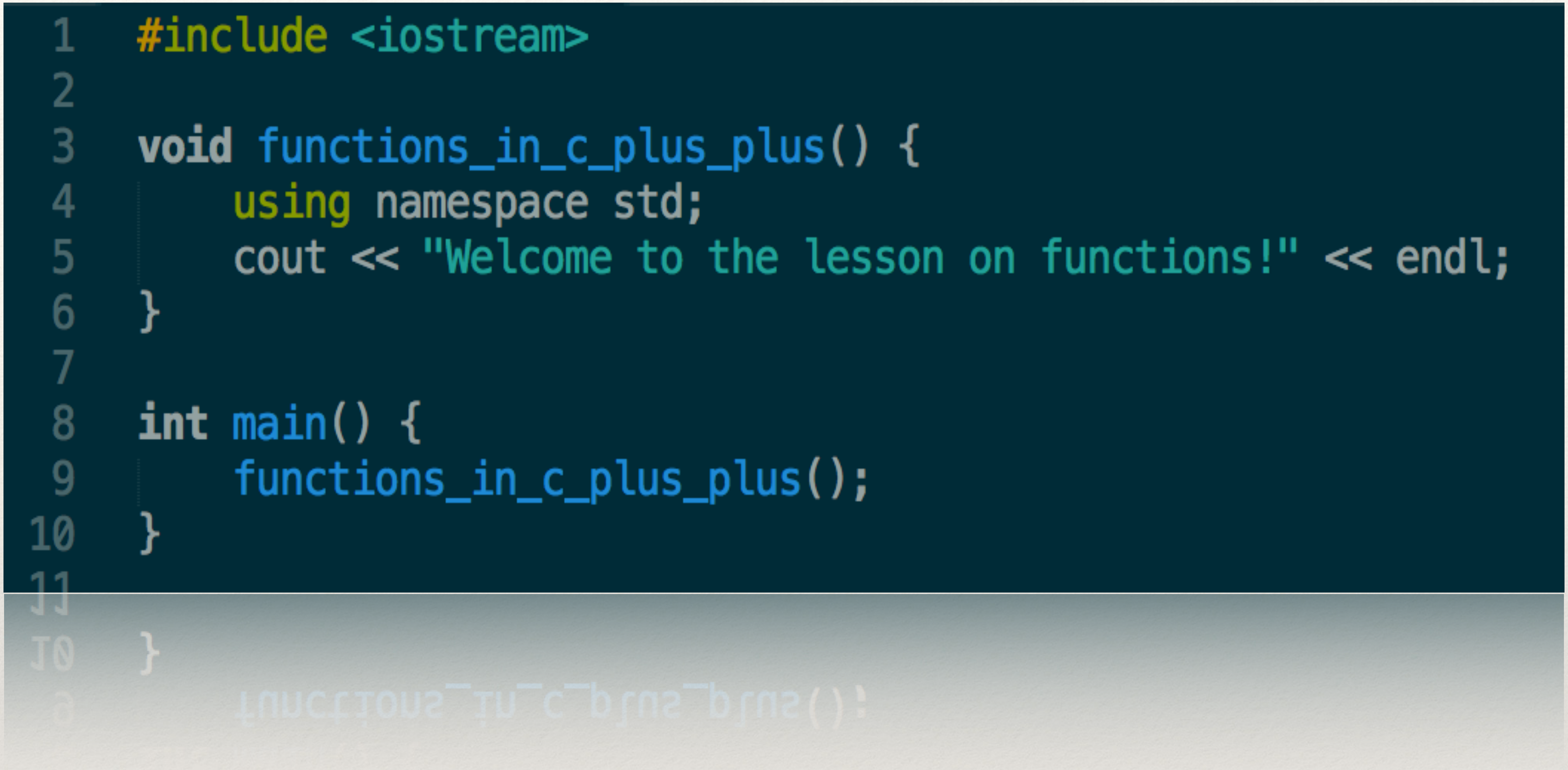

```cpp
#include <iostream>

void functions_in_c_plus_plus() {
    using namespace std;
    cout << "Welcome to the lesson on functions!" << endl;
}

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

Functions

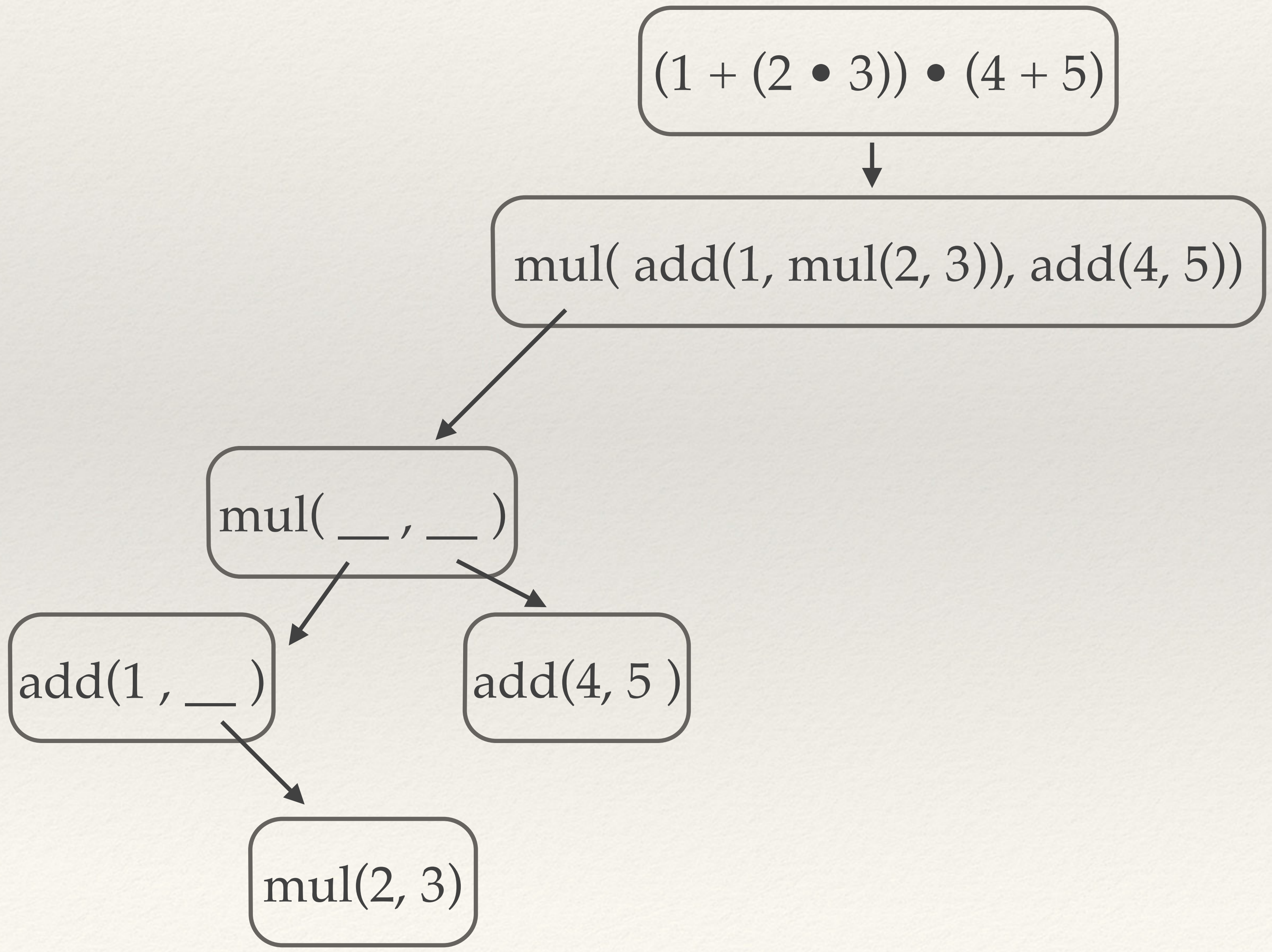
Matthew Mussomele

Expressions

- Expressions are computable statements
  - $3 + 10$
  - $\sqrt{(169)}$
  - $|-13|$
  - $f(x)$

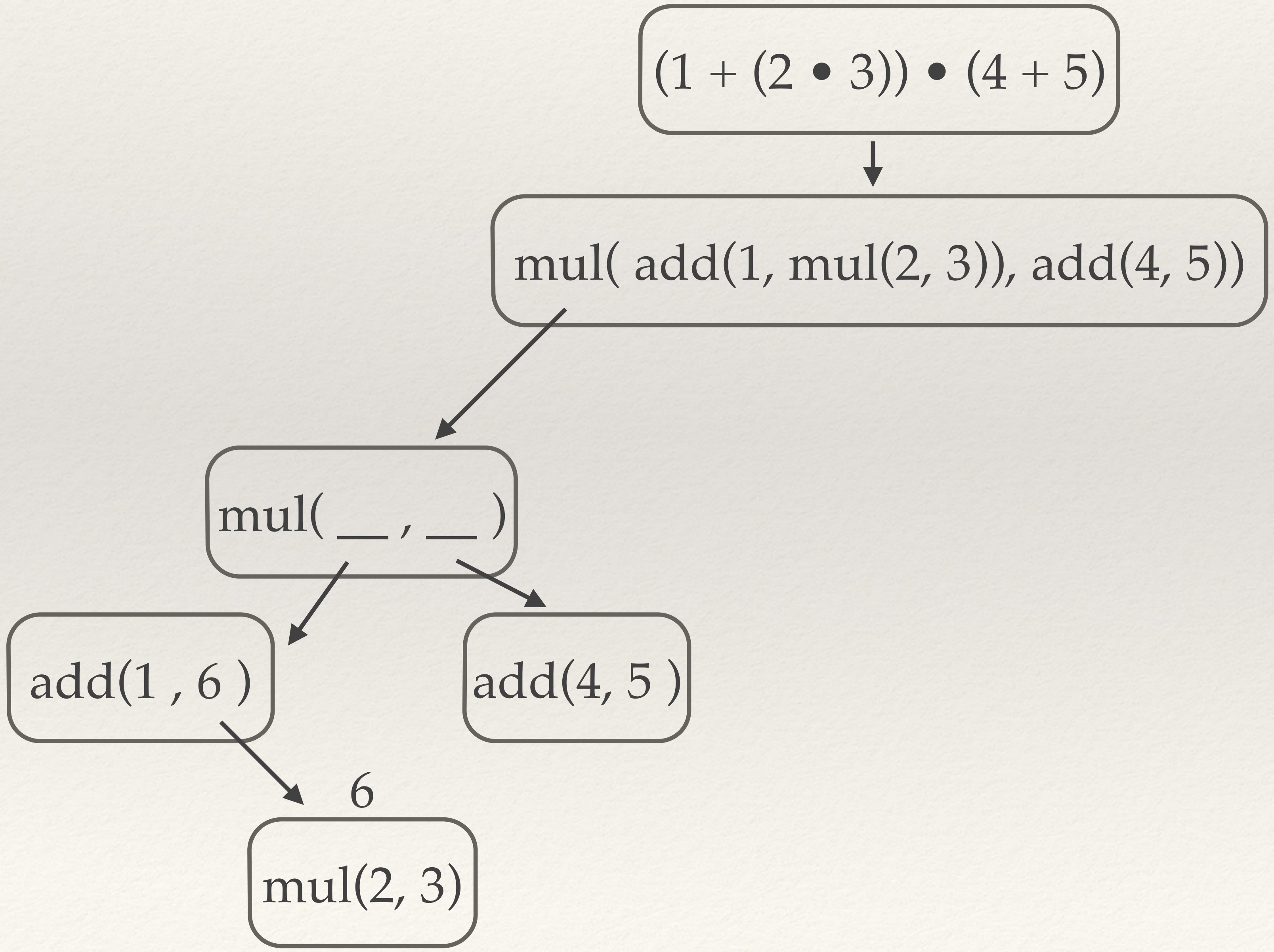
Evaluating Expressions

When evaluating expressions, any sub-expressions within them must be evaluated first.



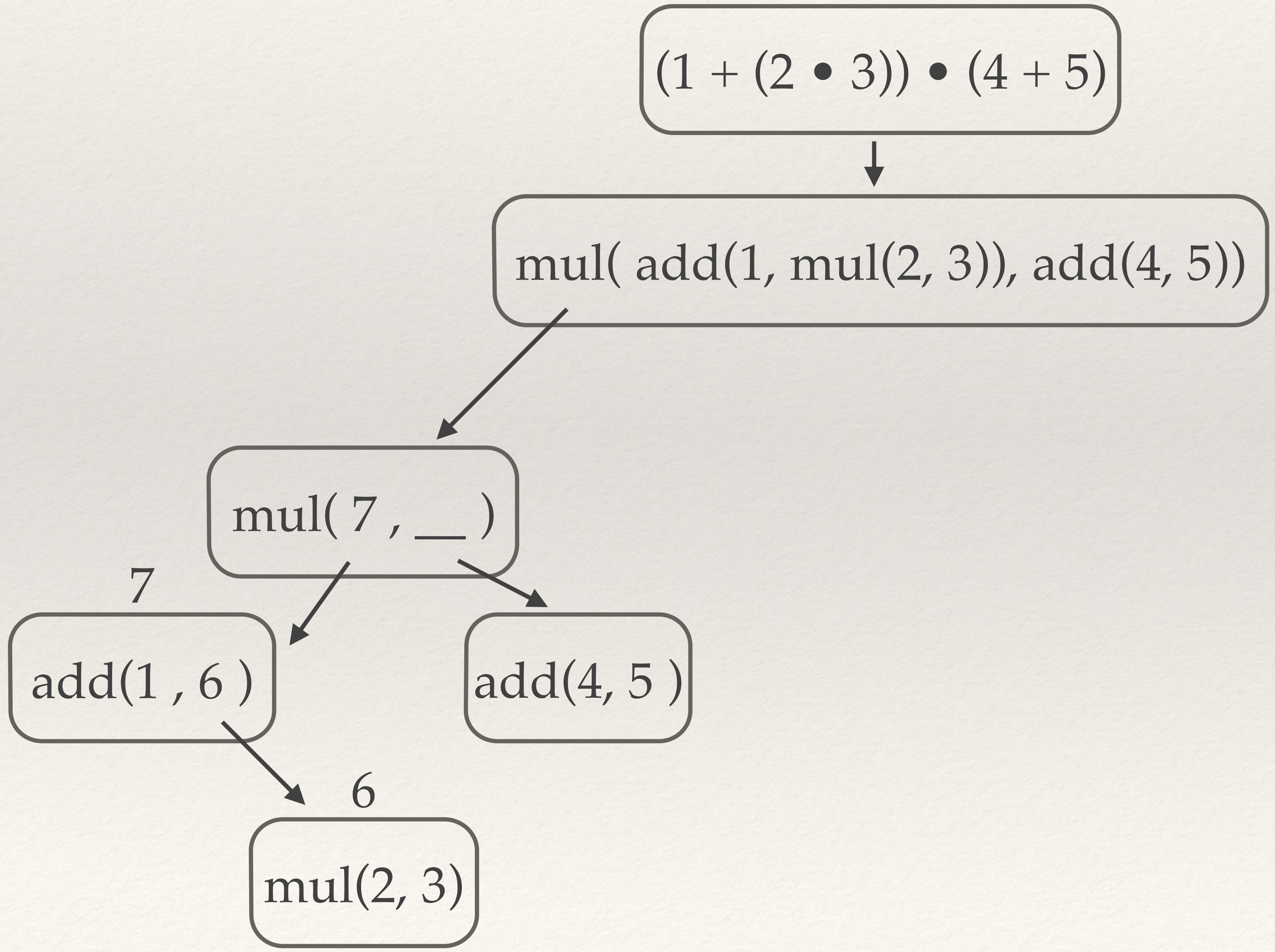
Evaluating Expressions

When evaluating expressions, any sub-expressions within them must be evaluated first.



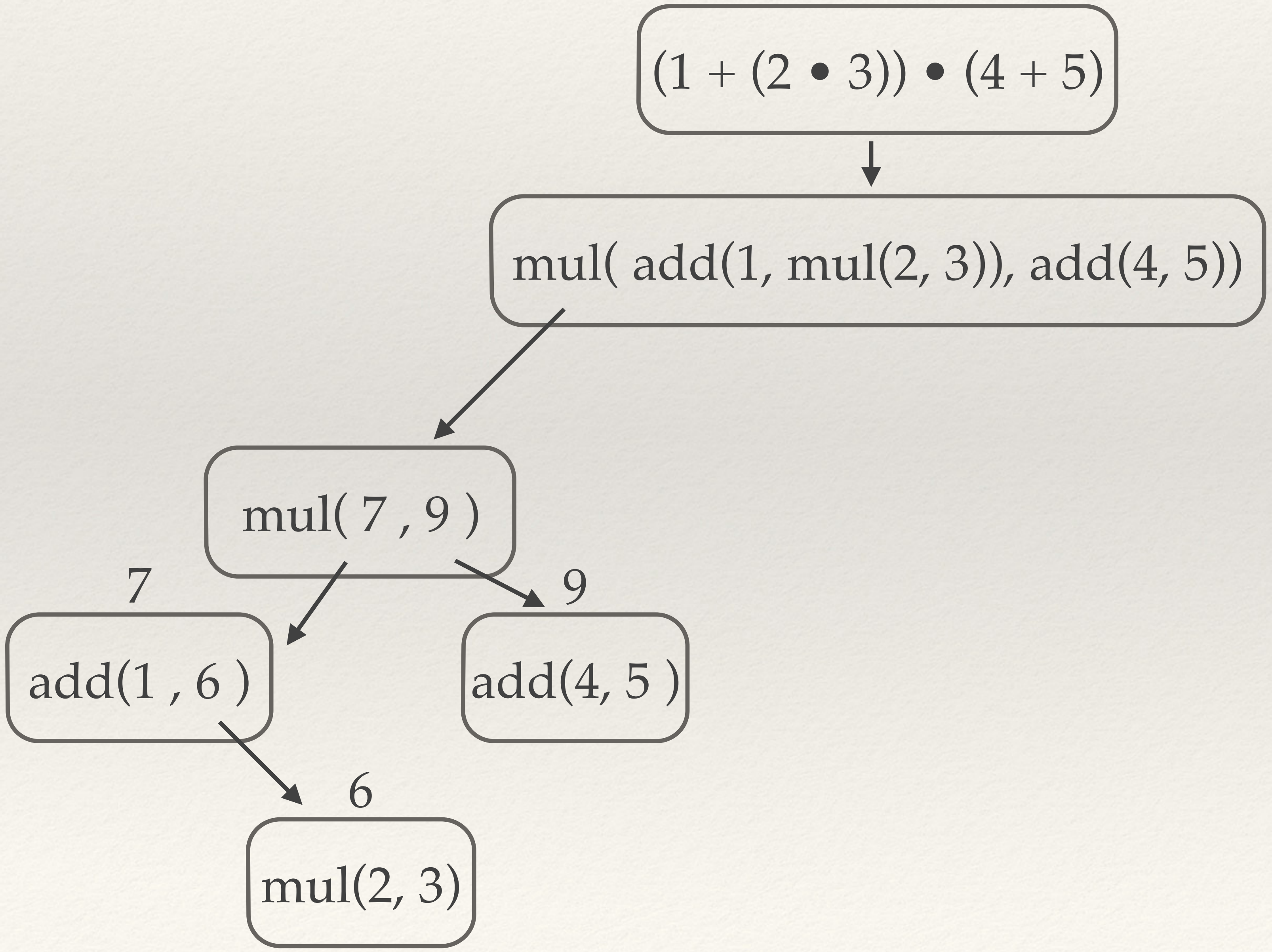
Evaluating Expressions

When evaluating expressions, any sub-expressions within them must be evaluated first.



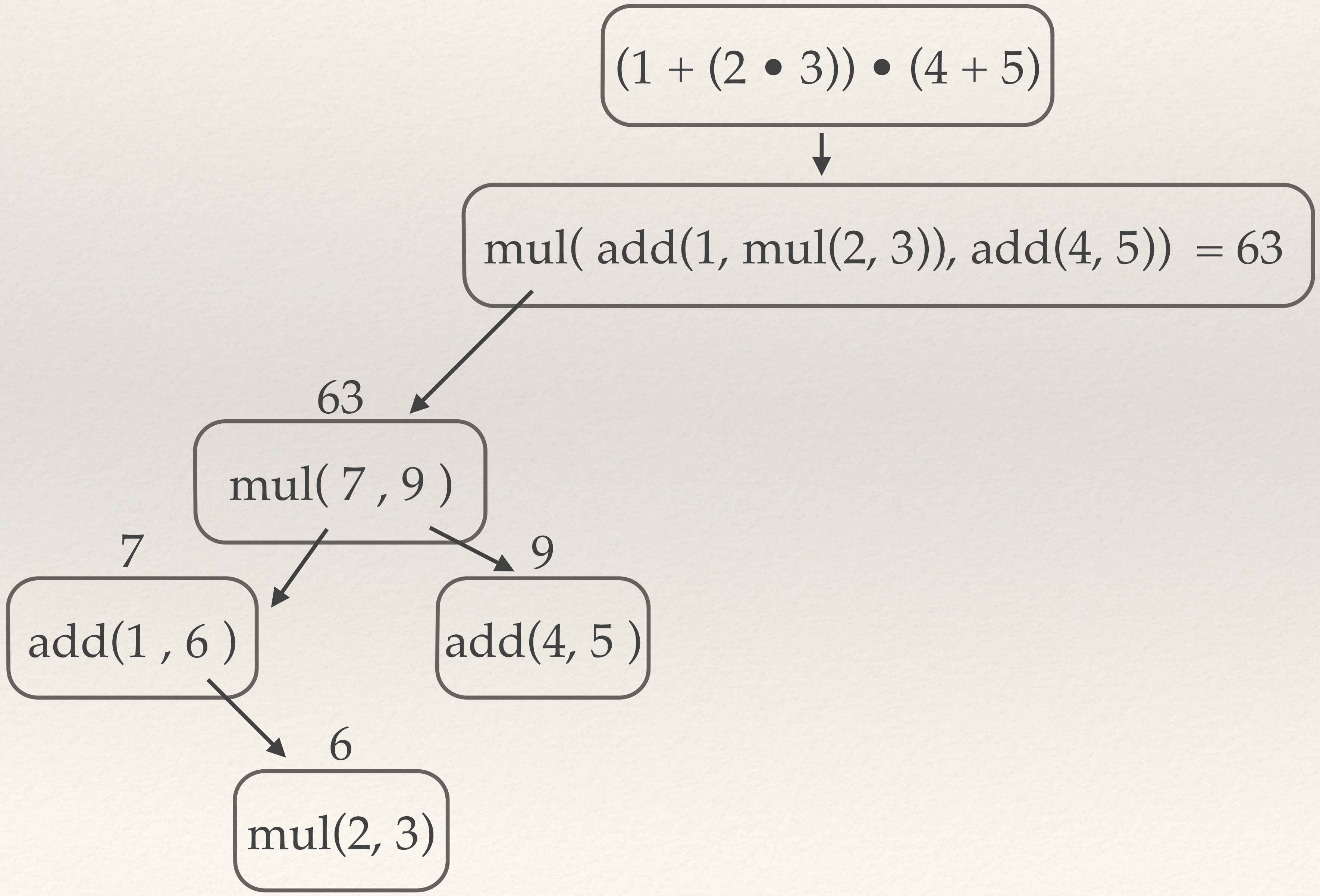
Evaluating Expressions

When evaluating expressions, any sub-expressions within them must be evaluated first.



Evaluating Expressions

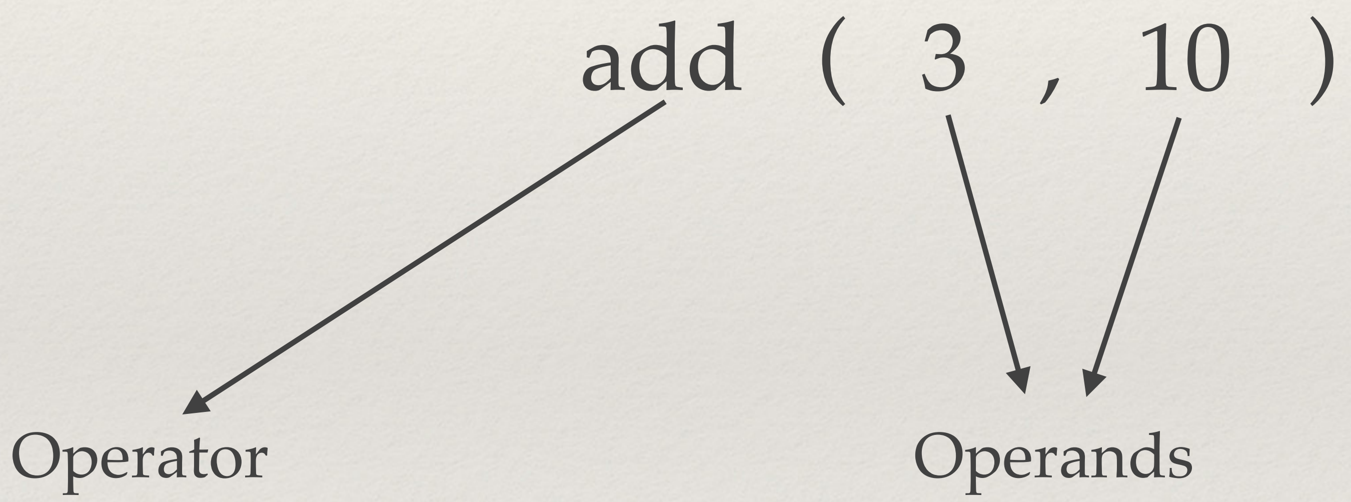
When evaluating expressions, any sub-expressions within them must be evaluated first.



Functions

- Function calls are types of expressions

This is a function call:



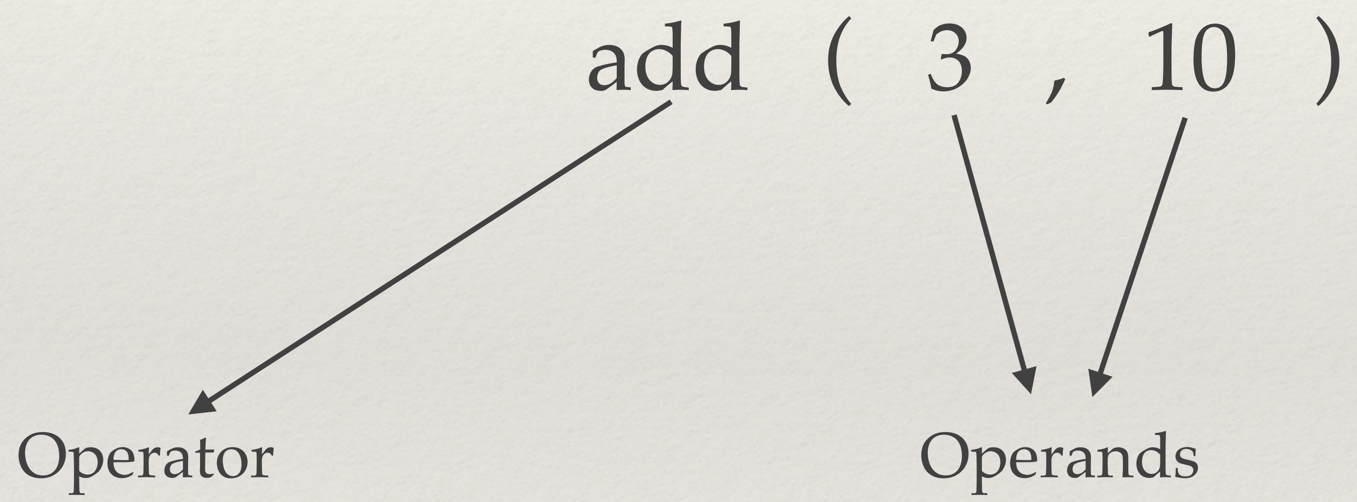
```
add ( 3 , 10 )
```

Operators and Operands are expressions themselves and evaluate to values.

Functions

- Function calls are types of expressions

This is a function call:



```
add ( 3 , 10 )
```

Order of Operations:

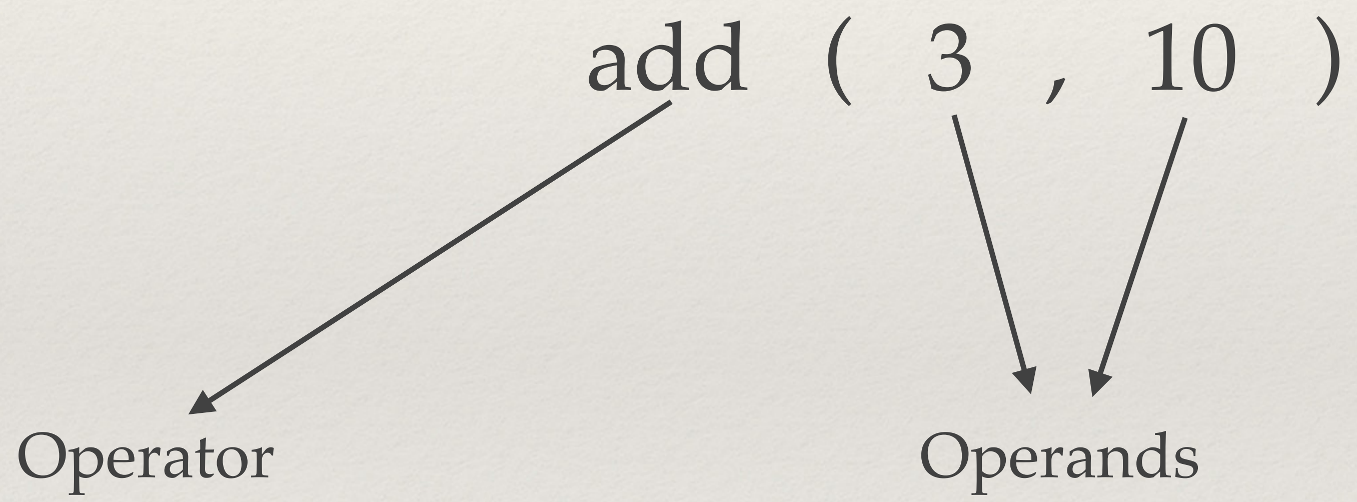
1. Evaluate the Operator and the Operands*

*The order of evaluation for operands is undefined.

Functions

- Function calls are types of expressions

This is a function call:



```
add ( 3 , 10 )
```

Order of Operations:

2. Pass the evaluated components to the function.

Defining Your Own Functions

In C++, all functions match a general form.

```
<return type> <name>(<formal paramters>) {
       <body>
}
```

❖ Return Type	- The type of expression that the function will evaluate to
❖ Name	- The name used to call and refer to the function
❖ Formal Parameters	- The arguments that are to be passed to the function
❖ Body	- An expression (1+ lines) that is to be evaluated when the function is called

Return Type

The return type of a function is the type it should evaluate to.

For example, a function that adds two integers should return an integer.

Example Return Types:

- Integer - `int`
- Double - `double`
- Boolean - `bool`
- Nothing - `void`

Name

You can name your functions whatever you like as long as they follow the following rules:

1. Only alphanumeric and underscore characters may be used.
2. Names cannot start with a number.
3. C++ keywords cannot be used (more on this later)

Names are also case sensitive. For example, `myvariable` is a different variable than `myVariable`.

Formal Parameters

Formal parameters are the arguments your function takes in to do things with.

These parameters consist of a type and a name (or identifier).

For example, a function with the header,

```
void printNum(int a)
```

would take a single integer parameter that could be referred to as ‘a’ inside the function.

Functions can have many formal parameters, but having more than 3 makes things messy.

Body

The body of a function is where all the processing and calculation happens.

Function bodies can have 1 or more lines and expressions.

If a function has a return type, then its body must contain a return statement.

Valid Function Body

```
int add_one(int a) {
    return a + 1;
}
```

Invalid Function Body

```
int add_one(int a) {
    int x = a + 1;
}
```

Using Functions

Like stated before, functions are a type of expression, meaning they can be used in more complicated expression and can be called many times.

To call a function, we use the syntax:

```
<name>(<arguments>)
```

Lets use the add_one function from the last slide as an example.

Using Functions

```
int add_one(int a) {
    return a + 1;
}
```

Here are some ways we might use `add_one`:

```
add_one(1); //this works, evaluates to 2
```

```
add_one(-1); //this works, evaluates to 0
```

```
add_one(0.5); //this works, evaluates to 1
```

```
add_one(1, 2); /* this won't compile! The
                  arguments given to the
                  function don't match the
                  formal parameters! */
```

Function Guidelines and Tips

- Every function should have exactly one job.
    - If your functions are getting longer than 10 lines and/or you can't describe what they do without using 'and', split them up.
- Use functions if you find yourself copying code over and over.
- Make your functions general.
    - Our `add_one(int a)` function was not general.
    - A general version would be just `add(int a, int b)`

Function Documentation

All functions should have a documentation comment above them.

These comments should follow the following format:

```
/**
 * @brief <short description of function>
 * @details <longer description>
 *
 * @param <name> <description>
 * @param  <name> <description>
 * …
 * @return <description of return value>
 */
```

Citations

- DeNero, John. "Composing Programs." Composing Programs. Chapters 1.2-1.4 Web. 10 June 2015.