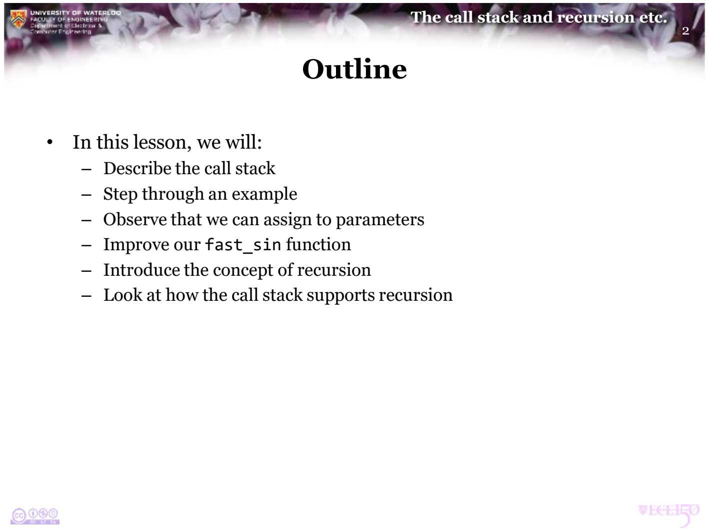
ECE 150 *Fundamentals of Programming*

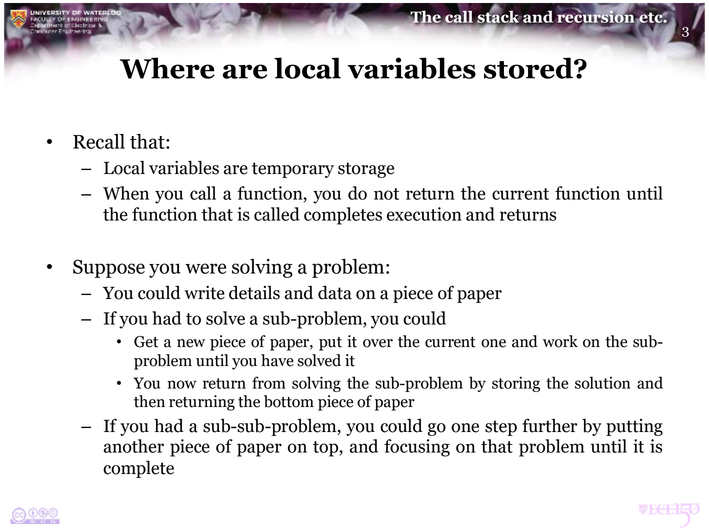
The call stack

Douglas Wilhelm Harder, M.Math.
Prof. Hiren Patel, Ph.D.
hiren.patel@uwaterloo.ca dwharder@uwaterloo.ca



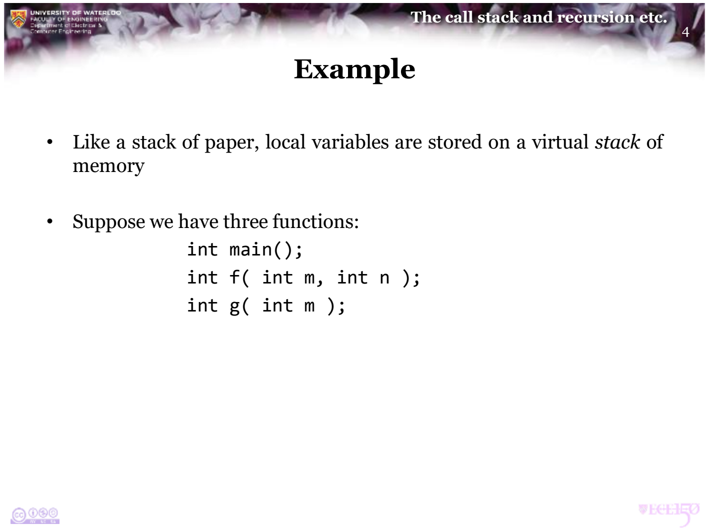
Outline

- In this lesson, we will:
  - Describe the call stack
  - Step through an example
  - Observe that we can assign to parameters
  - Improve our `fast_sin` function
  - Introduce the concept of recursion
  - Look at how the call stack supports recursion



Where are local variables stored?

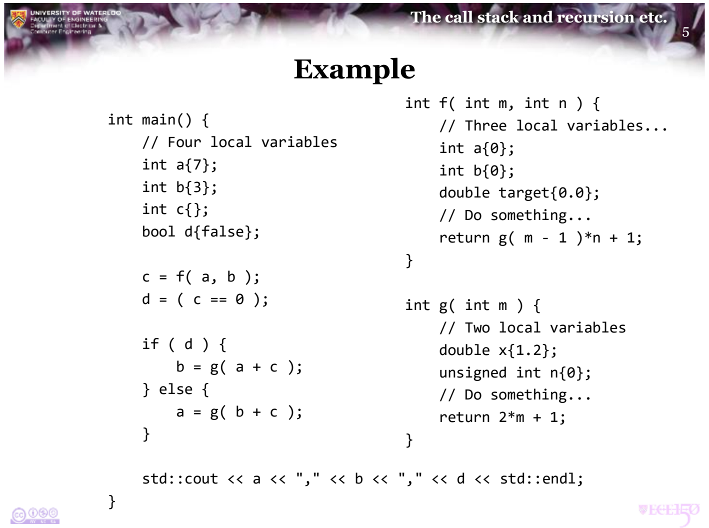
- Recall that:
  - Local variables are temporary storage
  - When you call a function, you do not return the current function until the function that is called completes execution and returns
- Suppose you were solving a problem:
  - You could write details and data on a piece of paper
  - If you had to solve a sub-problem, you could
    - Get a new piece of paper, put it over the current one and work on the sub-problem until you have solved it
    - You now return from solving the sub-problem by storing the solution and then returning the bottom piece of paper
  - If you had a sub-sub-problem, you could go one step further by putting another piece of paper on top, and focusing on that problem until it is complete



Example

- Like a stack of paper, local variables are stored on a virtual *stack* of memory
- Suppose we have three functions:

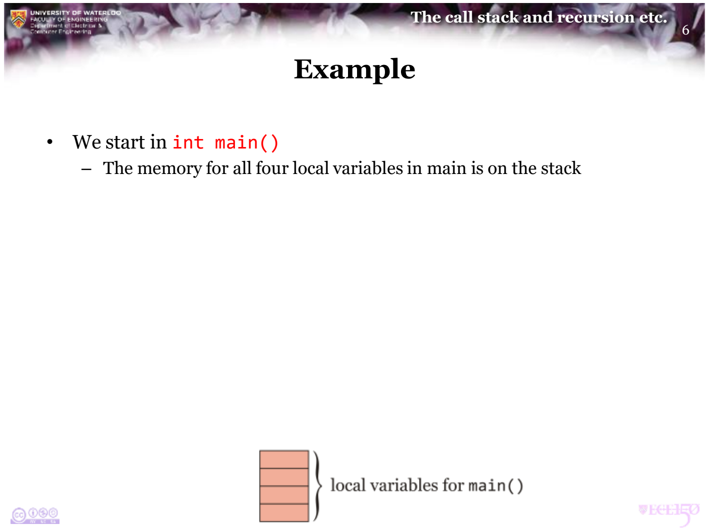
```
int main();
int f( int m, int n );
int g( int m );
```



Example

```
int main() {
    // Four local variables
    int a{7};
    int b{3};
    int c{};
    bool d{false};

    c = f( a, b );
    d = ( c == 0 );

    if ( d ) {
        b = g( a + c );
    } else {
        a = g( b + c );
    }

    std::cout << a << "," << b << "," << d << std::endl;
}
```

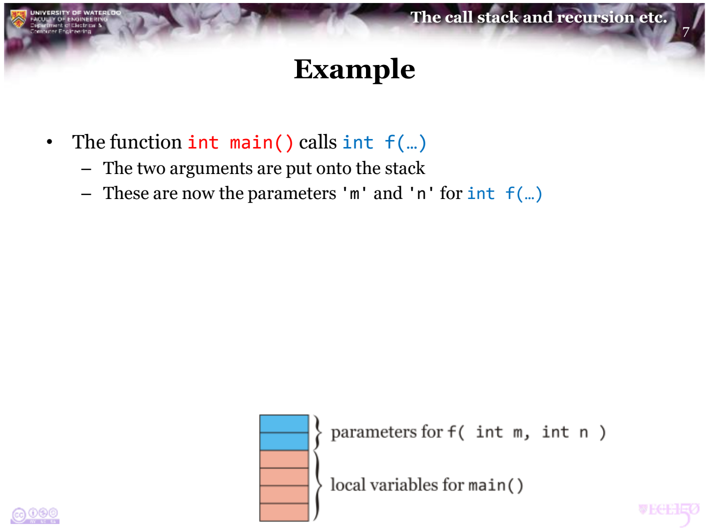
```
int f( int m, int n ) {
    // Three local variables...
    int a{0};
    int b{0};
    double target{0.0};
    // Do something...
    return g( m - 1 )*n + 1;
}

int g( int m ) {
    // Two local variables
    double x{1.2};
    unsigned int n{0};
    // Do something...
    return 2*m + 1;
}
```



Example

- We start in `int main()`
  - The memory for all four local variables in main is on the stack

local variables for `main()`

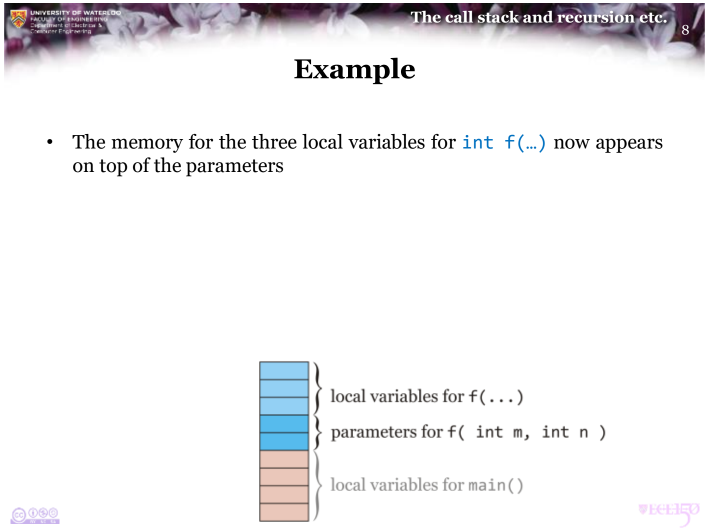


Example

- The function `int main()` calls `int f(…)`
  - The two arguments are put onto the stack
  - These are now the parameters `'m'` and `'n'` for `int f(…)`

parameters for `f( int m, int n )`

local variables for `main()`



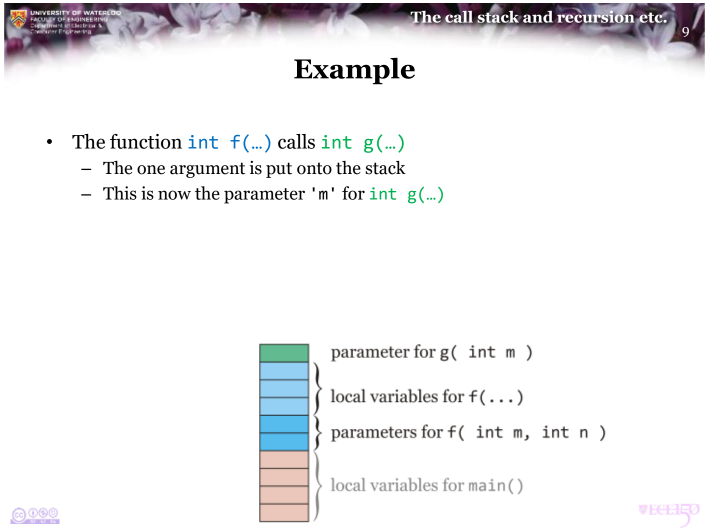
Example

- The memory for the three local variables for `int f(…)` now appears on top of the parameters

local variables for `f(...)`

parameters for `f( int m, int n )`

local variables for `main()`



The call stack and recursion etc. 9

Example

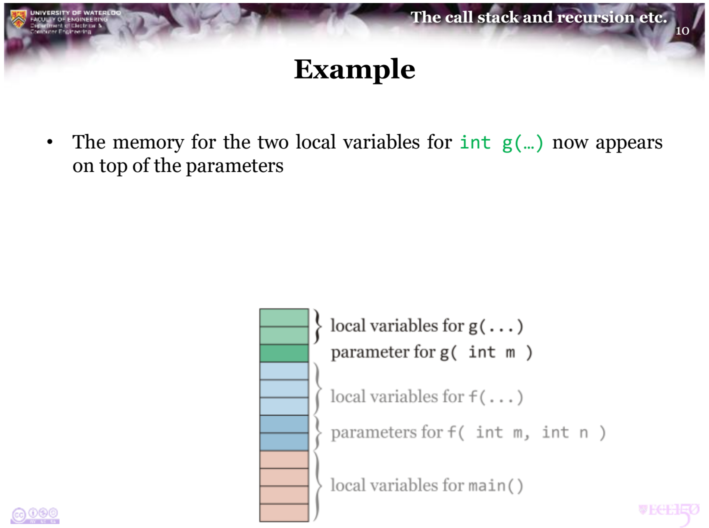
- The function `int f(…)` calls `int g(…)`
  - The one argument is put onto the stack
  - This is now the parameter `'m'` for `int g(…)`

parameter for `g( int m )`

local variables for `f(...)`

parameters for `f( int m, int n )`

local variables for `main()`



The call stack and recursion etc. 10

Example

- The memory for the two local variables for `int g(…)` now appears on top of the parameters

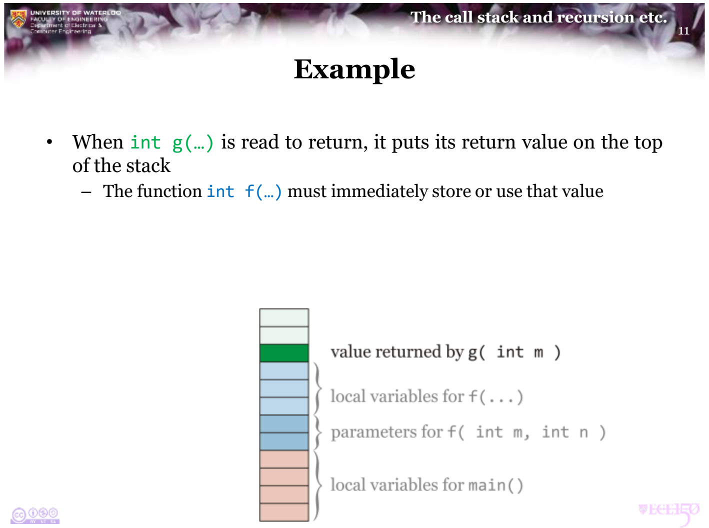
local variables for `g(...)`

parameter for `g( int m )`

local variables for `f(...)`

parameters for `f( int m, int n )`

local variables for `main()`



The call stack and recursion etc. 11

Example

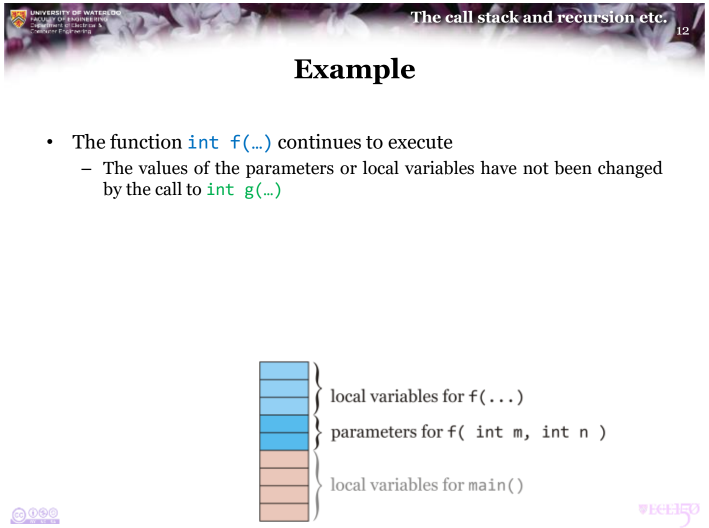
- When `int g(…)` is read to return, it puts its return value on the top of the stack
  - The function `int f(…)` must immediately store or use that value

value returned by `g( int m )`

local variables for `f(...)`

parameters for `f( int m, int n )`

local variables for `main()`



The call stack and recursion etc. 12

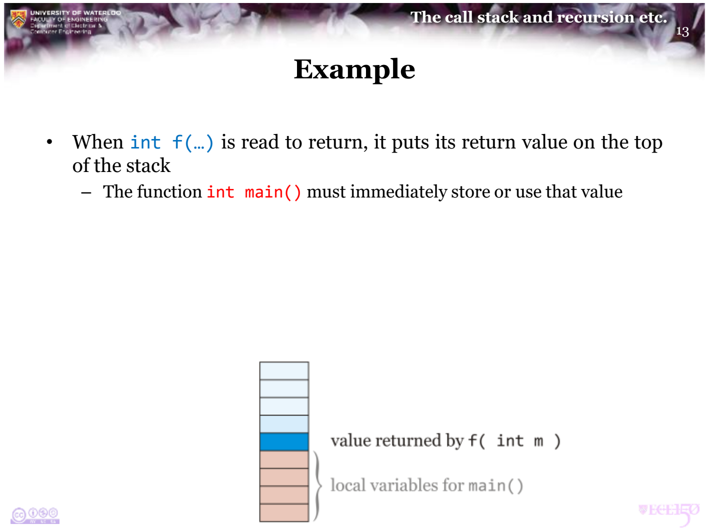
Example

- The function `int f(…)` continues to execute
  - The values of the parameters or local variables have not been changed by the call to `int g(…)`

local variables for `f(...)`

parameters for `f( int m, int n )`

local variables for `main()`



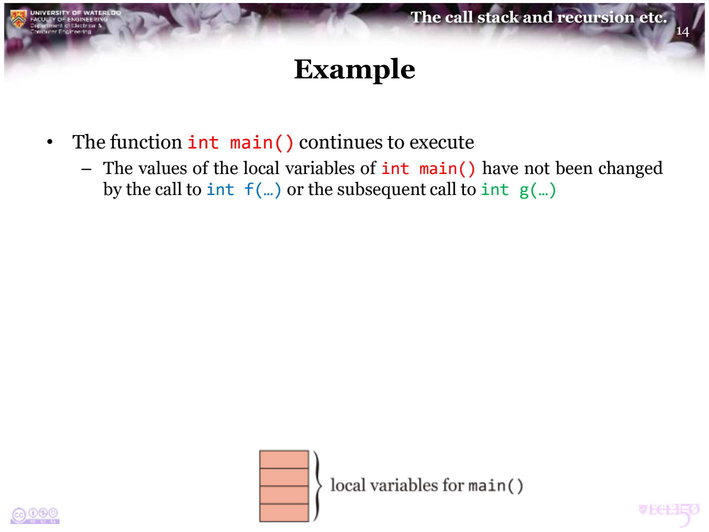
The call stack and recursion etc. 13

Example

- When `int f(…)` is read to return, it puts its return value on the top of the stack
  - The function `int main()` must immediately store or use that value

value returned by `f( int m )`

local variables for `main()`

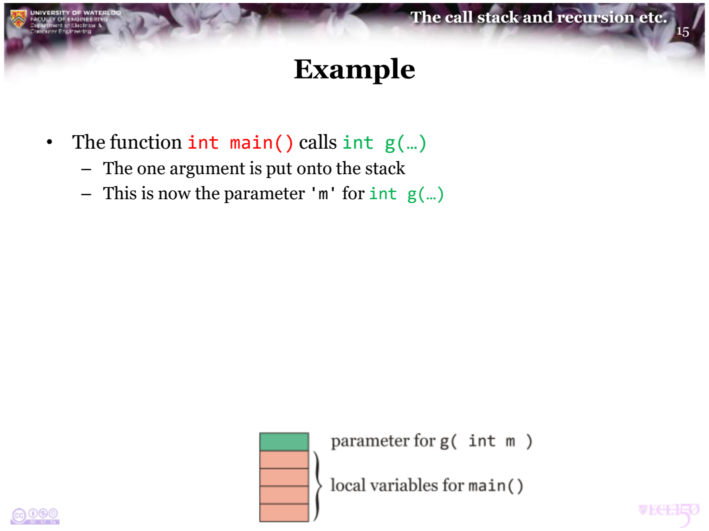


The call stack and recursion etc. 14

Example

- The function `int main()` continues to execute
  - The values of the local variables of `int main()` have not been changed by the call to `int f(…)` or the subsequent call to `int g(…)`

local variables for `main()`



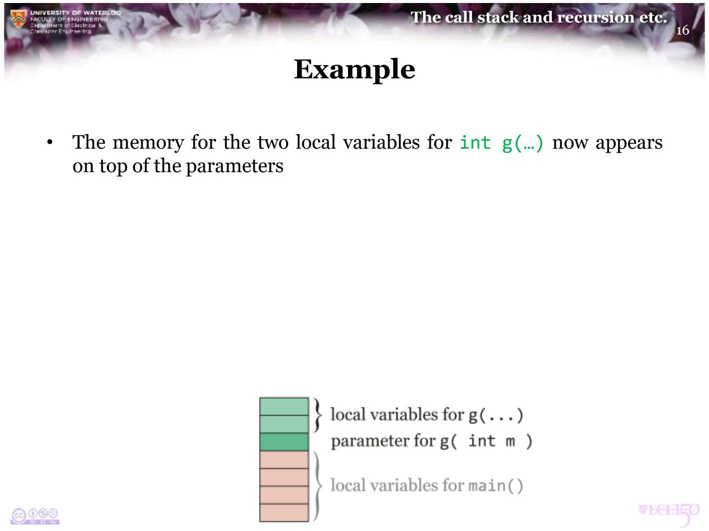
The call stack and recursion etc. 15

Example

- The function `int main()` calls `int g(…)`
  - The one argument is put onto the stack
  - This is now the parameter `'m'` for `int g(…)`

parameter for `g( int m )`

local variables for `main()`



The call stack and recursion etc. 16

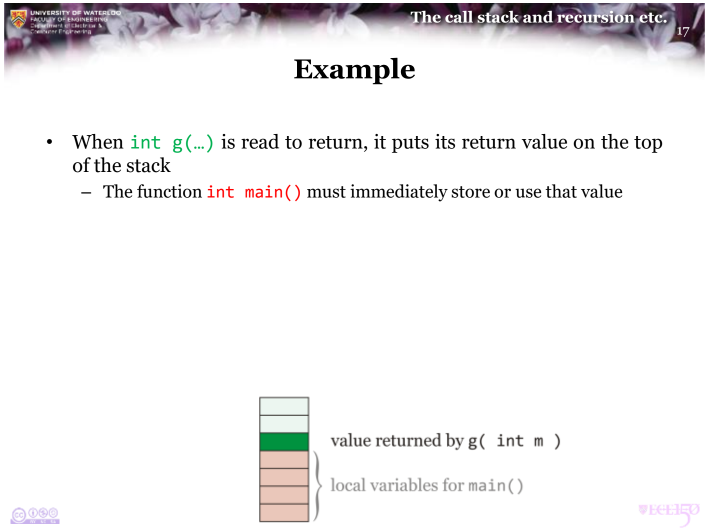
Example

- The memory for the two local variables for `int g(…)` now appears on top of the parameters

local variables for `g(...)`

parameter for `g( int m )`

local variables for `main()`

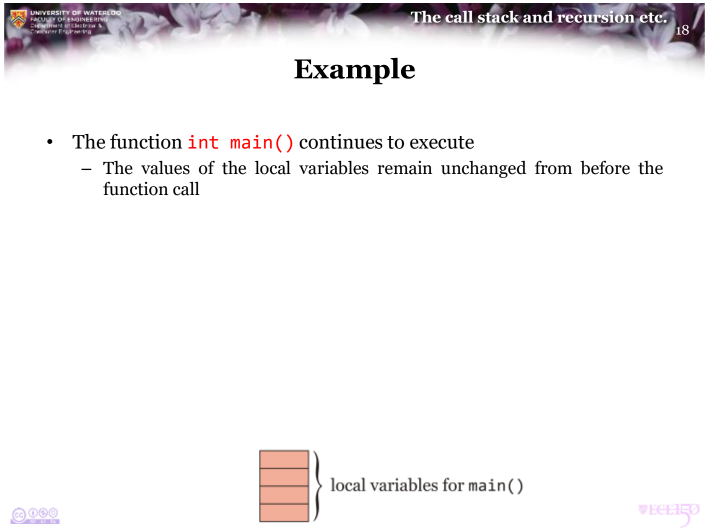


Example

- When `int g(…)` is read to return, it puts its return value on the top of the stack
  - The function `int main()` must immediately store or use that value

value returned by `g( int m )`

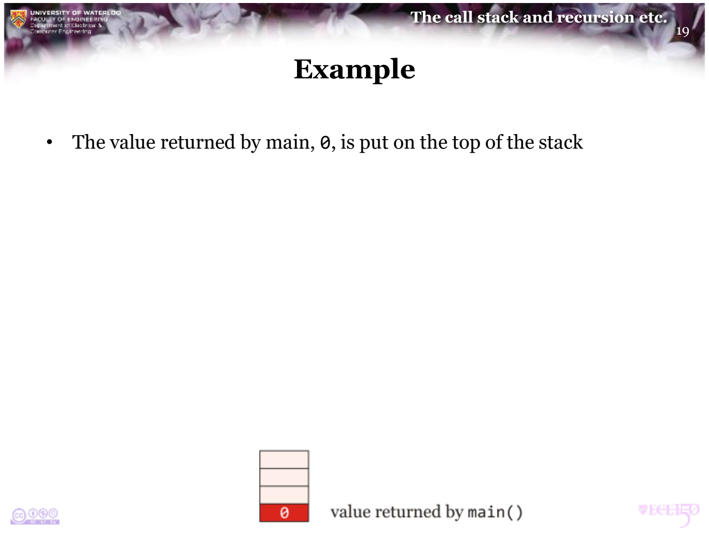
local variables for `main()`



Example

- The function `int main()` continues to execute
  - The values of the local variables remain unchanged from before the function call

local variables for `main()`

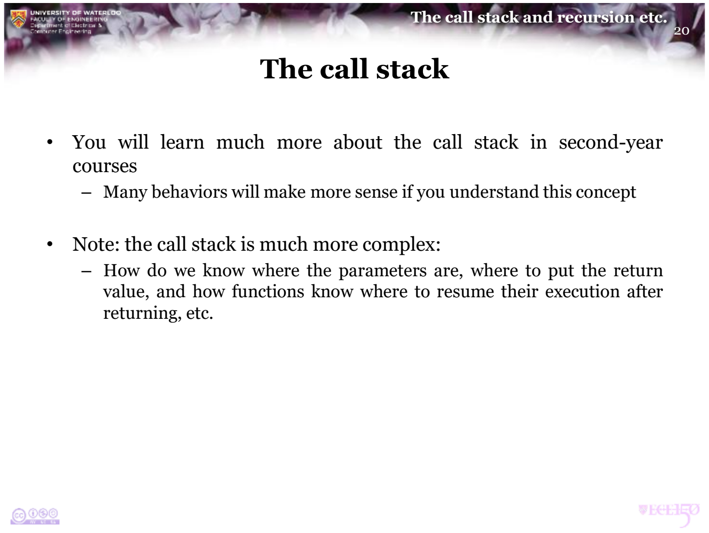


Example

- The value returned by main, `0`, is put on the top of the stack

0

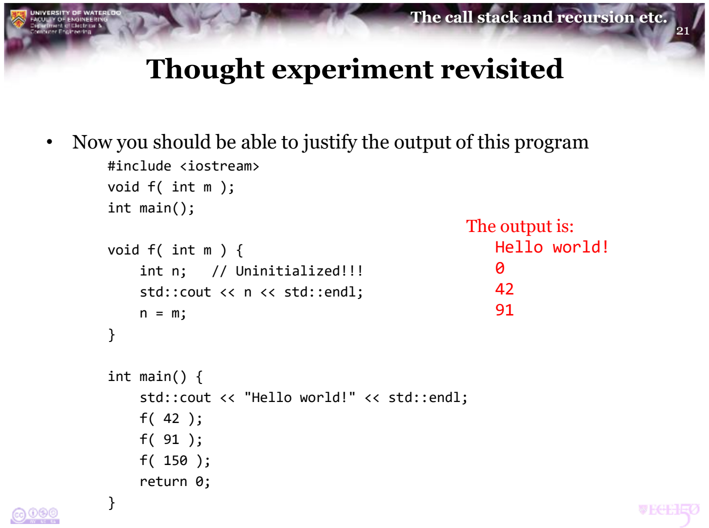
value returned by `main()`



The call stack

- You will learn much more about the call stack in second-year courses
  - Many behaviors will make more sense if you understand this concept

- Note: the call stack is much more complex:
  - How do we know where the parameters are, where to put the return value, and how functions know where to resume their execution after returning, etc.



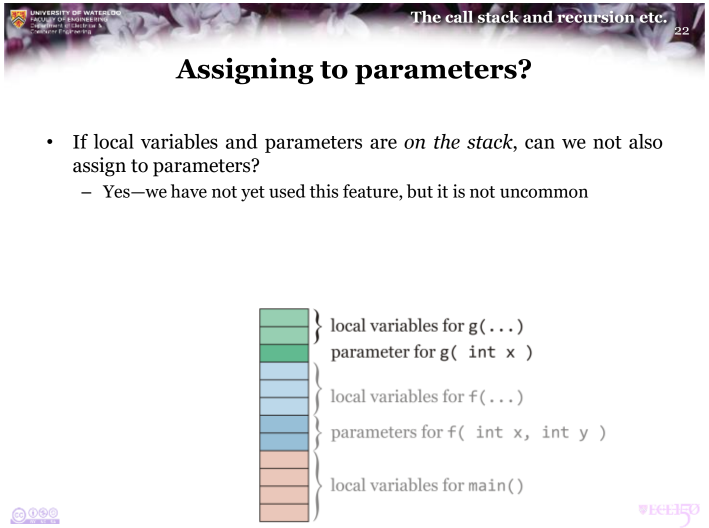
Thought experiment revisited

- Now you should be able to justify the output of this program

```
#include <iostream>
void f( int m );
int main();

void f( int m ) {
    int n;   // Uninitialized!!!
    std::cout << n << std::endl;
    n = m;
}

int main() {
    std::cout << "Hello world!" << std::endl;
    f( 42 );
    f( 91 );
    f( 150 );
    return 0;
}
```

The output is:

```
Hello world!
0
42
91
```



Assigning to parameters?

- If local variables and parameters are *on the stack*, can we not also assign to parameters?
  - Yes—we have not yet used this feature, but it is not uncommon

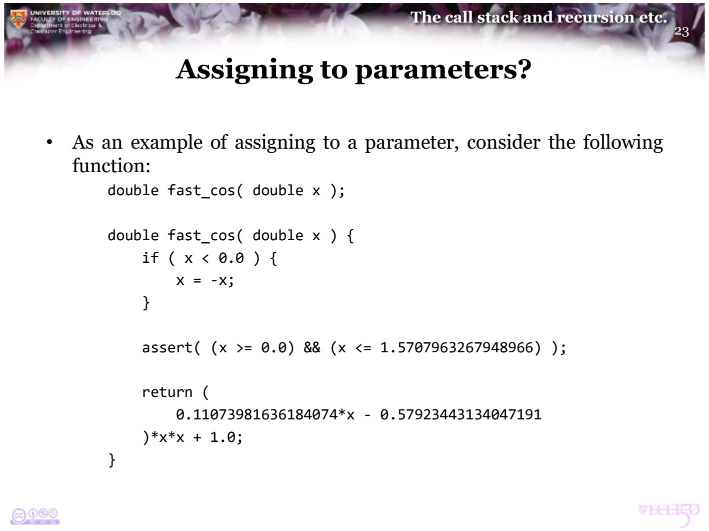
local variables for `g(...)`

parameter for `g( int x )`

local variables for `f(...)`

parameters for `f( int x, int y )`

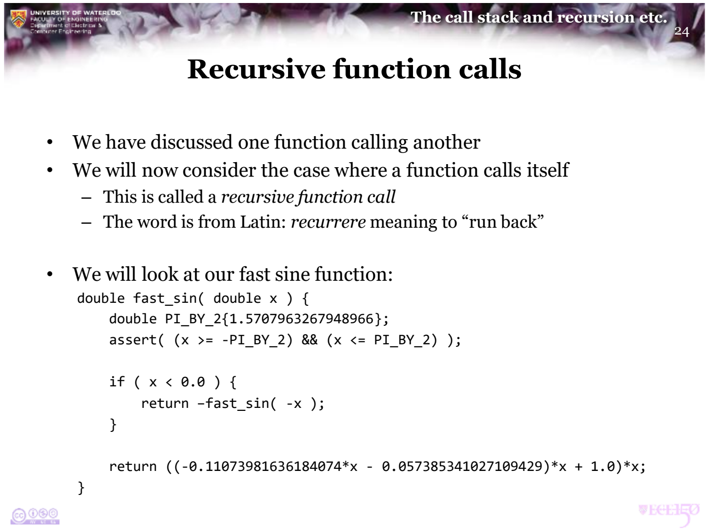
local variables for `main()`



Assigning to parameters?

- As an example of assigning to a parameter, consider the following function:

```
double fast_cos( double x );

double fast_cos( double x ) {
    if ( x < 0.0 ) {
        x = -x;
    }

    assert( (x >= 0.0) && (x <= 1.5707963267948966) );

    return (
        0.110739816361840 74*x - 0.57923443134047191
    )*x*x + 1.0;
}
```

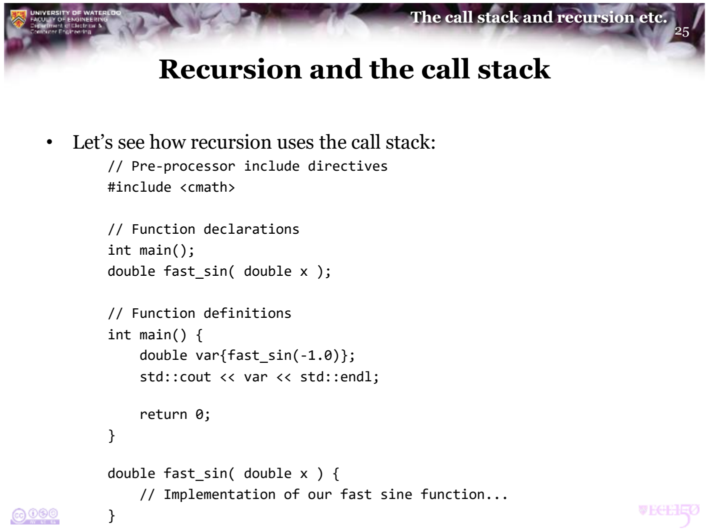


Recursive function calls

- We have discussed one function calling another
- We will now consider the case where a function calls itself
  - This is called a *recursive function call*
  - The word is from Latin: *recurrere* meaning to “run back”

- We will look at our fast sine function:

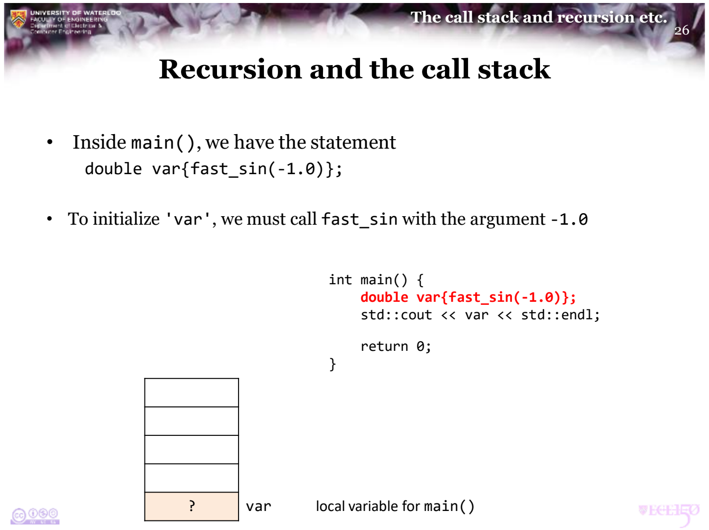
```
double fast_sin( double x ) {
    double PI_BY_2{1.5707963267948966};
    assert( (x >= -PI_BY_2) && (x <= PI_BY_2) );

    if ( x < 0.0 ) {
        return -fast_sin( -x );
    }

    return ((-0.110739816361840 74*x - 0.057385341027109429)*x + 1.0)*x;
}
```



Recursion and the call stack

- Let's see how recursion uses the call stack:

```
// Pre-processor include directives
#include <cmath>

// Function declarations
int main();
double fast_sin( double x );

// Function definitions
int main() {
    double var{fast_sin(-1.0)};
    std::cout << var << std::endl;

    return 0;
}

double fast_sin( double x ) {
    // Implementation of our fast sine function...
}
```

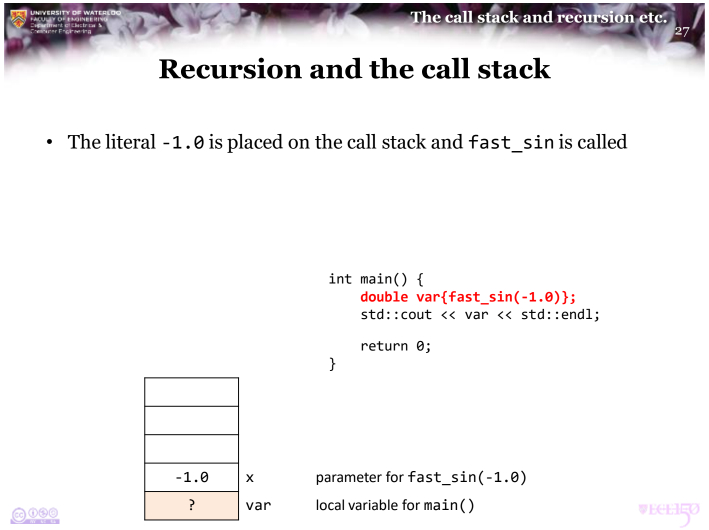


Recursion and the call stack

- Inside `main()`, we have the statement
  `double var{fast_sin(-1.0)};`
- To initialize `'var'`, we must call `fast_sin` with the argument `-1.0`

```
int main() {
    double var{fast_sin(-1.0)};
    std::cout << var << std::endl;

    return 0;
}
```

?	var	local variable for `main()`

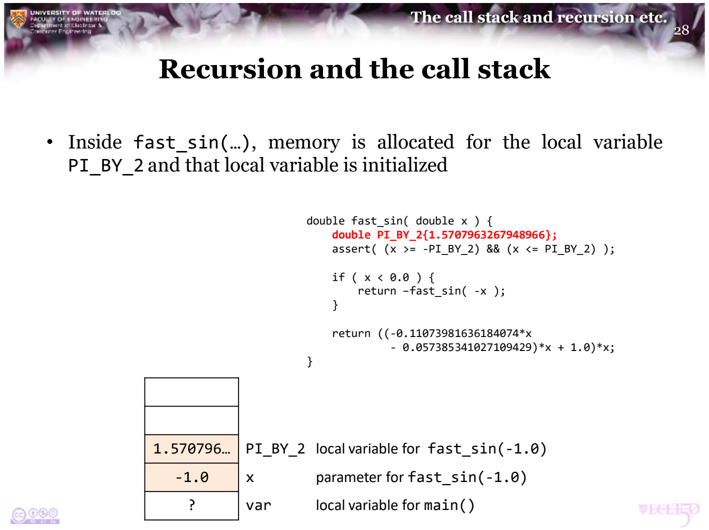


Recursion and the call stack

- The literal `-1.0` is placed on the call stack and `fast_sin` is called

```
int main() {
    double var{fast_sin(-1.0)};
    std::cout << var << std::endl;

    return 0;
}
```

-1.0	x	parameter for `fast_sin(-1.0)`
?	var	local variable for `main()`

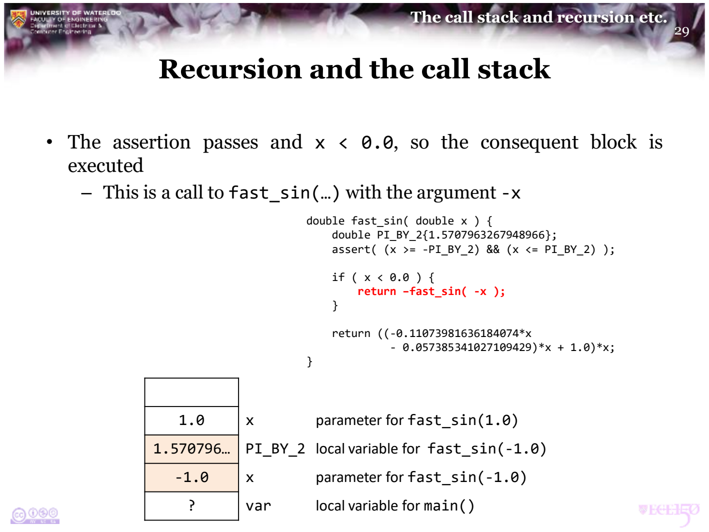


Recursion and the call stack

- Inside `fast_sin(…)`, memory is allocated for the local variable `PI_BY_2` and that local variable is initialized

```
double fast_sin( double x ) {
    double PI_BY_2{1.5707963267948966};
    assert( (x >= -PI_BY_2) && (x <= PI_BY_2) );

    if ( x < 0.0 ) {
        return –fast_sin( -x );
    }

    return ((-0.11073981636184074*x
               - 0.057385341027109429)*x + 1.0)*x;
}
```

1.570796…	PI_BY_2	local variable for `fast_sin(-1.0)`
-1.0	x	parameter for `fast_sin(-1.0)`
?	var	local variable for `main()`

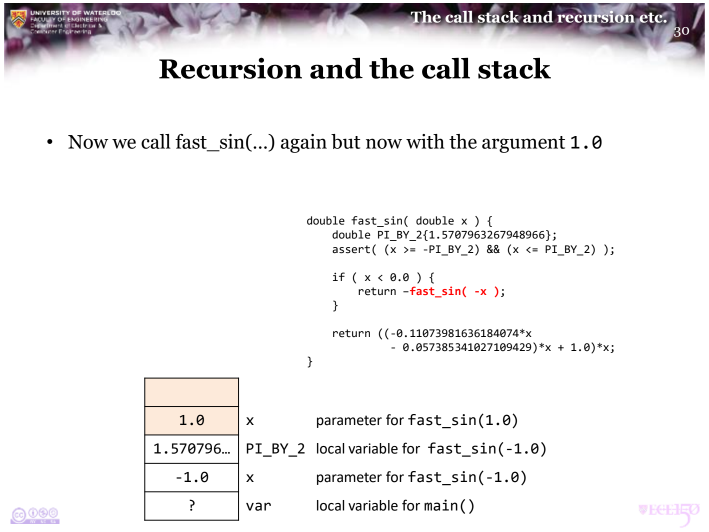


Recursion and the call stack

- The assertion passes and `x < 0.0`, so the consequent block is executed
  - This is a call to `fast_sin(…)` with the argument `-x`

```
double fast_sin( double x ) {
    double PI_BY_2{1.5707963267948966};
    assert( (x >= -PI_BY_2) && (x <= PI_BY_2) );

    if ( x < 0.0 ) {
        return -fast_sin( -x );
    }

    return ((-0.11073981636184074*x
             - 0.057385341027109429)*x + 1.0)*x;
}
```

1.0	x	parameter for `fast_sin(1.0)`
1.570796…	PI_BY_2	local variable for `fast_sin(-1.0)`
-1.0	x	parameter for `fast_sin(-1.0)`
?	var	local variable for `main()`

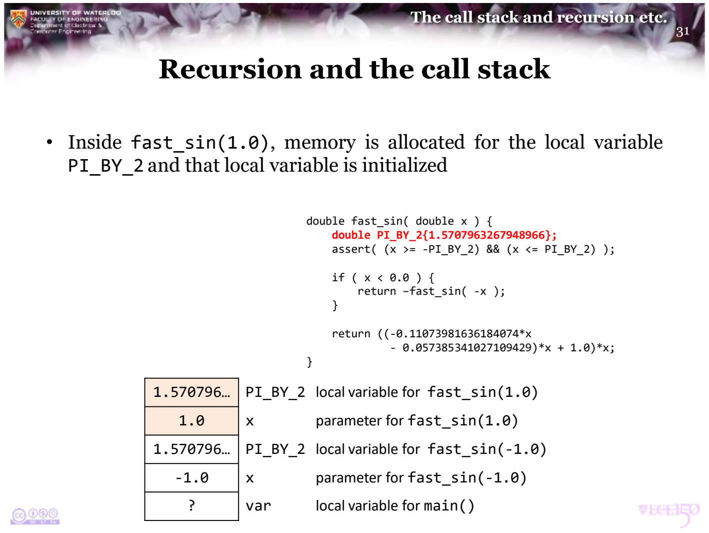


Recursion and the call stack

- Now we call fast_sin(…) again but now with the argument `1.0`

```
double fast_sin( double x ) {
    double PI_BY_2{1.5707963267948966};
    assert( (x >= -PI_BY_2) && (x <= PI_BY_2) );

    if ( x < 0.0 ) {
        return -fast_sin( -x );
    }

    return ((-0.11073981636184074*x
             - 0.057385341027109429)*x + 1.0)*x;
}
```

1.0	x	parameter for `fast_sin(1.0)`
1.570796…	PI_BY_2	local variable for `fast_sin(-1.0)`
-1.0	x	parameter for `fast_sin(-1.0)`
?	var	local variable for `main()`

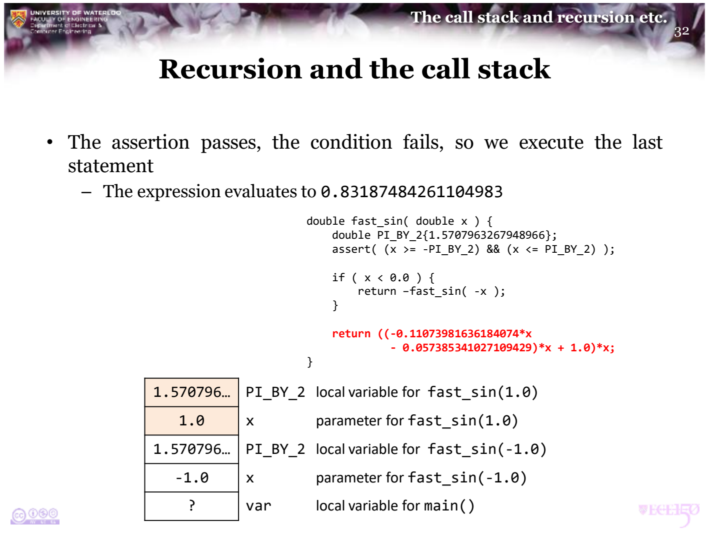


Recursion and the call stack

- Inside `fast_sin(1.0)`, memory is allocated for the local variable `PI_BY_2` and that local variable is initialized

```
double fast_sin( double x ) {
    double PI_BY_2{1.5707963267948966};
    assert( (x >= -PI_BY_2) && (x <= PI_BY_2) );

    if ( x < 0.0 ) {
        return -fast_sin( -x );
    }

    return ((-0.11073981636184074*x
             - 0.057385341027109429)*x + 1.0)*x;
}
```

1.570796…	PI_BY_2	local variable for `fast_sin(1.0)`
1.0	x	parameter for `fast_sin(1.0)`
1.570796…	PI_BY_2	local variable for `fast_sin(-1.0)`
-1.0	x	parameter for `fast_sin(-1.0)`
?	var	local variable for `main()`

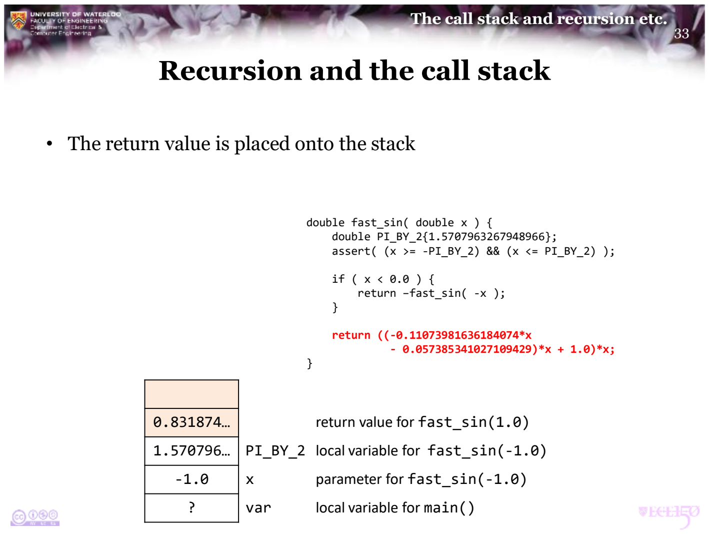


Recursion and the call stack

- The assertion passes, the condition fails, so we execute the last statement
  - The expression evaluates to `0.83187484261104983`

```
double fast_sin( double x ) {
    double PI_BY_2{1.5707963267948966};
    assert( (x >= -PI_BY_2) && (x <= PI_BY_2) );

    if ( x < 0.0 ) {
        return -fast_sin( -x );
    }

    return ((-0.11073981636184074*x
             - 0.057385341027109429)*x + 1.0)*x;
}
```

1.570796…	PI_BY_2	local variable for `fast_sin(1.0)`
1.0	x	parameter for `fast_sin(1.0)`
1.570796…	PI_BY_2	local variable for `fast_sin(-1.0)`
-1.0	x	parameter for `fast_sin(-1.0)`
?	var	local variable for `main()`

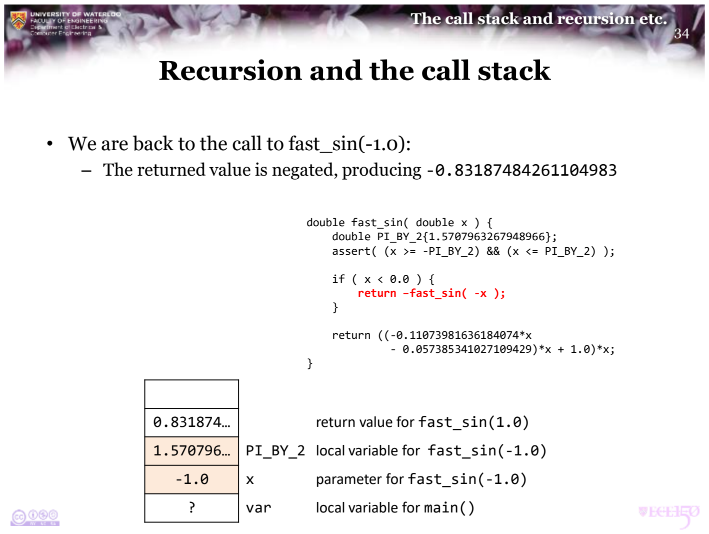


Recursion and the call stack

- The return value is placed onto the stack

```
double fast_sin( double x ) {
    double PI_BY_2{1.5707963267948966};
    assert( (x >= -PI_BY_2) && (x <= PI_BY_2) );

    if ( x < 0.0 ) {
        return -fast_sin( -x );
    }

    return ((-0.11073981636184074*x
             - 0.057385341027109429)*x + 1.0)*x;
}
```

0.831874…		return value for fast_sin(1.0)
1.570796…	PI_BY_2	local variable for fast_sin(-1.0)
-1.0	x	parameter for fast_sin(-1.0)
?	var	local variable for main()

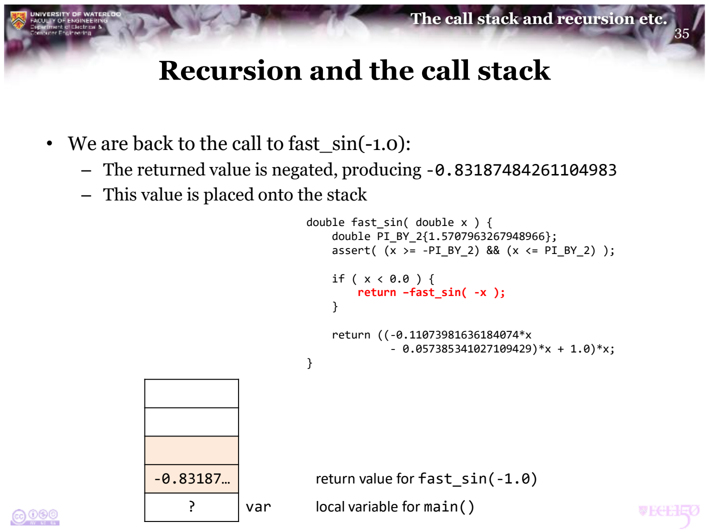


Recursion and the call stack

- We are back to the call to fast_sin(-1.0):
  - The returned value is negated, producing -0.831874842611049831

```
double fast_sin( double x ) {
    double PI_BY_2{1.5707963267948966};
    assert( (x >= -PI_BY_2) && (x <= PI_BY_2) );

    if ( x < 0.0 ) {
        return -fast_sin( -x );
    }

    return ((-0.11073981636184074*x
             - 0.057385341027109429)*x + 1.0)*x;
}
```

0.831874…		return value for fast_sin(1.0)
1.570796…	PI_BY_2	local variable for fast_sin(-1.0)
-1.0	x	parameter for fast_sin(-1.0)
?	var	local variable for main()

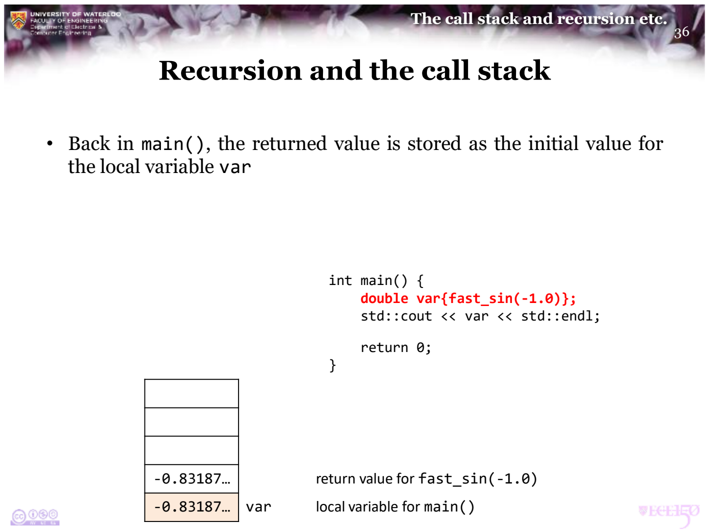


Recursion and the call stack

- We are back to the call to fast_sin(-1.0):
  - The returned value is negated, producing -0.831874842611049831
  - This value is placed onto the stack

```
double fast_sin( double x ) {
    double PI_BY_2{1.5707963267948966};
    assert( (x >= -PI_BY_2) && (x <= PI_BY_2) );

    if ( x < 0.0 ) {
        return -fast_sin( -x );
    }

    return ((-0.11073981636184074*x
             - 0.057385341027109429)*x + 1.0)*x;
}
```

-0.83187…		return value for fast_sin(-1.0)
?	var	local variable for main()

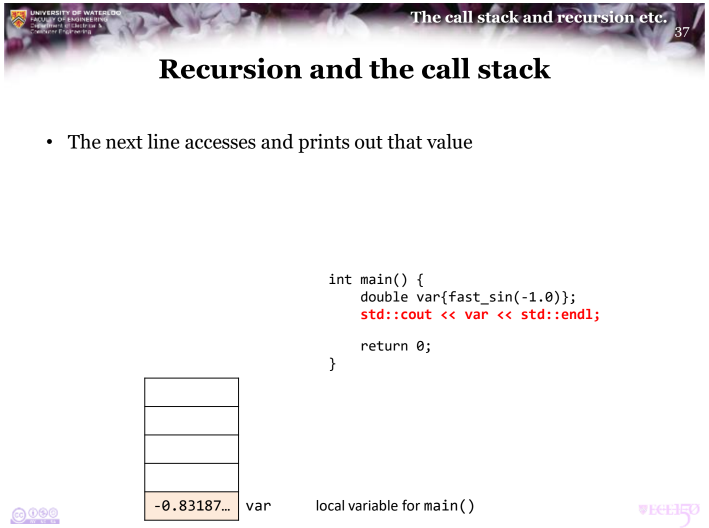


Recursion and the call stack

- Back in main(), the returned value is stored as the initial value for the local variable var

```
int main() {
    double var{fast_sin(-1.0)};
    std::cout << var << std::endl;

    return 0;
}
```

-0.83187…		return value for fast_sin(-1.0)
-0.83187…	var	local variable for main()



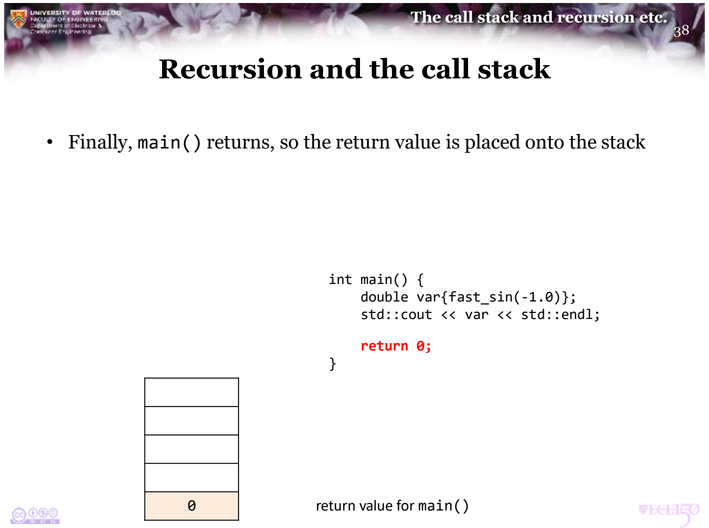
Recursion and the call stack

- The next line accesses and prints out that value

```
int main() {
    double var{fast_sin(-1.0)};
    std::cout << var << std::endl;

    return 0;
}
```

-0.83187… var

local variable for main()



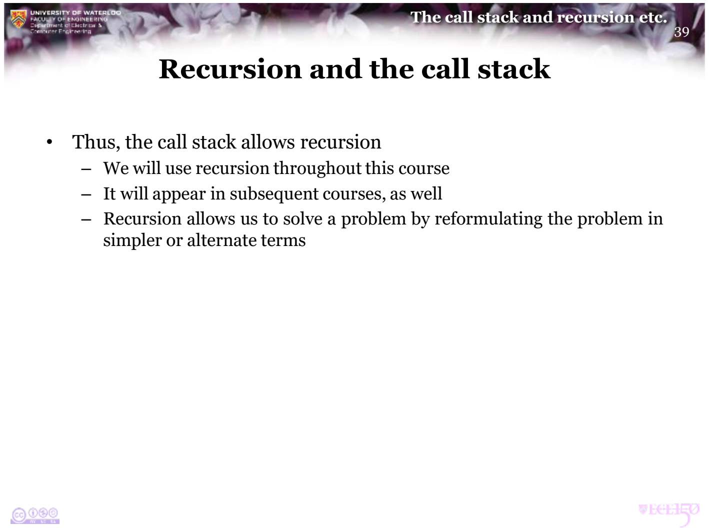
Recursion and the call stack

- Finally, `main()` returns, so the return value is placed onto the stack

```
int main() {
    double var{fast_sin(-1.0)};
    std::cout << var << std::endl;

    return 0;
}
```

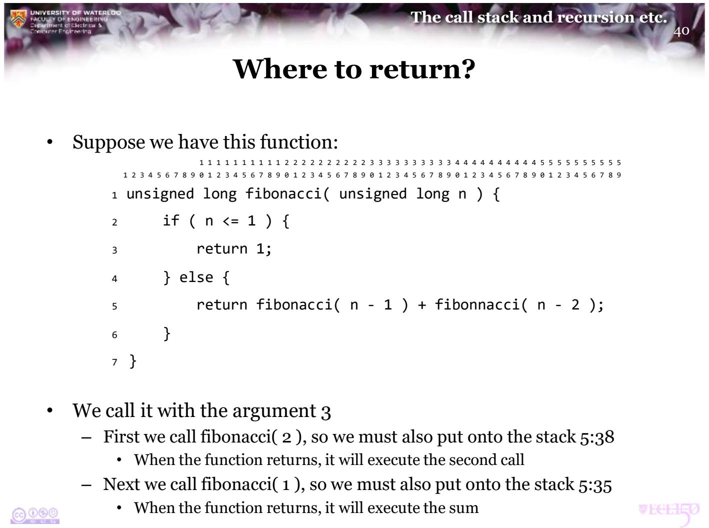
0

return value for main()



Recursion and the call stack

- Thus, the call stack allows recursion
  - We will use recursion throughout this course
  - It will appear in subsequent courses, as well
  - Recursion allows us to solve a problem by reformulating the problem in simpler or alternate terms

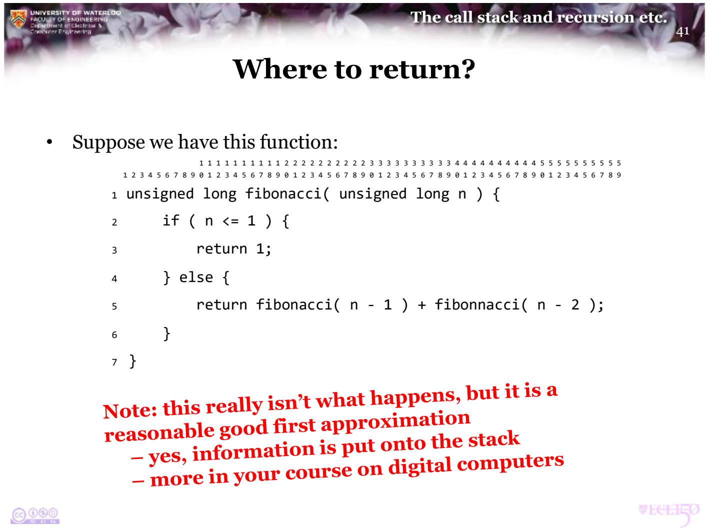


Where to return?

- Suppose we have this function:

```
           11111111112222222222333333333344444444445555555555
  12345678901234567890123456789012345678901234567890123456789
1 unsigned long fibonacci( unsigned long n ) {
2     if ( n <= 1 ) {
3         return 1;
4     } else {
5         return fibonacci( n - 1 ) + fibonnacci( n - 2 );
6     }
7 }
```

- We call it with the argument 3
  - First we call fibonacci(2), so we must also put onto the stack 5:38
    - When the function returns, it will execute the second call
  - Next we call fibonacci(1), so we must also put onto the stack 5:35
    - When the function returns, it will execute the sum

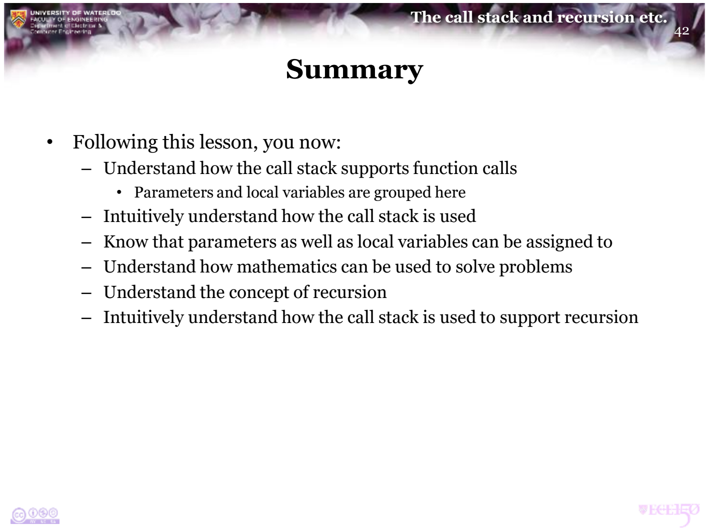


Where to return?

- Suppose we have this function:

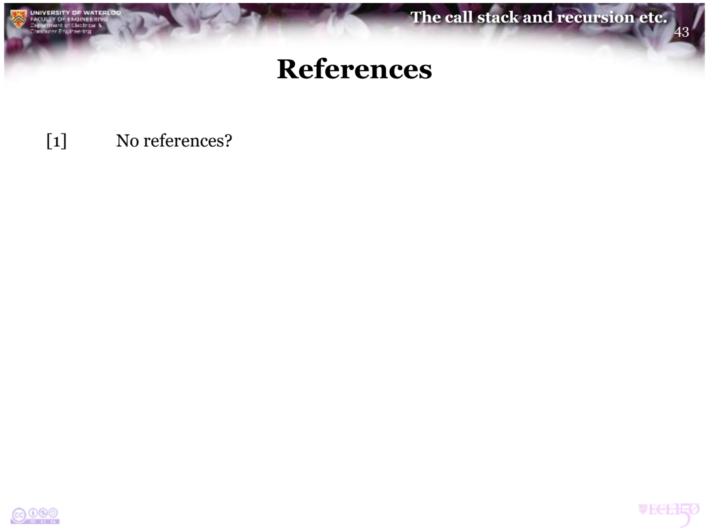
```
unsigned long fibonacci( unsigned long n ) {
    if ( n <= 1 ) {
        return 1;
    } else {
        return fibonacci( n - 1 ) + fibonnacci( n - 2 );
    }
}
```

Note: this really isn’t what happens, but it is a reasonable good first approximation
– yes, information is put onto the stack
– more in your course on digital computers



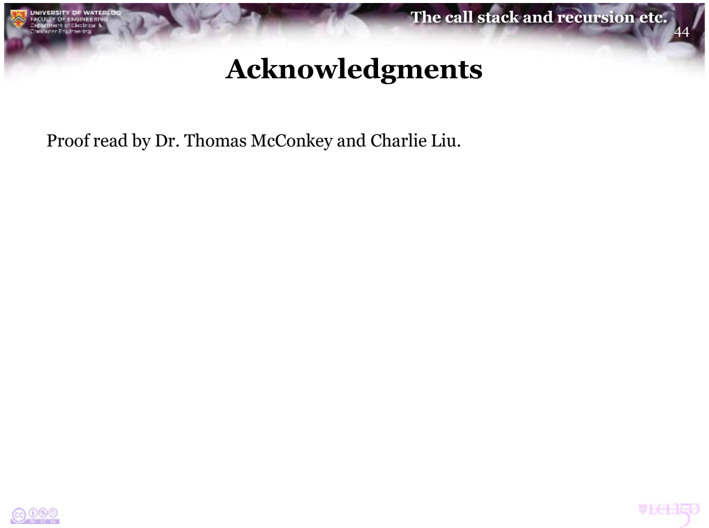
Summary

- Following this lesson, you now:
  - Understand how the call stack supports function calls
    - Parameters and local variables are grouped here
  - Intuitively understand how the call stack is used
  - Know that parameters as well as local variables can be assigned to
  - Understand how mathematics can be used to solve problems
  - Understand the concept of recursion
  - Intuitively understand how the call stack is used to support recursion



References

[1] No references?



Acknowledgments

Proof read by Dr. Thomas McConkey and Charlie Liu.

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.

Disclaimer

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.