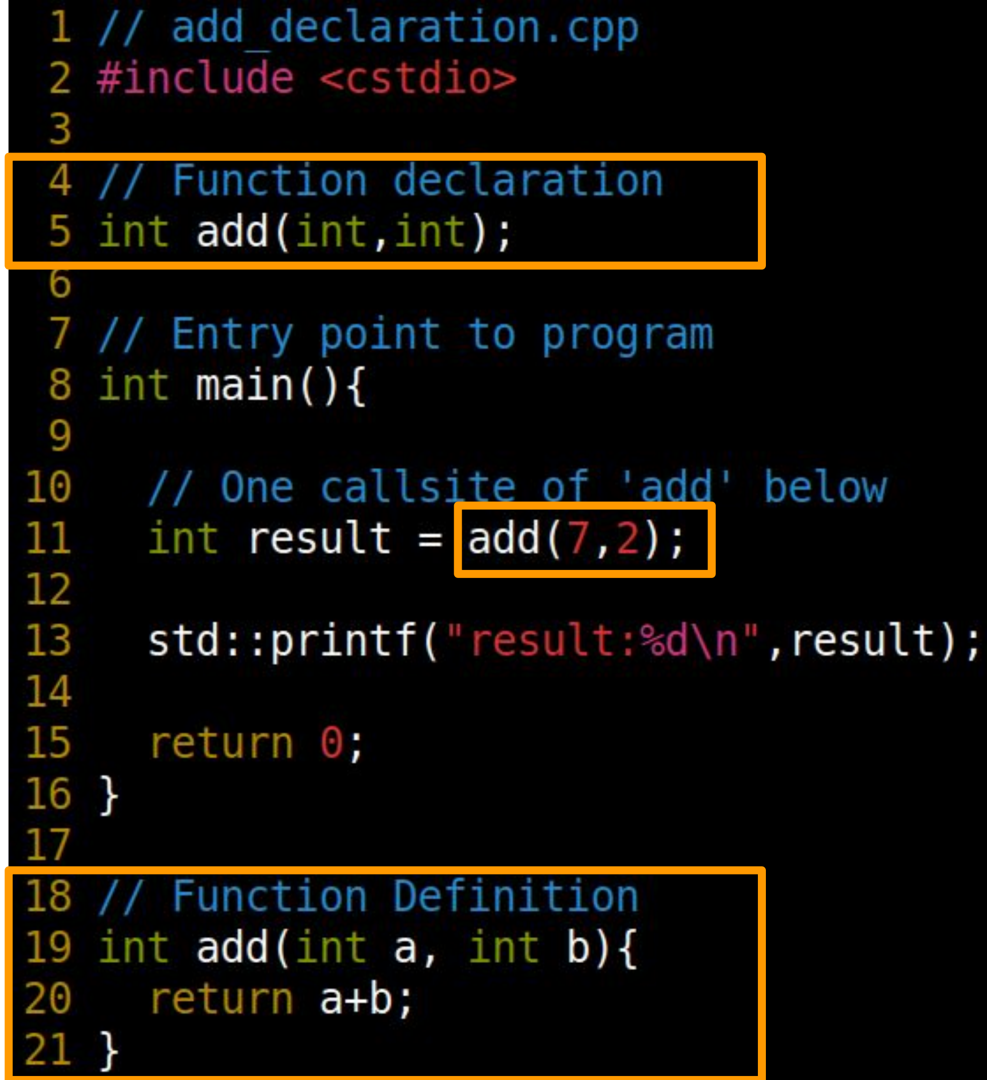
- We can see this from our previous dive into assembly
  - Functions have an ‘address’ where they are stored in memory.
    - This means we can take the ‘address’ of a function
      - (e.g. &add).
    - But before we get into that idea -- I want to show an example of how functions get organized in .cpp and .hpp files.



```
        add(int, int):
         55
401126   push   %rbp
         48 89 e5
401127   mov    %rsp,%rbp
         89 7d fc
40112a   mov    %edi,-0x4(%rbp)
         89 75 f8
40112d   mov    %esi,-0x8(%rbp)
         8b 55 fc
401130   mov    -0x4(%rbp),%edx
         8b 45 f8
401133   mov    -0x8(%rbp),%eax
         01 d0
401136   add    %edx,%eax
         5d
401138   pop    %rbp
         c3
401139   ret
        main:
         55
40113a   push   %rbp
         48 89 e5
40113b   mov    %rsp,%rbp
         48 83 ec 10
40113e   sub    $0x10,%rsp
         be 02 00 00 00
401142   mov    $0x2,%esi
         bf 07 00 00 00
401147   mov    $0x7,%edi
         e8 d5 ff ff ff
40114c   call   401126 <add(int, int)>
         89 45 fc
401151   mov    %eax,-0x4(%rbp)
         8b 45 fc
401154   mov    -0x4(%rbp),%eax
         89 c6
401157   mov    %eax,%esi
         bf 04 20 40 00
```

Function Declaration (1/2)

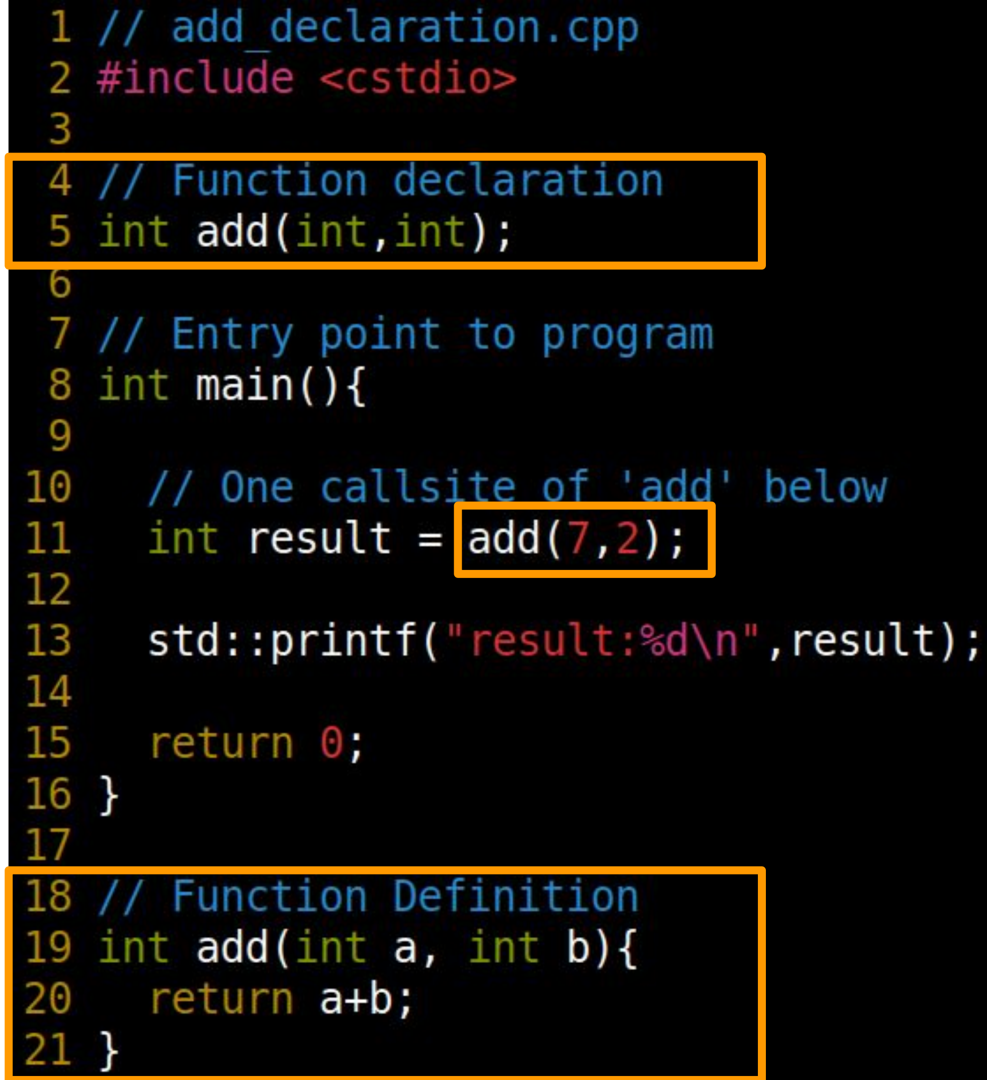
- Observe at line 5 we have a ‘Function Declaration’
  - The includes the function signature and return type
  - There is no ‘body’ of the function
- The purpose of providing a function declaration in this case is known as a **forward declaration**
  - We must parse our file from top-to-bottom -- thus forward declarations allow the use of `add(7,2)` to compile without issue.
  - So long as at the link stage of compilation we find a definition , we will successfully build a program.



```
 1 // add_declaration.cpp
 2 #include <cstdio>
 3
 4 // Function declaration
 5 int add(int,int);
 6
 7 // Entry point to program
 8 int main(){
 9
10   // One callsite of 'add' below
11   int result = add(7,2);
12
13   std::printf("result:%d\n",result);
14
15   return 0;
16 }
17
18 // Function Definition
19 int add(int a, int b){
20   return a+b;
21 }
```

A forward declaration is effectively a 'promise to the compiler and/or linker' that you will in fact find the function definition before everything is assembled.

- Observe at line 5 we have a 'Function Declaration'
  - The includes the function signature and return type
  - There is no 'body' of the function
- The purpose of providing a function declaration in this case is known as a **forward declaration**
  - We must parse our file from top-to-bottom -- thus forward declarations allow the use of `add(7,2)` to compile without issue.
  - So long as at the link stage of compilation we find a definition , we will successfully build a program.



```
 1 // add_declaration.cpp
 2 #include <cstdio>
 3
 4 // Function declaration
 5 int add(int,int);
 6
 7 // Entry point to program
 8 int main(){
 9
10   // One callsite of 'add' below
11   int result = add(7,2);
12
13   std::printf("result:%d\n",result);
14
15   return 0;
16 }
17
18 // Function Definition
19 int add(int a, int b){
20   return a+b;
21 }
```

Creating Libraries (1/4)

- To the right I'm going to reveal a complete program separated out into three files
  - The header (add.hpp)
    - Provides the forward declarations for our function
    - At the linking stage, we'll need an implementation before we can use it.
  - The source for add.cpp
    - Provides the implementation
    - Note that the add.cpp also includes the add.hpp -- this is effectively the forward declaration being pasted in
  - Finally, the main.cpp
    - We #include "add.hpp" which gives us access to use add.cpp
    - So long as we link in the implementation of add (from add.cpp, which will be an add.o file), we can use the add function.

```cpp
// add_library/add.hpp
#pragma once

// Contains only function declarations
// Note: int add(int,int) is equivalent
//       but I think it's best to give
//       names to parameters in declaration.
int add(int a, int b);
```

```cpp
// add_library/add.cpp

// Include 'add.hpp' and provide
// definition for our function.
#include "add.hpp"

int add(int a, int b){
  return a+b;
}
```

```cpp
// add_library/main.cpp
#include <iostream>
// include 'add.hpp' from local dir.
#include "add.hpp"

int main(){
  std::cout << add(7,2) << std::endl;
  return 0;
}
```

```
g++ add.cpp main.cpp -o prog
```

Creating Libraries (2/4)

- To the right I'm going to reveal a complete program separated out into three files
  - The header (add.hpp)
    - Provides the forward declarations for our function
    - At the linking stage, we'll need an implementation before we can use it.
  - The source for add.cpp
    - Provides the implementation
    - Note that the add.cpp also includes the add.hpp -- this is effectively the forward declaration being pasted in
  - Finally, the main.cpp
    - We #include "add.hpp" which gives us access to use add.cpp
    - So long as we link in the implementation of add (from add.cpp, which will be an add.o file), we can use the add function.

```cpp
// add_library/add.hpp
#pragma once

// Contains only function declarations
// Note: int add(int,int) is equivalent
//       but I think it's best to give
//       names to parameters in declaration.
int add(int a, int b);
```

```cpp
// add_library/add.cpp

// Include 'add.hpp' and provide
// definition for our function.
#include "add.hpp"

int add(int a, int b){
  return a+b;
}
```

```cpp
// add_library/main.cpp
#include <iostream>
// include 'add.hpp' from local dir.
#include "add.hpp"

int main(){
  std::cout << add(7,2) << std::endl;
  return 0;
}
```

```
g++ add.cpp main.cpp -o prog
```

Creating Libraries (3/4)

- To the right I'm going to reveal a complete program separated out into three files
  - The header (add.hpp)
    - Provides the forward declarations for our function
    - At the linking stage, we'll need an implementation before we can use it.
  - The source for add.cpp
    - Provides the implementation
    - Note that the add.cpp also includes the add.hpp -- this is effectively the forward declaration being pasted in
  - Finally, the main.cpp
    - We #include "add.hpp" which gives us access to use add.cpp
    - So long as we link in the implementation of add (from add.cpp, which will be an add.o file), we can use the add function.

```cpp
// add_library/add.hpp
#pragma once

// Contains only function declarations
// Note: int add(int,int) is equivalent
//       but I think it's best to give
//       names to parameters in declaration.
int add(int a, int b);
```

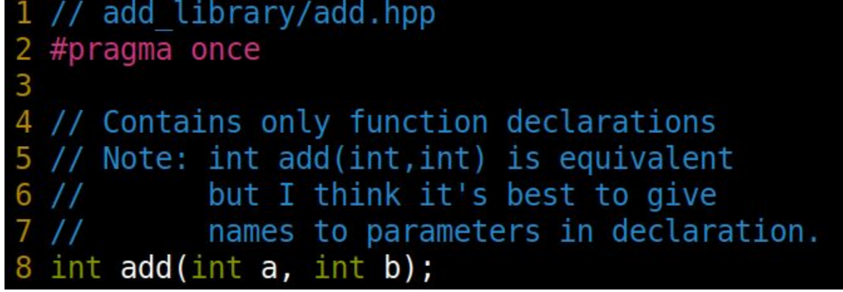
```cpp
// add_library/add.cpp

// Include 'add.hpp' and provide
// definition for our function.
#include "add.hpp"

int add(int a, int b){
  return a+b;
}
```

```cpp
// add_library/main.cpp
#include <iostream>
// include 'add.hpp' from local dir.
#include "add.hpp"

int main(){
  std::cout << add(7,2) << std::endl;
  return 0;
}
```

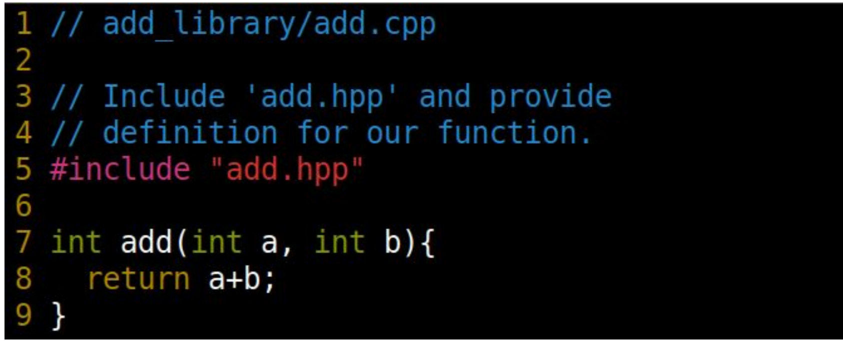
```
g++ add.cpp main.cpp -o prog
```

Creating Libraries (4/4)

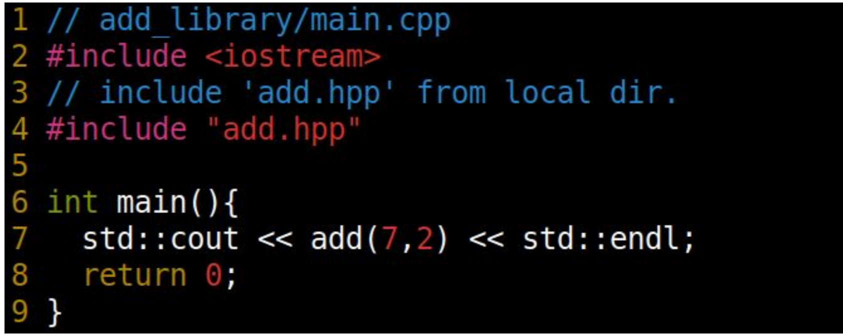
- To the right I'm going to reveal a complete program separated out into three files
  - The header (add.hpp)
    - Provides the forward declarations for our function
    - At the linking stage, we'll need an implementation before we can use it.
  - The source (add.cpp)
    - Provides the implementation
    - Note that the add.cpp also includes the add.hpp -- this is effectively the forward declaration being pasted in
  - Finally, the main.cpp
    - We #include "add.hpp" which gives us access to use add.cpp
    - So long as we link in the implementation of add (from add.cpp, which will be an add.o file), we can use the add function.



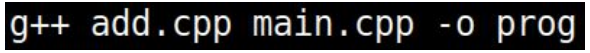
```cpp
// add_library/add.hpp
#pragma once

// Contains only function declarations
// Note: int add(int,int) is equivalent
//       but I think it's best to give
//       names to parameters in declaration.
int add(int a, int b);
```



```cpp
// add_library/add.cpp

// Include 'add.hpp' and provide
// definition for our function.
#include "add.hpp"

int add(int a, int b){
  return a+b;
}
```



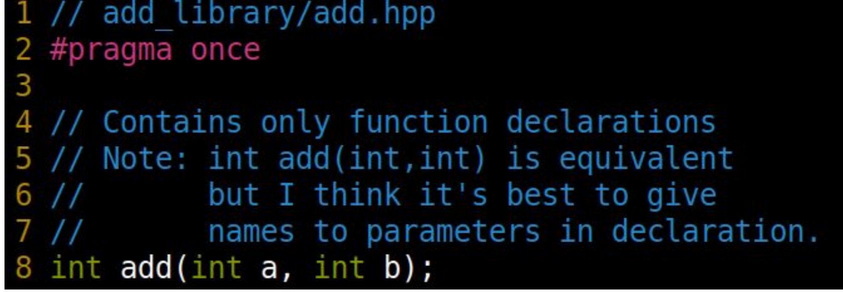
```cpp
// add_library/main.cpp
#include <iostream>
// include 'add.hpp' from local dir.
#include "add.hpp"

int main(){
  std::cout << add(7,2) << std::endl;
  return 0;
}
```



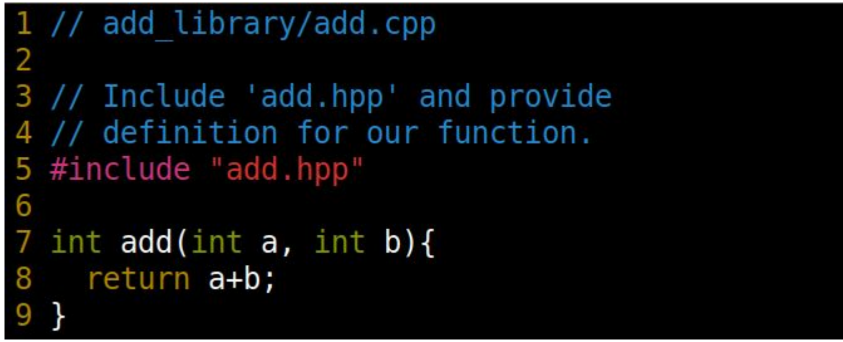
```
g++ add.cpp main.cpp -o prog
```

Our First Library of Functions

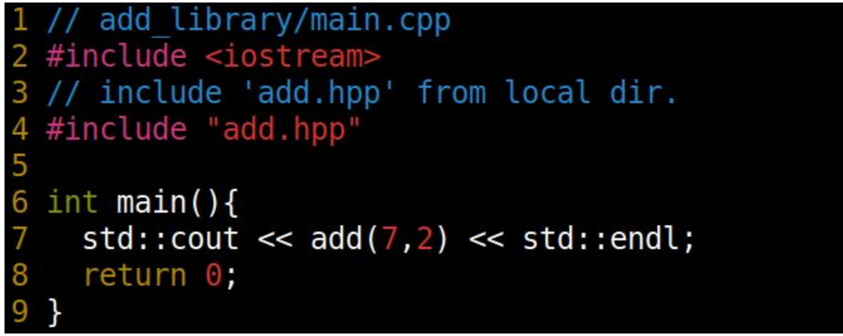
- We have effectively built a (tiny) library at this point
- Separating functions into separate files has a few advantages
    - Reuse your functions in other projects
        - (While maintaining and testing one version)
    - Hide your implementation details from users
    - Potentially speed up compilation
    - Utilize only the functionality you need by breaking up source into modules of related functions



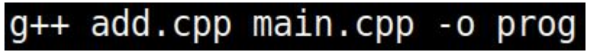
```cpp
// add_library/add.hpp
#pragma once

// Contains only function declarations
// Note: int add(int,int) is equivalent
//       but I think it's best to give
//       names to parameters in declaration.
int add(int a, int b);
```



```cpp
// add_library/add.cpp

// Include 'add.hpp' and provide
// definition for our function.
#include "add.hpp"

int add(int a, int b){
  return a+b;
}
```



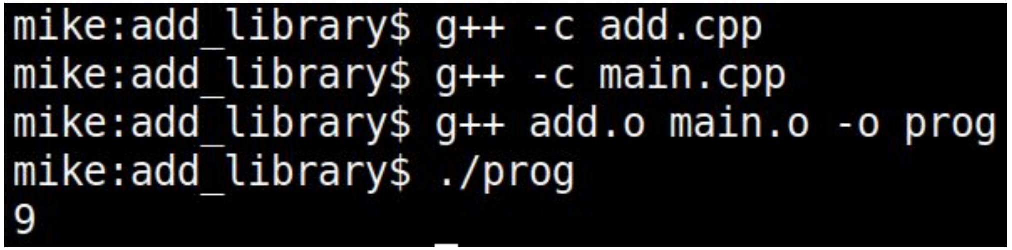
```cpp
// add_library/main.cpp
#include <iostream>
// include 'add.hpp' from local dir.
#include "add.hpp"

int main(){
  std::cout << add(7,2) << std::endl;
  return 0;
}
```



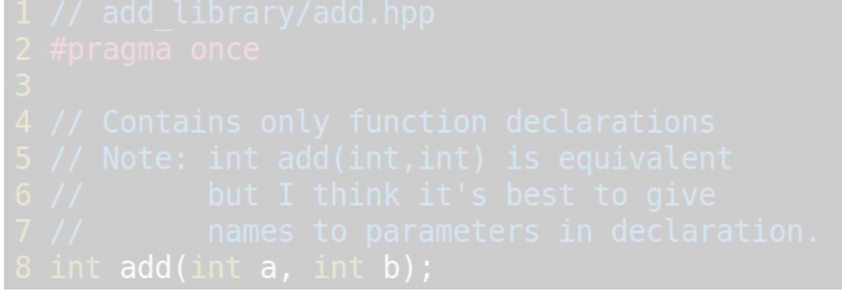
```
g++ add.cpp main.cpp -o prog
```

Separate Compilation of our Function Library

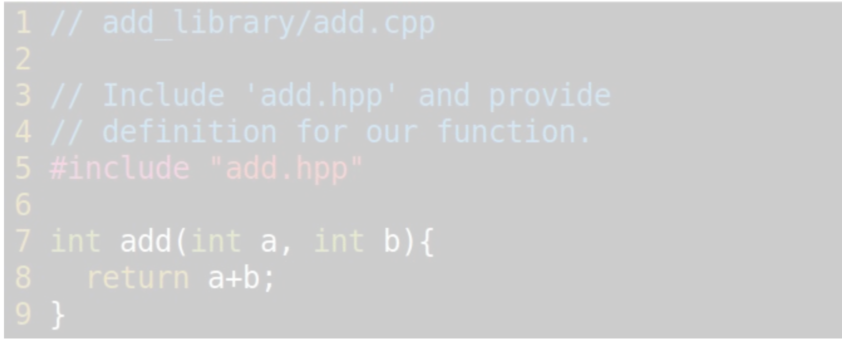
- Observe to the below an example of compiling our source (.cpp) files individually
- Here is an example of separate compilation and linking together the object files (.o) together to build our final executable.



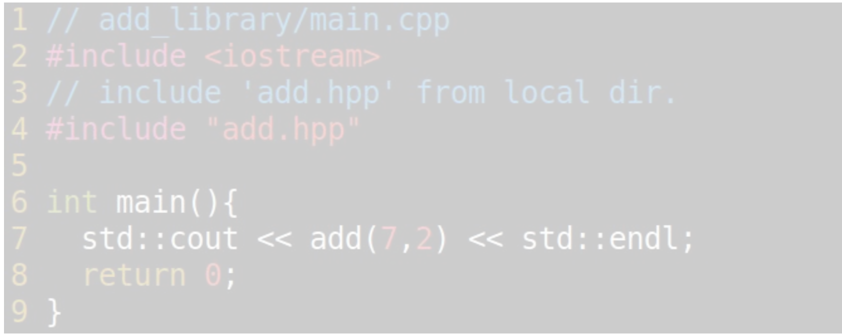
```
mike:add_library$ g++ -c add.cpp
mike:add_library$ g++ -c main.cpp
mike:add_library$ g++ add.o main.o -o prog
mike:add_library$ ./prog
9
```



```cpp
// add_library/add.hpp
#pragma once

// Contains only function declarations
// Note: int add(int,int) is equivalent
//       but I think it's best to give
//       names to parameters in declaration.
int add(int a, int b);
```



```cpp
// add_library/add.cpp

// Include 'add.hpp' and provide
// definition for our function.
#include "add.hpp"

int add(int a, int b){
  return a+b;
}
```

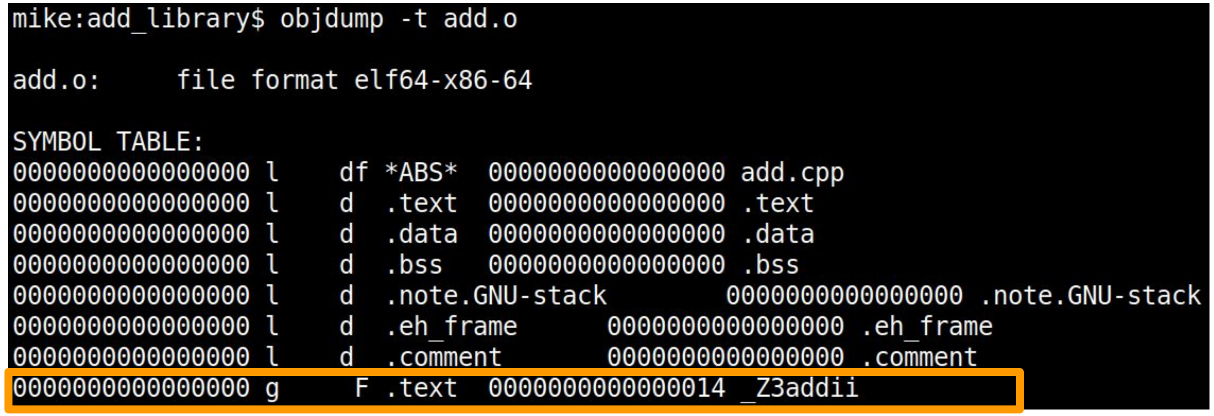


```cpp
// add_library/main.cpp
#include <iostream>
// include 'add.hpp' from local dir.
#include "add.hpp"

int main(){
  std::cout << add(7,2) << std::endl;
  return 0;
}
```

```
g++ add.cpp main.cpp -o prog
```

(Quick Detour) Taking a deeper look (1/2)

- Various tools allow us to 'inspect' object code such as objdump -- we can see the functions available to ensure they are there.
    - What I am displaying to you is we have a global ('g') function ('F') that has been identified.



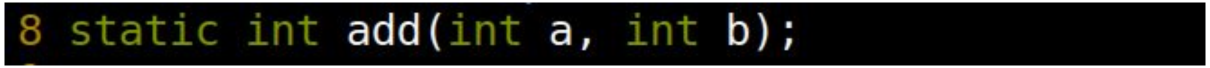
```
mike:add_library$ objdump -t add.o

add.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 add.cpp
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     F .text  0000000000000014 _Z3addii
```

Notes on Library Building:

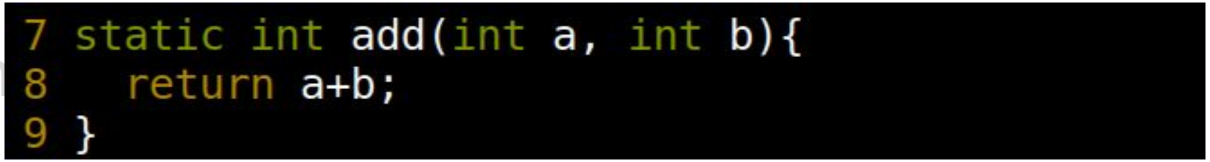
- If today is your first day with functions -- ignore these details
    - Bookmark this slide and revisit it at a later date when you build your first or second library :)

(Quick Detour) Takin

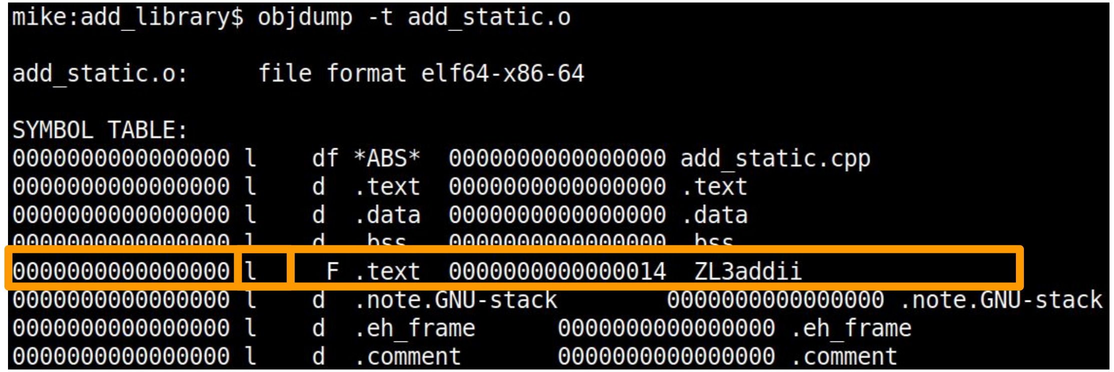
- If we add the ‘static’ qualifier to our function -- this effectively makes the function private to that source file.
  - That means this is only callable within the ‘.cpp’ file it is implemented in.
  - You can take a peek at the linker error on the bottom-right
- **Why?**
  - The reason you might want to do this, is if you have other functions that help you ultimately implement the function you want to expose to a user.



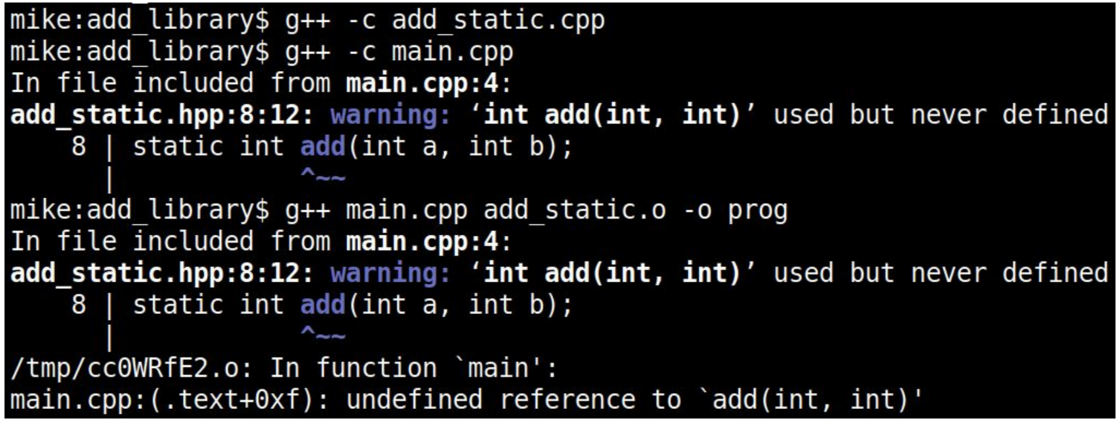
```
8 static int add(int a, int b);
```



```
7 static int add(int a, int b){
8   return a+b;
9 }
```



```
mike:add_library$ objdump -t add_static.o

add_static.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 add_static.cpp
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l     F .text  0000000000000014 _ZL3addii
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
```



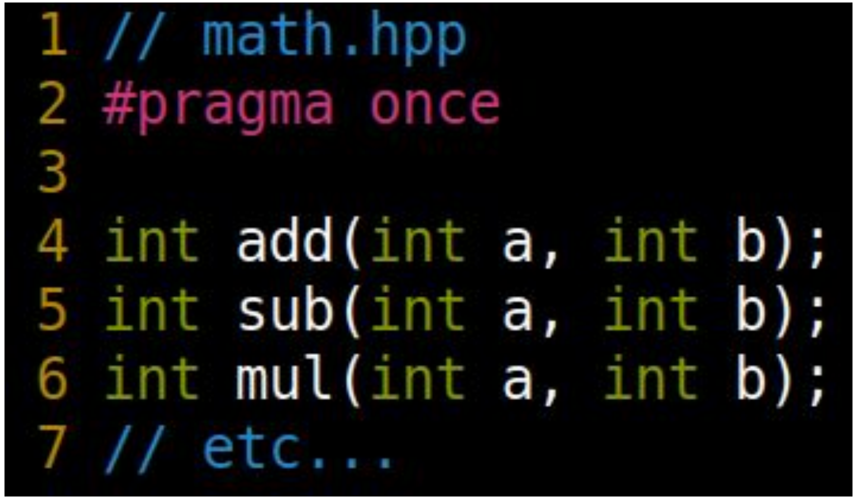
```
mike:add_library$ g++ -c add_static.cpp
mike:add_library$ g++ -c main.cpp
In file included from main.cpp:4:
add_static.hpp:8:12: warning: ‘int add(int, int)’ used but never defined
    8 | static int add(int a, int b);
      |            ^~~
mike:add_library$ g++ main.cpp add_static.o -o prog
In file included from main.cpp:4:
add_static.hpp:8:12: warning: ‘int add(int, int)’ used but never defined
    8 | static int add(int a, int b);
      |            ^~~
/tmp/cc0WRfE2.o: In function `main':
main.cpp:(.text+0xf): undefined reference to `add(int, int)'
```

Functions

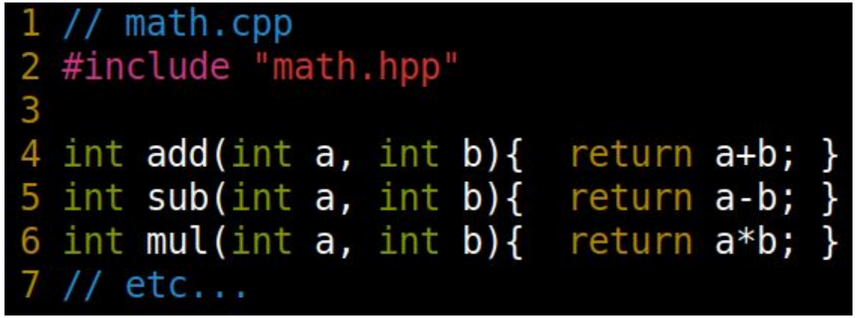
How might we group related functions together?

Grouping Functions Together (1/3)

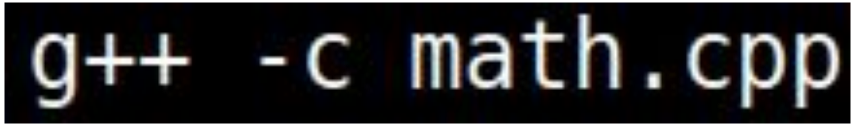
- At some point you'll want to collect related files into a single source file -- that is probably a good idea!
  - So at the least, we can group files together in one file
- A C-like strategy is to add a uniform prefix to each function name.
  - That is possibly reasonable if you foresee your functions being used in many different languages.



```cpp
// math.hpp
#pragma once

int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
// etc...
```



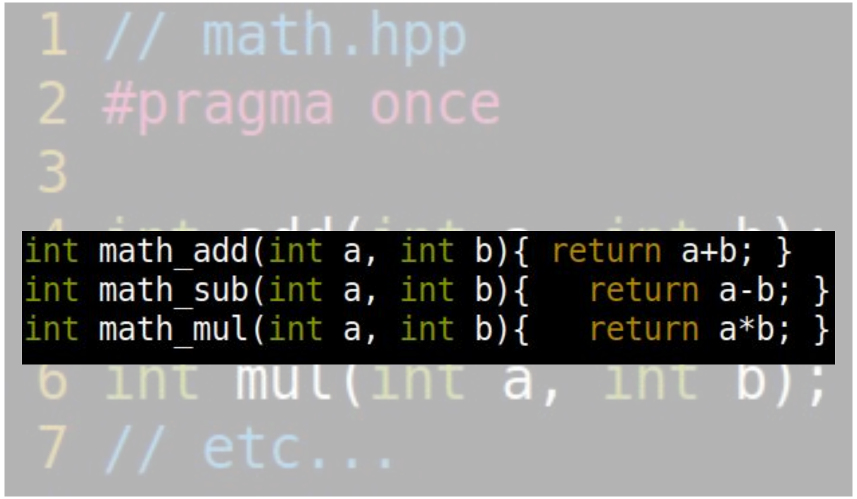
```cpp
// math.cpp
#include "math.hpp"

int add(int a, int b){  return a+b; }
int sub(int a, int b){  return a-b; }
int mul(int a, int b){  return a*b; }
// etc...
```



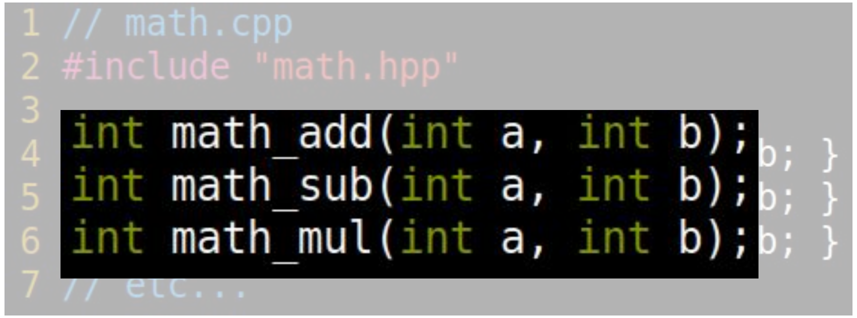
```
g++ -c math.cpp
```

Grouping Functions Together (2/3)

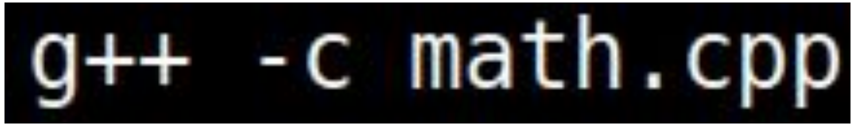
- At some point you'll want to collect related files into a single source file -- that is probably a good idea!
  - So at the least, we can group files together in one file
- A C-like strategy is to add a uniform prefix to each function name.
  - That is possibly reasonable if you foresee your functions being used in many different languages.



```
// math.hpp
#pragma once

int math_add(int a, int b){ return a+b; }
int math_sub(int a, int b){   return a-b; }
int math_mul(int a, int b){   return a*b; }
int mul(int a, int b);
// etc...
```



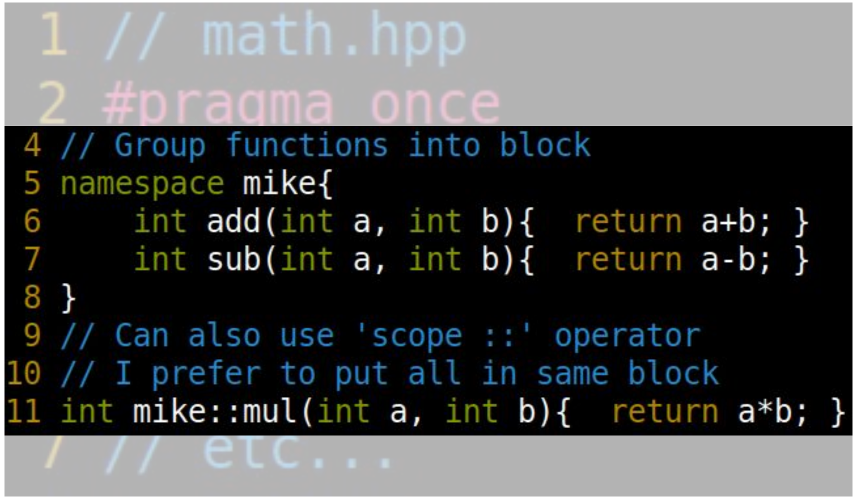
```
// math.cpp
#include "math.hpp"

int math_add(int a, int b);
int math_sub(int a, int b);
int math_mul(int a, int b);
// etc...
```



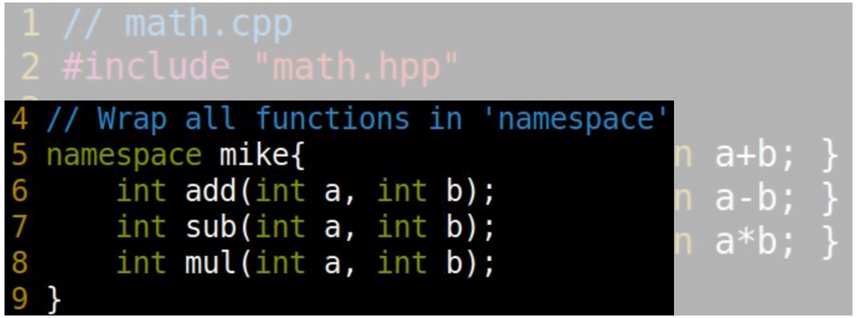
```
g++ -c math.cpp
```

Grouping Functions Together (3/3)

- A better C++ approach is to group functions together in a namespace
    - This makes it easy to avoid naming collisions
        - (Someone else probably wrote an ‘add’ function at some point in a large enough project)
    - Refactoring becomes easier as well.
        - If nested namespaces were to get too long -- at a local scope you can use:
            - `using namespace mike;`
            - (Note: I recommend avoiding ‘using namespace’ at a global scope)

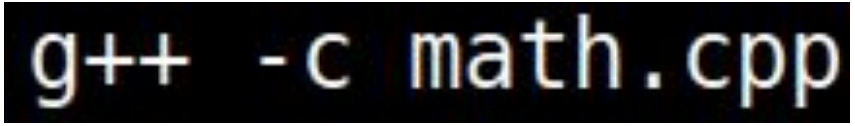


```
// math.hpp
#pragma once
// Group functions into block
namespace mike{
    int add(int a, int b){  return a+b; }
    int sub(int a, int b){  return a-b; }
}
// Can also use 'scope ::' operator
// I prefer to put all in same block
int mike::mul(int a, int b){  return a*b; }
// etc...
```



```
// math.cpp
#include "math.hpp"
// Wrap all functions in 'namespace'
namespace mike{
    int add(int a, int b);
    int sub(int a, int b);
    int mul(int a, int b);
}
```

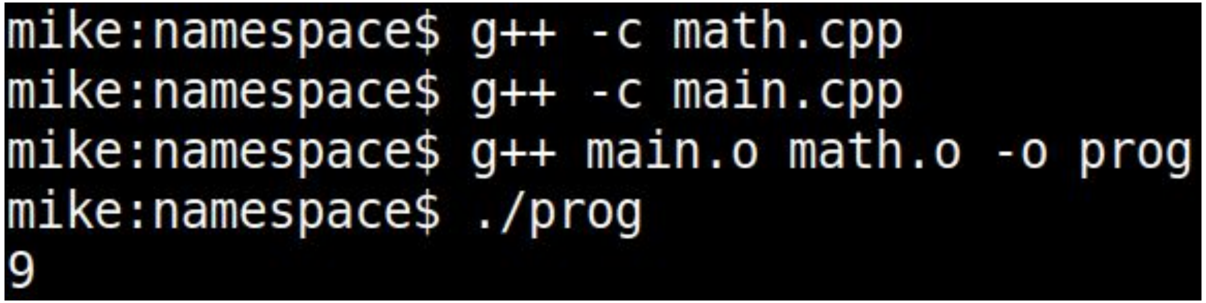
```
n a+b; }
n a-b; }
n a*b; }
```



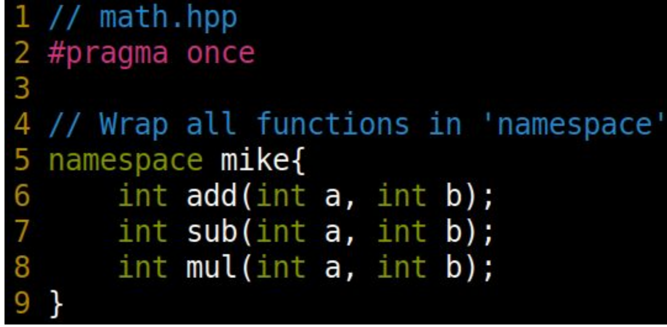
```
g++ -c math.cpp
```

(Aside) Example Usage

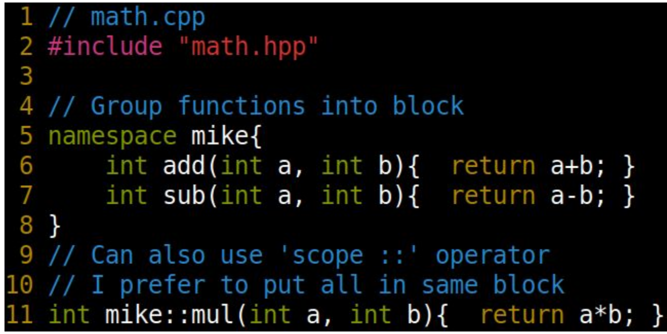
- And here’s a full example if you like:



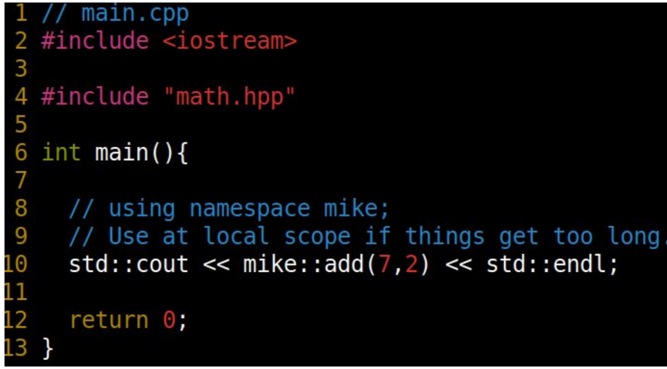
```
mike:namespace$ g++ -c math.cpp
mike:namespace$ g++ -c main.cpp
mike:namespace$ g++ main.o math.o -o prog
mike:namespace$ ./prog
9
```



```cpp
// math.hpp
#pragma once

// Wrap all functions in 'namespace'
namespace mike{
    int add(int a, int b);
    int sub(int a, int b);
    int mul(int a, int b);
}
```



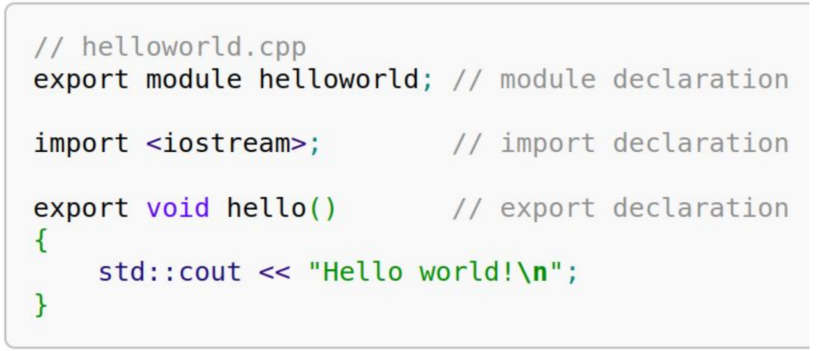
```cpp
// math.cpp
#include "math.hpp"

// Group functions into block
namespace mike{
    int add(int a, int b){  return a+b; }
    int sub(int a, int b){  return a-b; }
}
// Can also use 'scope ::' operator
// I prefer to put all in same block
int mike::mul(int a, int b){  return a*b; }
```



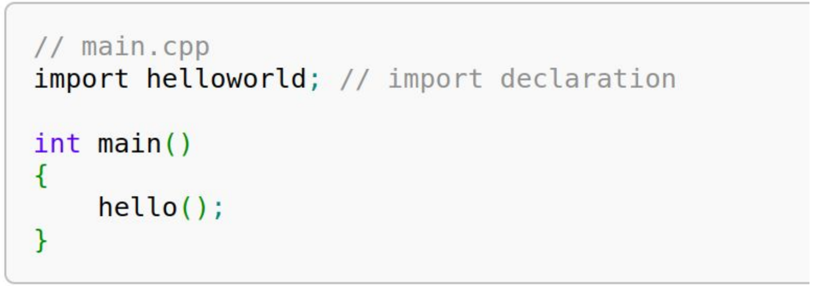
```cpp
// main.cpp
#include <iostream>

#include "math.hpp"

int main(){

  // using namespace mike;
  // Use at local scope if things get too long
  std::cout << mike::add(7,2) << std::endl;

  return 0;
}
```

(Aside: Modules)

- Modules in C++ 20 *should* help resolve some of the organization of source files.
  - Note: ‘export’ appears to be a better way than the previous ‘static’ trick I showed you to determine what functions are exposed.
- Note: I am not yet a C++ modules expert, but I will learn more as compiler support continues advancing.



```
// helloworld.cpp
export module helloworld; // module declaration

import <iostream>;         // import declaration

export void hello()        // export declaration
{
    std::cout << "Hello world!\n";
}
```



```
// main.cpp
import helloworld; // import declaration

int main()
{
    hello();
}
```

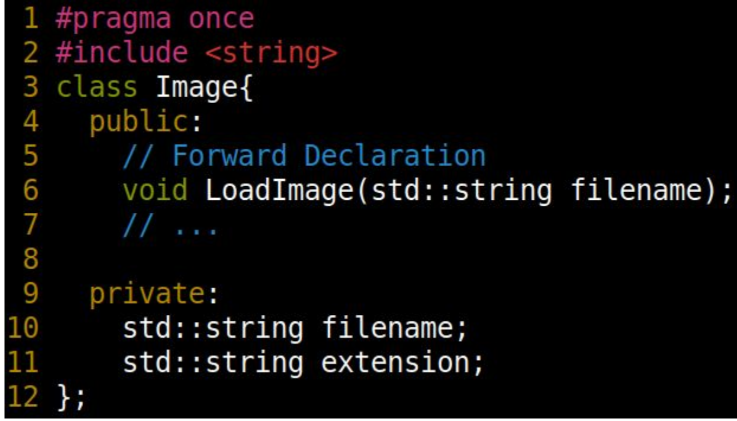
Example from:
https://en.cppreference.com/w/cpp/language/modules

Function (Member Functions)

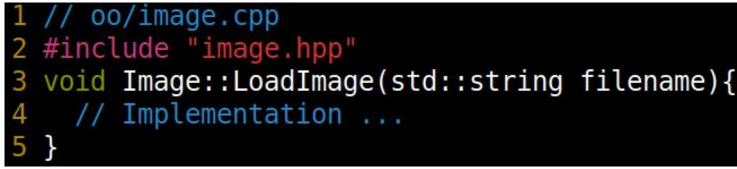
Another way to ‘group’ functions -- Object-Oriented Programming

Object-Oriented (Actions + Attributes)

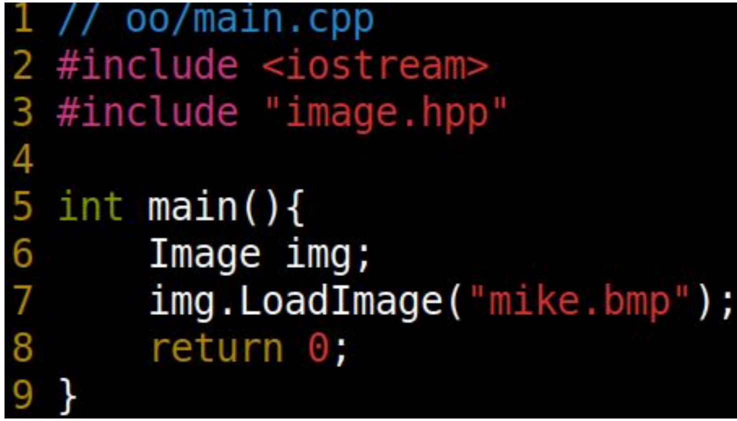
- Beyond grouping functions into namespaces
- We can **group related functions and data together** to form a new user-defined data type -- an object
  - Typically we call the functions ‘**member functions**’ (other languages may call these ‘methods’)
    - Member functions perform the ‘work’ based on arguments provides, and possibly internal state (member variables(
      - Note: member variables sometimes also called either fields or attributes.



```cpp
#pragma once
#include <string>
class Image{
  public:
    // Forward Declaration
    void LoadImage(std::string filename);
    // ...

  private:
    std::string filename;
    std::string extension;
};
```



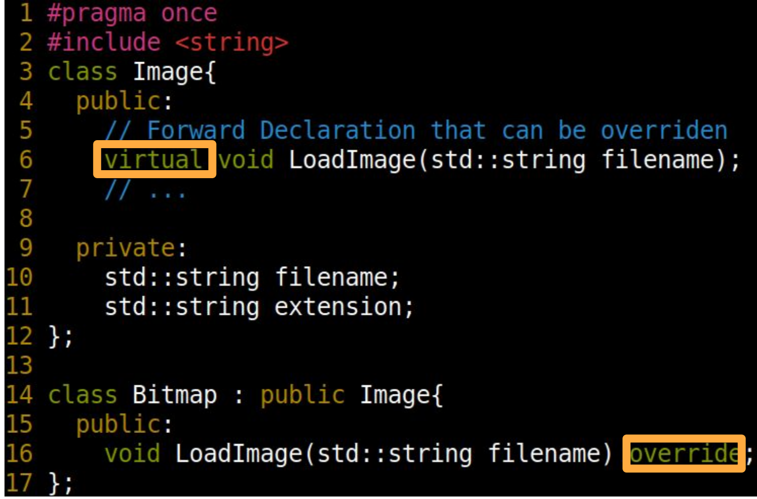
```cpp
// oo/image.cpp
#include "image.hpp"
void Image::LoadImage(std::string filename){
  // Implementation ...
}
```



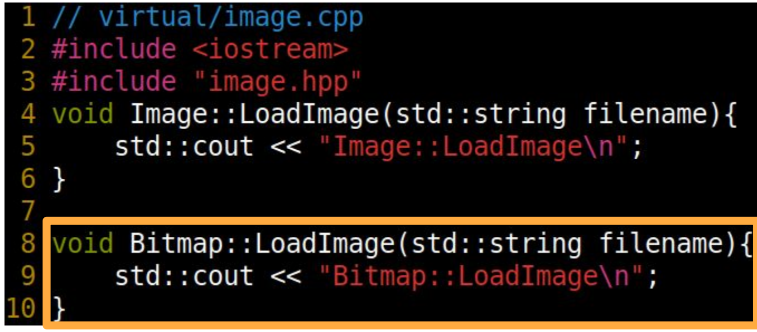
```cpp
// oo/main.cpp
#include <iostream>
#include "image.hpp"

int main(){
    Image img;
    img.LoadImage("mike.bmp");
    return 0;
}
```

Virtual Functions (or Virtual Member Functions) (1/3)

- Member Functions role become interesting when we start talking about inheritance.
- Member Functions can be ‘overridden’ in derived classes.
    - Observe the **‘virtual’** keyword on line ‘6’ signaling that the function may be overridden
    - Observe the **‘override’** keyword at line ‘16’ which specifically indicates a function will be overridden.
    - In the image.cpp file (bottom-right image) you will then see the implementation provided for the new derived class



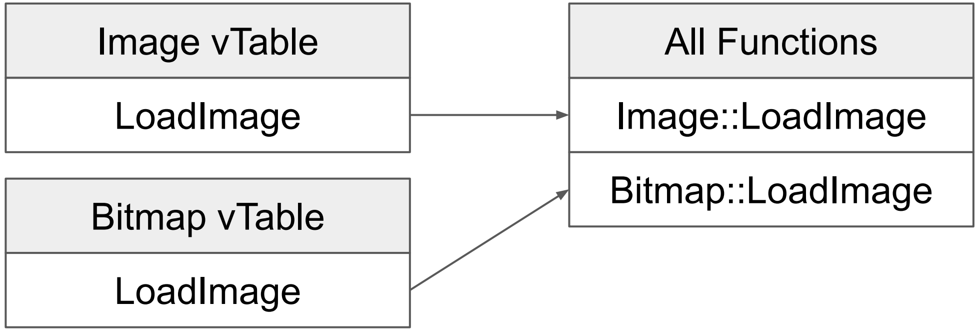
```
 1 #pragma once
 2 #include <string>
 3 class Image{
 4   public:
 5     // Forward Declaration that can be overriden
 6     virtual void LoadImage(std::string filename);
 7     // ...
 8
 9   private:
10     std::string filename;
11     std::string extension;
12 };
13
14 class Bitmap : public Image{
15   public:
16     void LoadImage(std::string filename) override;
17 };
```

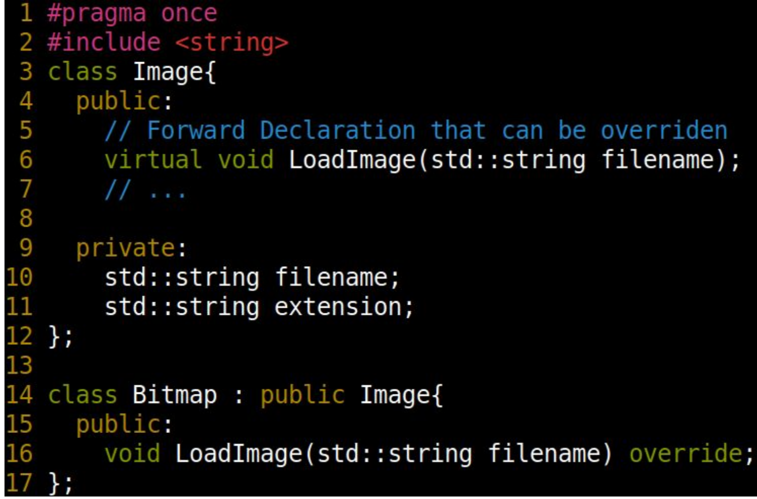


```
 1 // virtual/image.cpp
 2 #include <iostream>
 3 #include "image.hpp"
 4 void Image::LoadImage(std::string filename){
 5     std::cout << "Image::LoadImage\n";
 6 }
 7
 8 void Bitmap::LoadImage(std::string filename){
 9     std::cout << "Bitmap::LoadImage\n";
10 }
```

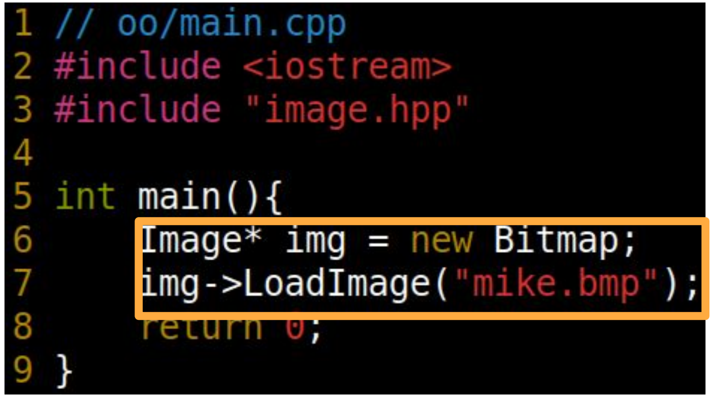
Virtual Functions (or Virtual Member Functions) (2/3)

- When calling a specific '`::LoadImage`' member function, the correct implementation will be called based on the allocated object.
    - Classes and structs containing virtual functions have a 'virtual table' of pointers to functions.





```cpp
#pragma once
#include <string>
class Image{
  public:
    // Forward Declaration that can be overriden
    virtual void LoadImage(std::string filename);
    // ...

  private:
    std::string filename;
    std::string extension;
};

class Bitmap : public Image{
  public:
    void LoadImage(std::string filename) override;
};
```



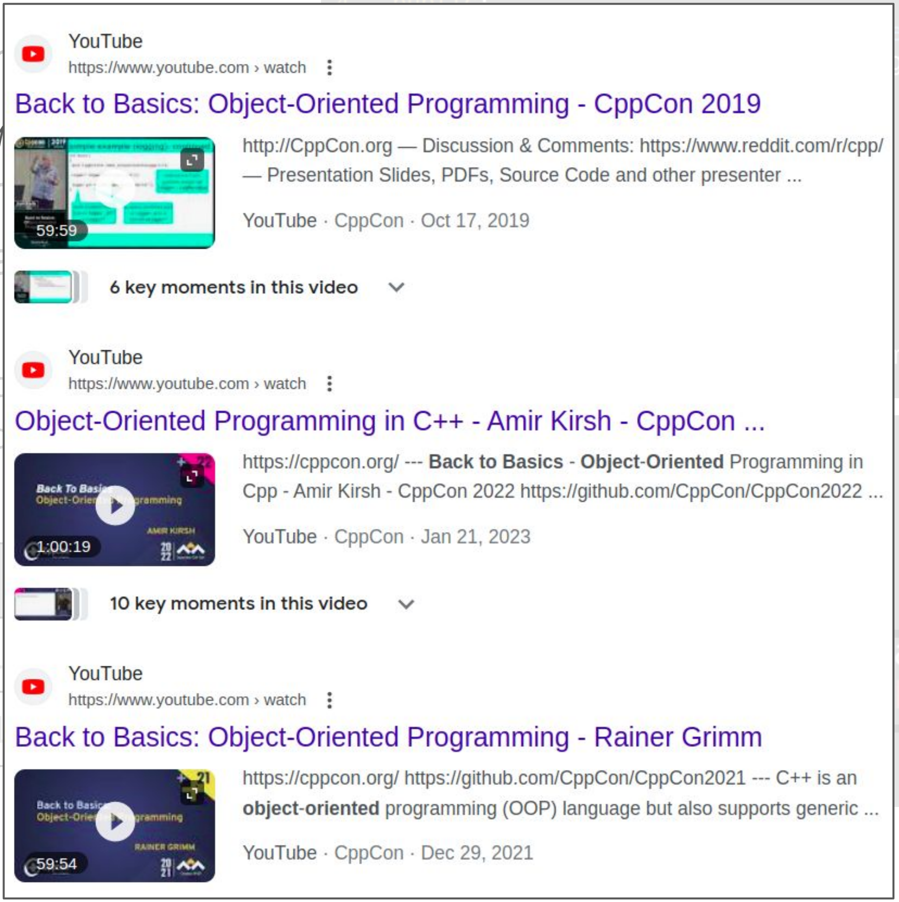
```cpp
// oo/main.cpp
#include <iostream>
#include "image.hpp"

int main(){
    Image* img = new Bitmap;
    img->LoadImage("mike.bmp");
    return 0;
}
```

Virtual Functions (or Virtual Member

For a full treatment of Object-Oriented Programming check out the following videos or otherwise my C++ collection on YouTube

Going Further:

- Pure Virtual Functions
  - (For interfaces)
- Friend functions
- Static member functions


YouTube
https://www.youtube.com › watch
Back to Basics: Object-Oriented Programming - CppCon 2019
http://CppCon.org — Discussion & Comments: https://www.reddit.com/r/cpp/ — Presentation Slides, PDFs, Source Code and other presenter ...
YouTube · CppCon · Oct 17, 2019
6 key moments in this video

YouTube
https://www.youtube.com › watch
Object-Oriented Programming in C++ - Amir Kirsh - CppCon ...
https://cppcon.org/ --- **Back to Basics** - **Object-Oriented** Programming in Cpp - Amir Kirsh - CppCon 2022 https://github.com/CppCon/CppCon2022 ...
YouTube · CppCon · Jan 21, 2023
10 key moments in this video

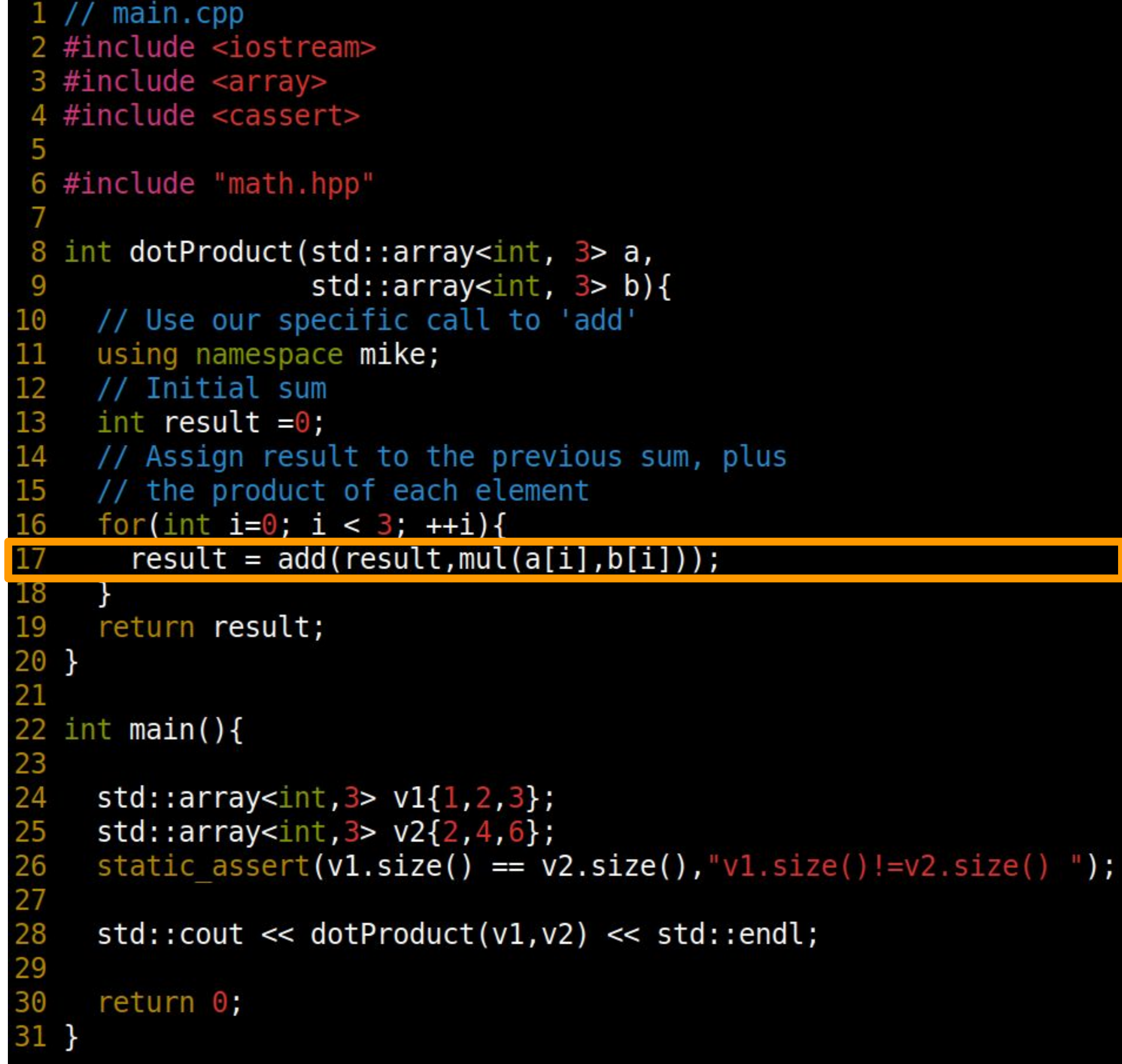
YouTube
https://www.youtube.com › watch
Back to Basics: Object-Oriented Programming - Rainer Grimm
https://cppcon.org/ https://github.com/CppCon/CppCon2021 --- C++ is an **object-oriented** programming (OOP) language but also supports generic ...
YouTube · CppCon · Dec 29, 2021

Function Composition

Functions are our building blocks

Functions Compose (1/3)

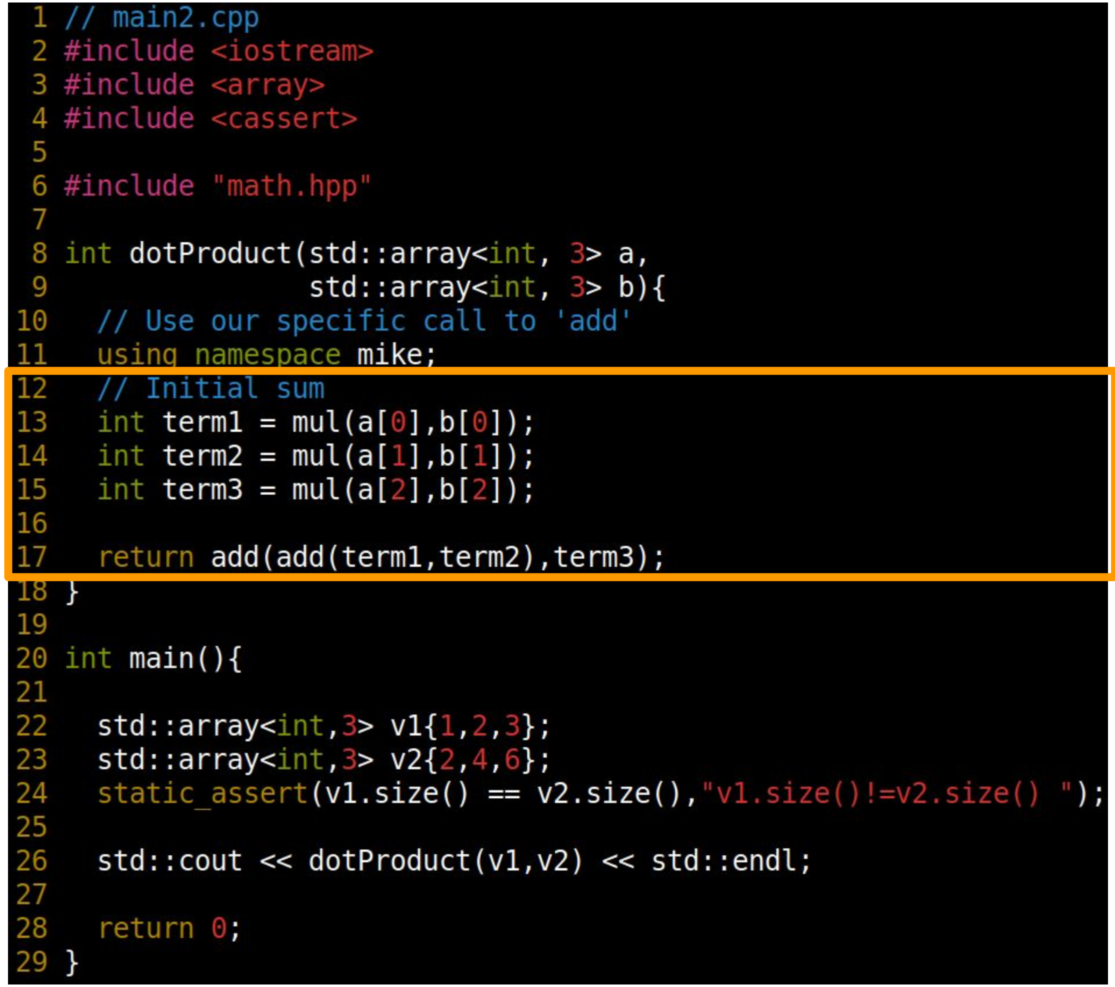
- Here's a *somewhat* silly example of composing with functions.
  - That is to say, we are using the result of one function as an argument into another.
  - We know we already have '+' and '*' operators for primitive types
    - But this type of code is still useful
    - What if 'add' and 'mul' check for integer overflow for example?



```cpp
// main.cpp
#include <iostream>
#include <array>
#include <cassert>

#include "math.hpp"

int dotProduct(std::array<int, 3> a,
               std::array<int, 3> b){
  // Use our specific call to 'add'
  using namespace mike;
  // Initial sum
  int result =0;
  // Assign result to the previous sum, plus
  // the product of each element
  for(int i=0; i < 3; ++i){
    result = add(result,mul(a[i],b[i]));
  }
  return result;
}

int main(){

  std::array<int,3> v1{1,2,3};
  std::array<int,3> v2{2,4,6};
  static_assert(v1.size() == v2.size(),"v1.size()!=v2.size() ");

  std::cout << dotProduct(v1,v2) << std::endl;

  return 0;
}
```

Functions Compose (2/3)

- Many ways to think about this implementation
  - The point being however -- hopefully you think in terms of **functions as building blocks that you can compose together**
- Brief aside -- for advanced users -- yes, you can use a variadic template and evaluate this at compile-time in that manner.



```cpp
// main2.cpp
#include <iostream>
#include <array>
#include <cassert>

#include "math.hpp"

int dotProduct(std::array<int, 3> a,
               std::array<int, 3> b){
  // Use our specific call to 'add'
  using namespace mike;
  // Initial sum
  int term1 = mul(a[0],b[0]);
  int term2 = mul(a[1],b[1]);
  int term3 = mul(a[2],b[2]);

  return add(add(term1,term2),term3);
}

int main(){

  std::array<int,3> v1{1,2,3};
  std::array<int,3> v2{2,4,6};
  static_assert(v1.size() == v2.size(),"v1.size()!=v2.size() ");

  std::cout << dotProduct(v1,v2) << std::endl;

  return 0;
}
```

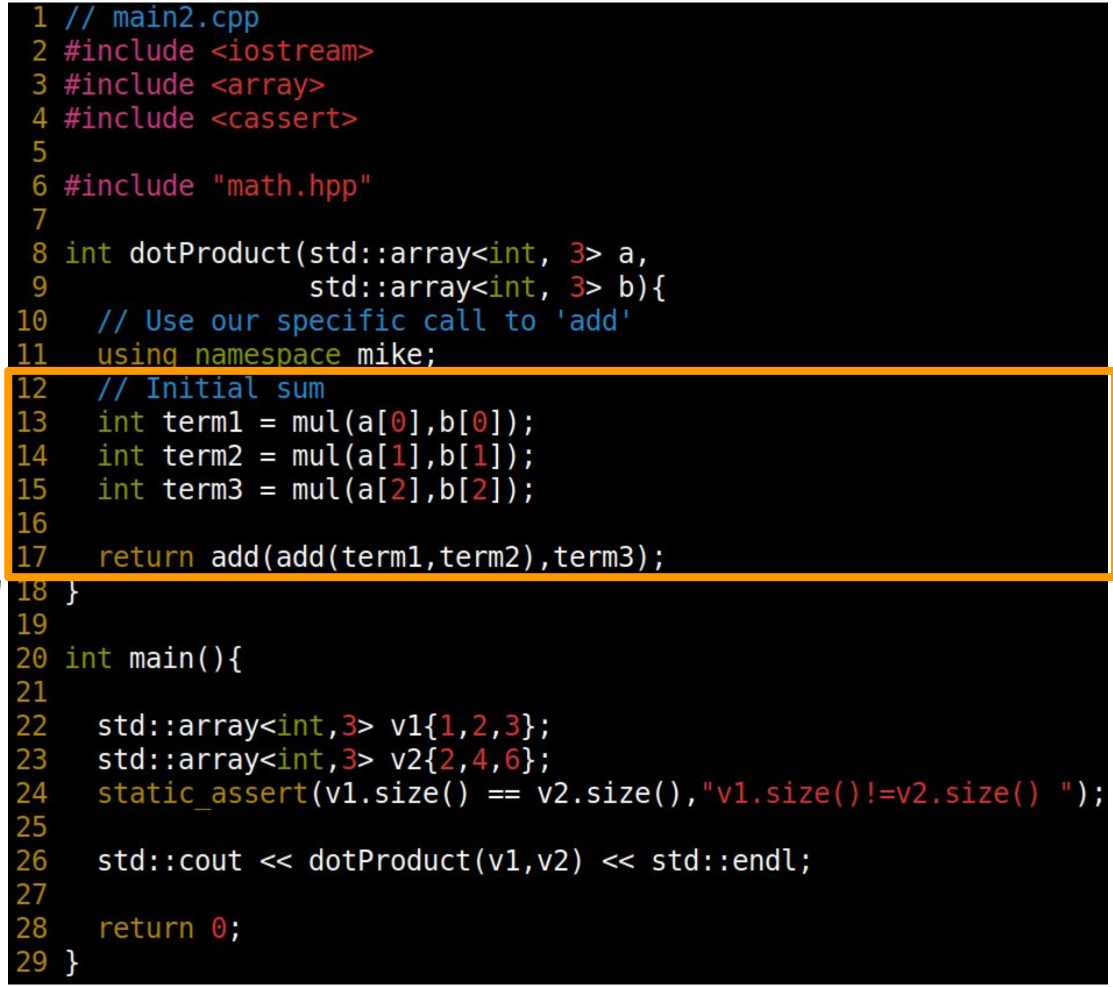
Functions Compose (3/3)

Something that becomes more apparent here -- is that our functions are very simple.

Simple -- but it also looks like we *may* have all the information at compile-time to compute the result.

C++ 11 introduced just that feature -- constexpr

compile-time.



```cpp
// main2.cpp
#include <iostream>
#include <array>
#include <cassert>

#include "math.hpp"

int dotProduct(std::array<int, 3> a,
               std::array<int, 3> b){
  // Use our specific call to 'add'
  using namespace mike;
  // Initial sum
  int term1 = mul(a[0],b[0]);
  int term2 = mul(a[1],b[1]);
  int term3 = mul(a[2],b[2]);

  return add(add(term1,term2),term3);
}

int main(){

  std::array<int,3> v1{1,2,3};
  std::array<int,3> v2{2,4,6};
  static_assert(v1.size() == v2.size(),"v1.size()!=v2.size() ");

  std::cout << dotProduct(v1,v2) << std::endl;

  return 0;
}
```

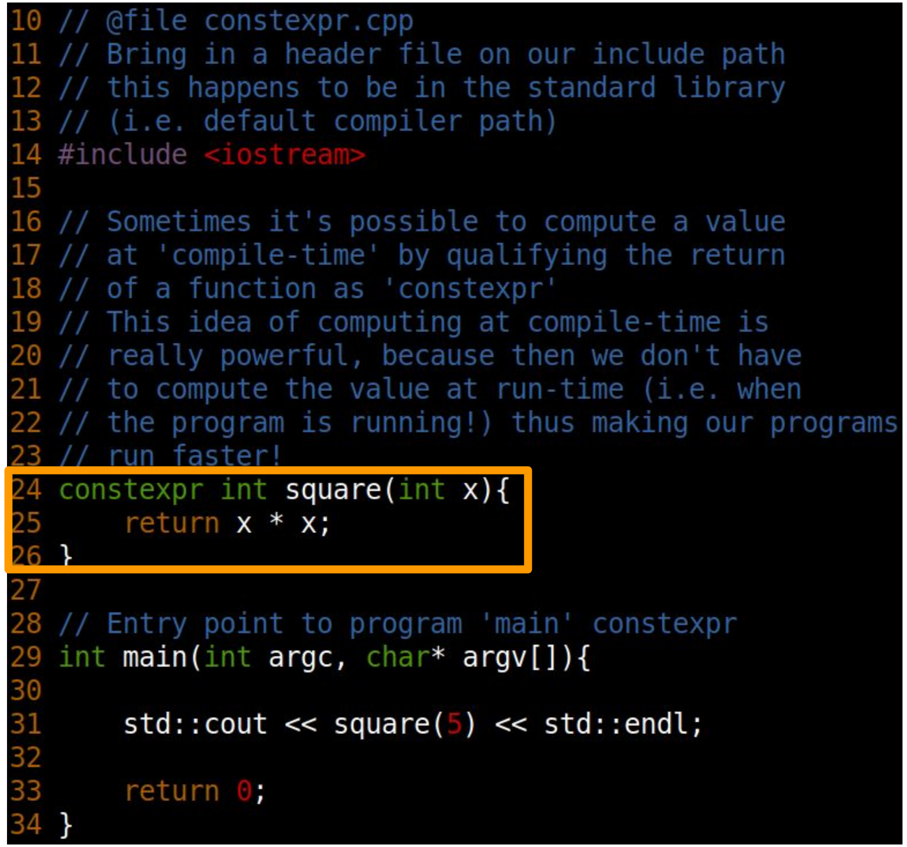
constexpr Functions

Compute at compile-time

https://en.cppreference.com/w/cpp/language/constexpr

constexpr return value

- We can **qualify our return type** of a function with **'constexpr'**
- This makes it possible (*but not necessarily guaranteed*) that we can evaluate an expression (i.e. return value of a function, a computation, etc.) before your code runs!
  - From cppreference
  - *"The constexpr specifier declares that it is possible to evaluate the value of the function or variable at compile time. Such variables and functions can then be used where only compile time [constant expressions](#) are allowed (provided that appropriate function arguments are given)."*



```
// @file constexpr.cpp
// Bring in a header file on our include path
// this happens to be in the standard library
// (i.e. default compiler path)
#include <iostream>

// Sometimes it's possible to compute a value
// at 'compile-time' by qualifying the return
// of a function as 'constexpr'
// This idea of computing at compile-time is
// really powerful, because then we don't have
// to compute the value at run-time (i.e. when
// the program is running!) thus making our programs
// run faster!
constexpr int square(int x){
    return x * x;
}

// Entry point to program 'main' constexpr
int main(int argc, char* argv[]){

    std::cout << square(5) << std::endl;

    return 0;
}
```

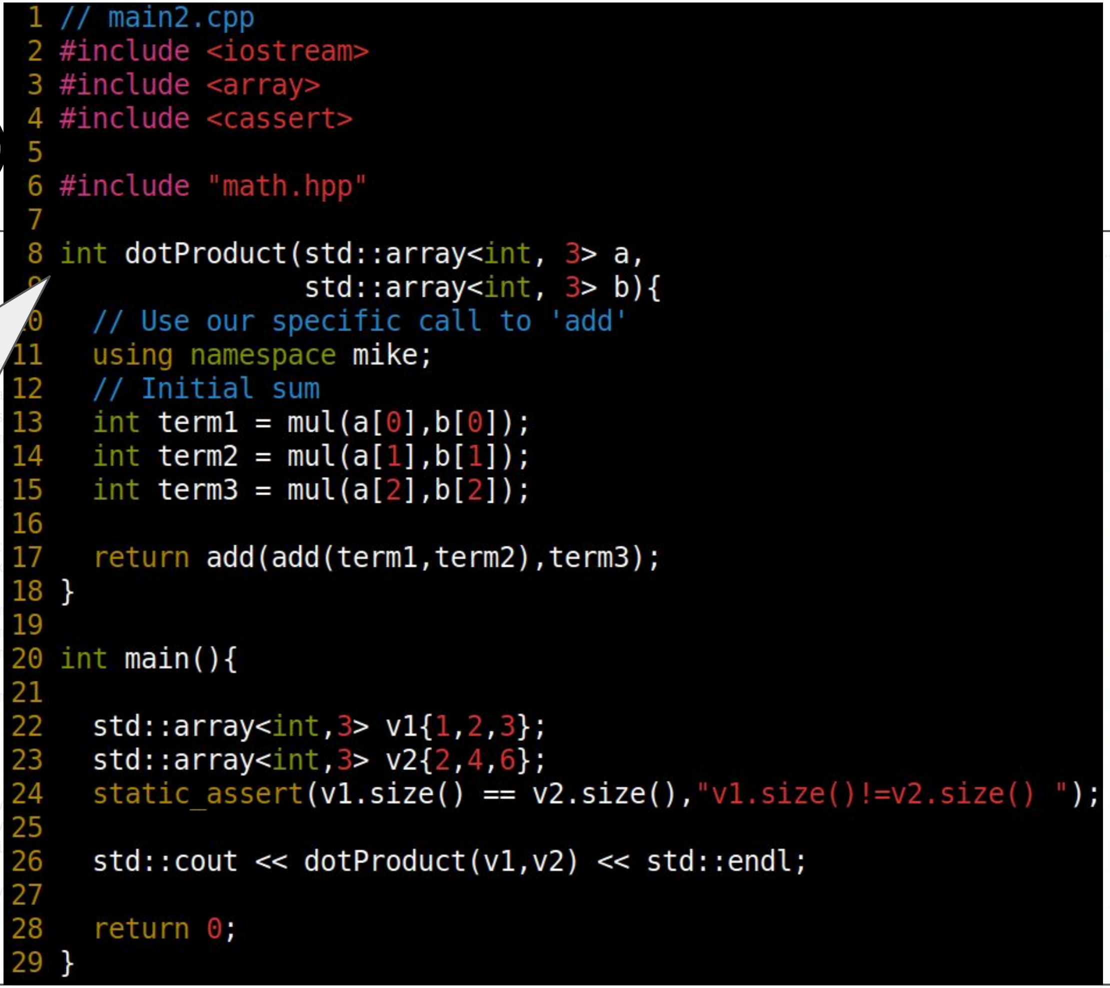
https://godbolt.org/z/hhxcWj6Gq

constexpr functions (0/3)

We can actually improve this particular function we previously looked at with our new ‘constexpr’ knowledge

returns 28.

- The key is the ‘constexpr’ used on the return type qualifier.

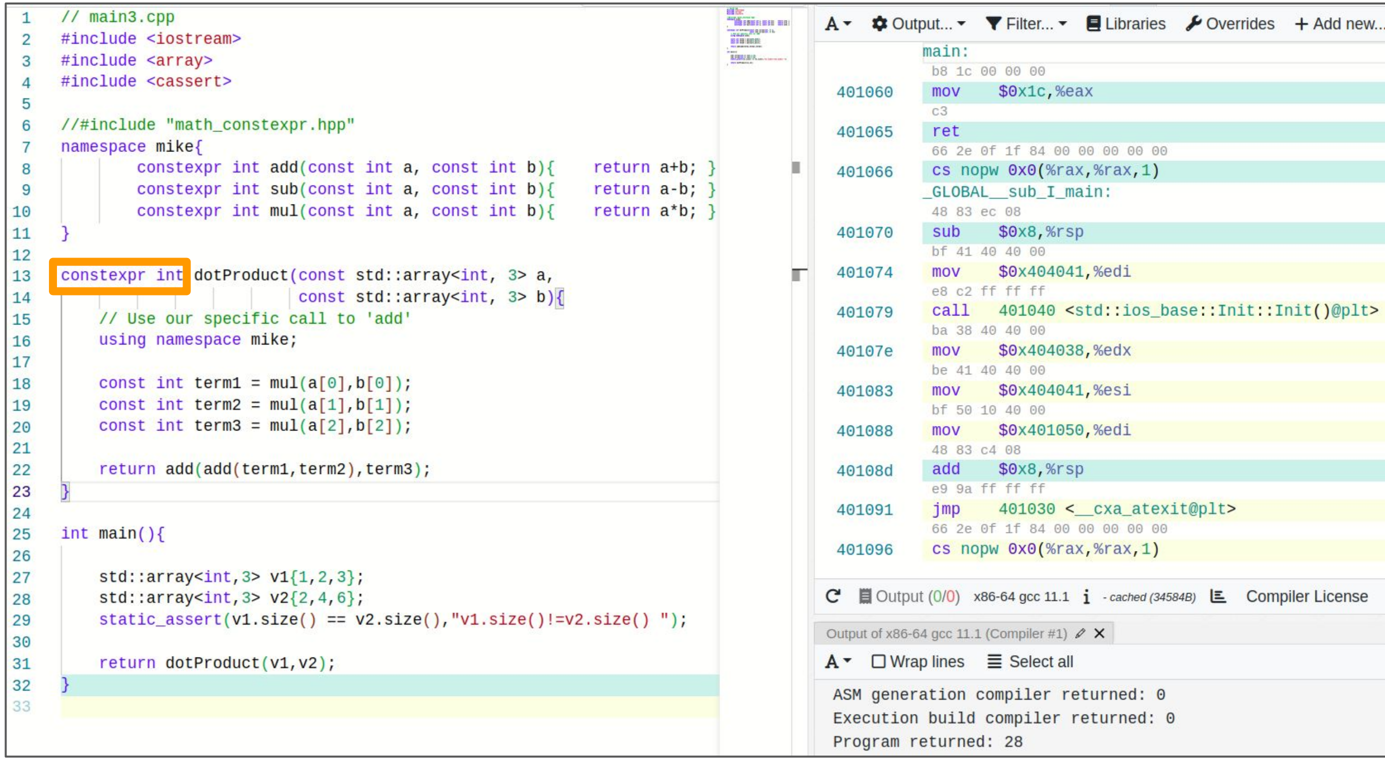


```cpp
// main2.cpp
#include <iostream>
#include <array>
#include <cassert>

#include "math.hpp"

int dotProduct(std::array<int, 3> a,
                     std::array<int, 3> b){
  // Use our specific call to 'add'
  using namespace mike;
  // Initial sum
  int term1 = mul(a[0],b[0]);
  int term2 = mul(a[1],b[1]);
  int term3 = mul(a[2],b[2]);

  return add(add(term1,term2),term3);
}

int main(){

  std::array<int,3> v1{1,2,3};
  std::array<int,3> v2{2,4,6};
  static_assert(v1.size() == v2.size(),"v1.size()!=v2.size() ");

  std::cout << dotProduct(v1,v2) << std::endl;

  return 0;
}
```

https://godbolt.org/z/hhxcWj6Gq

constexpr functions (1/3)

- We can evaluate some functions at compile-time with ‘constexpr’
  - This effectively makes our program to the right a ‘no-op’ and it just returns 28.
- The key is the ‘constexpr’ used on the return type qualifier.



```
// main3.cpp
#include <iostream>
#include <array>
#include <cassert>

//#include "math_constexpr.hpp"
namespace mike{
        constexpr int add(const int a, const int b){    return a+b; }
        constexpr int sub(const int a, const int b){    return a-b; }
        constexpr int mul(const int a, const int b){    return a*b; }
}

constexpr int dotProduct(const std::array<int, 3> a,
                         const std::array<int, 3> b){
    // Use our specific call to 'add'
    using namespace mike;

    const int term1 = mul(a[0],b[0]);
    const int term2 = mul(a[1],b[1]);
    const int term3 = mul(a[2],b[2]);

    return add(add(term1,term2),term3);
}

int main(){

    std::array<int,3> v1{1,2,3};
    std::array<int,3> v2{2,4,6};
    static_assert(v1.size() == v2.size(),"v1.size()!=v2.size() ");

    return dotProduct(v1,v2);
}
```

```
        main:
         b8 1c 00 00 00
401060   mov    $0x1c,%eax
         c3
401065   ret
         66 2e 0f 1f 84 00 00 00 00 00
401066   cs nopw 0x0(%rax,%rax,1)
        _GLOBAL__sub_I_main:
         48 83 ec 08
401070   sub    $0x8,%rsp
         bf 41 40 40 00
401074   mov    $0x404041,%edi
         e8 c2 ff ff ff
401079   call   401040 <std::ios_base::Init::Init()@plt>
         ba 38 40 40 00
40107e   mov    $0x404038,%edx
         be 41 40 40 00
401083   mov    $0x404041,%esi
         bf 50 10 40 00
401088   mov    $0x401050,%edi
         48 83 c4 08
40108d   add    $0x8,%rsp
         e9 9a ff ff ff
401091   jmp    401030 <__cxa_atexit@plt>
         66 2e 0f 1f 84 00 00 00 00 00
401096   cs nopw 0x0(%rax,%rax,1)
```

Output (0/0) x86-64 gcc 11.1 - cached (34584B) Compiler License
Output of x86-64 gcc 11.1 (Compiler #1)
A Wrap lines Select all

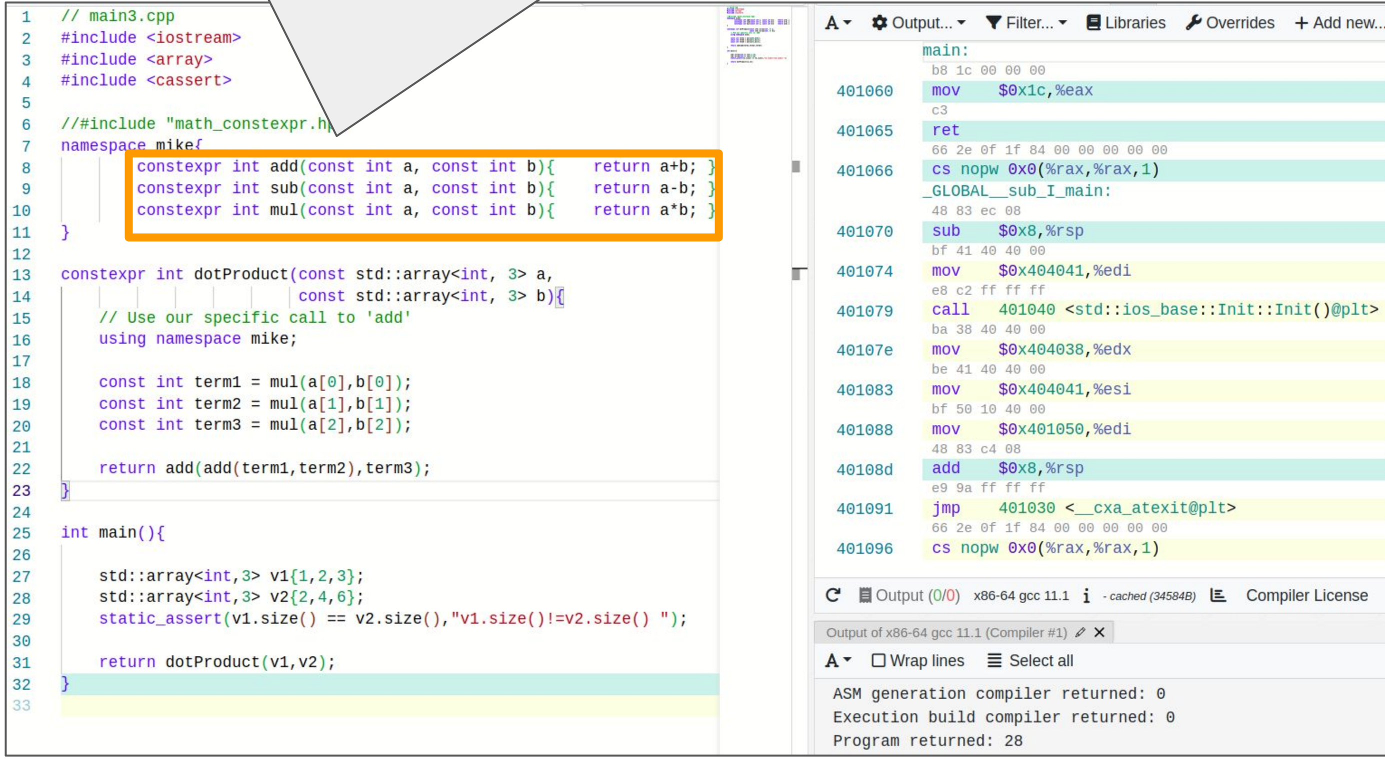
```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 28
```

https://godbolt.org/z/hhxcWj6Gq

constexpr functi

- We can evaluate some functions at compile-time with ‘constexpr’
  - This effectively makes our program to the right a ‘no-op’ and it just returns 28.
- The key is the ‘constexpr’ used on the return type qualifier.

The purpose of this slide is again to show you -- if I break my program ino small composable pieces, it becomes more clear when I can make something constexpr.



```cpp
// main3.cpp
#include <iostream>
#include <array>
#include <cassert>

//#include "math_constexpr.h
namespace mike{
        constexpr int add(const int a, const int b){    return a+b; }
        constexpr int sub(const int a, const int b){    return a-b; }
        constexpr int mul(const int a, const int b){    return a*b; }
}

constexpr int dotProduct(const std::array<int, 3> a,
                         const std::array<int, 3> b){
    // Use our specific call to 'add'
    using namespace mike;

    const int term1 = mul(a[0],b[0]);
    const int term2 = mul(a[1],b[1]);
    const int term3 = mul(a[2],b[2]);

    return add(add(term1,term2),term3);
}

int main(){

    std::array<int,3> v1{1,2,3};
    std::array<int,3> v2{2,4,6};
    static_assert(v1.size() == v2.size(),"v1.size()!=v2.size() ");

    return dotProduct(v1,v2);
}
```

A ▾ Output... ▾ Filter... ▾ Libraries Overrides + Add new...

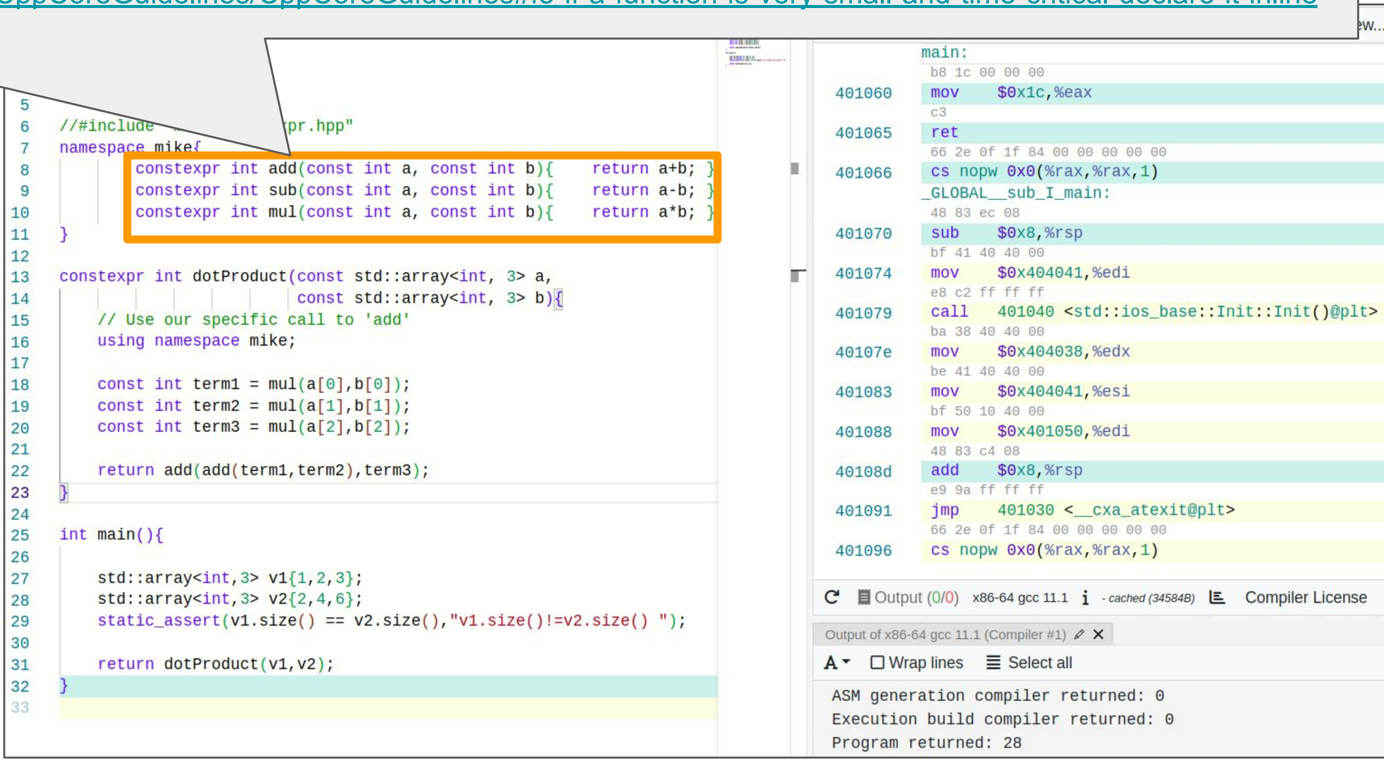
```
main:
        b8 1c 00 00 00
401060  mov    $0x1c,%eax
        c3
401065  ret
        66 2e 0f 1f 84 00 00 00 00 00
401066  cs nopw 0x0(%rax,%rax,1)
_GLOBAL__sub_I_main:
        48 83 ec 08
401070  sub    $0x8,%rsp
        bf 41 40 40 00
401074  mov    $0x404041,%edi
        e8 c2 ff ff ff
401079  call   401040 <std::ios_base::Init::Init()@plt>
        ba 38 40 40 00
40107e  mov    $0x404038,%edx
        be 41 40 40 00
401083  mov    $0x404041,%esi
        bf 50 10 40 00
401088  mov    $0x401050,%edi
        48 83 c4 08
40108d  add    $0x8,%rsp
        e9 9a ff ff ff
401091  jmp    401030 <__cxa_atexit@plt>
        66 2e 0f 1f 84 00 00 00 00 00
401096  cs nopw 0x0(%rax,%rax,1)
```

Output (0/0) x86-64 gcc 11.1 - cached (34584B) Compiler License
Output of x86-64 gcc 11.1 (Compiler #1)
A ▾ Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 28
```

- Note that making something ‘constexpr’ also implies it is inline -- you have the computed value!
- Core Guideline(s):
  - https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#f4-if-a-function-might-have-to-be-evaluated-at-compile-time-declare-it-constexpr
  - https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#f5-if-a-function-is-very-small-and-time-critical-declare-it-inline

- We can eva
  some functions at compile-time with ‘constexpr’
  - This effectively makes our program to the right a ‘no-op’ and it just returns 28.
- The key is the ‘constexpr’ used on the return type qualifier.



```cpp
//#include "                  pr.hpp"
namespace mike{
        constexpr int add(const int a, const int b){    return a+b; }
        constexpr int sub(const int a, const int b){    return a-b; }
        constexpr int mul(const int a, const int b){    return a*b; }
}

constexpr int dotProduct(const std::array<int, 3> a,
                         const std::array<int, 3> b){
    // Use our specific call to 'add'
    using namespace mike;

    const int term1 = mul(a[0],b[0]);
    const int term2 = mul(a[1],b[1]);
    const int term3 = mul(a[2],b[2]);

    return add(add(term1,term2),term3);
}

int main(){

    std::array<int,3> v1{1,2,3};
    std::array<int,3> v2{2,4,6};
    static_assert(v1.size() == v2.size(),"v1.size()!=v2.size() ");

    return dotProduct(v1,v2);
}
```

```
          main:
           b8 1c 00 00 00
  401060   mov    $0x1c,%eax
           c3
  401065   ret
           66 2e 0f 1f 84 00 00 00 00 00
  401066   cs nopw 0x0(%rax,%rax,1)
          _GLOBAL__sub_I_main:
           48 83 ec 08
  401070   sub    $0x8,%rsp
           bf 41 40 40 00
  401074   mov    $0x404041,%edi
           e8 c2 ff ff ff
  401079   call   401040 <std::ios_base::Init::Init()@plt>
           ba 38 40 40 00
  40107e   mov    $0x404038,%edx
           be 41 40 40 00
  401083   mov    $0x404041,%esi
           bf 50 10 40 00
  401088   mov    $0x401050,%edi
           48 83 c4 08
  40108d   add    $0x8,%rsp
           e9 9a ff ff ff
  401091   jmp    401030 <__cxa_atexit@plt>
           66 2e 0f 1f 84 00 00 00 00 00
  401096   cs nopw 0x0(%rax,%rax,1)
```

Output (0/0) x86-64 gcc 11.1 - cached (34584B) Compiler License
Output of x86-64 gcc 11.1 (Compiler #1)
Wrap lines Select all

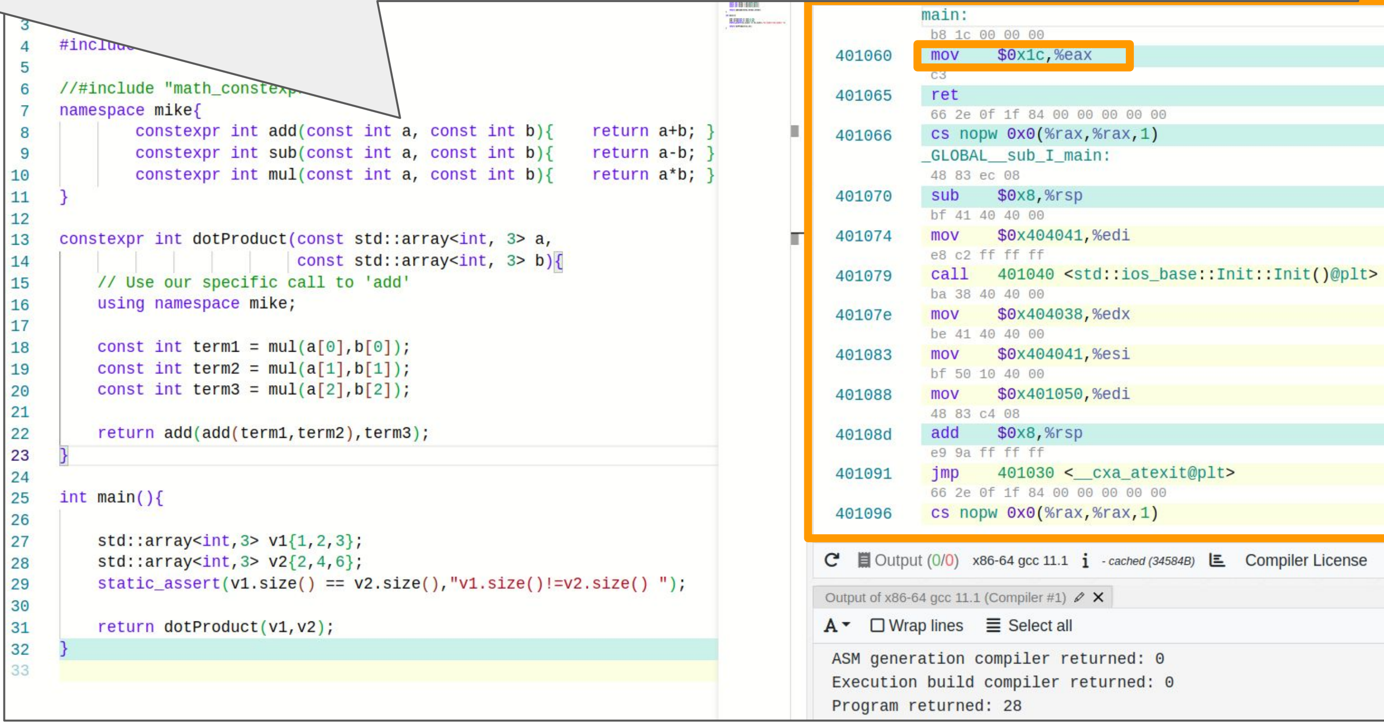
```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 28
```

https://godb

- By the way -- this is essentially all of the assembly code for our program
  - Note there's no call to our 'dotProduct' function -- just moving the value 28 into a register :)

cons

- We can evaluate some functions at compile-time with 'constexpr'
  - This effectively makes our program to the right a 'no-op' and it just returns 28.
- The key is the 'constexpr' used on the return type qualifier.



```cpp
#inclu

//#include "math_constexp
namespace mike{
        constexpr int add(const int a, const int b){    return a+b; }
        constexpr int sub(const int a, const int b){    return a-b; }
        constexpr int mul(const int a, const int b){    return a*b; }
}

constexpr int dotProduct(const std::array<int, 3> a,
                         const std::array<int, 3> b){
    // Use our specific call to 'add'
    using namespace mike;

    const int term1 = mul(a[0],b[0]);
    const int term2 = mul(a[1],b[1]);
    const int term3 = mul(a[2],b[2]);

    return add(add(term1,term2),term3);
}

int main(){

    std::array<int,3> v1{1,2,3};
    std::array<int,3> v2{2,4,6};
    static_assert(v1.size() == v2.size(),"v1.size()!=v2.size() ");

    return dotProduct(v1,v2);
}
```

```
        main:
          b8 1c 00 00 00
401060    mov    $0x1c,%eax
          c3
401065    ret
          66 2e 0f 1f 84 00 00 00 00 00
401066    cs nopw 0x0(%rax,%rax,1)
        _GLOBAL__sub_I_main:
          48 83 ec 08
401070    sub    $0x8,%rsp
          bf 41 40 40 00
401074    mov    $0x404041,%edi
          e8 c2 ff ff ff
401079    call   401040 <std::ios_base::Init::Init()@plt>
          ba 38 40 40 00
40107e    mov    $0x404038,%edx
          be 41 40 40 00
401083    mov    $0x404041,%esi
          bf 50 10 40 00
401088    mov    $0x401050,%edi
          48 83 c4 08
40108d    add    $0x8,%rsp
          e9 9a ff ff ff
401091    jmp    401030 <__cxa_atexit@plt>
          66 2e 0f 1f 84 00 00 00 00 00
401096    cs nopw 0x0(%rax,%rax,1)
```

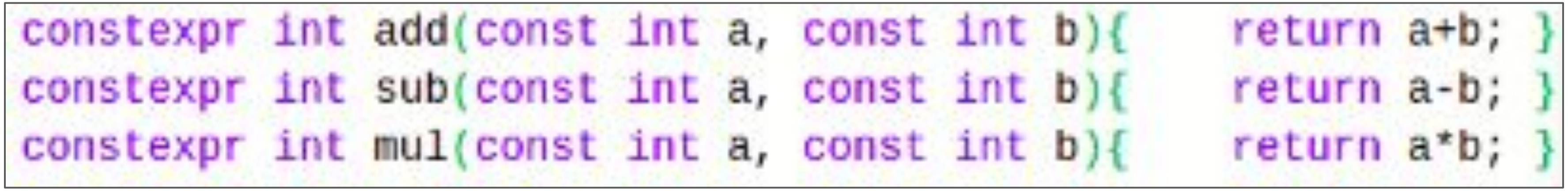
ew...
Output (0/0) x86-64 gcc 11.1 - cached (34584B) Compiler License
Output of x86-64 gcc 11.1 (Compiler #1)
A Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 28
```

https://godbolt.org/z/hhxcWj6Gq

(Aside) pure functions

- Functions that don’t have side effects (i.e. immutable functions) in the argument or return value are known as **pure functions**
    - These are good, because they are not dependent on run-time ‘state’, all values could be known at compile-time
    - You can think of pure functions as most of the common math functions you started out learning in school
        - The same inputs generate the same output values.
        - Note: I believe many cmath functions in either C++23/26 are becoming constexpr



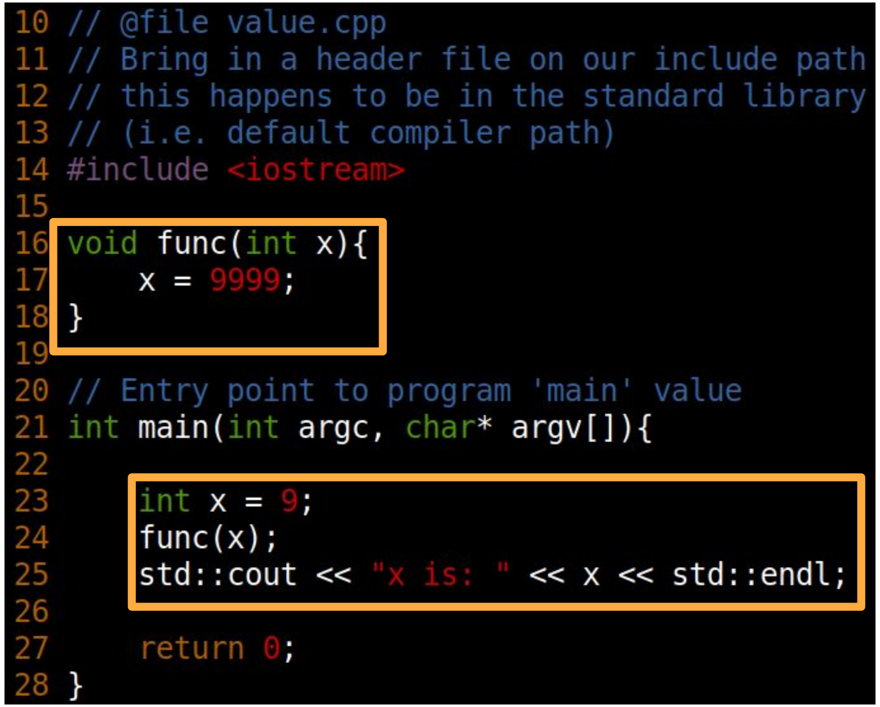
```
constexpr int add(const int a, const int b){    return a+b; }
constexpr int sub(const int a, const int b){    return a-b; }
constexpr int mul(const int a, const int b){    return a*b; }
```

Function Parameters

Understanding pass-by-value and pass-by-reference (and ‘const’)

Quick Check: What do you think the value of x will be? (1/2)

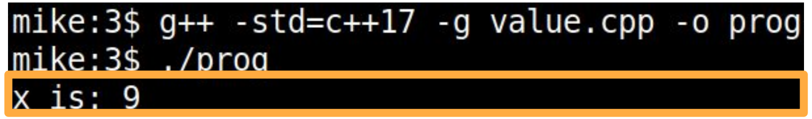
- What will the value of x be?



```cpp
// @file value.cpp
// Bring in a header file on our include path
// this happens to be in the standard library
// (i.e. default compiler path)
#include <iostream>

void func(int x){
    x = 9999;
}

// Entry point to program 'main' value
int main(int argc, char* argv[]){

    int x = 9;
    func(x);
    std::cout << "x is: " << x << std::endl;

    return 0;
}
```

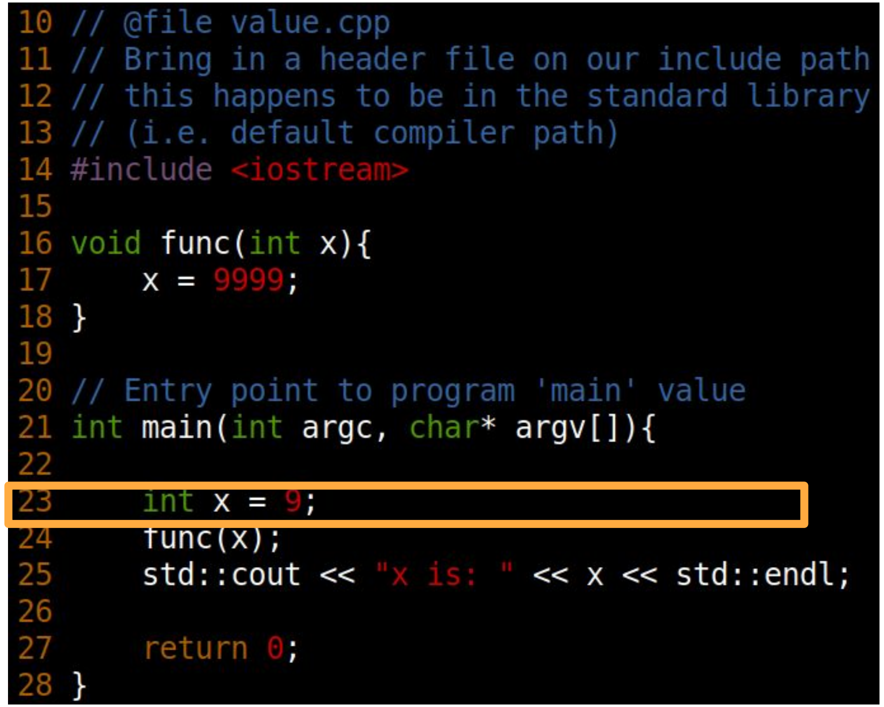
Quick Check: What do you think the value of x will be? (2/2)

- What will the value of x be?



```
mike:3$ g++ -std=c++17 -g value.cpp -o prog
mike:3$ ./prog
x is: 9
```

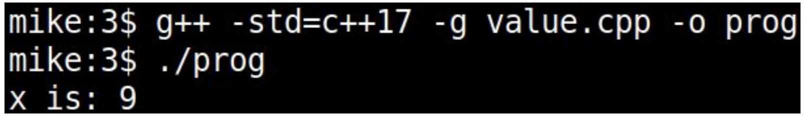
- Hmm, why is this? (Next slide!)



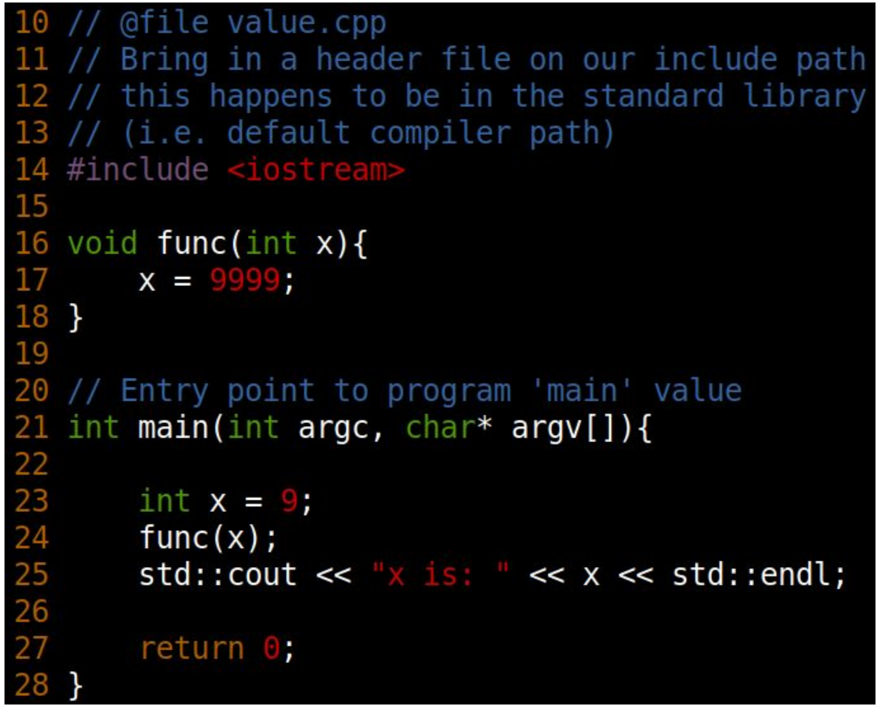
```cpp
// @file value.cpp
// Bring in a header file on our include path
// this happens to be in the standard library
// (i.e. default compiler path)
#include <iostream>

void func(int x){
    x = 9999;
}

// Entry point to program 'main' value
int main(int argc, char* argv[]){

    int x = 9;
    func(x);
    std::cout << "x is: " << x << std::endl;

    return 0;
}
```

Pass by Value (Also known as pass by ‘copy-value’)

- In C++ we have control over what happens when we pass in a variable into a function.
- At line 24, we actually get a ‘copy’ of ‘x’.



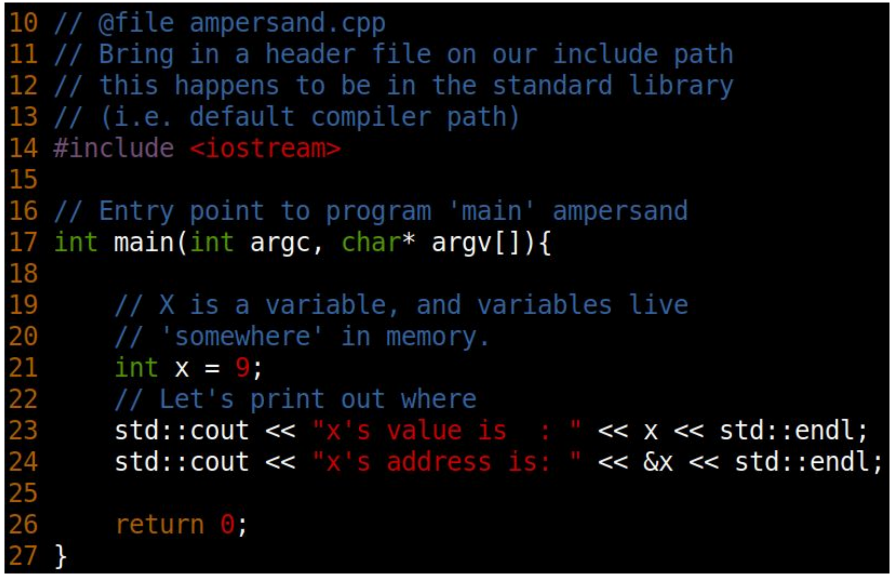
```
mike:3$ g++ -std=c++17 -g value.cpp -o prog
mike:3$ ./prog
x is: 9
```



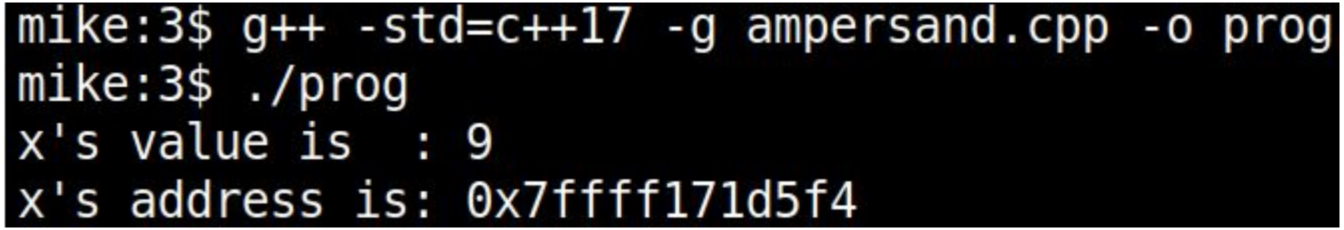
```
10 // @file value.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15
16 void func(int x){
17     x = 9999;
18 }
19
20 // Entry point to program 'main' value
21 int main(int argc, char* argv[]){
22
23     int x = 9;
24     func(x);
25     std::cout << "x is: " << x << std::endl;
26
27     return 0;
28 }
```

(Review) & Operator (‘Address of function’)

- The ampersand operator (‘&’) in C++ retrieves the address of a variable in memory.
- We can use it to figure out where exactly in memory (i.e. the address) our variables are located.
  - You can thus see below, the hexadecimal address in memory of ‘x’



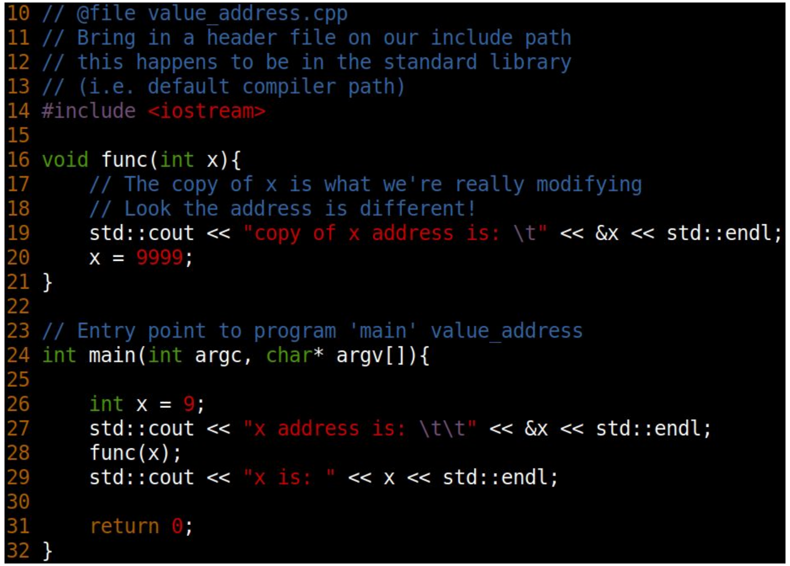
```cpp
// @file ampersand.cpp
// Bring in a header file on our include path
// this happens to be in the standard library
// (i.e. default compiler path)
#include <iostream>

// Entry point to program 'main' ampersand
int main(int argc, char* argv[]){

    // X is a variable, and variables live
    // 'somewhere' in memory.
    int x = 9;
    // Let's print out where
    std::cout << "x's value is   : " << x << std::endl;
    std::cout << "x's address is: " << &x << std::endl;

    return 0;
}
```



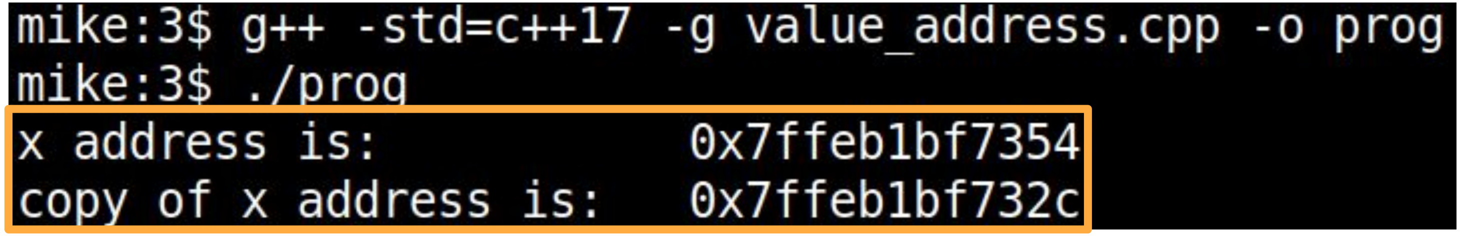
```
mike:3$ g++ -std=c++17 -g ampersand.cpp -o prog
mike:3$ ./prog
x's value is  : 9
x's address is: 0x7ffff171d5f4
```

(Aside) Using & to understand pass-by-value

- Notice below that the addresses are different
- Thus, if the address is different, than when we modify



```
// @file value_address.cpp
// Bring in a header file on our include path
// this happens to be in the standard library
// (i.e. default compiler path)
#include <iostream>

void func(int x){
    // The copy of x is what we're really modifying
    // Look the address is different!
    std::cout << "copy of x address is: \t" << &x << std::endl;
    x = 9999;
}

// Entry point to program 'main' value_address
int main(int argc, char* argv[]){

    int x = 9;
    std::cout << "x address is: \t\t" << &x << std::endl;
    func(x);
    std::cout << "x is: " << x << std::endl;

    return 0;
}
```



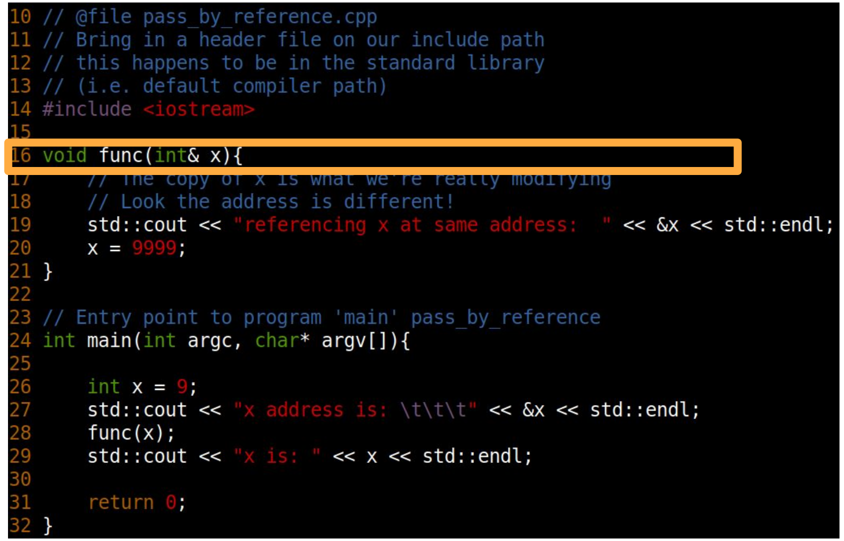
```
mike:3$ g++ -std=c++17 -g value_address.cpp -o prog
mike:3$ ./prog
x address is:           0x7ffeb1bf7354
copy of x address is:   0x7ffeb1bf732c
```

(Aside) Quick Tip for ‘&’

- & is an operator (i.e. function) for getting the ‘address of’ a variable or function that exists.

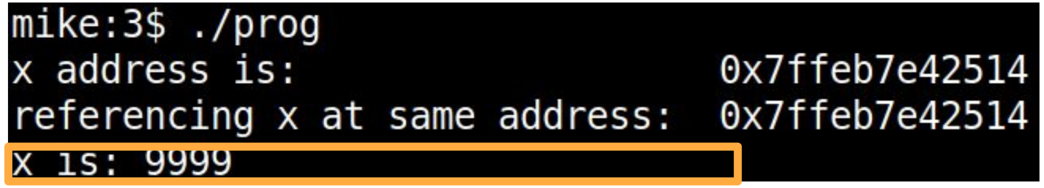
Pass by Reference (1/2)

- In C++, if you want to modify the value, you can instead ‘pass by reference’
- Notice very subtly the function signature at line 16
  - `void func(int& x)`
    - Think of the int& as a ‘reference type’
  - The ampersand states that we are passing an actual reference to something that exists.
  - The parameter is thus an ‘alias’ to something that exists.
  - Now the actual ‘x’ in memory will be modified



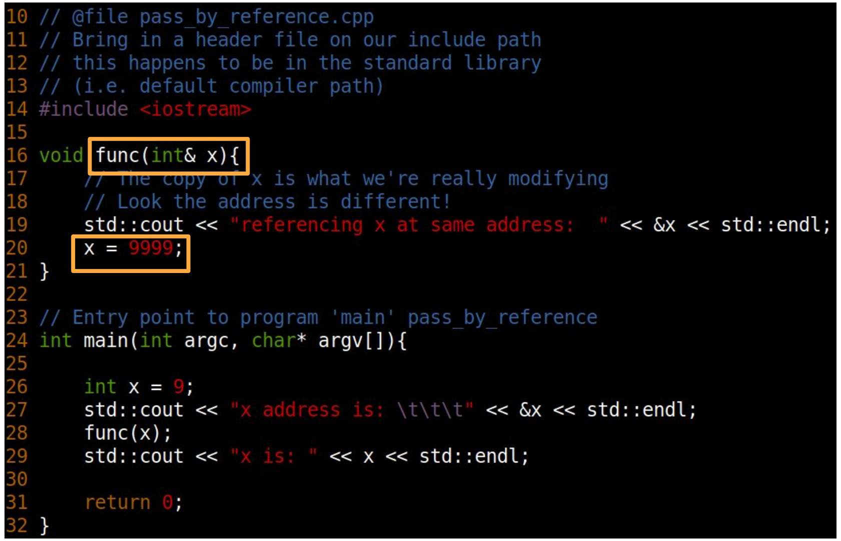
```
10 // @file pass_by_reference.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15
16 void func(int& x){
17     // The copy of x is what we're really modifying
18     // Look the address is different!
19     std::cout << "referencing x at same address:  " << &x << std::endl;
20     x = 9999;
21 }
22
23 // Entry point to program 'main' pass_by_reference
24 int main(int argc, char* argv[]){
25
26     int x = 9;
27     std::cout << "x address is: \t\t\t" << &x << std::endl;
28     func(x);
29     std::cout << "x is: " << x << std::endl;
30
31     return 0;
32 }
```

Pass by Reference (2/2)



```
mike:3$ ./prog
x address is:                     0x7ffeb7e42514
referencing x at same address:    0x7ffeb7e42514
x is: 9999
```

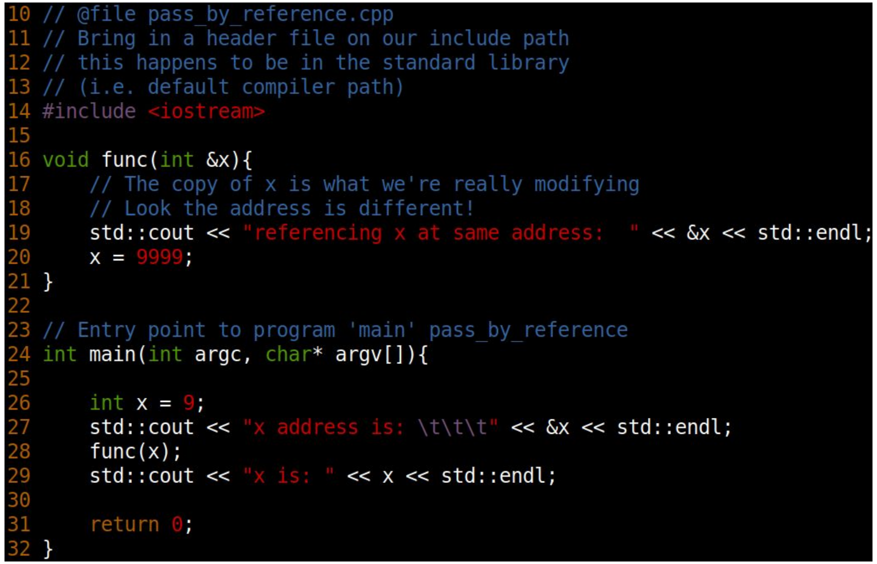
- In C++, if you want to modify the value, you can instead ‘pass by reference’
- Notice very subtly the function signature at line 16
  - `void func(int& x)`
    - Think of the int& as a ‘reference type’
  - The ampersand states that we are passing an actual reference to something that exists.
  - The parameter is thus an ‘alias’ to something that exists.
  - Now the actual ‘x’ in memory will be modified



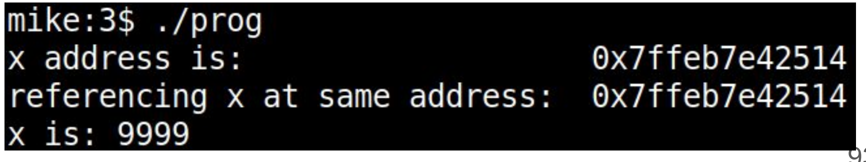
```
10 // @file pass_by_reference.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15
16 void func(int& x){
17     // The copy of x is what we're really modifying
18     // Look the address is different!
19     std::cout << "referencing x at same address:  " << &x << std::endl;
20     x = 9999;
21 }
22
23 // Entry point to program 'main' pass_by_reference
24 int main(int argc, char* argv[]){
25
26     int x = 9;
27     std::cout << "x address is: \t\t\t" << &x << std::endl;
28     func(x);
29     std::cout << "x is: " << x << std::endl;
30
31     return 0;
32 }
```

Why Pass by Reference?

- Reason 1:
  - Sometimes we want to modify the actual variable being passed in!
- Reason 2:
  - We avoiding making a copy of our data
    - You'll notice the performance if you pass big or expensive to copy data structures
    - (e.g. a vector of 10,000,000 big objects would all have to be copied)
- Reason 3:
  - It's a bit safer than a pointer -- meaning it's a lot harder to get a NULL value



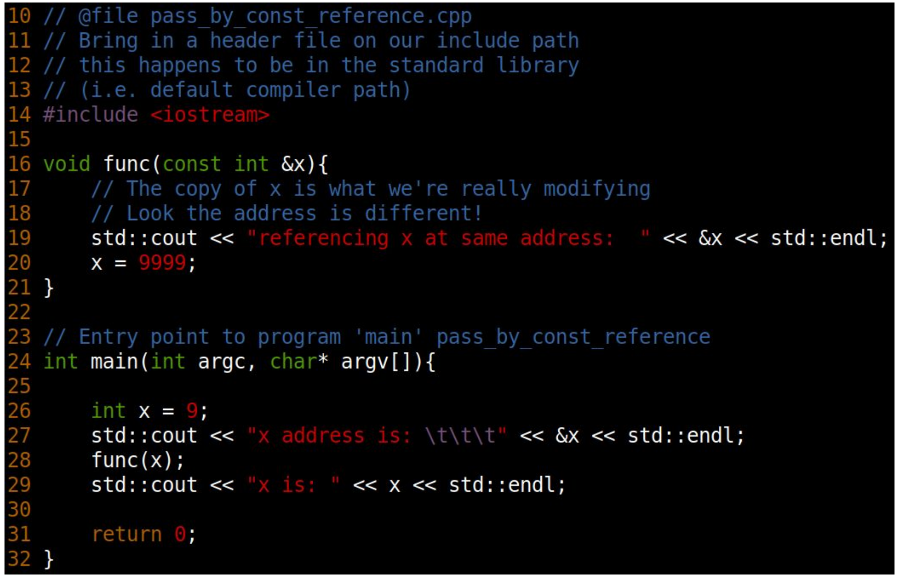
```cpp
// @file pass_by_reference.cpp
// Bring in a header file on our include path
// this happens to be in the standard library
// (i.e. default compiler path)
#include <iostream>

void func(int &x){
    // The copy of x is what we're really modifying
    // Look the address is different!
    std::cout << "referencing x at same address:  " << &x << std::endl;
    x = 9999;
}

// Entry point to program 'main' pass_by_reference
int main(int argc, char* argv[]){

    int x = 9;
    std::cout << "x address is: \t\t\t" << &x << std::endl;
    func(x);
    std::cout << "x is: " << x << std::endl;

    return 0;
}
```



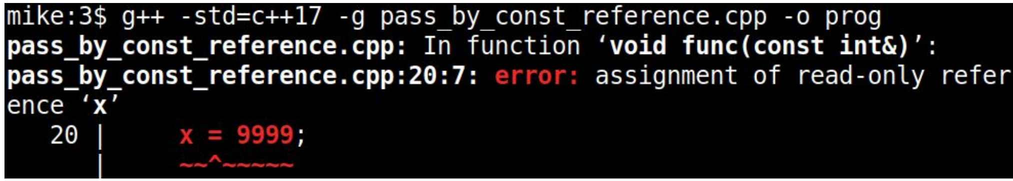
```
mike:3$ ./prog
x address is:                   0x7ffeb7e42514
referencing x at same address:  0x7ffeb7e42514
x is: 9999
```

const reference parameter

- Just like when we declare variables with ‘const’ we can also do so for our function arguments.
- In this case, we can:
  - **`const int &x`**
  - This means we cannot modify that value of x
- If you try to run this example, it will not let you, because you are trying to reassign the value of the int that you passed in the function.



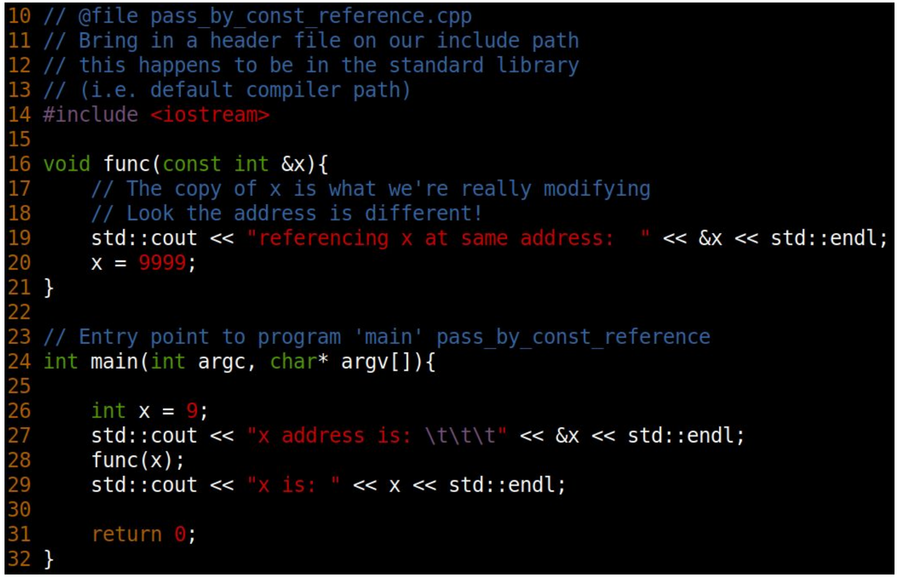
```cpp
10 // @file pass_by_const_reference.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15
16 void func(const int &x){
17     // The copy of x is what we're really modifying
18     // Look the address is different!
19     std::cout << "referencing x at same address:  " << &x << std::endl;
20     x = 9999;
21 }
22
23 // Entry point to program 'main' pass_by_const_reference
24 int main(int argc, char* argv[]){
25
26     int x = 9;
27     std::cout << "x address is: \t\t\t" << &x << std::endl;
28     func(x);
29     std::cout << "x is: " << x << std::endl;
30
31     return 0;
32 }
```



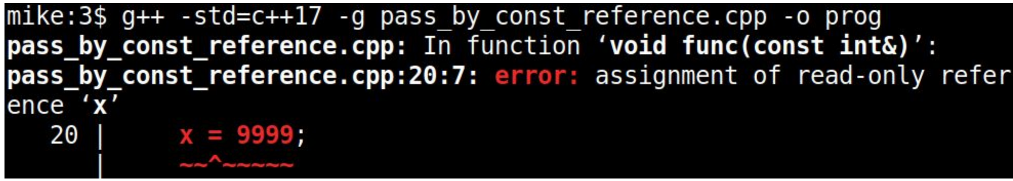
```
mike:3$ g++ -std=c++17 -g pass_by_const_reference.cpp -o prog
pass_by_const_reference.cpp: In function ‘void func(const int&)’:
pass_by_const_reference.cpp:20:7: error: assignment of read-only refer
ence ‘x’
   20 |     x = 9999;
      |     ~~^~~~~~
```

const reference parameter - Why would we do this?

- Again we pass-by-reference to avoid a copy
  - The ‘const’ part is a ‘security’ (i.e. contract) that ensures that whatever data we are passing into that function will not be mutated (i.e. change its state in anyway).
- As we work with bigger data structures, this is more important!



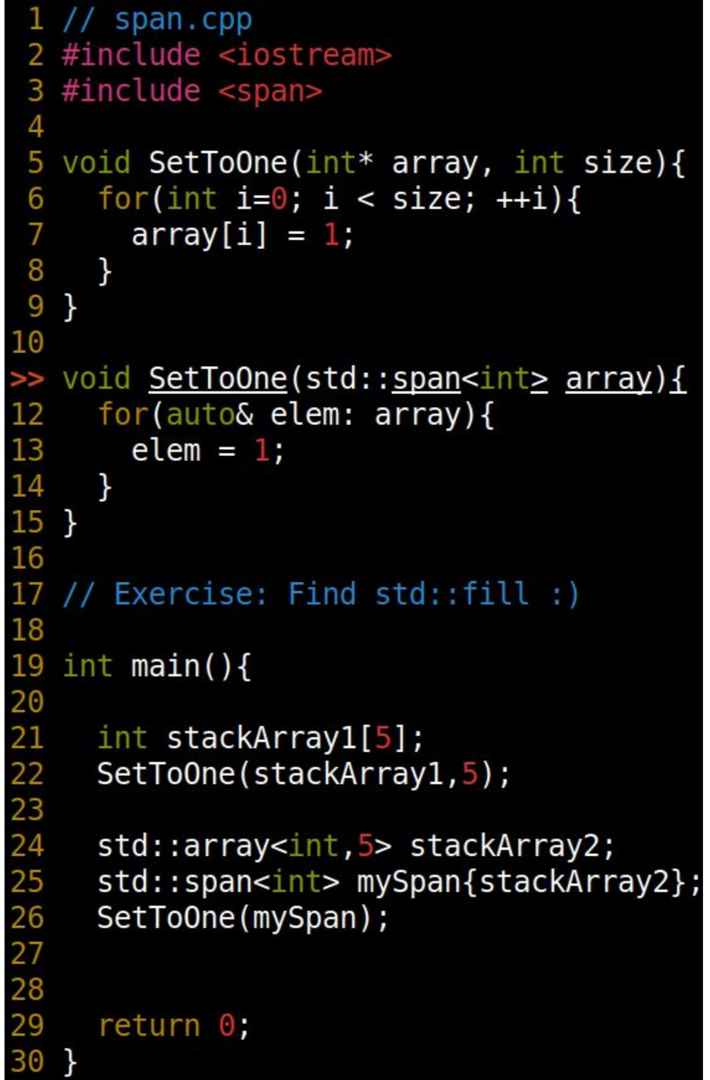
```cpp
10 // @file pass_by_const_reference.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15
16 void func(const int &x){
17     // The copy of x is what we're really modifying
18     // Look the address is different!
19     std::cout << "referencing x at same address:  " << &x << std::endl;
20     x = 9999;
21 }
22
23 // Entry point to program 'main' pass_by_const_reference
24 int main(int argc, char* argv[]){
25
26     int x = 9;
27     std::cout << "x address is: \t\t\t" << &x << std::endl;
28     func(x);
29     std::cout << "x is: " << x << std::endl;
30
31     return 0;
32 }
```



```
mike:3$ g++ -std=c++17 -g pass_by_const_reference.cpp -o prog
pass_by_const_reference.cpp: In function ‘void func(const int&)’:
pass_by_const_reference.cpp:20:7: error: assignment of read-only refer
ence ‘x’
   20 |     x = 9999;
      |     ~~^~~~~~
```

Other Tips -- Take a look at std::span

- Pass-by-Pointer (line 5) is still pass-by-value
  - (i.e. making a copy of the pointer)
  - Passing in a pointer copies the pointer, but both pointers point to the same underlying address -- thus we can modify the value.
- Prefer in Modern C++ codebases however to use std::span (C++20) as an argument in your functions if you do have to pass a pointer and a size
  - std::span is a pointer and a length
  - Can handle dynamic data structures as well.



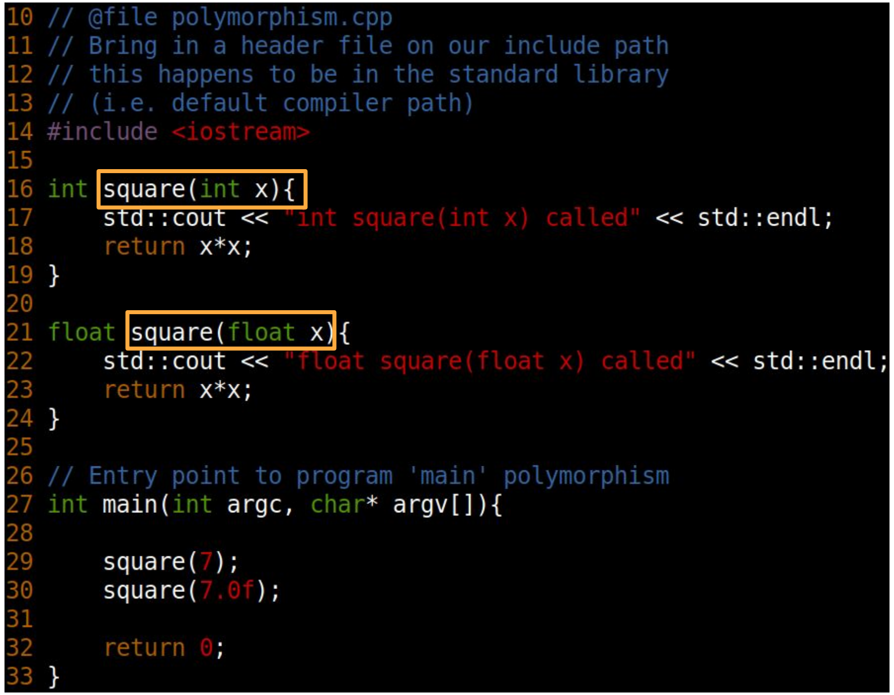
```
 1 // span.cpp
 2 #include <iostream>
 3 #include <span>
 4
 5 void SetToOne(int* array, int size){
 6   for(int i=0; i < size; ++i){
 7     array[i] = 1;
 8   }
 9 }
10
>> void SetToOne(std::span<int> array){
12   for(auto& elem: array){
13     elem = 1;
14   }
15 }
16
17 // Exercise: Find std::fill :)
18
19 int main(){
20
21   int stackArray1[5];
22   SetToOne(stackArray1,5);
23
24   std::array<int,5> stackArray2;
25   std::span<int> mySpan{stackArray2};
26   SetToOne(mySpan);
27
28
29   return 0;
30 }
```

Function Polymorphism and Overloading

Function with the same name, with potentially a different implementation (often because of different parameter types)

Function polymorphism

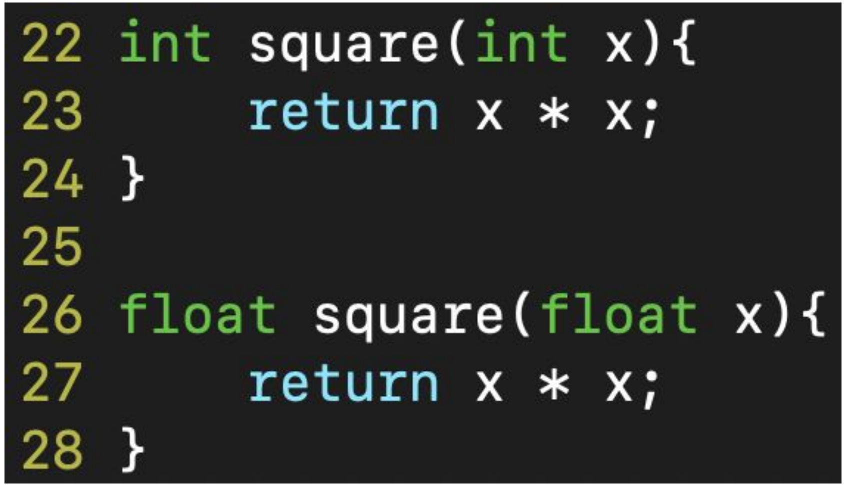
- In C++, we can reuse the same name for multiple functions where the parameters are different.
  - Note: In languages like C we have to uniquely name our functions
- When we make a call to the function (i.e. square), C++ can automatically deduce which function to call based on the data types or arguments used.
  - This is a type of function polymorphism



```cpp
// @file polymorphism.cpp
// Bring in a header file on our include path
// this happens to be in the standard library
// (i.e. default compiler path)
#include <iostream>

int square(int x){
    std::cout << "int square(int x) called" << std::endl;
    return x*x;
}

float square(float x){
    std::cout << "float square(float x) called" << std::endl;
    return x*x;
}

// Entry point to program 'main' polymorphism
int main(int argc, char* argv[]){

    square(7);
    square(7.0f);

    return 0;
}
```

(Aside) Argument Dependent Lookup (ADL)

- C++ compilers perform something known as **a**rgument-**d**ependent **l**ookup (ADL) when resolving which function
  - https://en.cppreference.com/w/cpp/language/adl
  - You can read through this if you want a bit more detail on how function calls are resolved (or which version of square will be called if we pass in a 'double')
  - ADL specifically helps us figure out which functions to call within scope.



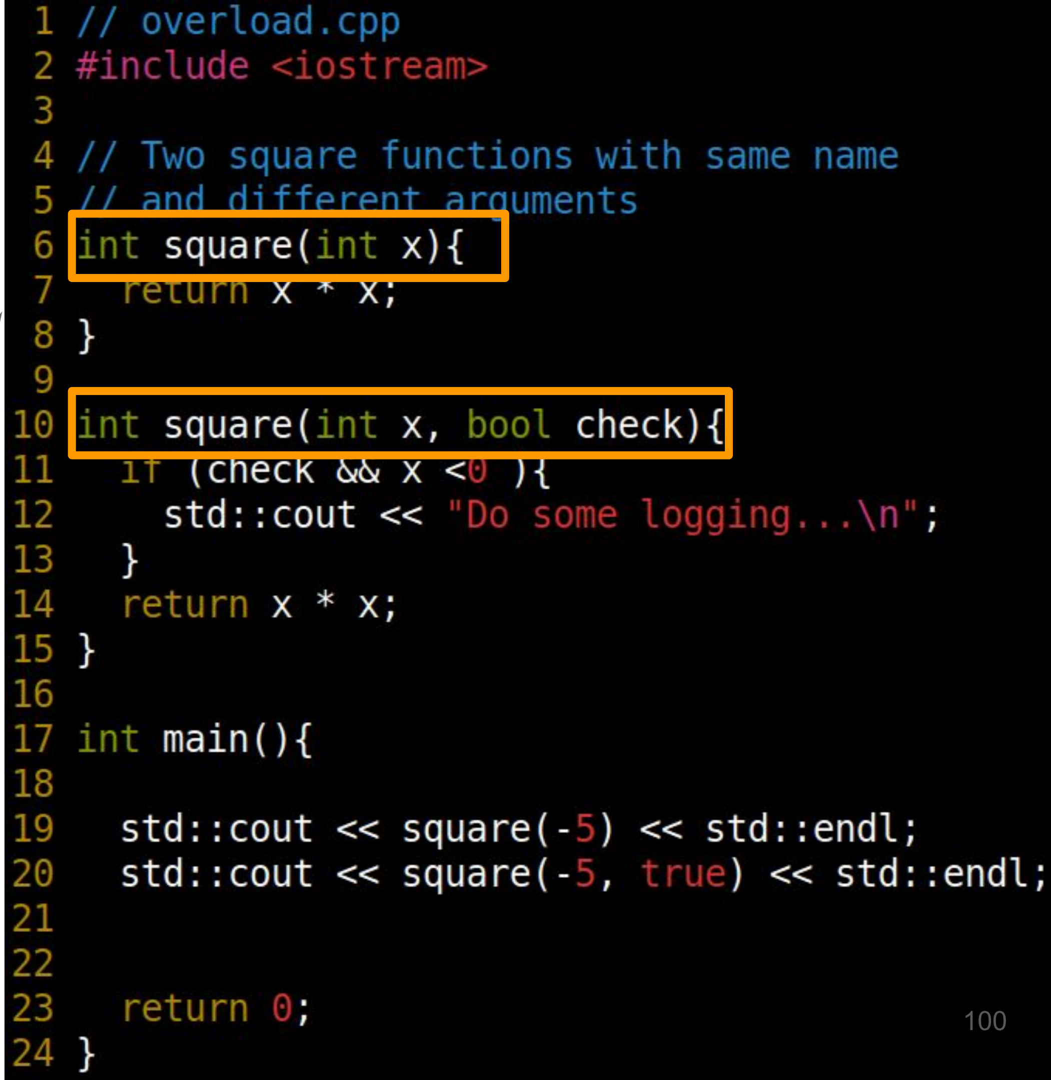
```
int square(int x){
        return x * x;
}

float square(float x){
        return x * x;
}
```

Function Overloads

Sometimes it's useful to provide different 'types' and different number of parameters into functions but use the same function name.

This is known as function 'overloading.

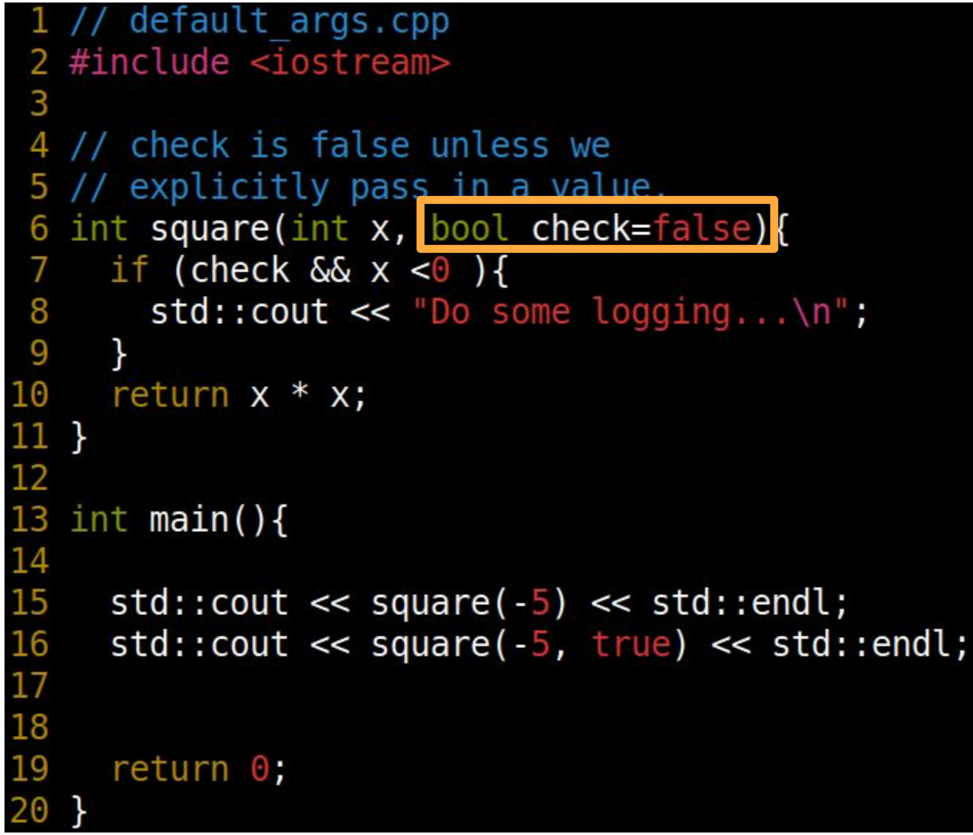
Here is an example with two functions with the same name, but different parameter lists.



```
// overload.cpp
#include <iostream>

// Two square functions with same name
// and different arguments
int square(int x){
  return x * x;
}

int square(int x, bool check){
  if (check && x <0 ){
    std::cout << "Do some logging...\n";
  }
  return x * x;
}

int main(){

  std::cout << square(-5) << std::endl;
  std::cout << square(-5, true) << std::endl;


  return 0;
}
```

Function Default Parameters

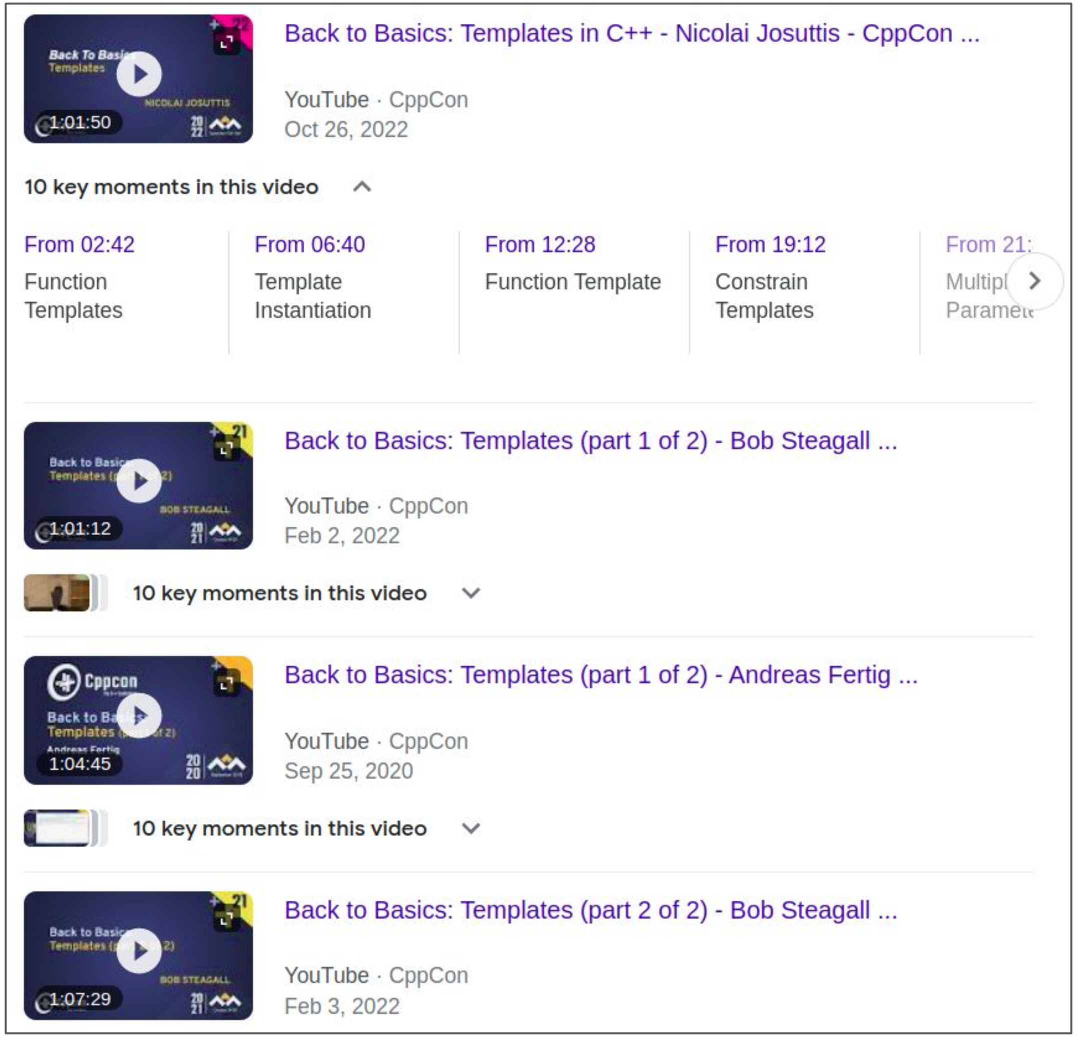
- Note: We can also provide default parameters to our functions when it makes sense
  - i.e. if we have some option that is not always needed, then provide a default value.
  - This is sometimes preferred versus creating lots of different functions -- as it may be preferable to have one implementation.
  - There's a better tool for specific implementations if the implementation is dependent on types however (next slide on templates!)



```cpp
// default_args.cpp
#include <iostream>

// check is false unless we
// explicitly pass in a value.
int square(int x, bool check=false){
  if (check && x <0 ){
    std::cout << "Do some logging...\n";
  }
  return x * x;
}

int main(){

  std::cout << square(-5) << std::endl;
  std::cout << square(-5, true) << std::endl;


  return 0;
}
```

Function Templates

- Templates are a mechanism for generating code and working with generic types.
- Templates (and Concepts) are a big topic in C++
    - I can again refer you to talks from the past and this current conference on the topic.



Back to Basics: Templates in C++ - Nicolai Josuttis - CppCon ...

YouTube · CppCon
Oct 26, 2022

10 key moments in this video

From 02:42 Function Templates
From 06:40 Template Instantiation
From 12:28 Function Template
From 19:12 Constrain Templates
From 21: Multipl Paramet

Back to Basics: Templates (part 1 of 2) - Bob Steagall ...

YouTube · CppCon
Feb 2, 2022

10 key moments in this video

Back to Basics: Templates (part 1 of 2) - Andreas Fertig ...

YouTube · CppCon
Sep 25, 2020

10 key moments in this video

Back to Basics: Templates (part 2 of 2) - Bob Steagall ...

YouTube · CppCon
Feb 3, 2022

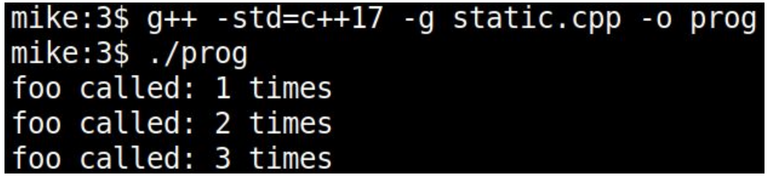
Functions & State

Understanding lifetime and state in function-like functions

Functions and State

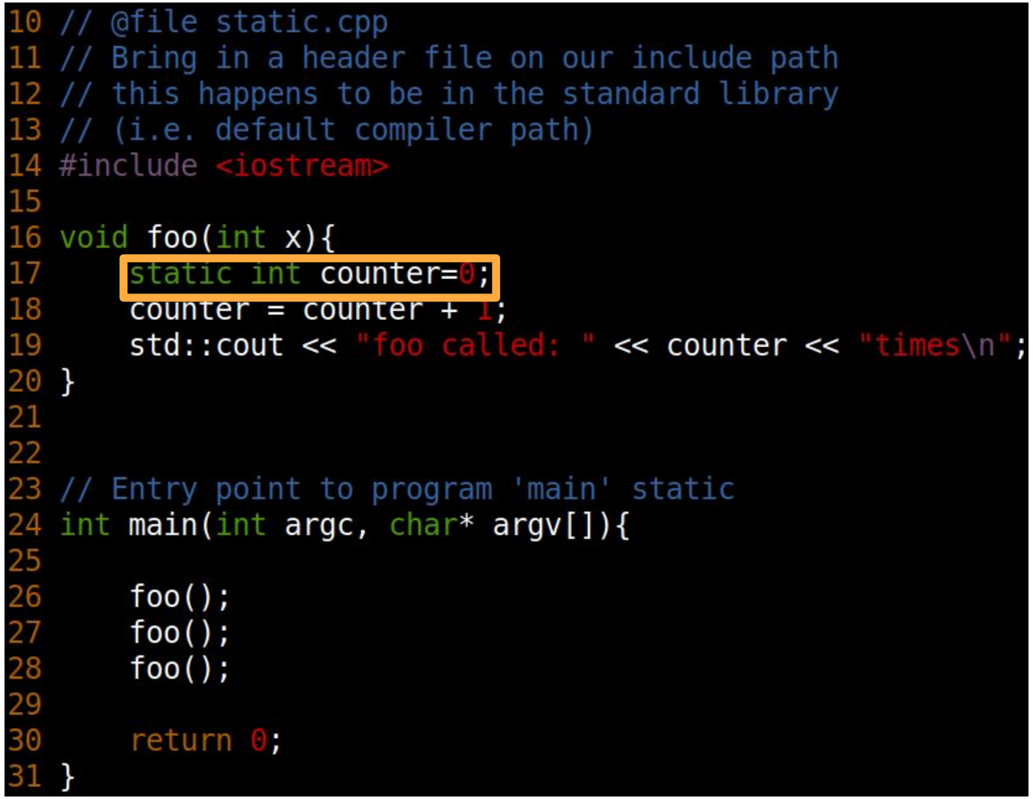
- No State
  - We’ve seen some previous examples of ‘pure’ functions (using constexpr)
  - These functions compose well
- State Changes
  - We’ve seen some functions that allow for mutation
  - (passing by reference or pass-by-pointer) that allow for mutation.
- Holding State
  - We have seen how objects can be used to hold state and even change behavior of object your dynamic dispatch (run-time type polymorphism)

Extending local lifetime with ‘static’



```
mike:3$ g++ -std=c++17 -g static.cpp -o prog
mike:3$ ./prog
foo called: 1 times
foo called: 2 times
foo called: 3 times
```

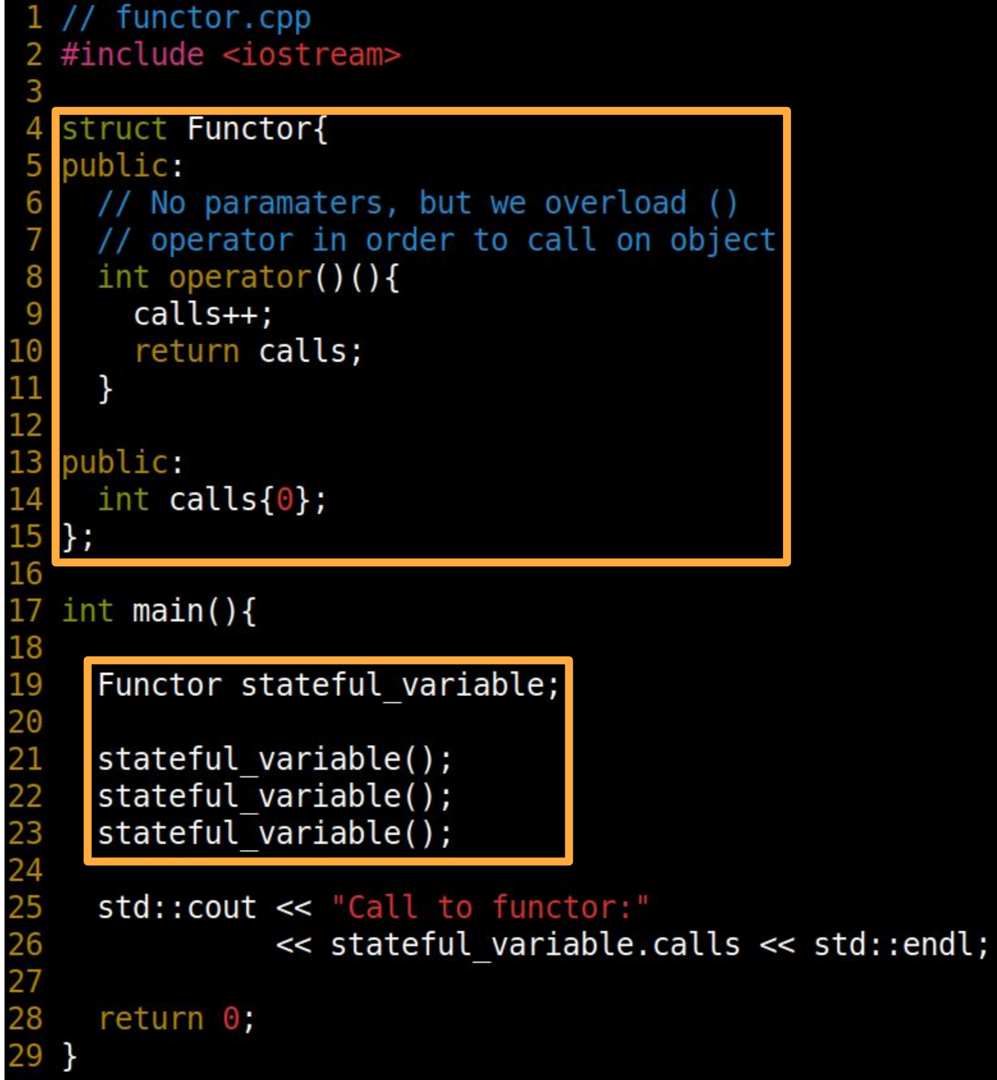
- There is a way to ‘extend’ the lifetime of a variable within a function
  - The variable is effectively a global variable
  - The scope is still within the function however.
- This means that when you call a function, it will retain its value.
- This is done with the keyword ‘static’
  - Notice how ‘counter’ does not get redeclared each time.
  - It is allocated exactly once in the compiled code, and C++ retains the local variable ‘counter’ in foo().



```cpp
// @file static.cpp
// Bring in a header file on our include path
// this happens to be in the standard library
// (i.e. default compiler path)
#include <iostream>

void foo(int x){
    static int counter=0;
    counter = counter + 1;
    std::cout << "foo called: " << counter << "times\n";
}


// Entry point to program 'main' static
int main(int argc, char* argv[]){

    foo();
    foo();
    foo();

    return 0;
}
```

Functors (Function Objects)

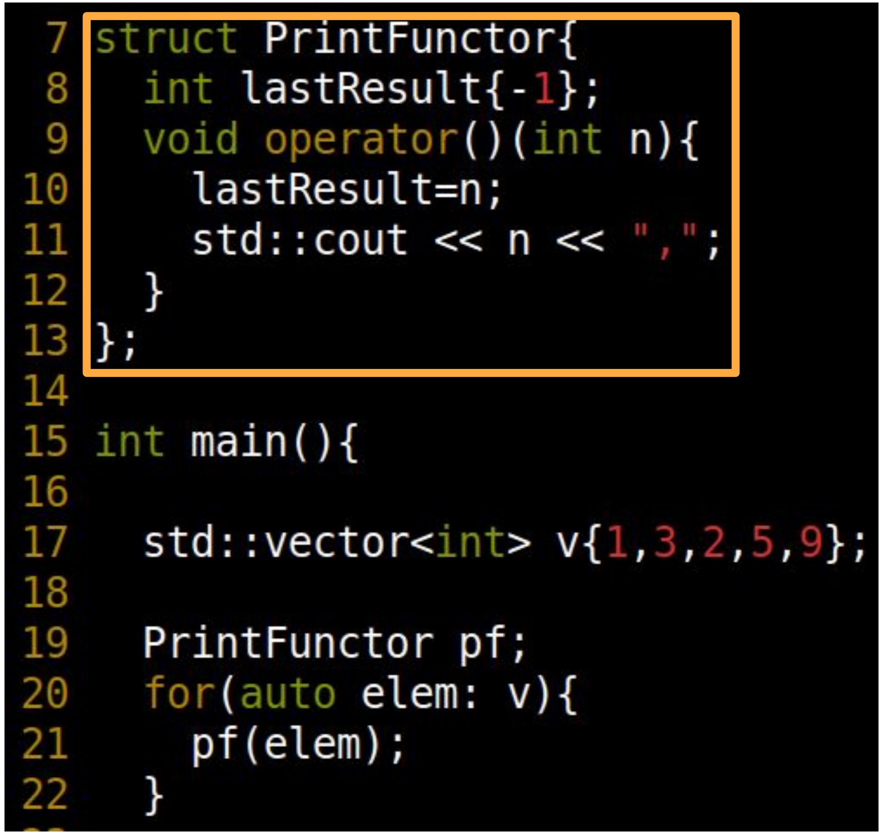
- Functors are ‘function objects’
  - You are ‘allocating’ some separate persistent memory to hold ‘state’ for your function
    - This memory lives (i.e. is in scope) for the duration of the objects lifetime, as opposed to the call stack.
  - This way you can ‘save’ state within specific invocations of your functor.



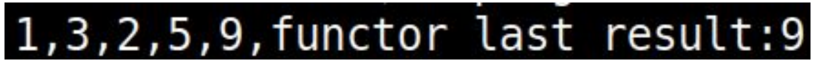
```cpp
// functor.cpp
#include <iostream>

struct Functor{
public:
  // No paramaters, but we overload ()
  // operator in order to call on object
  int operator()(){
    calls++;
    return calls;
  }

public:
  int calls{0};
};

int main(){

  Functor stateful_variable;

  stateful_variable();
  stateful_variable();
  stateful_variable();

  std::cout << "Call to functor:"
            << stateful_variable.calls << std::endl;

  return 0;
}
```

Evolving Functors to ...

- Here's another example of a 'functor' that 'captures' (i.e. stores) the last value in a member variable called `lastResult`.
    - Again, this code is perfectly reasonable
    - But how 'modifiable is this functor?
        - What if I want similar functors?
        - What if I don't want the scope to be 'global' ?



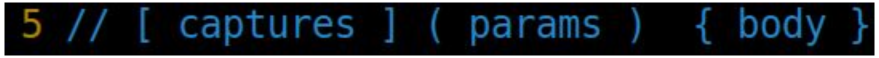
```
struct PrintFunctor{
  int lastResult{-1};
  void operator()(int n){
    lastResult=n;
    std::cout << n << ",";
  }
};

int main(){

  std::vector<int> v{1,3,2,5,9};

  PrintFunctor pf;
  for(auto elem: v){
    pf(elem);
  }
```



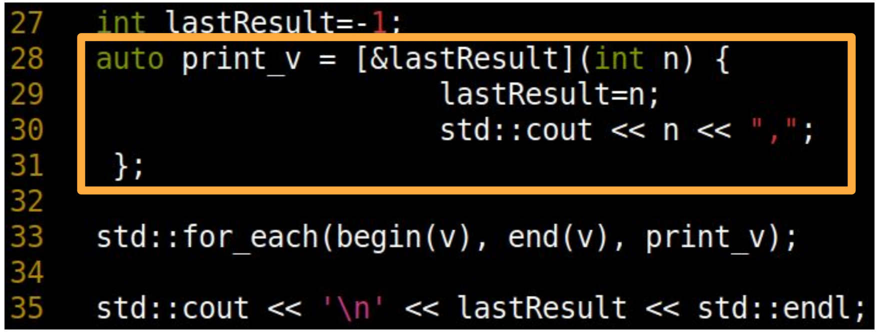
```
1,3,2,5,9,functor last result:9
```

Lambda's (Effectively Functors behind the scenes)

- Lambda's are 'unnamed' functions.
  - Lambda's are a convenient way for us to create 'local' functions
    - Behind the scenes they are implemented as functors (as they can carry state)
- Lambda functions tend to be more local and help us break problems into smaller chunks
  - If you think you'll use the lambda more than once -- then it's okay to make it a function



```
// [ captures ] ( params )  { body }
```

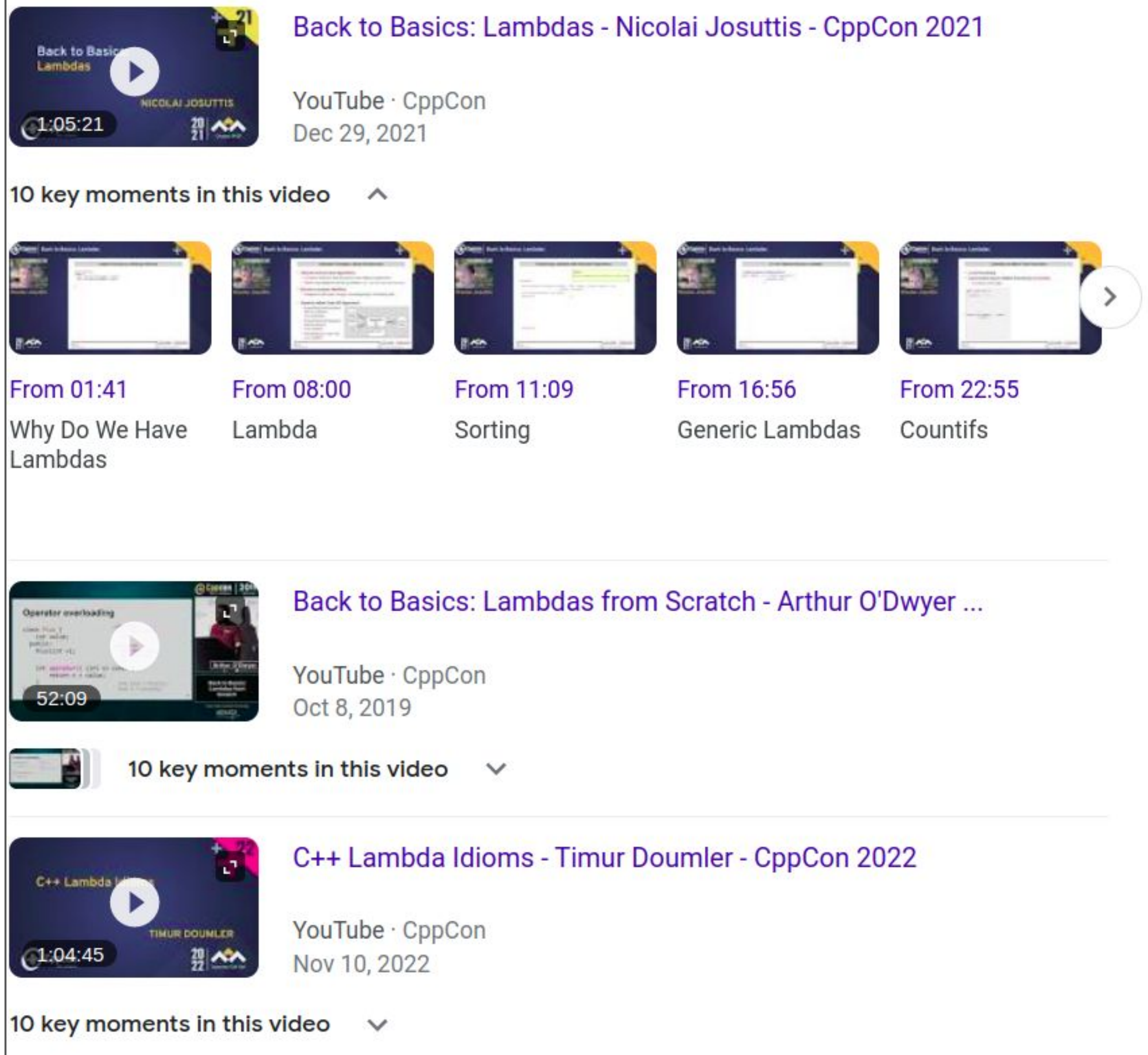


```
int lastResult=-1;
auto print_v = [&lastResult](int n) {
                        lastResult=n;
                        std::cout << n << ",";
 };

std::for_each(begin(v), end(v), print_v);

std::cout << '\n' << lastResult << std::endl;
```

(Same code as previously shown)

More on Lambda

- Lambda's are available in C++11 and beyond
  - They can very much help clean up your code.
- Lambda's themselves are quite nice -- but are yet again another separate talk.



Back to Basics: Lambdas - Nicolai Josuttis - CppCon 2021

YouTube · CppCon
Dec 29, 2021

10 key moments in this video

From 01:41 Why Do We Have Lambdas
From 08:00 Lambda
From 11:09 Sorting
From 16:56 Generic Lambdas
From 22:55 Countifs

Back to Basics: Lambdas from Scratch - Arthur O'Dwyer ...

YouTube · CppCon
Oct 8, 2019

10 key moments in this video

C++ Lambda Idioms - Timur Doumler - CppCon 2022

YouTube · CppCon
Nov 10, 2022

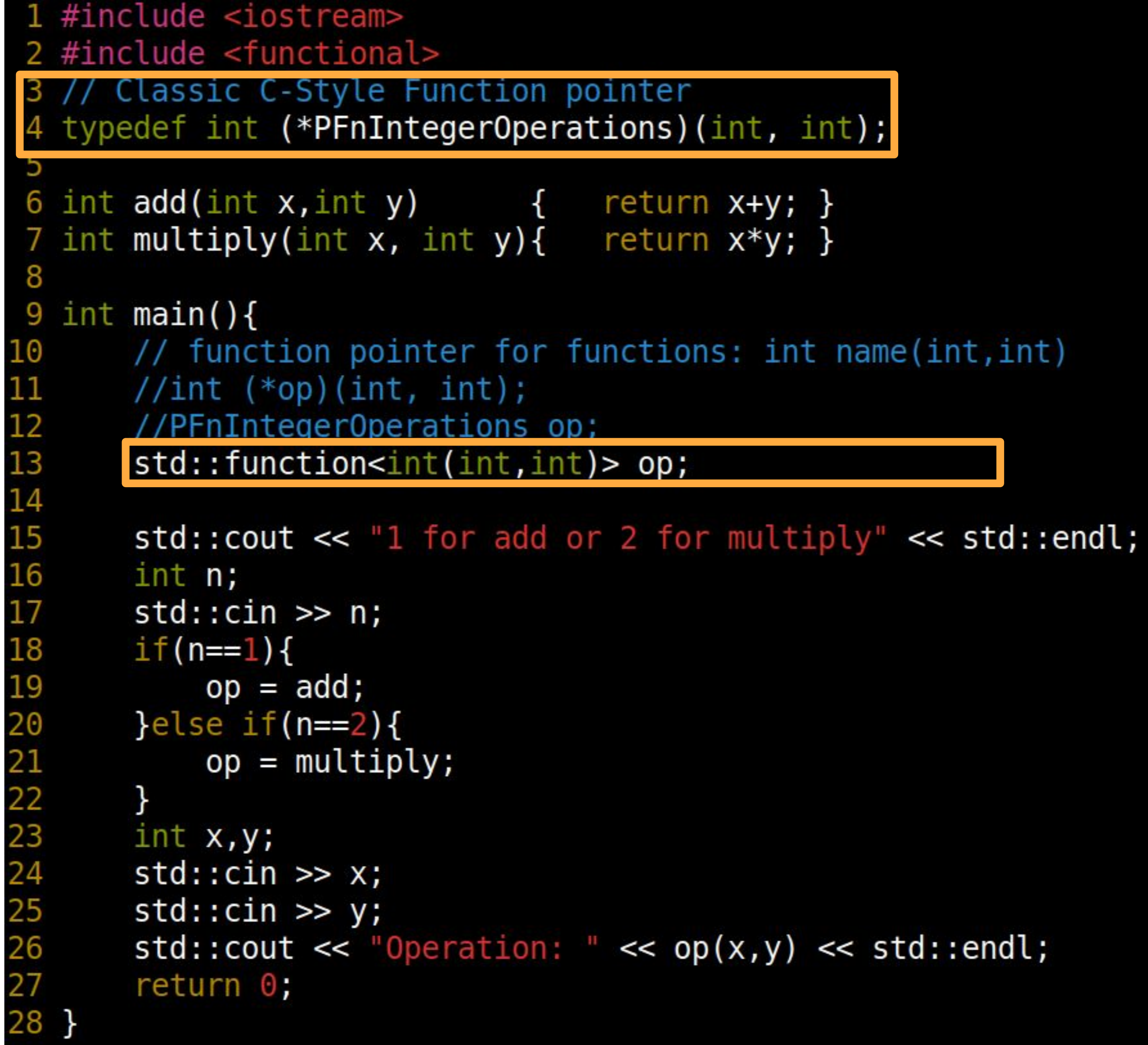
10 key moments in this video

Higher Order Functions (and more)

Passing Functions as Arguments

Function Pointers

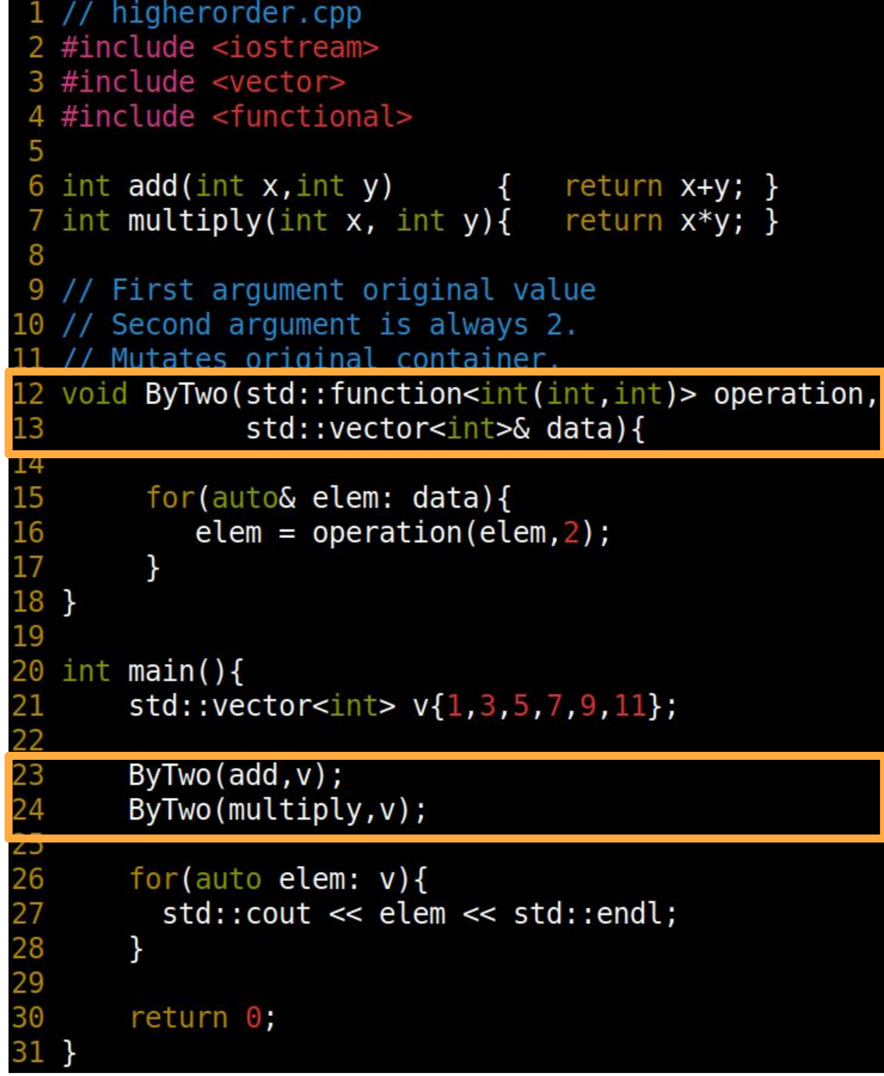
- With lambda's -- we open the door to 'pass functions around' as arguments in other functions.
- Of course this has been possible with:
  - function pointers
  - std::function -- available with C++ 11 and beyond



```cpp
#include <iostream>
#include <functional>
// Classic C-Style Function pointer
typedef int (*PFnIntegerOperations)(int, int);

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

int main(){
    // function pointer for functions: int name(int,int)
    //int (*op)(int, int);
    //PFnIntegerOperations op;
    std::function<int(int,int)> op;

    std::cout << "1 for add or 2 for multiply" << std::endl;
    int n;
    std::cin >> n;
    if(n==1){
        op = add;
    }else if(n==2){
        op = multiply;
    }
    int x,y;
    std::cin >> x;
    std::cin >> y;
    std::cout << "Operation: " << op(x,y) << std::endl;
    return 0;
}
```

Higher-Order Functions (HOF)

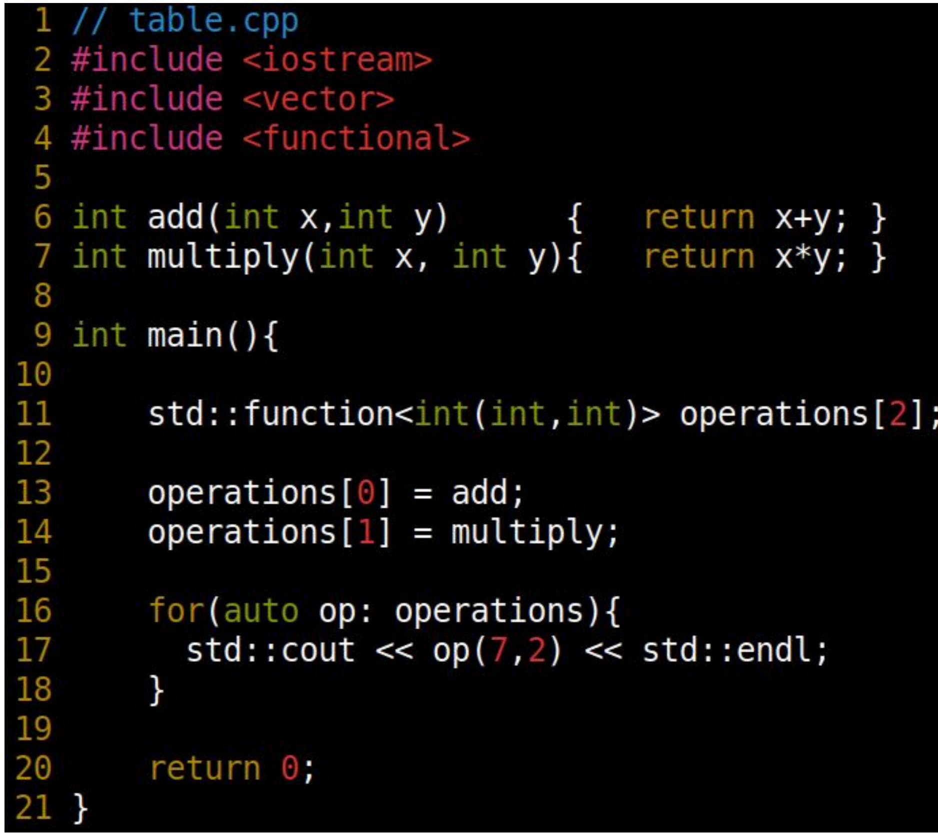
- A specific use case std::function, is to pass it as a function parameter
    - This is known as a **higher order function**
- Observe how we can pass in a `std::function` that affects the behavior of the function 'ByTwo' in this example
- Note:
    - std::function is a bit more powerful than regular function pointers
        - It's cleaner to type and easier to search for
        - It can hold any callable object
            - It *may* allocate memory



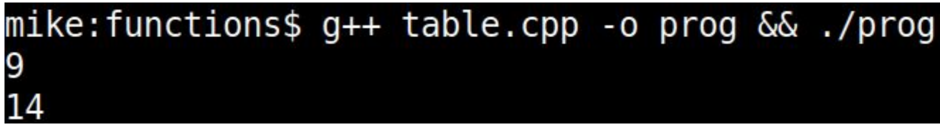
```cpp
// higherorder.cpp
#include <iostream>
#include <vector>
#include <functional>

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

// First argument original value
// Second argument is always 2.
// Mutates original container.
void ByTwo(std::function<int(int,int)> operation,
           std::vector<int>& data){

     for(auto& elem: data){
        elem = operation(elem,2);
     }
}

int main(){
    std::vector<int> v{1,3,5,7,9,11};

    ByTwo(add,v);
    ByTwo(multiply,v);

    for(auto elem: v){
      std::cout << elem << std::endl;
    }

    return 0;
}
```

Storing Functions (in tables)

- In a sense when we pass functions into other functions we are storing the function behavior
  - Same thing we saw earlier
- We can of course store functions in other data structures like an array
  - This can be incredibly useful for a ‘command like’ design pattern or ‘FIFO queue’ for executing a series of functions.
  - I’ve found this idea also very useful for generating tests



```cpp
// table.cpp
#include <iostream>
#include <vector>
#include <functional>

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

int main(){

    std::function<int(int,int)> operations[2];

    operations[0] = add;
    operations[1] = multiply;

    for(auto op: operations){
      std::cout << op(7,2) << std::endl;
    }

    return 0;
}
```



```
mike:functions$ g++ table.cpp -o prog && ./prog
9
14
```

Summary

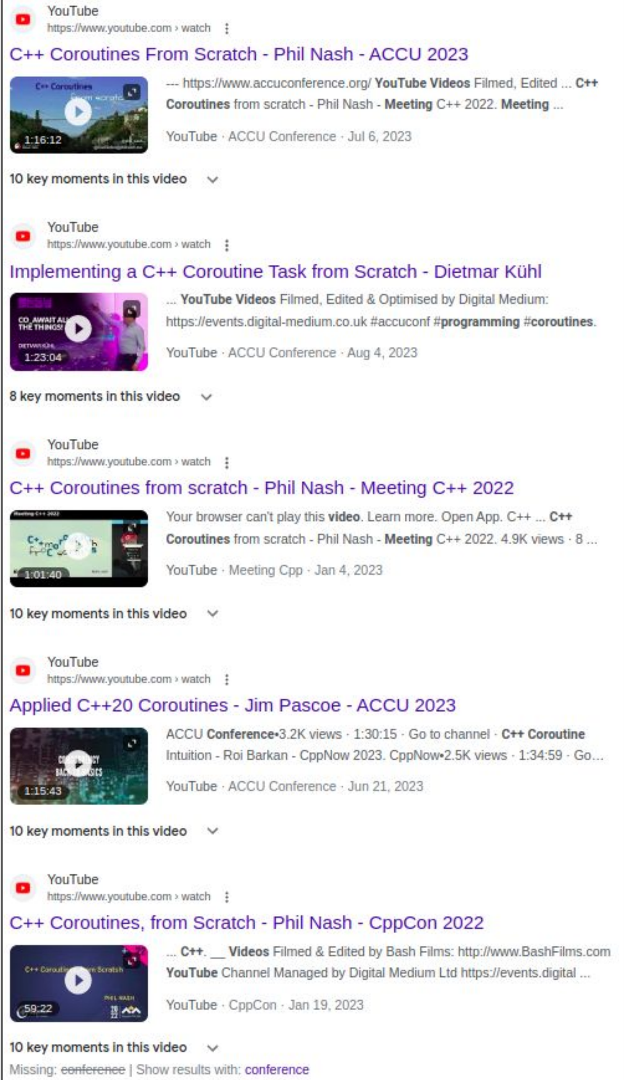
Nearly the end of our tour!

Summary

- We have touched on a lot of topics with functions -- but not yet all!
  - Primarily we have looked at functions as building blocks in our program
  - We have looked at how to organize functions into groups
  - We've talked a bit about 'state' of functions
- We can also think of functions
  - As forms of 'control' in our program
    - i.e. Coroutines are in C++20!
- There's plenty more to continue learning about functions!
  - Function Templates
  - Testing of Functions
  - Friend Functions (as related to Object-Oriented Programming)
  - Remote Procedure Calls
  - Again: I recommend checking out more of CPPCON's Back to Basics Talks ongoing this year and previous years (Keywords: Object-Oriented Programming, Lambdas, Templates)

Talks on Coroutines

- Last year’s Cppcon *Coroutines from Scratch* by Phil Nash, and several other folks have given nice talks.



YouTube
https://www.youtube.com › watch

C++ Coroutines From Scratch - Phil Nash - ACCU 2023

1:16:12

--- https://www.accuconference.org/ **YouTube Videos** Filmed, Edited ... **C++ Coroutines** from scratch - Phil Nash - **Meeting** C++ 2022. **Meeting** ...

YouTube · ACCU Conference · Jul 6, 2023

10 key moments in this video

YouTube
https://www.youtube.com › watch

Implementing a C++ Coroutine Task from Scratch - Dietmar Kühl

1:23:04

... **YouTube Videos** Filmed, Edited & Optimised by Digital Medium: https://events.digital-medium.co.uk #accuconf #**programming** #**coroutines**.

YouTube · ACCU Conference · Aug 4, 2023

8 key moments in this video

YouTube
https://www.youtube.com › watch

C++ Coroutines from scratch - Phil Nash - Meeting C++ 2022

1:01:40

Your browser can't play this **video**. Learn more. Open App. C++ ... **C++ Coroutines** from scratch - Phil Nash - **Meeting** C++ 2022. 4.9K views · 8 ...

YouTube · Meeting Cpp · Jan 4, 2023

10 key moments in this video

YouTube
https://www.youtube.com › watch

Applied C++20 Coroutines - Jim Pascoe - ACCU 2023

1:15:43

ACCU **Conference**•3.2K views · 1:30:15 · Go to channel · **C++ Coroutine** Intuition - Roi Barkan - CppNow 2023. CppNow•2.5K views · 1:34:59 · Go...

YouTube · ACCU Conference · Jun 21, 2023

10 key moments in this video

YouTube
https://www.youtube.com › watch

C++ Coroutines, from Scratch - Phil Nash - CppCon 2022

59:22

... **C++**. __ **Videos** Filmed & Edited by Bash Films: http://www.BashFilms.com **YouTube** Channel Managed by Digital Medium Ltd https://events.digital ...

YouTube · CppCon · Jan 19, 2023

10 key moments in this video

Missing: ~~conference~~ | Show results with: conference

Bonus (If Time Remains) Function Tips

A few quick tips and ideas for better functions

Function Naming Conventions

- It’s a good idea to give useful names to your function.
- Names should be descriptive enough to describe ‘the action’ the function is doing.
  - std::vector<uint8_t> Bitmap(); -- bad (‘I need more information’)
  - std::vector<uint8_t> GetBitmap() -- okay (‘Get’ tells us some action)
  - std::vector<uint8_t> GetBitmapAsByteVector() -- best (‘Documents action and value returned’)
- It may also be useful to uniformly name functions as well
  - i.e.
    - Include the word ‘Bitmap’ in all related functions operating on Bitmaps.
    - Some functions (even in the same namespace) prefix with some letters

Functions Should Have One Job

- Because the function returns at most ‘1’ value, that’s an indicator that our function should only do one thing
    - e.g. In the Standard Template Library (STL) pop() only removes an element, when in fact it could probably also return the value.
        - This makes functions more composable if you want a ‘popAndGetValue()’ function
        - This makes the function more testable on expected behavior
    - Core Guideline: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#f2-a-function-should-perform-a-single-logical-operation

Keep Functions Short

- “Short” here is subjective to one's domain and experience
  - I learned < 50 lines -- ‘50’ was arbitrary and probably the right number in university in which programs are not massive.
    - I’ve seen perfectly fine functions 1,000 lines long.
      - Initialization code of some system tends to be the common use case.
  - The point is -- if you have too much code in a function, it may be doing either:
    - Too many jobs
    - Be overly complex and difficult to maintain
      - i.e. If there are no git diff’s for years on a massive function -- is it because everyone is too afraid to modify that code? (Or is it actually perfect?)
  - Core Guideline:
    - https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#f3-keep-functions-short-and-simple
  - For Folks who want more performance -- consider inlining
    - Core Guideline: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#f5-if-a-function-is-very-small-and-time-critical-declare-it-inline

Function Testing Conventions

- It’s a good idea to then ‘test’ function as you write them as well
  - Test-Driven Development dictates that you write the test first, then implement the function body.
- Functions are also great to use as ‘pre’ and ‘post’ conditions (i.e. contracts)

Passing lots of parameters to your functions?

- It’s probably best to pass in a ‘struct’ or perhaps a ‘pointer to a struct’
- How many is ‘a lot’
  - Depends. I’ll say around 5 is when I personally get nervous and really have to think.
- Here’s one strategy -- pack everything into a struct
  - void myFunction(OptionsStruct options);
- Another
  - void myFunction(OptionsStruct* options);
  - Pass in as a pointer (or smart pointer) to ensure we’re always just passing in an ‘8-byte’ address (on a 64-bit architecture).

Cppcon The C++ Conference | 2023 October 01-06 Aurora, Colorado, USA

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube:
www.youtube.com/c/MikeShah

Back To Basics Functions

14:0-15:00 MDT
Mon, Oct. 1 2023

60 minutes
Introductory Audience

Thank you!

MIKE SHAH

Cppcon The C++ Conference

2023 October 01 - 06