

AMATH 483/583 High Performance Scientific Computing

Lecture 3: 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

NORTHWEST INSTITUTE for ADVANCED COMPUTING

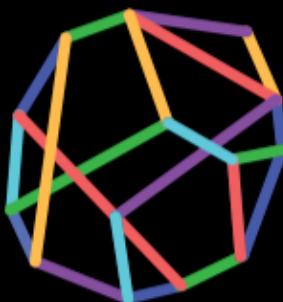
AMATH 483/583 High-Performance Scientific Computing Spring 2019
University of Washington by Andrew Lumsdaine

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

W
UNIVERSITY OF
WASHINGTON

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

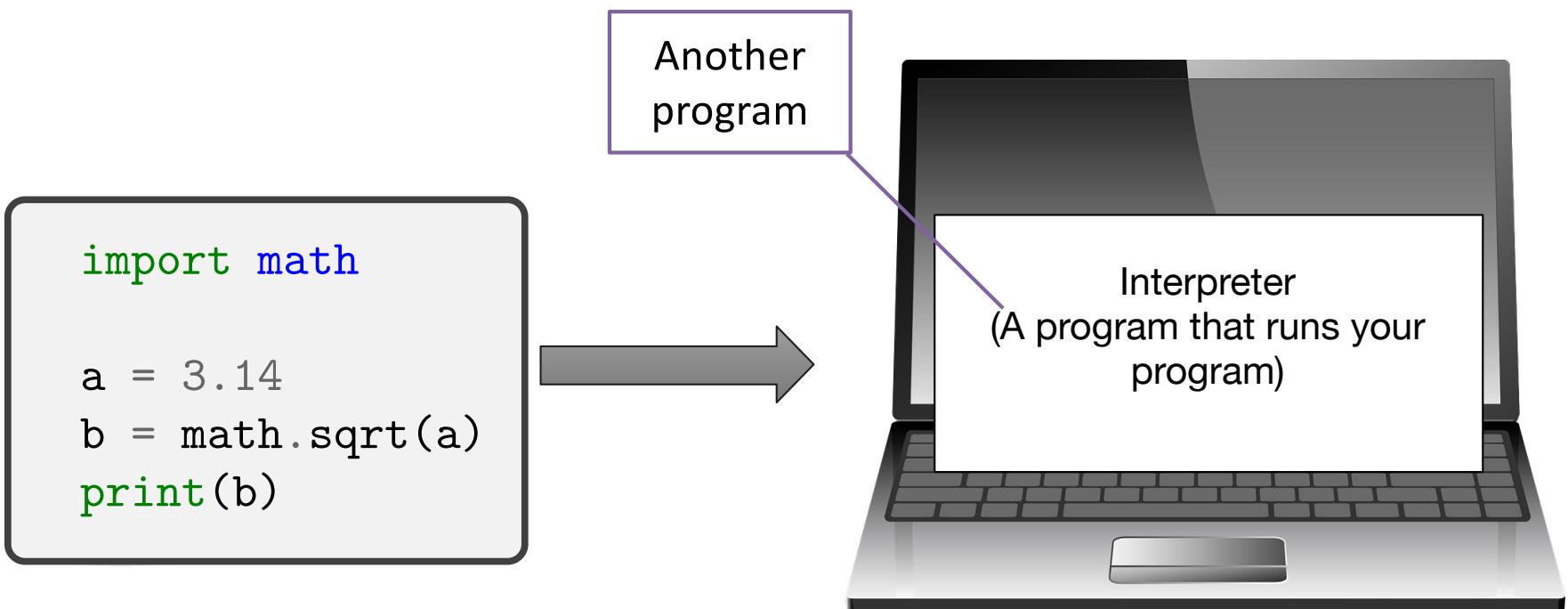
NORTHWEST INSTITUTE for ADVANCED COMPUTING

AMATH 483/583 High-Performance Scientific Computing Spring 2019
University of Washington by Andrew Lumsdaine

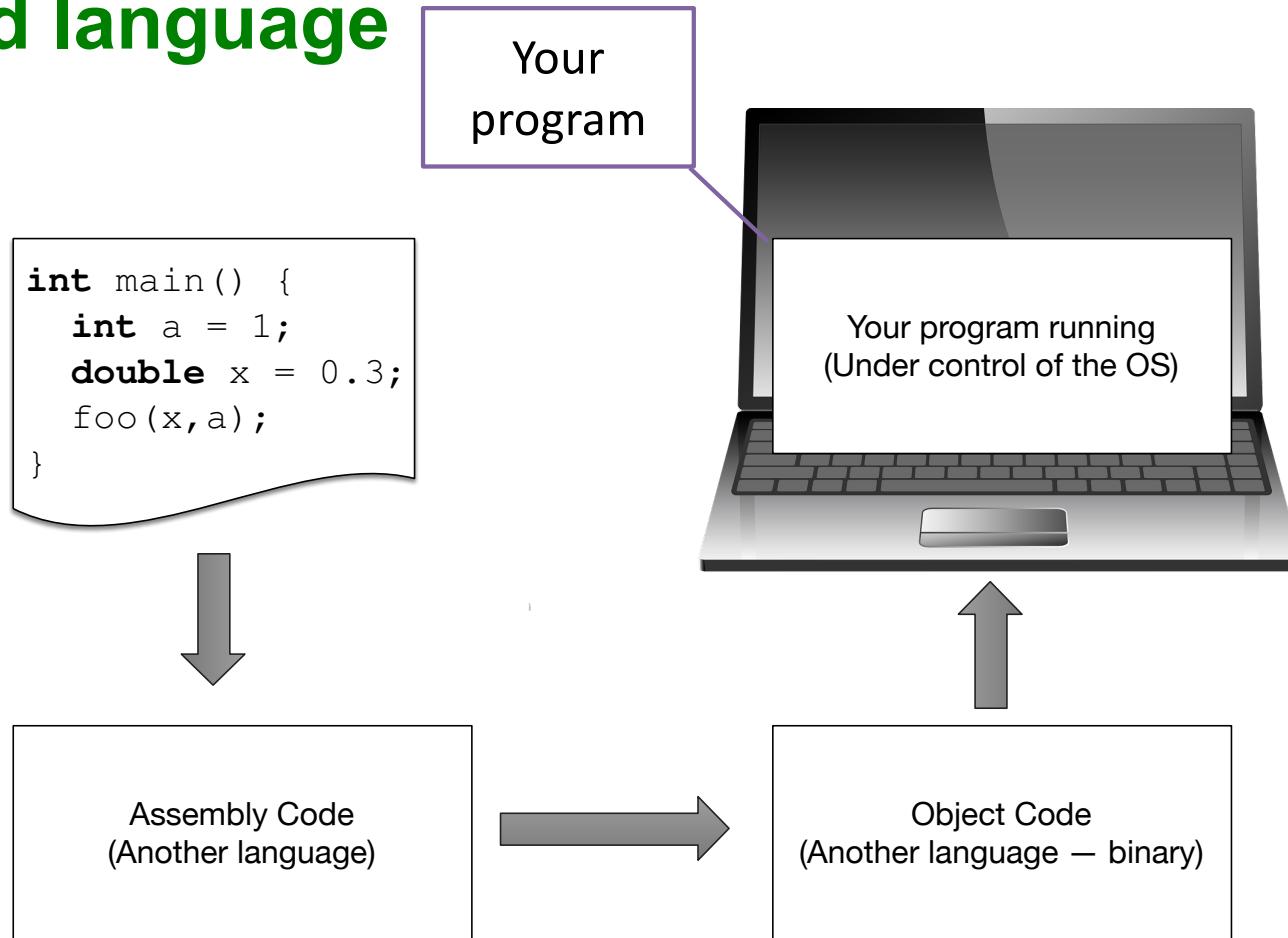
Pacific Northwest
NATIONAL LABORATORY
Partially Operated by Battelle
for the U.S. Department of Energy

W
UNIVERSITY OF
WASHINGTON

Interpreted language (Python)



Compiled language



Interpreted vs compiled

Use math library

Call function from math library

Use functions from iostream library

Use math library

```
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

int main()

double a = 3.14;
double b = std::sqrt(a);

std::cout << b << std::endl;
return 0;

Call function

"std" rather than "math"

#include <cmath>
<iostream>

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

Compile to
object file

Bits just for
this code

An
“executable”

Multi-step
process

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
- Now they can be defined anywhere in the block

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

Declaration

Use

```
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



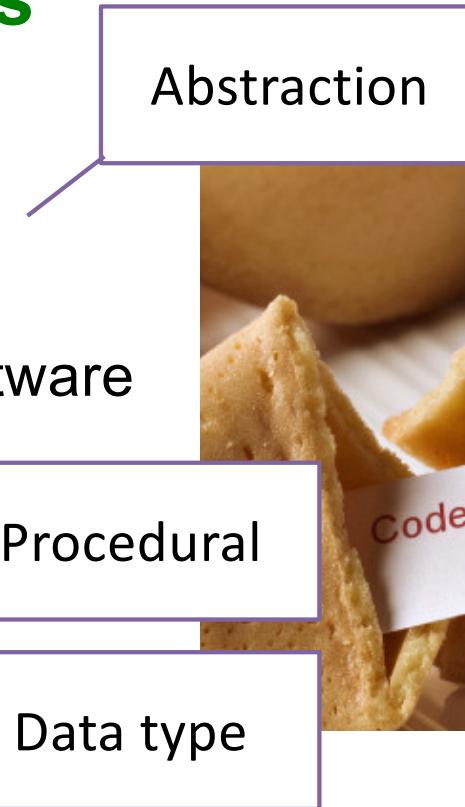
P.3: Express intent

```
= "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
 - 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: $f(x + \Delta x) \approx f(x) + \Delta x f'(x) = \Delta x f'(x)$

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

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

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>

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;
}
```

But they're not exactly the same

This is the only difference

This is the only difference

Procedural Abstraction

It returns
a double

It returns
a double

```
#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

Same code
as before

Except for
parameterization

Procedural Abstraction

It returns
a double

It returns
a double

```
#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?

Compiler can deduce return types

```
#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

Note auto is a
C++14 feature!

Square root of 2 and 3

Note initialization
and declaration of i

What is a size_t?

Pass
parameter 2

Pass
parameter 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;
}
```

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
```

```
#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

```
#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 () {
    std::cout << sqrt583(2.0) << std::endl;
    return 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;

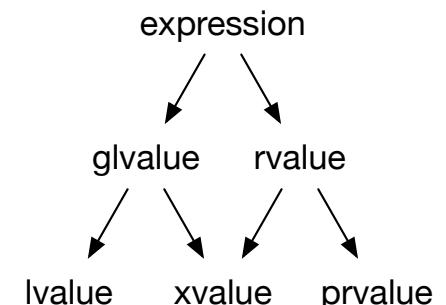
    return 0;
}
```

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

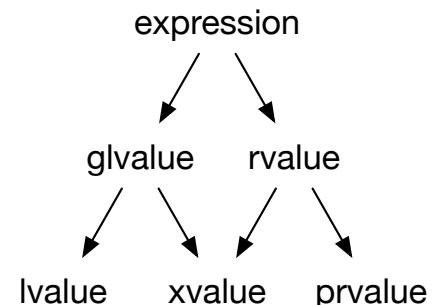
I-values and r-values

- 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



I-values and r-values

- More intuitively
- Ignore glvalue, xvalue, prvalue
- Ivalue 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



I-values and r-values

```
double x, y, z;
```

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

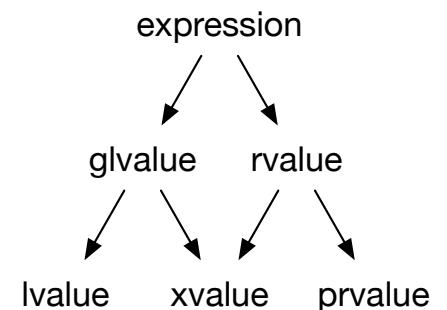
lvalue

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;
}
```

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

Compile main.cpp

Translate it into a language the cpu can run

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

The executable (program that the cpu can run)

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

Reusing in other programs

```
#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;
}
```

Put this function in its own file
amath583.cpp

Many programs (mains) can call it

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

    cout << sqrt583(3.0) << endl;

    return 0;
}
```

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

    cout << sqrt583(3.14) << endl;

    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;
}
```

Defined in a
different file

```
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

Didn't we declare it here?

This is *definition*

```
#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;
}
```

Reusing functions

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
1.4142
```

Compile main.cpp

Translate it into a
language the cpu can run

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

The executable (program
that the cpu can run)

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

Reusing functions across programs

Declare sqrt583 is a function that exists

~~clude <iostream>~~

Returns a double

```
double sqrt583(double);  
  
int main () {  
    std::cout << sqrt583(42.0) << std::endl;  
  
    return 0;  
}
```

Takes a double

Now we know how to call it

Reusing in other programs

```
#include <iostream>

double sqrt583(double);

int main () {

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

    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

```
#include <iostream>

double sqrt583(double);

int main () {

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

    return 0;
}
```

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

Compile main.cpp

Translate it into a language the cpu can run

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

The executable (program that the cpu can run)

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

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;
}
```

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

Translate it into a
language the cpu can run

```
#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;
}
```

The executable (program
that the cpu can run)

\$ c++ main.cpp sqrt583.cpp

\$./a.out

Reusing in other programs

```
#include <iostream>

double sqrt583(double);

int main () {

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

    return 0;
}
```

\$ c++ main.cpp

Compile main.cpp by itself

```
#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++ sqrt583.cpp

Compile sqrt583.cpp by itself

\$./a.out

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

    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

```
#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

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

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

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

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?

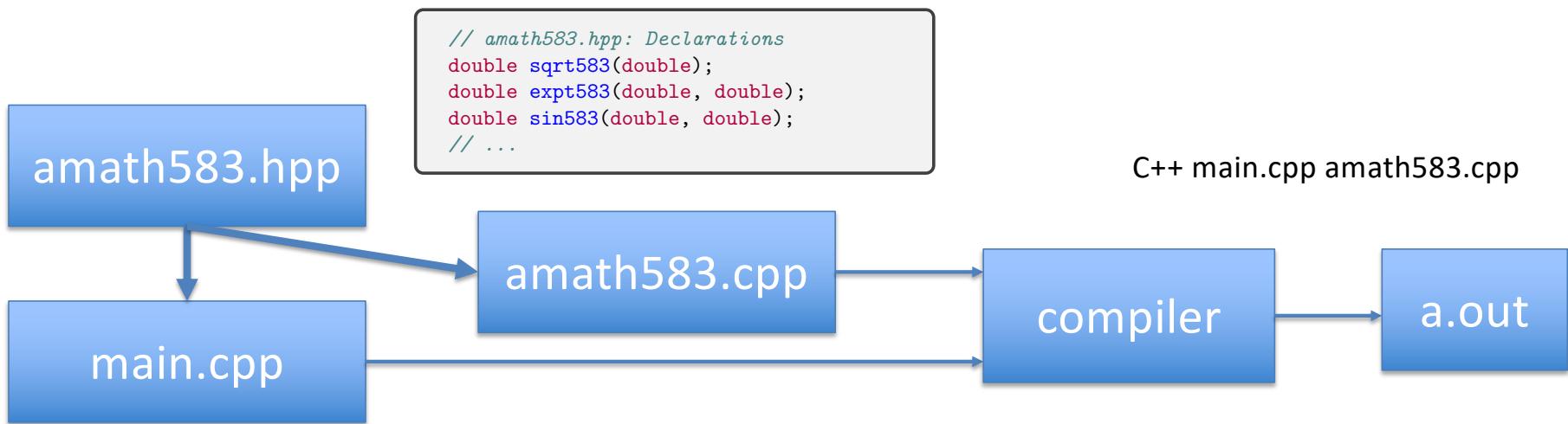
NORTHWEST INSTITUTE for ADVANCED COMPUTING

AMATH 483/583 High-Performance Scientific Computing Spring 2019
University of Washington by Andrew Lumsdaine

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

W
UNIVERSITY OF
WASHINGTON

Program (file) organization (in pictures)



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

int main () {

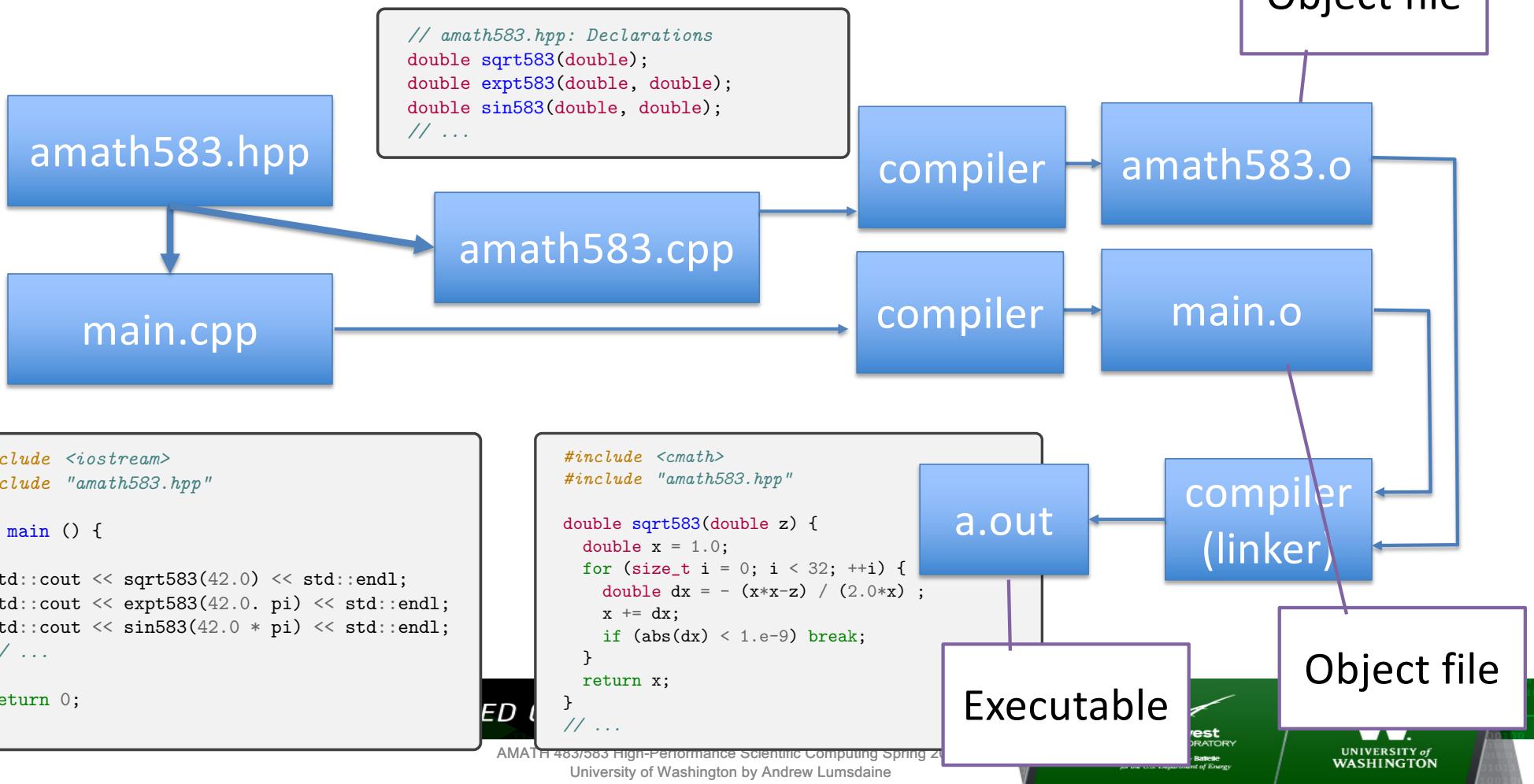
    std::cout << sqrt583(42.0) << std::endl;
    std::cout << expt583(42.0, pi) << std::endl;
    std::cout << sin583(42.0 * pi) << std::endl;
// ...

    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;
}
// ...
```

Refined program organization (in pictures)



Multifile Multistage Compilation

Compile main.cpp to
main.o object file

Tell the compiler to
generate object

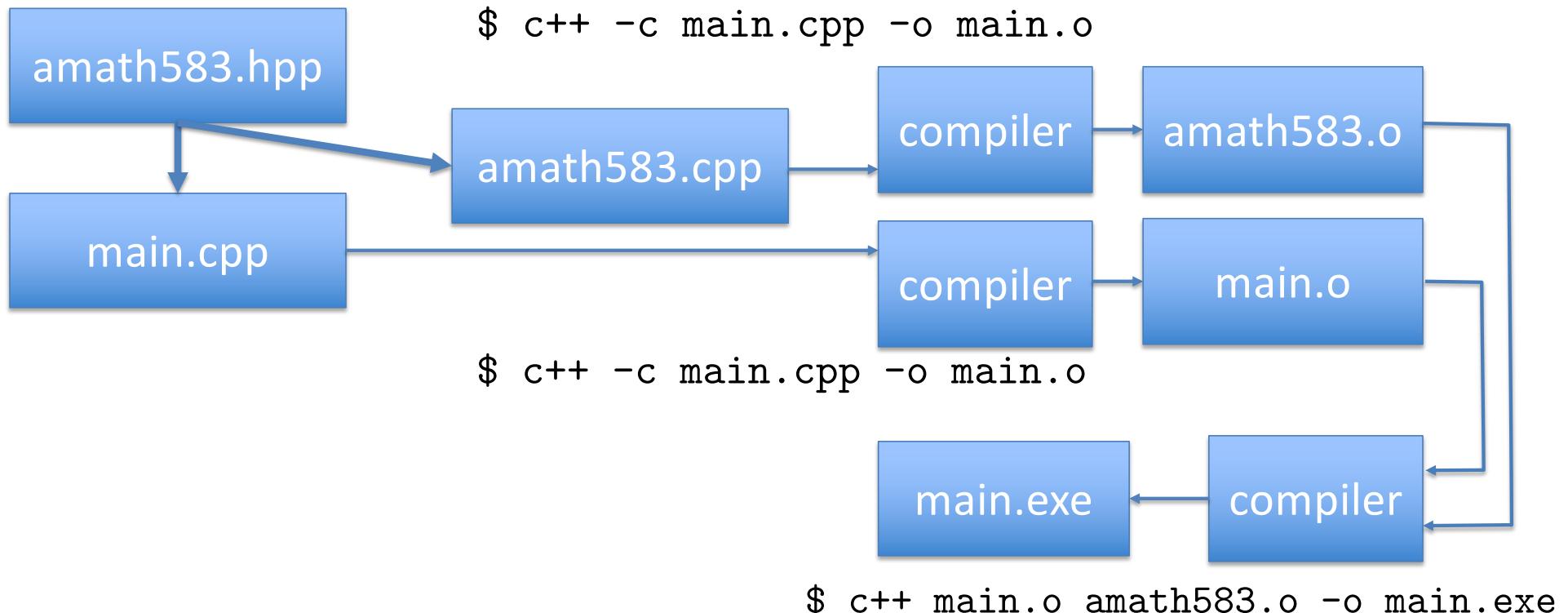
```
$ c++ -c main.cpp -o main.o
```

Tell the compiler
name of the object

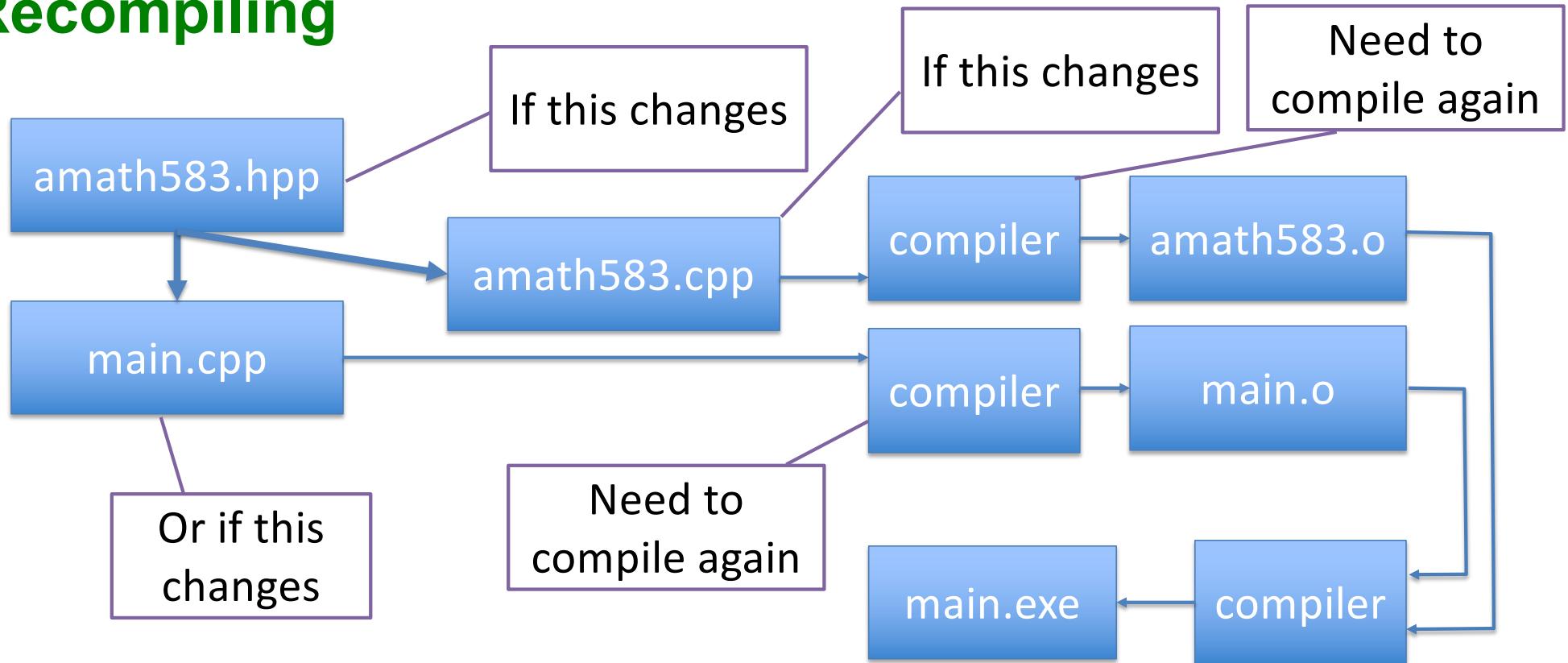
```
$ c++ -c amath583.cpp -o amath583.o
```

```
$ 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.o –
- If main.cpp is newer than main.o –
- If amath583.cpp is newer than amath583.o –
- If amath583.hpp is newer than main.h –



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
- Edit amath583.hpp

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

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

Computational Science

System of Partial
Differential Eqns

$$\begin{aligned}\nabla \cdot \mathbf{P} &= \mathbf{f}_0 \quad \text{in } \Omega_0 \\ [\mathbf{P} \cdot \mathbf{N}_0] &= [t_c] \quad \text{on } S_0 \\ \mathbf{P} \cdot \mathbf{N}_0 &= t_0 \quad \text{on } \partial\Omega_{t_0} \\ \mathbf{u} &= \mathbf{u}_p \quad \text{on } \partial\Omega_{u_0}\end{aligned}$$

Find \mathbf{P} that
satisfies this

(too hard)

System of
Nonlinear Eqns

$$F(x) = 0$$

Find x that
satisfies this

(too hard)

discretize

System of Linear
Eqns

$$Ax = b$$

Find x that
satisfies this

linearize

A problem we
can solve

Computational Science

- The fundamental computation at the core of many (most/all) computational science programs is solving $Ax = b$
- Assume $x, b \in R^N$ and $A \in R^{N \times N}$
- I.e., x and b are vectors with N real elements and A is a matrix with N by N real elements
- Solution process only requires basic arithmetic operations

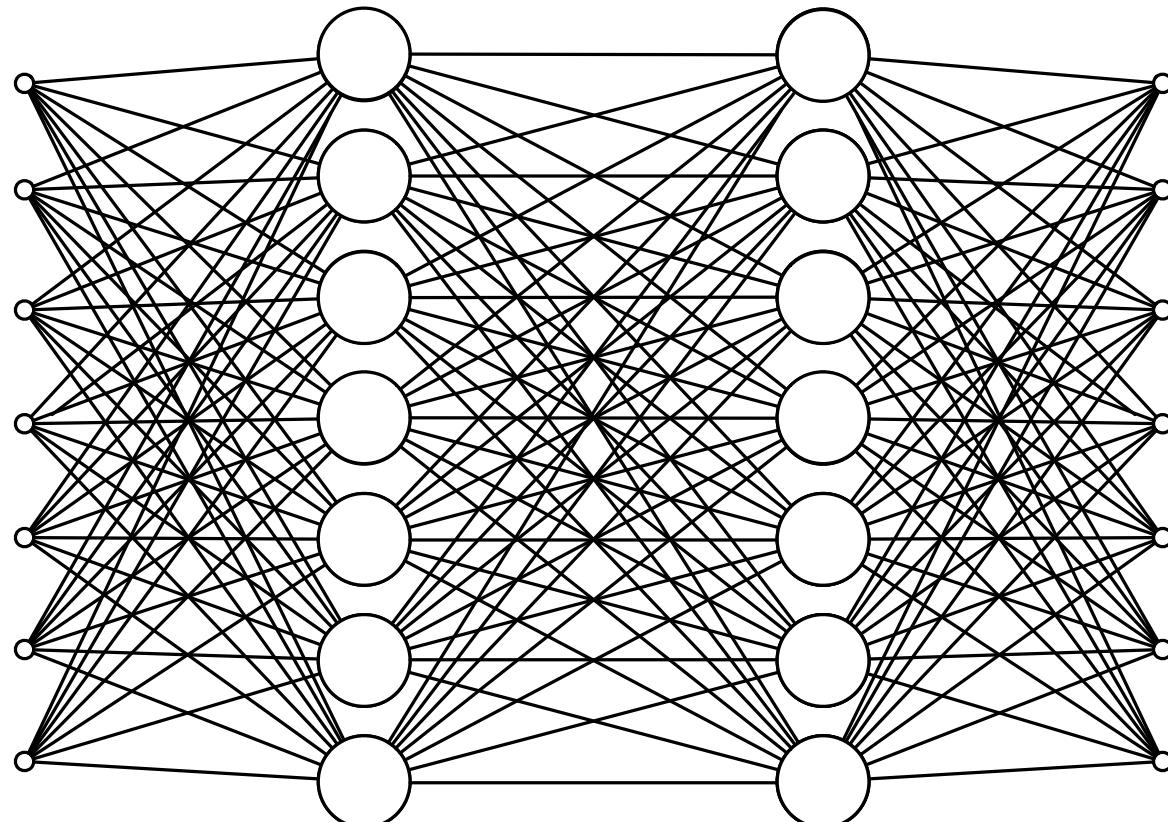
NORTHWEST INSTITUTE for ADVANCED COMPUTING

AMATH 483/583 High-Performance Scientific Computing Spring 2019
University of Washington by Andrew Lumsdaine

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

W
UNIVERSITY OF
WASHINGTON

Neural Network



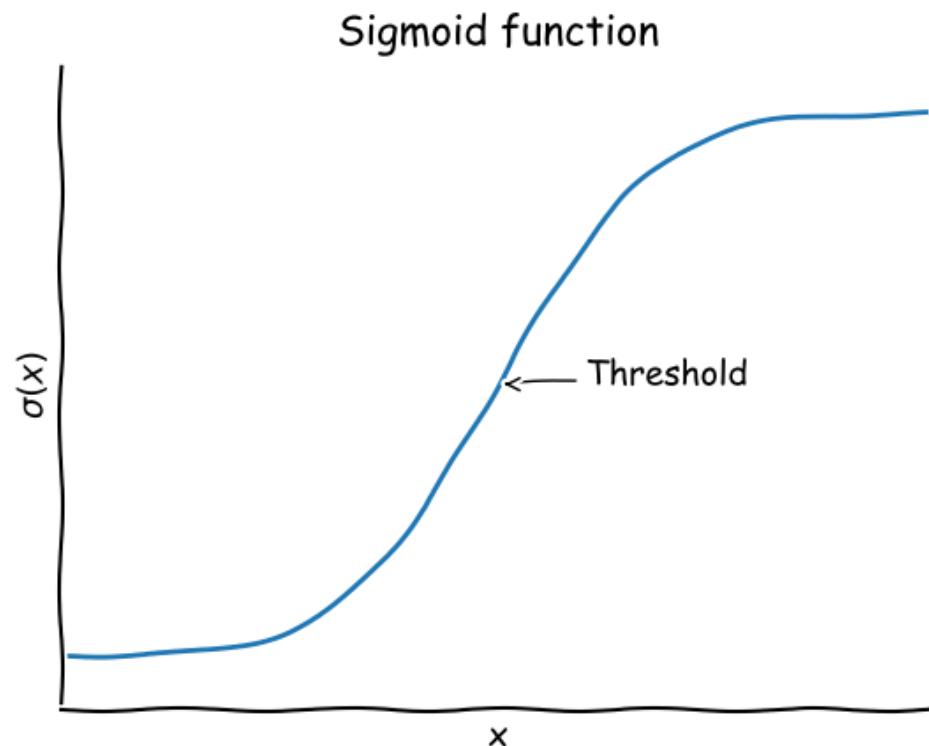
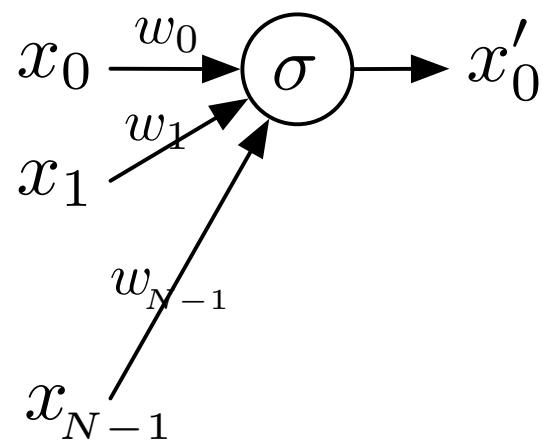
NORTHWEST INSTITUTE for ADVANCED COMPUTING

AMATH 483/583 High-Performance Scientific Computing Spring 2019
University of Washington by Andrew Lumsdaine

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

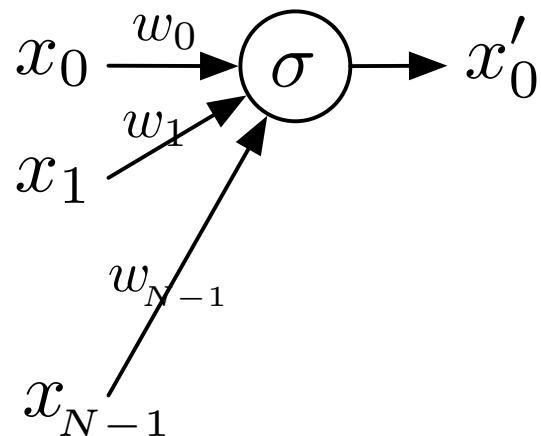
W
UNIVERSITY OF
WASHINGTON

Zoom In On One "Neuron"



Zoom In On One "Neuron"

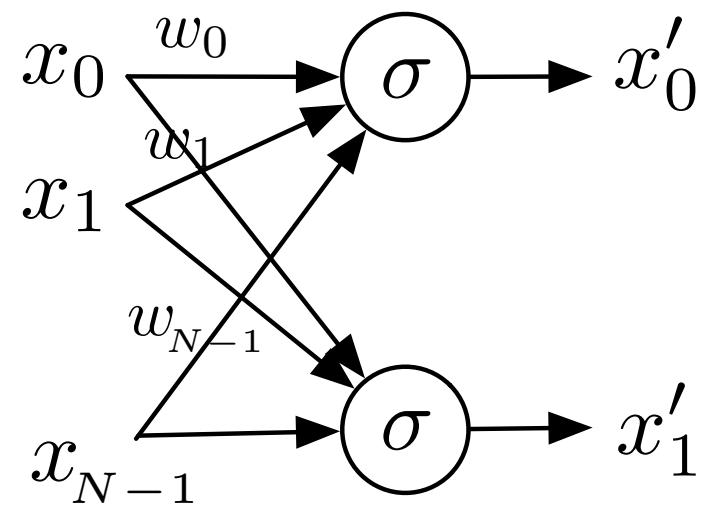
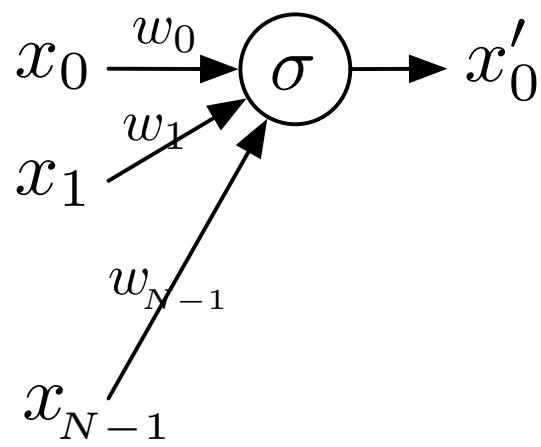
$$x'_0 = \sigma(t)$$



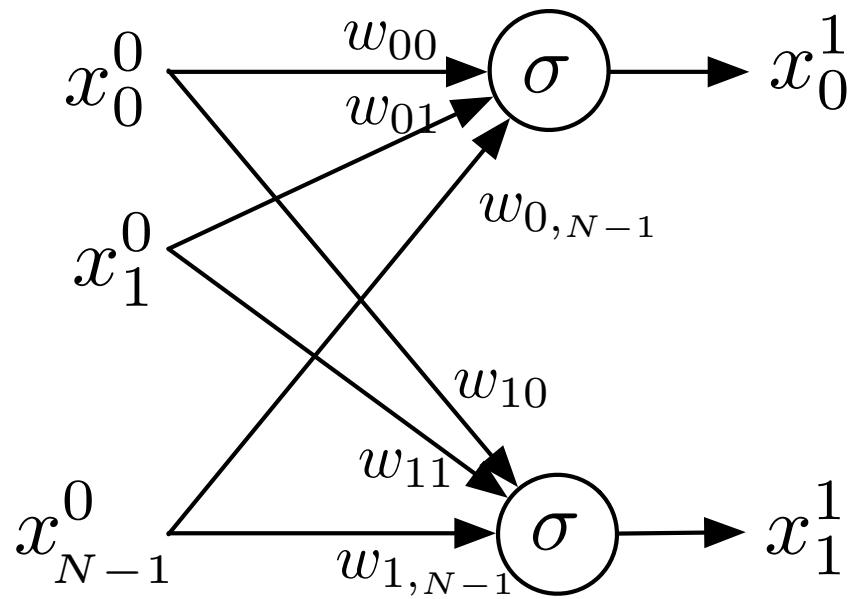
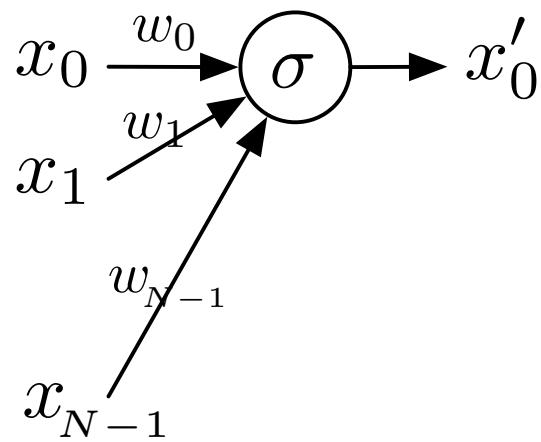
$$\begin{aligned} t &= w_0 x_0 + w_1 x_1 + \cdots + w_{n-1} x_{n-1} \\ &= \sum_{i=0}^{N-1} w_i x_i \end{aligned}$$

$$x'_0 = \sigma\left(\sum_{i=0}^{N-1} w_i x_i\right)$$

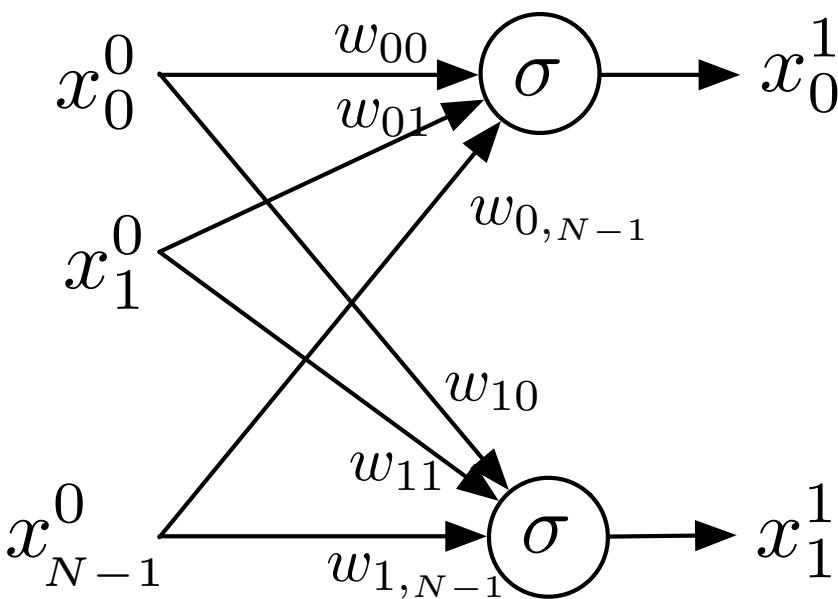
Zoom In On Two “Neurons”



Zoom In On Two “Neurons”



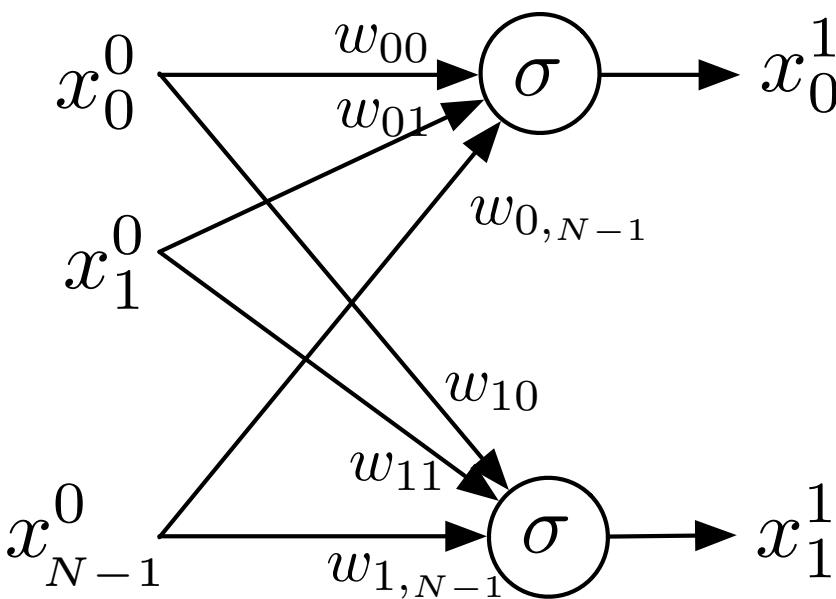
Zoom In On Two “Neurons”



$$x_0^1 = \sigma\left(\sum_{i=0}^{N-1} w_{0i} x_i^0\right)$$

$$x_1^1 = \sigma\left(\sum_{i=0}^{N-1} w_{1i} x_i^0\right)$$

Zoom In On Two “Neurons”



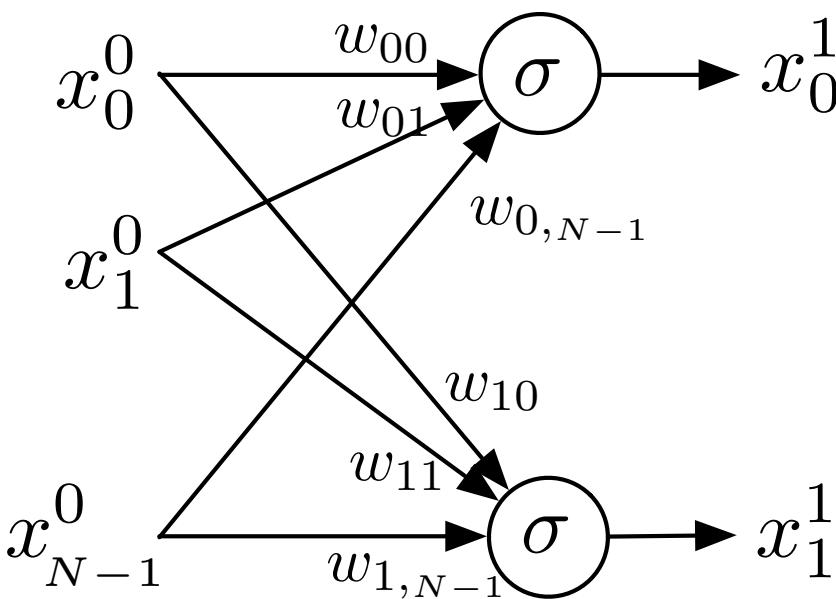
$$x_0^1 = \sigma\left(\sum_{i=0}^{N-1} w_{0i} x_i^0\right)$$

$$x_1^1 = \sigma\left(\sum_{i=0}^{N-1} w_{1i} x_i^0\right)$$

⋮

$$x_{N-1}^1 = \sigma\left(\sum_{i=0}^{N-1} w_{N-1,i} x_i^0\right)$$

Zoom In On Two “Neurons”



$$S(x) = \begin{bmatrix} \sigma(x_0) \\ \sigma(x_1) \\ \vdots \\ \sigma(x_{N-1}) \end{bmatrix}$$

$$x^1 = S(Wx^0)$$

vector

matrix

vector

Mathematical Vector Space

Definition. (Halmos) A vector space is a set V of elements called *vectors* satisfying the following axioms:

1. To every pair x and y of vectors in V there corresponds a vector $x + y$ called the *sum* of x and y in such a way that

commutative

associative

We need to be able to add 2 vectors → vector

- (a) addition is commutative, $x + y = y + x$
- (b) addition is associative, $x + (y + z) = (x + y) + z$
- (c) there exists in V a unique vector 0 (called the origin) such that $x + 0 = x$ for every vector x , and
- (d) to every vector x in V there corresponds a unique vector $-x$ such that $x + (-x) = 0$

2. To every pair a and x where a is a scalar and x is a vector in V , there corresponds a vector ax in V called the product of a and x in such a way that

Identity over +

- (a) multiplication by scalars is associative $a(bx) = (ab)x$, and

- (b) $1x = x$ for every vector x .

Identity over x

associative

distributive

3. (a) Multiplications by scalar is distributive with respect to vector addition. $a(x + y) \neq ax + ay$

- (b) multiplication by vectors is distributive with respect to scalar addition $(a + b)x = ax + by$

Mathematical Vector Space Examples

Definition. (Halmos) A vector space is a set V of elements called *vectors* satisfying the following axioms:

1. To every pair x and y of vectors in V there corresponds a vector $x + y$ called the *sum* of x and y in such a way that
 - (a) addition is commutative, $x + y = y + x$
 - (b) addition is associative, $x + (y + z) = (x + y) + z$
 - (c) there exists in V a unique vector 0 (called the origin) such that $x + 0 = x$ for every vector x , and
 - (d) to every vector x in V there corresponds a unique vector $-x$ such that $x + (-x) = 0$
2. To every pair a and x where a is a scalar and x is a vector in V , there corresponds a vector ax in V called the product of a and x in such a way that
 - (a) multiplication by scalars is associative $a(bx) = (ab)x$, and
 - (b) $1x = x$ for every vector x .
3. (a) Multiplications by scalar is distributive with respect to vector addition. $a(x + y) = ax + ay$
(b) multiplication by vectors is distributive with respect to scalar addition $(a + b)x = ax + by$

- Set of all complex numbers
- Set of all polynomials
- Set of all n-tuples of real numbers R^N

The vector space
used in scientific
computing

Computer Representation of Vector Space

Definition. (Halmos) A vector space is a set V of elements called *vectors* satisfying the following axioms:

1. To every pair x and y of vectors in V there corresponds a vector $x + y$ called the *sum* of x and y in such a way that

commutative

associative

We need to be able to add 2 vectors → vector

- (a) addition is commutative, $x + y = y + x$
- (b) addition is associative, $x + (y + z) = (x + y) + z$
- (c) there exists in V a unique vector 0 (called the origin) such that $x + 0 = x$ for every vector x , and
- (d) to every vector x in V there corresponds a unique vector $-x$ such that $x + (-x) = 0$

2. To every pair a and x where a is a scalar and x is a vector in V , there corresponds a vector ax in V called the product of a and x in such a way that

Identity over +

- (a) multiplication by scalars is associative $a(bx) = (ab)x$, and

- (b) $1x = x$ for every vector x .

Identity over x

associative

distributive

distributive

3. (a) Multiplications by scalar is distributive with respect to vector addition. $a(x + y) = ax + ay$

- (b) multiplication by vectors is distributive with respect to scalar addition $(a + b)x = ax + by$

Computer Representation of Vector Space

- In the bad old days, vectors represented as arrays

```
REAL X(N)
```

```
REAL Y(N)
```

- Add them `CALL SAXPY(N, ALPHA, X, Y) $Y \leftarrow \alpha X + Y$`

- Double precision

```
DOUBLE X(N)
```

```
DOUBLE Y(N)
```

Two different functions

For same operation

- Add them `CALL DAXPY(N, ALPHA, X, Y) $Y \leftarrow \alpha X + Y$`

```
for (int i = 0; int < N; ++i) y[i] += alpha * x[i];
```

For same implementation

Vectors Spaces in C++

- Despite the clumsiness of Fortran interface (or maybe because of it) the performance of vector operations was quite good
- In C/C++, there are numerous options for vectors (and matrices)

```
double x[N];
```

Not dynamically
sizable*

```
double *x = malloc(N * sizeof(double));
```

Memory
management hell

```
vector<double> x(N);
```

Limited to interface of
vector<double> (not a vector)

```
Vector x(N);
```

Just right, or very wrong

We can define interface and implementation

Vectors Spaces in C++

- Despite the clumsiness of Fortran interface (or maybe because of it) the performance of vector operations was quite good
- In C/C++, there are numerous options for vectors (and matrices)

```
double A [M] [N];
```

Not dynamically
sizable*

Memory management
hell squared

```
double **A = ??;
```

Really easy to get bad
performance

```
vector<vector<double>> x (N);
```

Not a matrix (or a 2D array for
that matter) at all

```
Matrix A (M, N);
```

Just right, or very wrong

We can define interface and implementation

Classes

- First principles: Abstraction, simplicity, consistent specification
 - Domain: Scientific computing
 - Domain abstractions: Matrices and vectors
 - Programming abstractions: Matrix and Vector
-
- C++ classes enable encapsulation of related data and functions
 - Provides visible interface
 - Hides implementation

std::vector<double>

- Before rushing off to implement fancy interfaces
- Understand what we are working with
- And how hardware and software interact
- std::vector<double> will be our storage
- But its interface won't be our interface
 - We will gradually build up to complete Vector
 - And complete Matrix



The Standard Template Library

- In early-mid 90s Stepanov, Musser, Lee applied principles of ***generic programming*** to C++
- Leveraged templates / parametric polymorphism

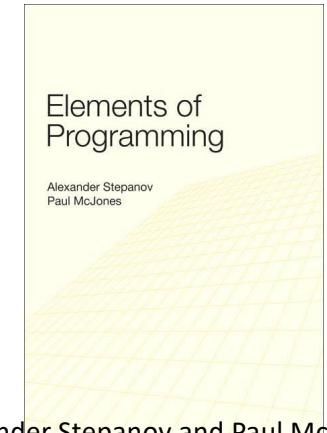
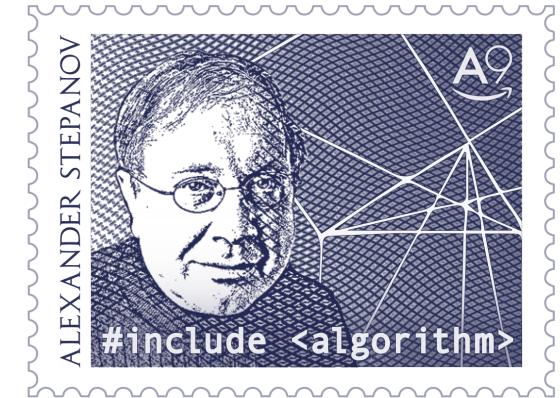
std::set
std::list
std::map
std::vector
...

ForwardIterator
ReverseIterator
RandomAccessIterator

std::for_each
std::sort
std::accumulate
std::min_element
...

Containers

Algorithms



Alexander Stepanov and Paul McJones.
2009. *Elements of Programming* (1st ed.). Addison-Wesley Professional.

Generic Programming

- Algorithms are *generic* (parametrically polymorphic)
- Algorithms can be used on *any* type that meets algorithmic reqts
 - Valid expressions, associated types
 - Not just std. ::types

```
vector<double> array (N);  
...  
std::accumulate(array.begin(), array.end(), 0.0);
```

Standard Library container

iterator

iterator

Initial value

std Containers

- Note that all containers have **same** interface
- (Actually a hierarchy, we'll come back to this)
- We will primarily be focusing on vector

Headers		<code><vector></code>	<code><deque></code>	<code><list></code>
Members		<code>vector</code>	<code>deque</code>	<code>list</code>
	constructor	<code>vector</code>	<code>deque</code>	<code>list</code>
	operator=	<code>operator=</code>	<code>operator=</code>	<code>operator=</code>
iterators	begin	<code>begin</code>	<code>begin</code>	<code>begin</code>
	end	<code>end</code>	<code>end</code>	<code>end</code>
capacity	size	<code>size</code>	<code>size</code>	<code>size</code>
	max_size	<code>max_size</code>	<code>max_size</code>	<code>max_size</code>
	empty	<code>empty</code>	<code>empty</code>	<code>empty</code>
	resize	<code>resize</code>	<code>resize</code>	<code>resize</code>
element access	front	<code>front</code>	<code>front</code>	<code>front</code>
	back	<code>back</code>	<code>back</code>	<code>back</code>
	operator[]	<code>operator[]</code>	<code>operator[]</code>	
modifiers	insert	<code>insert</code>	<code>insert</code>	<code>insert</code>
	erase	<code>erase</code>	<code>erase</code>	<code>erase</code>
	push_back	<code>push_back</code>	<code>push_back</code>	<code>push_back</code>
	pop_back	<code>pop_back</code>	<code>pop_back</code>	<code>pop_back</code>
	swap	<code>swap</code>	<code>swap</code>	<code>swap</code>

std Containers

- std containers “contain” elements

```
vector<double> array (N);
```

vector of doubles

```
vector<int> array (N);
```

vector of ints

```
list<vector<complex<double>> > thing;
```

list of vectors of complex doubles

- Implementation of list, vector, complex is the same regardless of what is being contained

Generic Programming

- Algorithms are *generic* (parametrically polymorphic)
- Algorithms can be used on *any* type that meets algorithmic reqts
 - Valid expressions, associated types
 - Not just std. ::types

```
list<vector<complex<double>>> thing(N);
```

```
...  
std::accumulate(thing.begin(), thing.end(), 0.0);
```

Standard Library container

iterator

iterator

Initial value

std Containers

- The std containers are **class templates** (not “template classes”)

```
template <typename T> class vector;  
template <typename T> class deque;  
template <typename T> class list;
```

What follows is
a template

The template
parameter is a
type placeholder

A class
template

- Don't need details for now

`vector<double>`

Our goal

- Extract maximal performance from one core, multiple cores, multiple machines for computational (and data) science
- Two algorithms: matrix-matrix product, (sparse) matrix-vector product

$$A, B, C \in R^{N \times N} \quad C = A \times B \quad C_{ij} = \sum_k A_{ik} B_{kj}$$

Matrix `A(M,N);`
...
`for (int i = 0; i < N; ++i)
 for (int j = 0; j < N; ++j)
 for (int k = 0; k < N; ++k)
 C(i,j) += A(i,k) * B(k,j)`

What does
the hard-
ware do?



Classes

- First principles: Abstraction, simplicity, consistent specification
 - Domain: Scientific computing
 - Domain abstractions: Matrices and vectors
 - Programming abstractions: Matrix and Vector
-
- C++ classes enable encapsulation of related data and functions
 - Provides visible interface
 - Hides implementation

Vector desiderata

- Mathematically we say let $v \in \mathbb{R}^N$
 - There are N real number elements
 - Accessed with subscript
 - (Vectors can be scaled, added)
-
- Programming abstraction
 - Create a Vector with N elements
 - Access elements with “subscript”

Using Vector class

```
int main() {
    size_t num_rows = 1024;

    Vector v1(num_rows);

    for (size_t i = 0; i < v1.num_rows(); ++i) {
        v1(i) = i;
    }

    return 0;
}
```

Using Vector class

```
int main() {  
    size_t num_rows = 1024;  
  
    Vector v1(num_rows);  
  
    for (size_t i = 0; i < v1.num_rows(); ++i) {  
        v1(i) = i;  
    }  
  
    Vector v2 (v1);  
    Vector v3 = v1;  
    v3 = v2;  
  
    return 0;  
}
```

Declare (construct) a Vector with num_rows elements

Get its size

Index each element

Copy (assign) in various ways

Using Vector class

```
int main() {  
    size_t num_rows = 1024;  
  
    Vector v1(num_rows);  
  
    for (size_t i = 0; i < v1.num_rows(); ++i) {  
        v1(i) = i;  
    }  
  
    Vector v2(v1);  
    Vector v3 = v1;  
    v3 = v2;  
  
    return 0;  
}
```

Declare (construct) a Vector with num_rows elements

Get its size

Index each element

Copy (assign) in various ways

Interface vs Implementation

Know nothing about **what** a Vector is – only how to use it



Anatomy of a C++ class

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Declares an interface

Hides implementation

Anatomy of a C++ class

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Define a new class

Name of the class

A class is a “recipe”
for objects

A class is a user-
defined type

And hides
implementation

Interface specifies
how to use objects

Objects are variables
of that type

Anatomy of a C++ class

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Create a Vector with
n elements (M)

Access elements
with a subscript

Anatomy of a C++ class

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator[](size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Constructor (function
that makes new object)

The name of a
constructor is the same
as the name of the class

This constructor function
takes one argument

Anatomy of a C++ class

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Everything following
the public: declaration
is public

Code outside of the object
can access public members
(functions or data)

Anatomy of a C++ class

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Three public member functions

Constructor

Subscript

“size”

Anatomy of a C++ class

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Everything following
the private: declaration
is private

Code outside of the object
can not access private
members (functions or data)

But member
functions can

Anatomy of a C++ class

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

And when?

How do we set these to
the right size, right value?

And to
what?

Store the size of
the Vector

Store the n elements of the
Vector as a
std::vector<double>

Anatomy of a C++ class

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Store the number
of elements

Store the n elements of the
Vector as a
std::vector<double>

Using Vector class

```
int main() {  
    size_t num_rows = 1024;  
  
    Vector v1(num_rows);  
  
    for (size_t i = 0; i < v1.num_rows(); ++i) {  
        v1(i) = i;  
    }  
  
    Vector v2(v1);  
    Vector v3 = v1;  
    v3 = v2;  
  
    return 0;  
}
```

Declare (construct) a Vector
with num_rows elements

Anatomy of a C++ class

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

The number of elements

In the constructor we want to set this to M

And make this num_rows_ elements long

Anatomy of a C++ class

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

The constructor is
a function

And it has a body

One option for initialization

```
class Vector {  
public:  
    Vector(size_t M) {  
        num_rows_ = M;  
        storage = std::vector<double>(num_rows_);  
    }  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Set num_rows_ to M

Construct storage_ with num_rows_ elements

Preferred initialization

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Set num_rows_ to M

Object is well-formed before body of function

Construct storage_

Using Vector class

```
int main() {  
    size_t num_rows = 1024;  
  
    Vector v1(num_rows);  
  
    for (size_t i = 0; i < v1.num_rows(); ++i) {  
        v1(i) = i;  
    }  
  
    Vector v2 (v1);  
    Vector v3 = v1;  
    v3 = v2;  
  
    return 0;  
}
```

Access num_rows

Call the num_rows()
member function for
object v1

Member function

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Just a function

Function body

Returns a size_t

Takes no arguments

Member function

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const;  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

`size_t Vector::num_rows() const { return num_rows_ ;};`

Interface in
Vector.hpp

Function declaration
(implementation
elsewhere)

Implementation
in Vector.cpp

Member function

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Subscript

In our next
exciting episode

C++ Core Guidelines related to classes

- C.1: Organize related data into structures (structs or classes)
- C.3: Represent the distinction between an interface and an implementation using a class
- C.4: Make a function a member only if it needs direct access to the representation of a class
- C.10: Prefer concrete types over class hierarchies
- C.11: Make concrete types regular

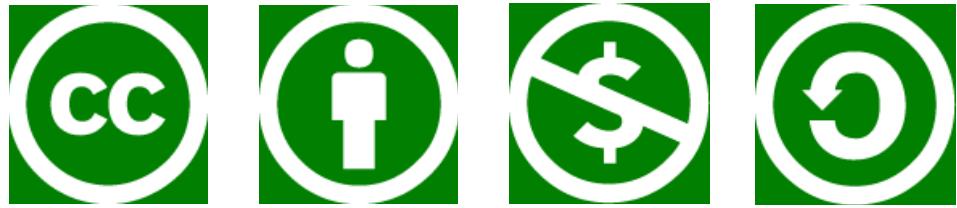
Thank you!

NORTHWEST INSTITUTE for ADVANCED COMPUTING

AMATH 483/583 High-Performance Scientific Computing Spring 2019
University of Washington by Andrew Lumsdaine

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

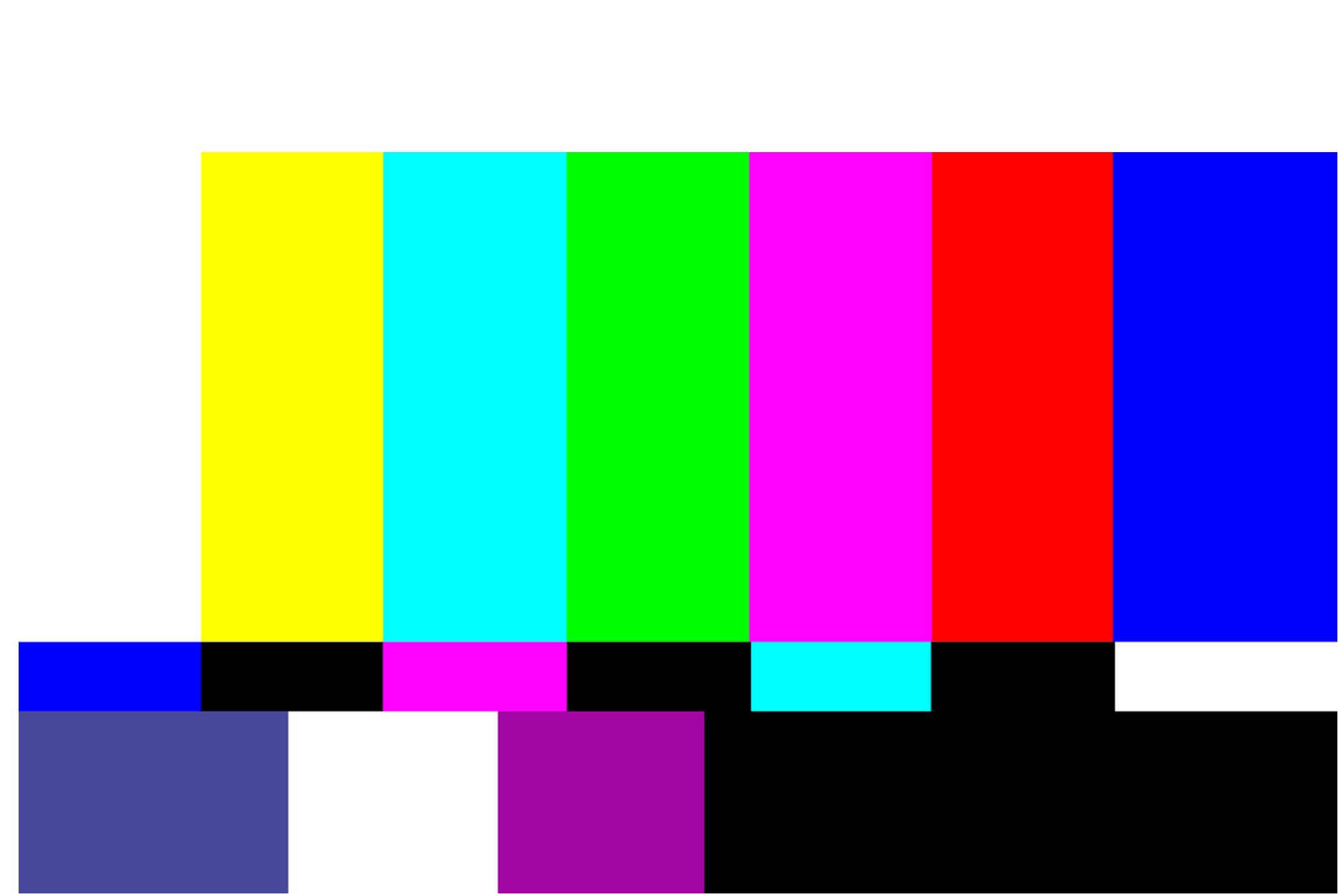
W
UNIVERSITY OF
WASHINGTON



© Andrew Lumsdaine, 2017-2018

Except where otherwise noted, this work is licensed under

<https://creativecommons.org/licenses/by-nc-sa/4.0/>



Example with Input

```
#include <iostream>
#include <string>

int main() {
    std::string contents;
    std::cin >> contents;
    std::cout << contents << std::endl;

    return 0;
}
```

All variables in C++ must be typed!

Variable declaration

Variable type is a std::string

Input Object

Variable name is contents

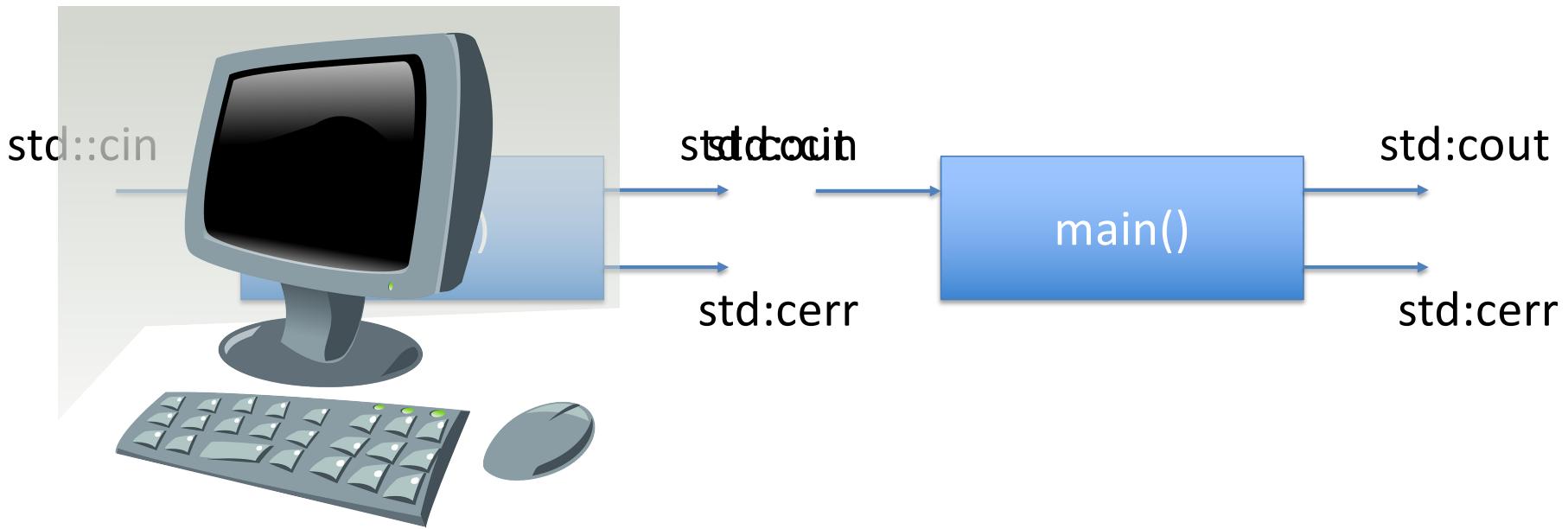
Result

```
$ c++ demo.cpp  
$ ./a.out  
Today is a good day for HPC!  
Today
```

- Explain

Aside (Standard I/O)

- When text is entered into bash, it is accumulated and sent to the program after CR is entered (there are ways to change this: stty)



Example

```
$ wc
```

Word count
(man wc)

```
int main() {
```

Tty input (all the
hello world text)

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

```
}
```

pipe

```
4 12 70
```

4 lines, 12 words,
70 characters

```
$ cat b.cpp | wc
```

4 12 70

Pipe the text from
b.cpp into wc

```
$ wc b.cpp
```

4 12 70 b.cpp

Read contents
from b.cpp

Explanation

- When text is entered into bash, it is accumulated and sent to the program after CR is entered (there are ways to change this: stty)
- This entire string is put into the input stream of the program

Today is a good day for HPC!

- **cin tokenizes the input stream**

```
int main() {  
    std::string contents;  
  
    std::cin >> contents;  
    std::cout << contents << std::endl;  
  
    return 0;  
}
```

Reads first token
only: Today

Today is a good day for HPC!



Prints contents
(first token: Today)

Next Attempt

```
int main() {
    std::string contents;

    std::cin >> contents;
    std::cout << contents << std::endl;

    return 0;
}
```

```
$ c++ demo2.cpp
$ ./a.out
Today is a good day for HPC!
TodayisagoooddayforHPC!
```

- Explain

Yet Another Attempt

```
#include <iostream>
#include <iomanip>

int main() {
    std::string contents;

    std::cin >> contents;
    std::cout << contents << "\n";
    std::cin >> contents;
    std::cout << contents << std::endl;

    return 0;
}
```

\$./a.out

Today is a good day for HPC!
Today is a good day for HPC!

\$./a.out

Today is a good day for
Today is a good day for

\$./a.out

Today is a good day for
Today is a good day for HPC
HPC

Stuck

One more token

Final token

What Else is Wrong?

```
#include <iostream>
#include <iomanip>

int main() {
    std::string contents;
    std::cin >> contents;
    std::cout << contents << "\n";
    std::cin >> contents;
    std::cout << contents << std::endl;
}

return 0;
}
```

```
$ ./a.out
Today is a good day for HPC!
Today is a good day for HPC!
```

NORTHWEST INSTITUTE for ADVANCED COMPUTING

AMATH 483/583 High-Performance Scientific Computing Spring 2019
University of Washington by Andrew Lumsdaine

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

W
UNIVERSITY OF
WASHINGTON

Getting a Line of Input

- Use `std::getline()`

```
#include <iostream>
#include <string>

int main() {
    std::string contents;
    std::getline(std::cin, contents);
    std::cout << contents << std::endl;

    return 0;
}
```

getline()
function

Stream to get
line from

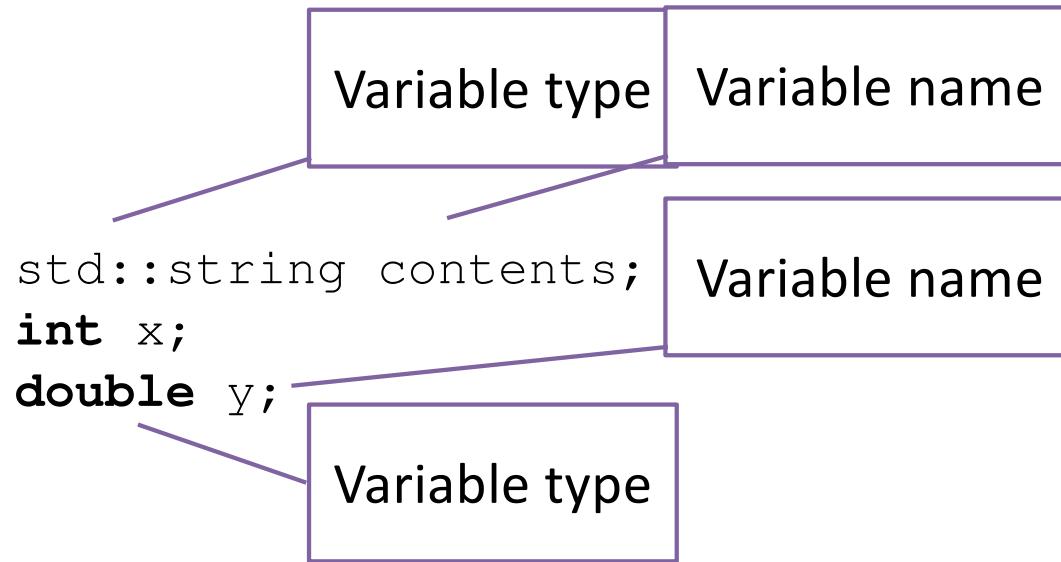
Where to put
the line

```
$ ./a.out
Today is a good day for HPC!
Today is a good day for HPC!
```

- Gets entire line of text, with no tokenization
- Make sure you understand `getline()` vs `>>`

Types

- Variable definition



- C++ has many **built-in** types: int, double, char, etc
- Other types are defined for **libraries** (accessed via #include)
- Almost always **class** definitions

Declaring and Initializing Variables

- In the old days variables were declared at the beginning of a block
- Now they can be defined anywhere in the block

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

Declaration

Use

```
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

More about string

```
std::string s;
```

Declare
empty string

```
std::string t = "Hello_World";
```

Declare string object and
initialize with characters
(Note “Hello World” is not
a C++ string object)

```
std::string u = t;
```

Declare string
and copy from t

```
std::string v = s + t;
```

+ operator concatenates
two string objects

```
int length = v.size();
```

size member function
returns length of string

Example

```
#include <iostream>
#include <string>

int main() {
    std::string msg_1      = "Hello";
    std::string msg_2      = "World";
    std::string message   = msg_1 + " " + msg_2;
    int msg_length        = message.size();

    std::cout << "There are " << msg_length << " characters in ";
    std::cout << "\"" << message << "\"" << std::endl;

    return 0;
}
```