NORTHWEST INSTITUTE for ADVANCED COMPUTING

Pacific Northwest NATIONAL LABORATORY
Proudly Operated by Battelle for the U.S. Department of Energy

UNIVERSITY of WASHINGTON

AMATH 483/583
High Performance Scientific Computing

Lecture 3:
Functions, Multiple Compilation, Data Abstraction

Andrew Lumsdaine
Northwest Institute for Advanced Computing
Pacific Northwest National Laboratory
University of Washington
Seattle, WA

Overview

- Recap of Lecture 2
  - Types and variables
  - Namespaces
- Functions and procedural abstraction
- Parameter passing
- Program / file organization
- Make and Makefile
- Back Propagation
- Vector and Matrix

SC'19 Student Cluster Competition Call-Out!

- Teams work with advisor and vendor to design and build a cutting-edge, commercially available cluster constrained by the 3000-watt power limit
- Cluster run a variety of HPC workflows, ranging from being limited by CPU performance to being memory bandwidth limited to I/O intensive
- Teams are comprised of six undergrad or high-school students plus advisor

SC19

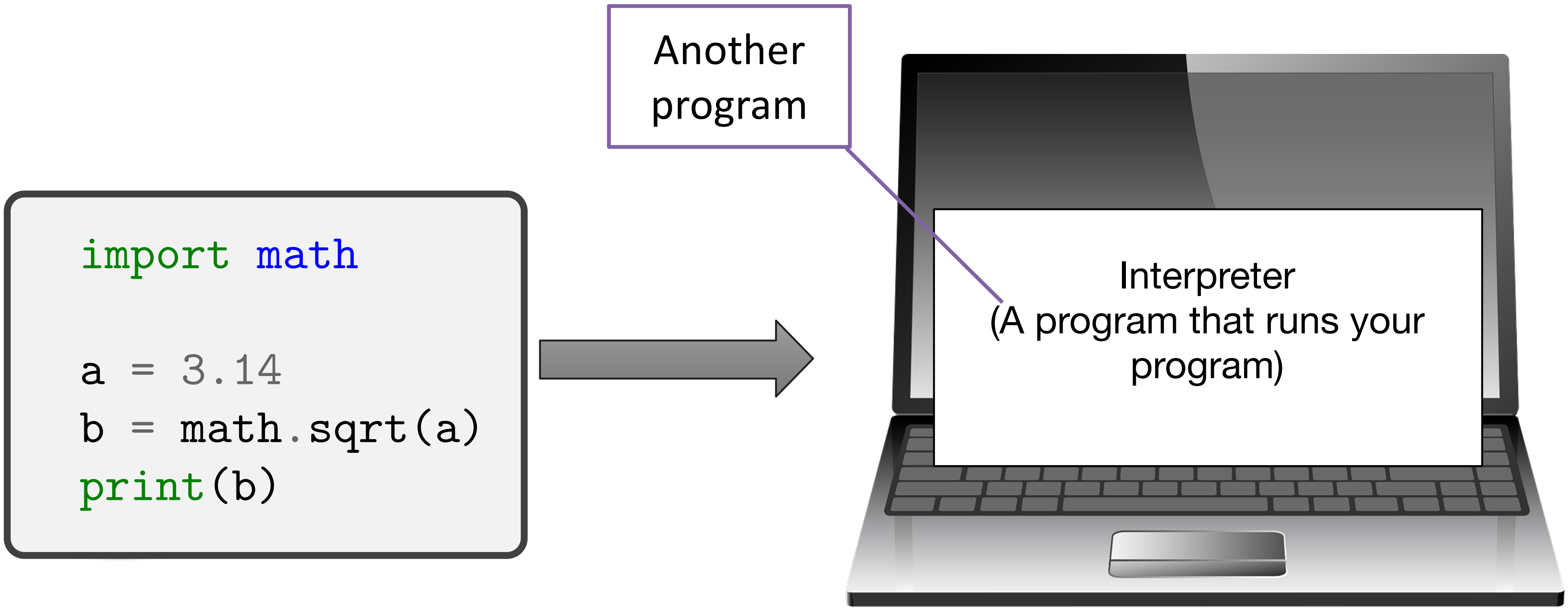
https://sc19.supercomputing.org/program/studentssc/student-cluster-competition/

Informational meeting:
Tu 5PM-6PM Allen 203
Th 5PM-6PM Allen 203

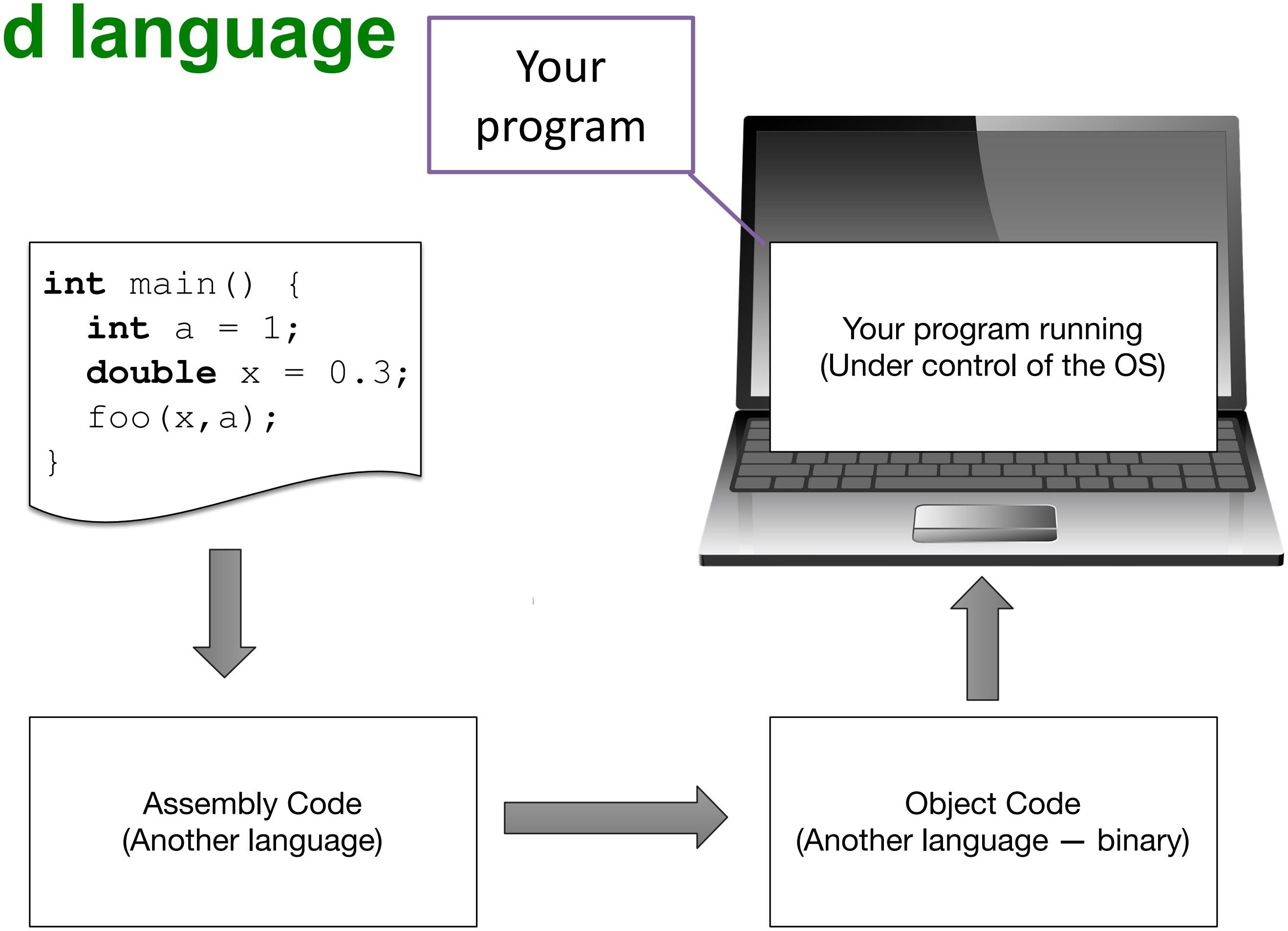
One Quick Definition

- FLOP

Interpreted language (Python)



Compiled language



Interpreted vs compiled

Use math library

Call function from math library

```python
import math

a = 3.14
b = math.sqrt(a)
print(b)
```

Curly braces for code blocks

Code must be in a function

Declare variables

Print result

Variables are typed

Use functions from iostream library

Use math library

```cpp
e <cmath>
#include <iostream>

int main() {

  double a = 3.14;
  double b = std::sqrt(a);
  std::cout << b << std::endl;

  return 0;
}
```

Call function

IO also in std

“std” rather than “math”

Compilation

```
#include <cmath>
#include <iostream>

int main() {

  double a = 3.14;
  double b = std::sqrt(a);
  std::cout << b << std::endl;

  return 0;
}
```

You can't run this code

It needs to be turned into code that can run

An "executable"

Multi-step process

Compile to **object file**

Bits just for this code

Then link in libraries for sqrt and IO

Declaring and Initializing Variables

- In the old days variables were declared at the beginning of a block

```
int main() {
  double x, y;
  // ...
  x = 3.14159;
  y = x * 2.0;
  // ...
  return 0;
}
```

Declaration

Use

- Now they can be defined anywhere in the block

```
int main() {
  // ...
  double x = 3.14159;
  double y = x * 2.0;
  // ...
  return 0;
}
```

Declaration with initialization

- Best practice: Don't declare variables before they are needed and **always** initialize if possible

Namespace Recommendation for AMATH 483/583

[P.3: Express intent]

```
= "Hello World";
<< std::endl;
```

Too much typing?

Organizing your programs

- Software development is difficult
- How do humans attack complex problems?
- Apply the same principles to software

Abstraction

Procedural

Data type

- Modular / reusable
- Well defined interfaces and functionality
- Understandable

Procedural Abstraction

Separate functionality into well-defined, reusable, pieces of parameterized code (aka “functions”)

Newton's Method for Square Root

- To solve $f(x) = 0$ for x
- Linearize (approximate the nonlinear problem with a linear one) and solve the linear problem
- Iterate
- Taylor: $$f(x + \Delta x) \approx f(x) + \Delta x f'(x) = \Delta x f'(x)$$

$$\Delta x = -\frac{f(x)}{f'(x)}$$

$$f(x) = x^2 - y = 0 \rightarrow y = \sqrt{x} \qquad f'(x) = 2x \qquad \Delta x = -\frac{x^2 - y}{2x}$$

Compute square root of 2

```
#include <iostream>
#include <cmath>

int main () {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-2.0) / (2.0*x) ;
    x += dx;
    if (std::abs(dx) < 1.e-9) break;
  }

  std::cout << x << std::endl;

  return 0;
}
```

Compute square root of 3

```
#include <iostream>
#include <cmath>

int main () {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-3.0) / (2.0*x) ;
    x += dx;
    if (std::abs(dx) < 1.e-9) break;
  }

  std::cout << x << std::endl;

  return 0;
}
```

Compute square root of 2 and 3

```
#include <iostream>
#include <cmath>

int main () {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-2.0) / (2.0*x) ;
    x += dx;
    if (std::abs(dx) < 1.e-9) break;
  }

  std::cout << x << std::endl;

  return 0;
}
```

Don’t do the same thing twice in different places

```
include <iostream>
include <cmath>

nt main () {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-3.0) / (2.0*x) ;
    x += dx;
    if (std::abs(dx) < 1.e-9) break;
  }

  std::cout << x << std::endl;

  return 0;
}
```

But they’re not exactly the same

This is the only difference

This is the only difference

Procedural Abstraction

```
#include <cmath>

double sqrt583(double y) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-y) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  return x;
}
```

Define function named sqrt583

The function is ***parameterized*** by y

Which is a double

It returns a double

It returns a double

Same code as before

Except for parameterization

Procedural Abstraction

```
#include <cmath>

double sqrt583(double y) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-y) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  return x;
}
```

Redundant?

It returns a double

It returns a double

Compiler can deduce return types

Note auto is a C++14 feature!

```
#include <cmath>

auto sqrt583(double y) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-y) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  return x;
}
```

It returns a double

It returns a double

Square root of 2 and 3

```
#include <iostream>
#include <cmath>

double sqrt583(double y) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-y) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;

int main () {
              sqrt583(2.0) << std::endl;
              sqrt583(3.0) << std::endl;
```

Note initialization and declaration of i

What is a size_t?

Pass parameter 2

Pass parameter 3

Thought experiment

Change value of y

Print y

What will print?

```
#include <iostream>
#include <cmath>

double sqrt583(double y) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-y) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  y = x;

  return x;
}


int main () {
  double y = 2.0;
  std::cout << sqrt583(y) << std::endl;
  std::cout << y << std::endl;

  return 0;
}
```

```
$ ./a.out
1.41421
2
```

Parameter Passing in C++

y is passed **by value** (copied), so only the copy is changed, not the original

C++ has “pass by value” semantics

```
#include <iostream>
#include <cmath>

double sqrt583(double y) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-y) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  y = x;

  return x;
}


int main () {
  double y = 2.0;
  std::cout << sqrt583(y) << std::endl;
  std::cout << y << std::endl;

  return 0;
}
```

Parameter Passing in C++

y is passed **by value** (copied), so only the copy is changed, not the original

C++ has "pass by value" semantics

Just to be clear, the parameter can have any name (don't confuse with y declared in main)

```
#include <iostream>
#include <cmath>

double sqrt583(double z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  z = x;

  return x;
}


int main () {
  double y = 2.0;
  std::cout << sqrt583(y) << std::endl;
  std::cout << y << std::endl;

  return 0;
}
```

Before

```
$ ./a.out
1.41421
2
```

```
#include <iostream>
#include <cmath>

double sqrt583(double z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  z = x;

  return x;
}


int main () {
  double y = 2.0;
  std::cout << sqrt583(y) << std::endl;
  std::cout << y << std::endl;

  return 0;
}
```

After

```
$ ./a.out
1.41421
1.41421
```

```cpp
#include <iostream>
#include <cmath>

double sqrt583(double& z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  z = x;

  return x;
}


int main () {
  double y = 2.0;
  std::cout << sqrt583(y) << std::endl;
  std::cout << y << std::endl;

  return 0;
}
```

After

```
$ ./a.out
1.41421
1.41421
```

y is passed **by reference** (not copied), so the original is changed

This variable

Is this variable

```cpp
#include <iostream>
#include <cmath>

double sqrt583(double& z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  z = x;

  return x;
}


int main () {
  double y = 2.0;
  std::cout << sqrt583(y) << std::endl;
  std::cout << y << std::endl;

  return 0;
}
```

Thought experiment

This variable

Is this variable

Which isn't a variable

```
#include <iostream>
#include <cmath>

double sqrt583(double &z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  z = x;

  return x;
}


int main () {

  ...d::cout << sqrt583(2.0) << std::endl;

  ...turn 0;
```

```
sqrtr2.cpp:21:16: error: no matching function for call to 'sqrt583'
  std::cout << sqrt583(2.0) << std::endl;
               ^~~~~~~
sqrtr2.cpp:4:8: note: candidate function not viable: expects an l-value for 1st argument
double sqrt583(double &z) {
       ^
1 error generated.
```

Thought experiment

Why would we want to pass a reference?

"Out parameters"

Efficiency (no copy)

How can we do this?

```
#include <iostream>
#include <cmath>

double sqrt583(double &z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  z = x;

  return x;
}


int main () {

  std::cout << sqrt583(2.0) << std::endl;


  return 0;
}
```

Before

```
#include <iostream>
#include <cmath>

double sqrt583(double &z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  z = x;

  return x;
}


int main () {

  std::cout << sqrt583(2.0) << std::endl;


  return 0;
}
```

After

```
#include <iostream>
#include <cmath>

double sqrt583(const double &z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  z = x;

  return x;
}


int main () {

  std::cout << sqrt583(2.0) << std::endl;


  return 0;
}
```

After

Promise not to change z

A reference to a constant is okay

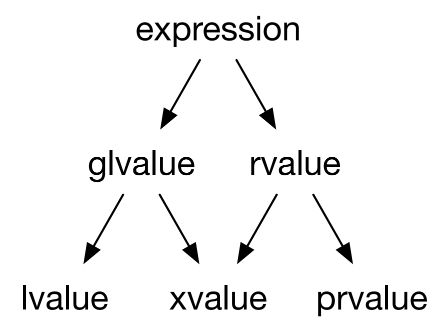
```
#include <iostream>
#include <cmath>

double sqrt583(const double &z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  z = x;

  return x;
}


int main () {

  std::cout << sqrt583(2.0) << std::endl;


  return 0;
}
```

Functions

- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple
- F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to const
- F.17: For "in-out" parameters, pass by reference to non-const
- F.20: For "out" output values, prefer return values to output parameters

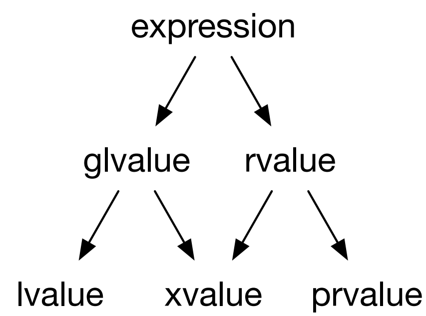
l-values and r-values

- Section 3.10 of C++ standard
  - A *glvalue* is an expression whose evaluation determines the identity of an object, bit-field, or function.
  - A *prvalue* is an expression whose evaluation initializes an object or a bit-field, or computes the value of the operand of an operator, as specified by the context in which it appears.
  - An *xvalue* is a glvalue that denotes an object or bit-field whose resources can be reused (usually because it is near the end of its lifetime).
  - An *lvalue* is a glvalue that is not an xvalue.
  - An *rvalue* is a prvalue or an xvalue



l-values and r-values

- More intuitively
- Ignore glvalue, xvalue, prvalue
- lvalue is something that can go on the ***left*** of an assignment (correctly)
  - "Lives" beyond an expression
- Rvalue is something that can go on the ***right*** of an assignment (correctly)
  - Does not "live" beyond an expression



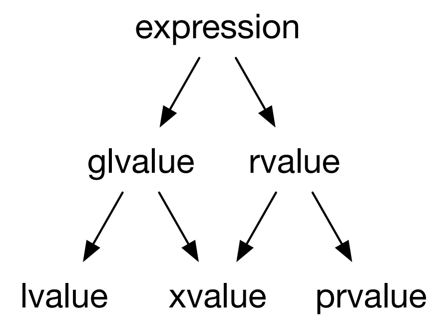
l-values and r-values

```
double x, y, z;

x = y;
x = 1.0;
y = x + z;
```

lvalue

rvalue

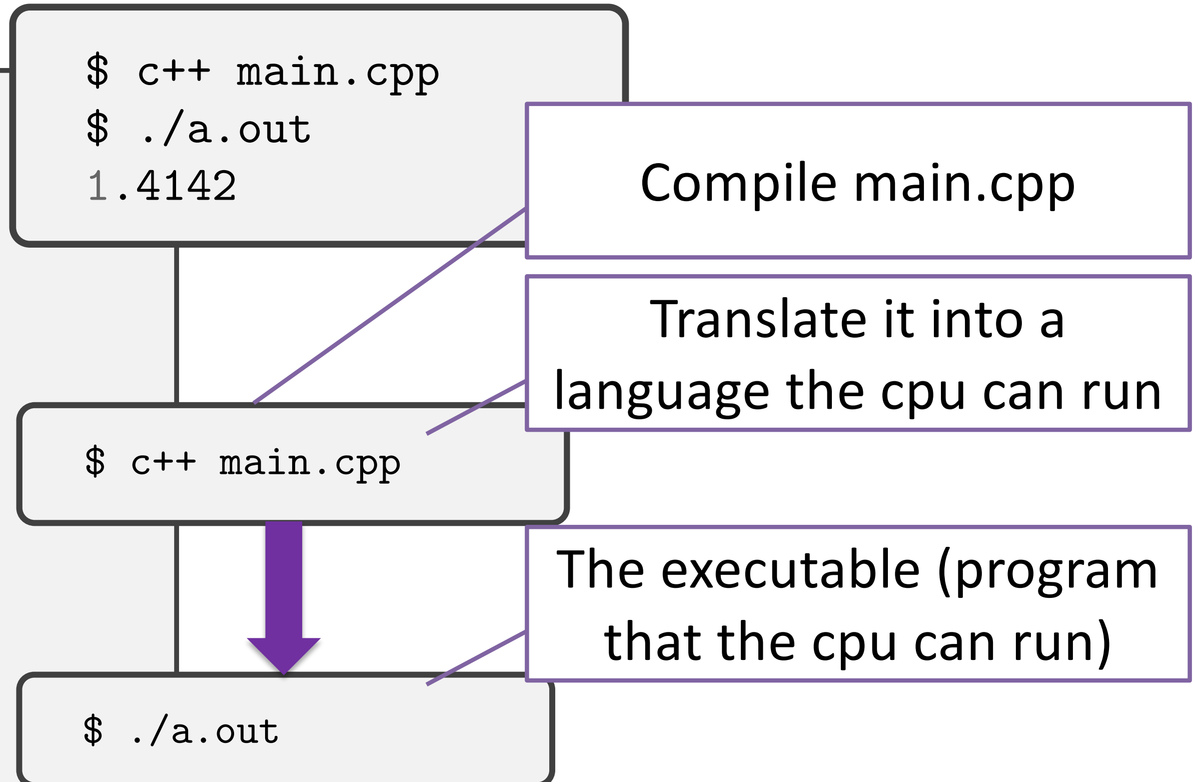
```
double x, y, z;

x = y;
1.0 = x;
x + z = y;
```




```
% c++ sl7.cpp
c++ sl7.cpp
sl7.cpp:7:9: error: expression is not assignable
  x + z = y;
  ~~~~~ ^
1 error generated.
```

Reusing functions

```
#include <iostream>
#include <cmath>

double sqrt583(double z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  return x;
}


int main () {

  std::cout << sqrt583(2.0) << std::endl;


  return 0;
}
```



Reusing in other programs

Put this function in its own file amath583.cpp

```
#include <cmath>

double sqrt583(double& z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  return x;
}
```

Many programs (mains) can call it

```
#include <iostream>
#using namespace std;
int main () {

  cout << sqrt583(3.0) << e

  return 0;
}
```

```
#include <iostream>
#using namespace std;
int main () {

  cout << sqrt583(3.14) <

  return 0;
}
```

```
#include <iostream>
#using namespace std;
int main () {

  cout << sqrt583(42.0) << endl;

  return 0;
}
```

Reusing in other programs

Many mains can call it

```
#include <iostream>
#using namespace std;
int main () {

  cout << sqrt583(42.0) << endl;

  return 0;
}
```

```
sqrt3.cpp:8:11: error: use of undeclared identifier 'sqrt583'
  cout << sqrt583(2.0) << endl;
          ^
sqrt3.cpp:9:11: error: use of undeclared identifier 'sqrt583'
  cout << sqrt583(3.0) << endl;
          ^
2 errors generated.
```

Undeclared identifier

```
#include <cmath>

double sqrt583(double z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  return x;
}
```

Defined in a different file

Didn't we declare it here?

This is ***definition***

Reusing functions

```
$ c++ main.cpp
$ ./a.out
1.4142
```

Doesn't know how to translate this

```
#include <iostream>
#using namespace std;
int main () {

  cout << sqrt583(42.0) << endl;

  return 0;
}
```

```
$ c++ main.cpp
```

```
$ ./a.out
```

Compile main.cpp

Translate it into a language the cpu can run

The executable (program that the cpu can run)

Reusing functions across programs

Declare sqrt583 is a function that exists

Takes a double

Returns a double

Now we know how to call it

```
clude <iostream>

double sqrt583(double);

nt main () {

  std::cout << sqrt583(42.0) << std::endl;

  return 0;
}
```

Reusing in other programs

```
#include <iostream>

double sqrt583(double);

int main () {

  std::cout << sqrt583(42.0) << std::e

  return 0;
}
```

Many mains can call sqrt583

Undefined symbol

Linker command failed

```
Undefined symbols for architecture x86_64:
  "sqrt583(double const&)", referenced from:
      _main in sqrt3-1d1d35.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Reusing functions

```
$ c++ main.cpp
$ ./a.out
1.4142
```

```
#include <iostream>

double sqrt583(double);

int main () {

  std::cout << sqrt583(42.0) << std::endl;

  return 0;
}
```

```
$ c++ main.cpp
```

```
$ ./a.out
```

Compile main.cpp

Translate it into a language the cpu can run

The executable (program that the cpu can run)

Needs to find sqrt583 somewhere

Reusing in other programs

```
#include <iostream>

double sqrt583(double);

int main () {

  std::cout << sqrt583(42.0) << std::endl;

  return 0;
}
```

```
#include <cmath>

double sqrt583(double z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  return x;
}
```

```
$ c++ main.cpp sqrt583.cpp
```

Compile main.cpp ***with*** sqrt583.cpp

Translate it into a language the cpu can run

```
$ ./a.out
```

The executable (program that the cpu can run)

Reusing in other programs

```cpp
#include <iostream>

double sqrt583(double);

int main () {

  std::cout << sqrt583(42.0) << std::endl;

  return 0;
}
```

```cpp
#include <cmath>

double sqrt583(double z) {
  double x = 1.0;

  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }

  return x;
}
```

```
$ c++ main.cpp
```

```
$ c++ sqrt583.cpp
```

```
$ ./a.out
```

Compile main.cpp by itself

Compile sqrt583.cpp by itself

Another step here

Generate executable

Reusing in other programs

```
#include <iostream>

double sqrt583(double);

int main () {

  std::cout << sqrt583(42.0) << std::endl;

  return 0;
}
```

I need to declare it

If I am going to call this

But a real program uses many functions

```
#include <iostream>

double sqrt583(double);

int main () {

  std::cout << sqrt583(42.0) << std::endl;
  std::cout << expt583(42.0. pi) << std::endl;
  std::cout << sin583(42.0 * pi) << std::endl;
  // ...

  return 0;
}
```

Reusing in other programs

```
#include <iostream>

double sqrt583(double);
double expt583(double, double);
double sin583(double, double);
// ...

int main () {

  std::cout << sqrt583(42.0) << std::endl;
  std::cout << expt583(42.0. pi) << std::en
  std::cout << sin583(42.0 * pi) << std::en
  // ...

  return 0;
}
```

And I could declare each of them individually

But why?

But a real program uses many functions

But if not, how are these declarations found?

Hint: iostream

Header files: Interface declarations

```
#include <iostream>
#include "amath583.hpp"

int main () {

  std::cout << sqrt583(42.0) << std::endl;
  std::cout << expt583(42.0. pi) << std::endl;
  std::cout << sin583(42.0 * pi) << std::endl;
  // ...

  return 0;
}
```

Include amath583.hpp

```
// amath583.hpp: Declarations
double sqrt583(double);
double expt583(double, double);
double sin583(double, double);
// ...
```

Declare all functions in amath583.hpp

```
$ c++ main.cpp
```

```
$ c++ sqrt583.cpp
```

```
$ ./a.out
```

```
#include <cmath>
#include "amath583.hpp"

double sqrt583(double z) {
  double x = 1.0;
  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }
  return x;
}
// ...
```

Include amath583.hpp

Implement all functions in amath583.cpp

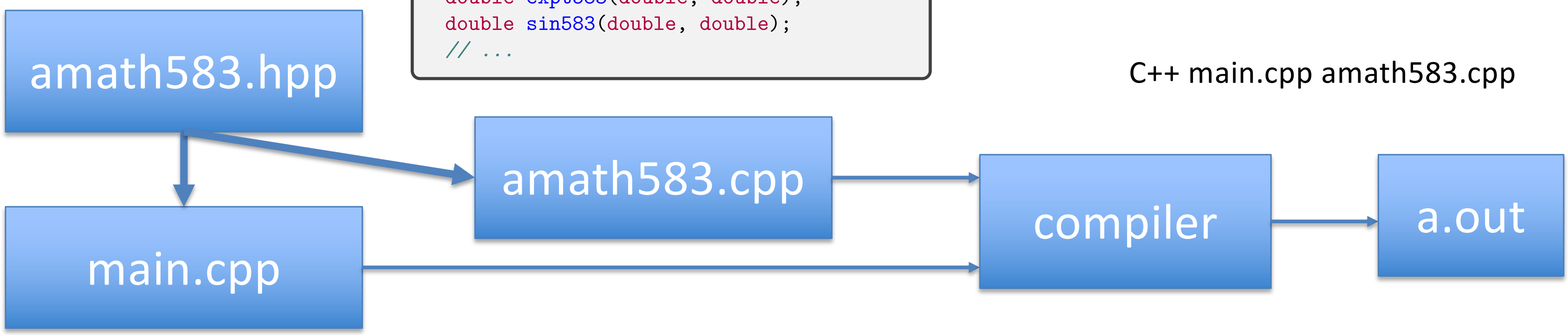
Review

- What is the difference between a function declaration and a function definition?
- Which do you need in order to be able to call a function from your code?
- Where do function declarations usually go?
- Where do function definitions usually go?

Program (file) organization (in pictures)

```cpp
// amath583.hpp: Declarations
double sqrt583(double);
double expt583(double, double);
double sin583(double, double);
// ...
```

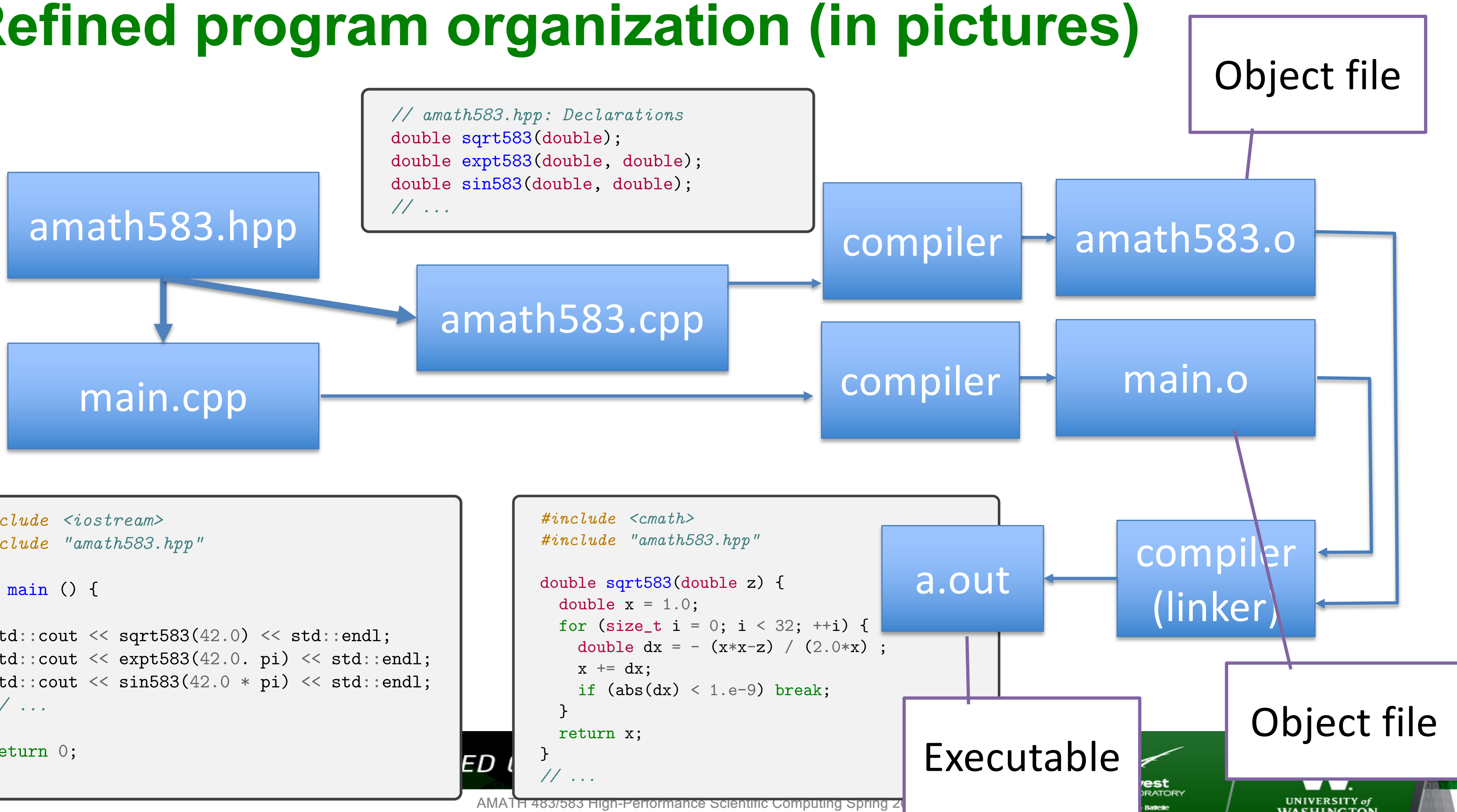
C++ main.cpp amath583.cpp




```cpp
#include <iostream>
#include "amath583.hpp"

int main () {

  std::cout << sqrt583(42.0) << std::endl;
  std::cout << expt583(42.0. pi) << std::endl;
  std::cout << sin583(42.0 * pi) << std::endl;
  // ...

  return 0;
}
```

```cpp
#include <cmath>
#include "amath583.hpp"

double sqrt583(double z) {
  double x = 1.0;
  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }
  return x;
}
// ...
```

Refined program organization (in pictures)




```
// amath583.hpp: Declarations
double sqrt583(double);
double expt583(double, double);
double sin583(double, double);
// ...
```

```
#include <iostream>
#include "amath583.hpp"

int main () {

  std::cout << sqrt583(42.0) << std::endl;
  std::cout << expt583(42.0. pi) << std::endl;
  std::cout << sin583(42.0 * pi) << std::endl;
  // ...

  return 0;
}
```

```
#include <cmath>
#include "amath583.hpp"

double sqrt583(double z) {
  double x = 1.0;
  for (size_t i = 0; i < 32; ++i) {
    double dx = - (x*x-z) / (2.0*x) ;
    x += dx;
    if (abs(dx) < 1.e-9) break;
  }
  return x;
}
// ...
```

Multifile Multistage Compilation

Compile main.cpp to main.o object file

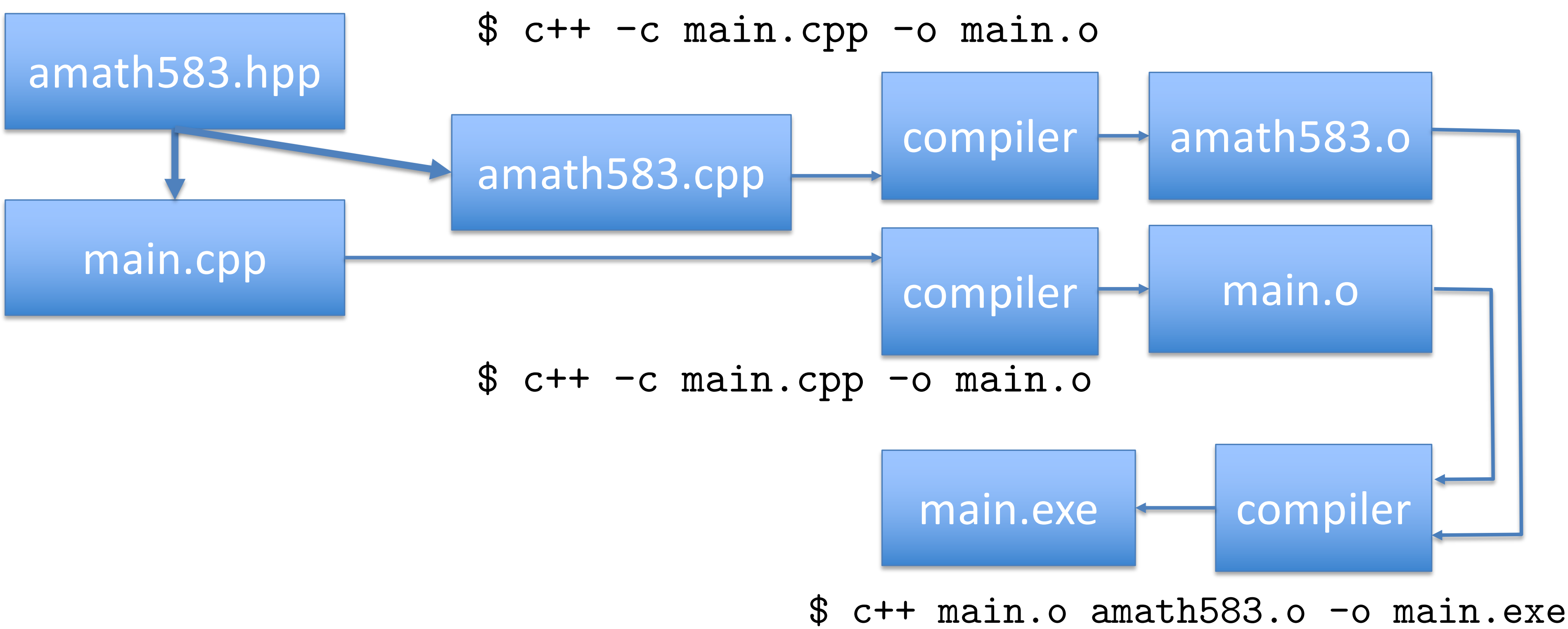
Tell the compiler to generate object

```
$ c++ -c main.cpp -o main.o
```

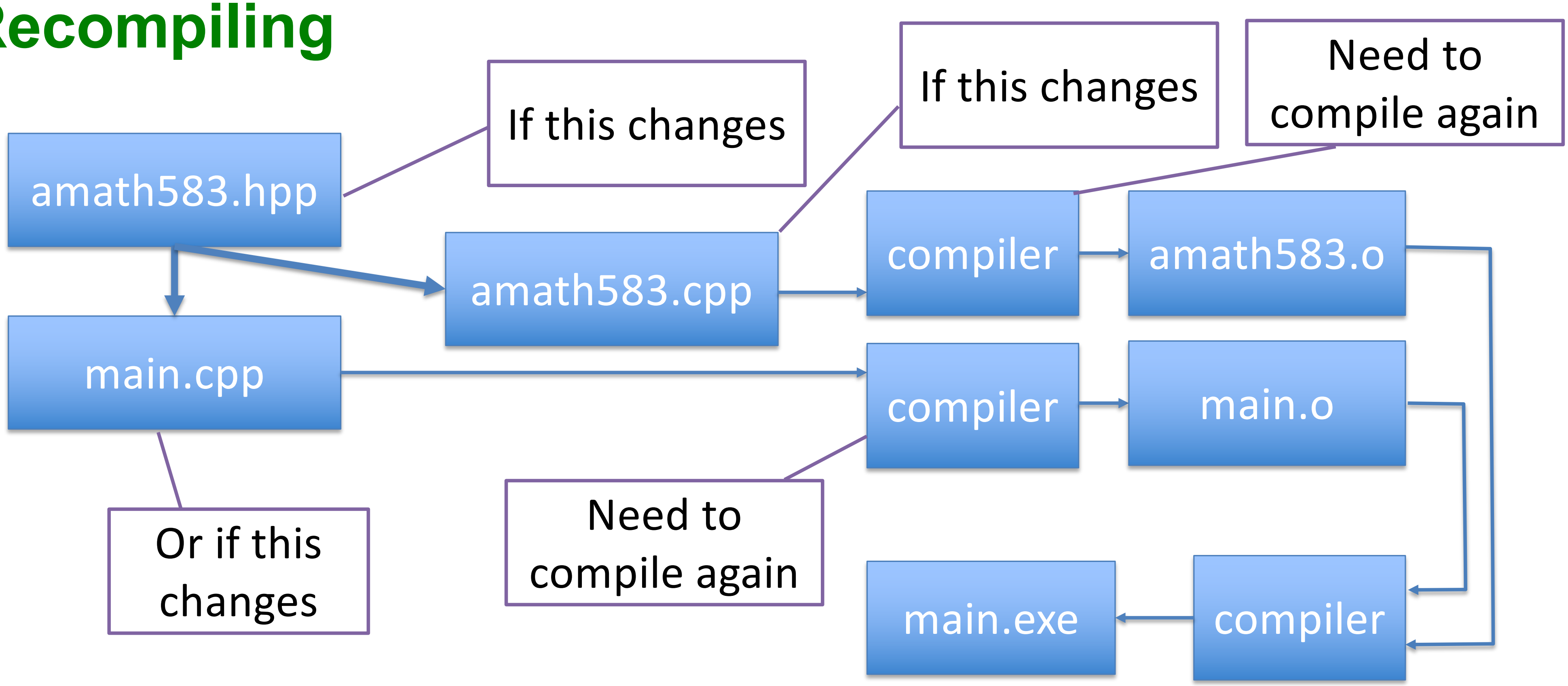
Tell the compiler name of the object

```
$ c++ -c amath583.cpp -o amath583.o

$ c++ main.o amath583.o -o main.exe
```

Multistage compilation (pictorially)



Recompiling



Dependencies

- main.o depends on main.cpp and amath583.hpp
- amath583.o depends on amath583.cpp
- main.exe depends on amath583.o and main.o

Automating: The Rules

- If main.o is newer than main.exe –
- If amath583.o is newer than main.
- If main.cpp is newer than main.o –
- If amath583.cpp is newer than am
- If amath583.hpp is newer than ma

Make

- Tool for automating compilation (or any other rule-driven tasks)
- Rules are specified in a makefile (usually named "Makefile")
- Rules include
  - Dependency
  - Target
  - Consequent

```
main.exe: main.o amath583.o
	c++ main.o amath583.o -o main.exe

main.o: main.cpp amath583.hpp
	c++ -c main.cpp -o main.o

amath583.o: amath583.cpp
	c++ -c amath583.cpp -o amath583.o
```

Target

Dependencies

Consequent

Make

- Tool for automating compilation (or any other rule-driven tasks)
- Rules are specified in a makefile (usually named "Makefile")
- Rules include
  - Dependency
  - Target
  - Consequent

```
$ make
c++ -c main.cpp -o main.o
c++ -c amath583.cpp -o amath583.o
c++ main.o amath583.o -o main.exe
```

- Edit amath583.hpp

```
$ make
c++ -c main.cpp -o main.o
c++ main.o amath583.o -o main.exe
```

Thank you!