AMATH 483/583
High Performance Scientific Computing

Lecture 3:
Functions, Multiple Compilation, Data Abstraction

Xu Tony Liu, PhD

Paul G. Allen School of Computer Science & Engineering

University of Washington

Seattle, WA

Overview

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

Previous episode

- Hello world in C++
  - #include, comments
- Types and variables
  - Built-in types (int, double, char, etc), other types in libraries, class types
- Namespaces
  - Provide a scope, organize code, prevent name collision
  - Explicitly use variables from namespace std
- Functions and procedural abstraction
  - Parameter passing (by value, by reference, by const reference)

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


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

[P.3: Express intent](#)

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

Abstraction

Procedural

Data type

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 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;
}
```

```
#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;
}
```

Don’t do the same thing twice in different places

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

Compiler can deduce return t

```
#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;
}
```

Note auto is a C++14 feature!

It returns a double

It returns a double

Square root of 2 and 3

Note initialization and declaration of i

What is a size_t?

Pass parameter 2.0

Pass parameter 3.0

```cpp
#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;
  }

  return x;
}


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

“In parameters”

```
#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++

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

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

“Out parameters”

```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

```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 () {

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

      urn 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.
```

Before

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;
}
```

After

Passing **by const reference** (not copied), 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;
}
```

Passing parameter by value vs by reference

- With following declarations:
  - Passing by value
    ```
    double sqrt583(double x);
    ```
  - Passing by reference
    ```
    double sqrt583(double& x);
    ```
  - Passing by const reference (with promise not to modify x)
    ```
    double sqrt583(const double& x);
    ```

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

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

Passing **by const reference** (not copied), promise not to change z

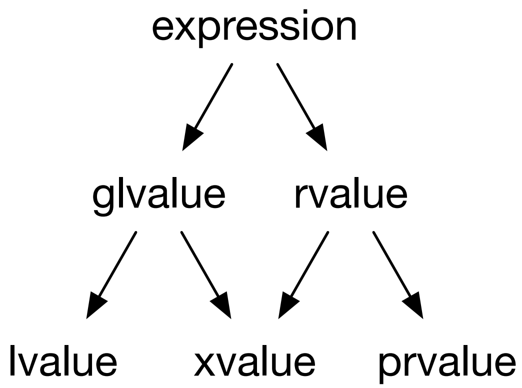
- Which one to choose?
  - What’s the overhead of copying?
  - Does the application need to modify the original value?

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

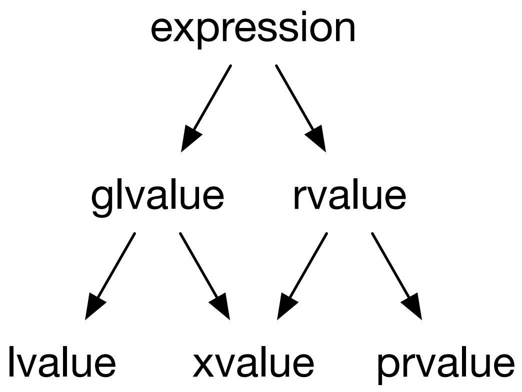
l-values and r-values

- 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



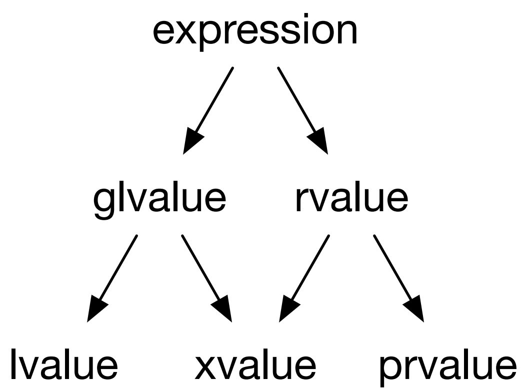
l-values and r-values

```
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.
```



l-values and r-values

- Consider following program:

```
int main() {
  double y = 2.0, z = 3.0;

  double x = sqrt583(y+z);

  return 0;
}
```

rvalue

Copy: x = y + z (lvalue = rvalue)

- With following declarations:

```
double sqrt583(double x);
```

OK to copy rvalue

```
double sqrt583(double& x);
```

Not OK to reference rvalue

```
double sqrt583(const double& x);
```

OK to reference const rvalue

l-values and r-values

- How is the value used in the function

```
double sqrt583(double y) {
  double x = 0.0, dx;

  do {
    dx = - (x*x-y) / (2.0*x);
    x += dx;
  } while (abs(dx) > 1.e-9);

  y = x;

  return x;
}
```

lvalue

lvalue

- With following declarations:

```
double sqrt583(double x);
```

OK to copy rvalue

```
double sqrt583(double& x);
```

Not OK to reference rvalue

```
double sqrt583(const double& x);
```

OK to reference const rvalue

l-values and r-values

- How is the value used in the function

```
double sqrt583(double& y) {
  double x = 0.0, dx;

  do {
    dx = - (x*x-y) / (2.0*x);
    x += dx;
  } while (abs(dx) > 1.e-9);

  y = x;

  return x;
}
```

lvalue ref

lvalue ref

- With following declarations:

```
double sqrt583(double x);
```

Temp result of x+y is references

```
double sqrt583(double& x);
```

Not OK to reference rvalue

```
double sqrt583(const double& x);
```

l-values and r-values

- How is the value used in the function

```
double sqrt583(const double& y) {
  double x = 0.0, dx;

  do {
    dx = - (x*x-y) / (2.0*x);
    x += dx;
  } while (abs(dx) > 1.e-9);

  return x;
}
```

Const lvalue ref

- With following declarations:

```
double sqrt583(double x);
```

```
double sqrt583(double& x);
```

Temp x+y can be referenced if read only

```
double sqrt583(const double& x);
```

OK to reference const rvalue

l-values and r-values

- How is the value used in the function

```
double sqrt583(const double& y) {
  double x = 0.0, dx;

  do {
    dx = - (x*x-y) / (2.0*x);
    x += dx;
  } while (abs(dx) > 1.e-9);

  y = x;

  return x;
}
```

Const lvalue ref

```
$ c++ -c sl11.cpp
error: cannot assign to variable ’y’ with
const-qualified type ’const double &’
  y = x;
  ~ ^
sl11.cpp:3:30: note: variable ’y’ declared const here
double sqrt583(const double& y) {
               ~~~~~~~~~~~~~~~^
1 error generated.
```

Reusing functions

```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;
  }

  return x;
}


int main () {

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


  return 0;
}
```

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

Compile main.cpp

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

Translate it into a language the cpu can run

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

The executable (program that the cpu can run)

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

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

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

Translate it into a language the cpu can run

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

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
```

```cpp
#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

```
#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
```

```
$ 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
```

```
#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

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

Implement all functions in amath583.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?

Program (file) organization (in pictures)

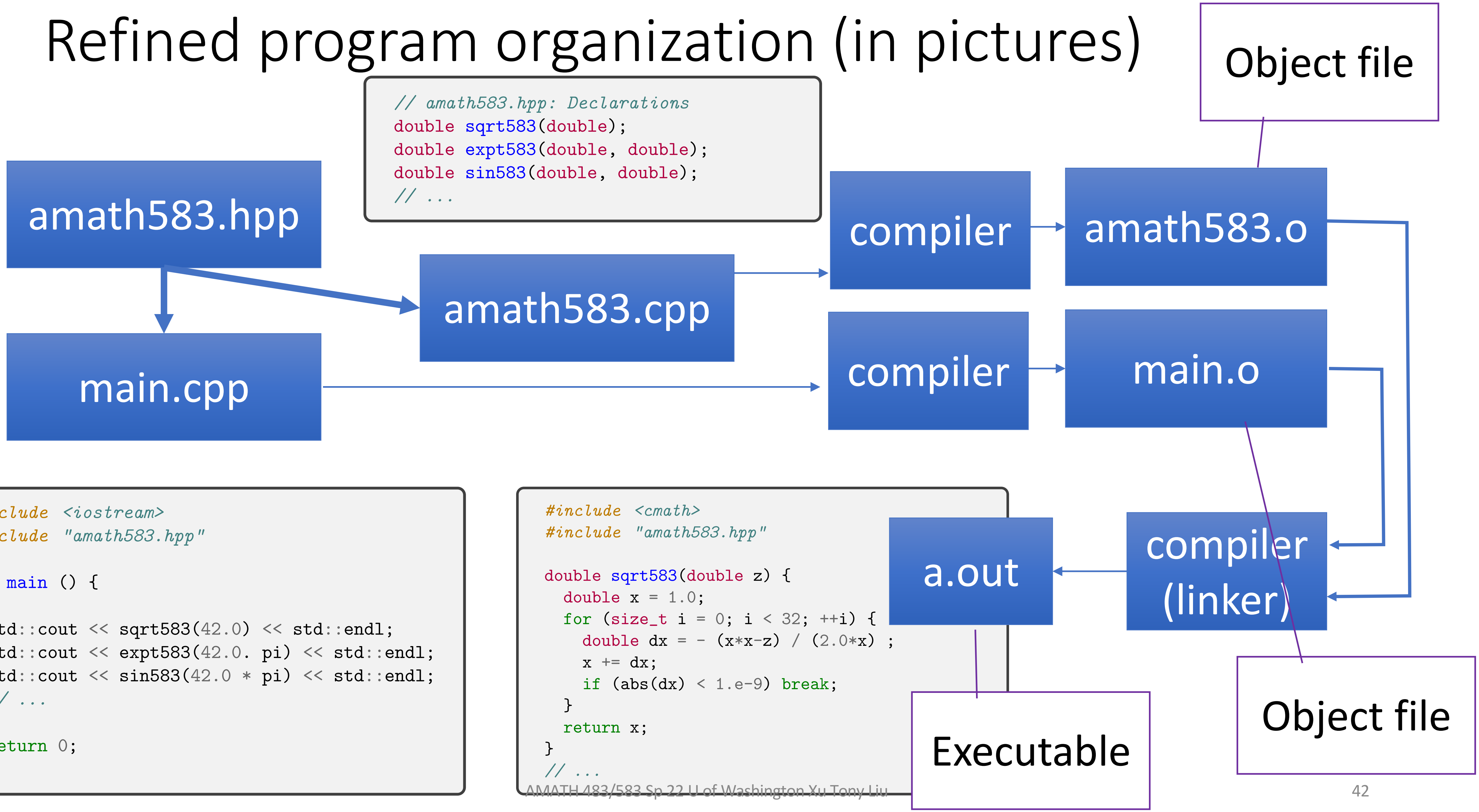
```cpp
// amath583.hpp: Declarations
double sqrt583(double);
double expt583(double, double);
double sin583(double, double);
// ...
```


```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)

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




```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;
}
// ...
```

Multifile Multistage Compilation

Compile main.cpp to main.o object file

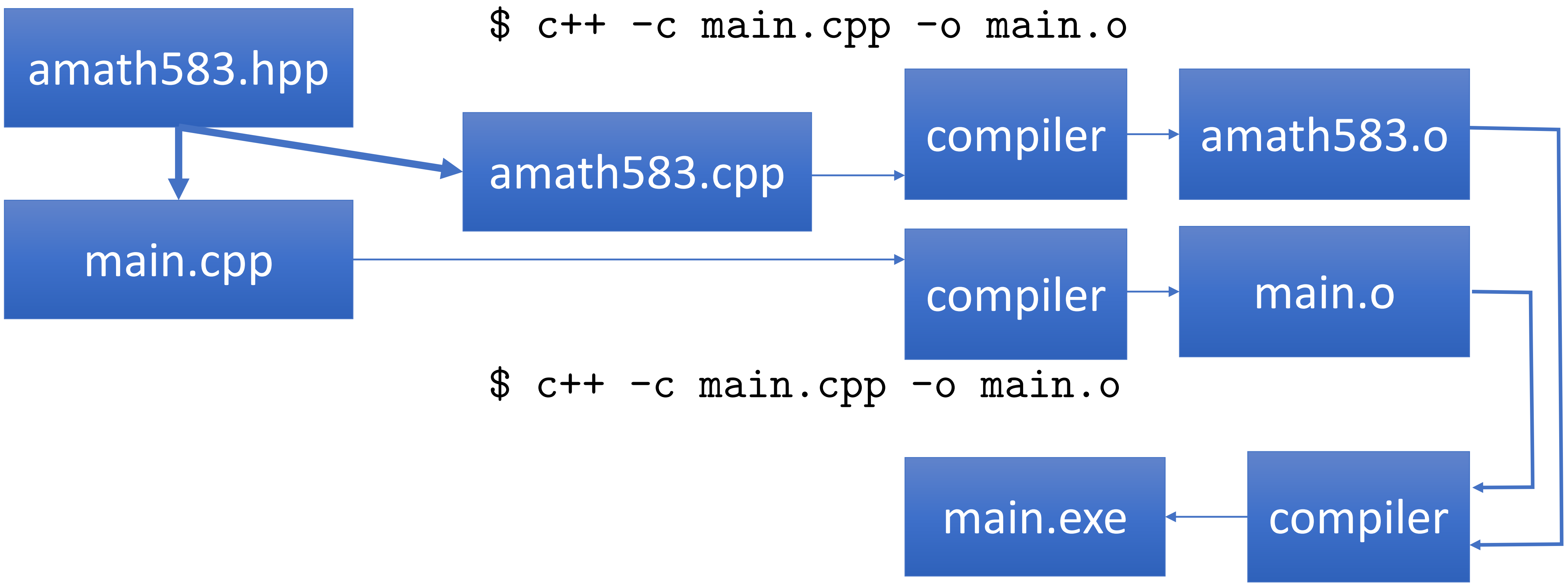
Tell the compiler to generate object

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

Tell the compiler name of the object

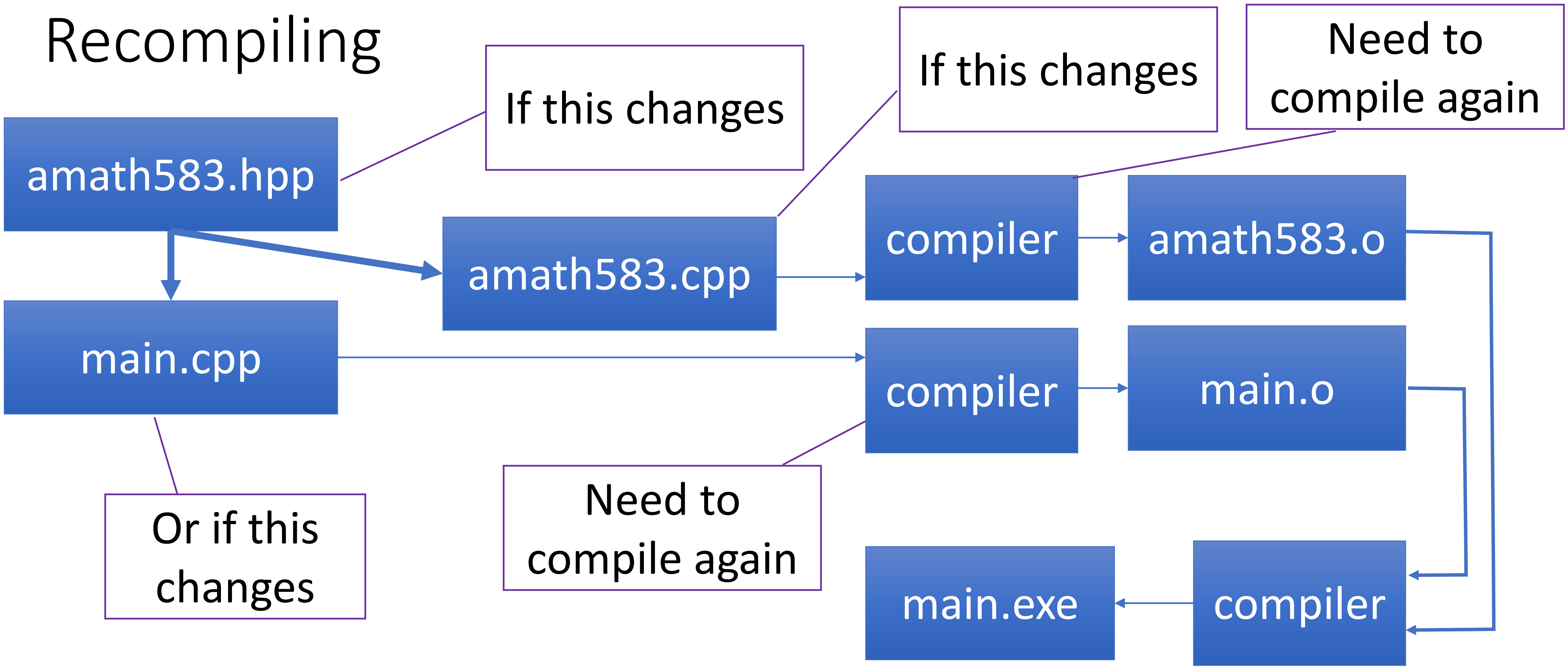
```
$ c++ -c amath583.cpp -o amath583.o

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

Multistage compilation (pictorially)




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

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
```

Dependencies

Consequent

Target

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
```

Computational Science

System of Partial Differential Eqns

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

Find P that satisfies this

(too hard)

discretize

System of Nonlinear Eqns

$$F(x) = 0$$

Find x that satisfies this

(too hard)

linearize

System of Linear Eqns

$$Ax = b$$

Find x that satisfies this

A problem we can solve

Computational Science

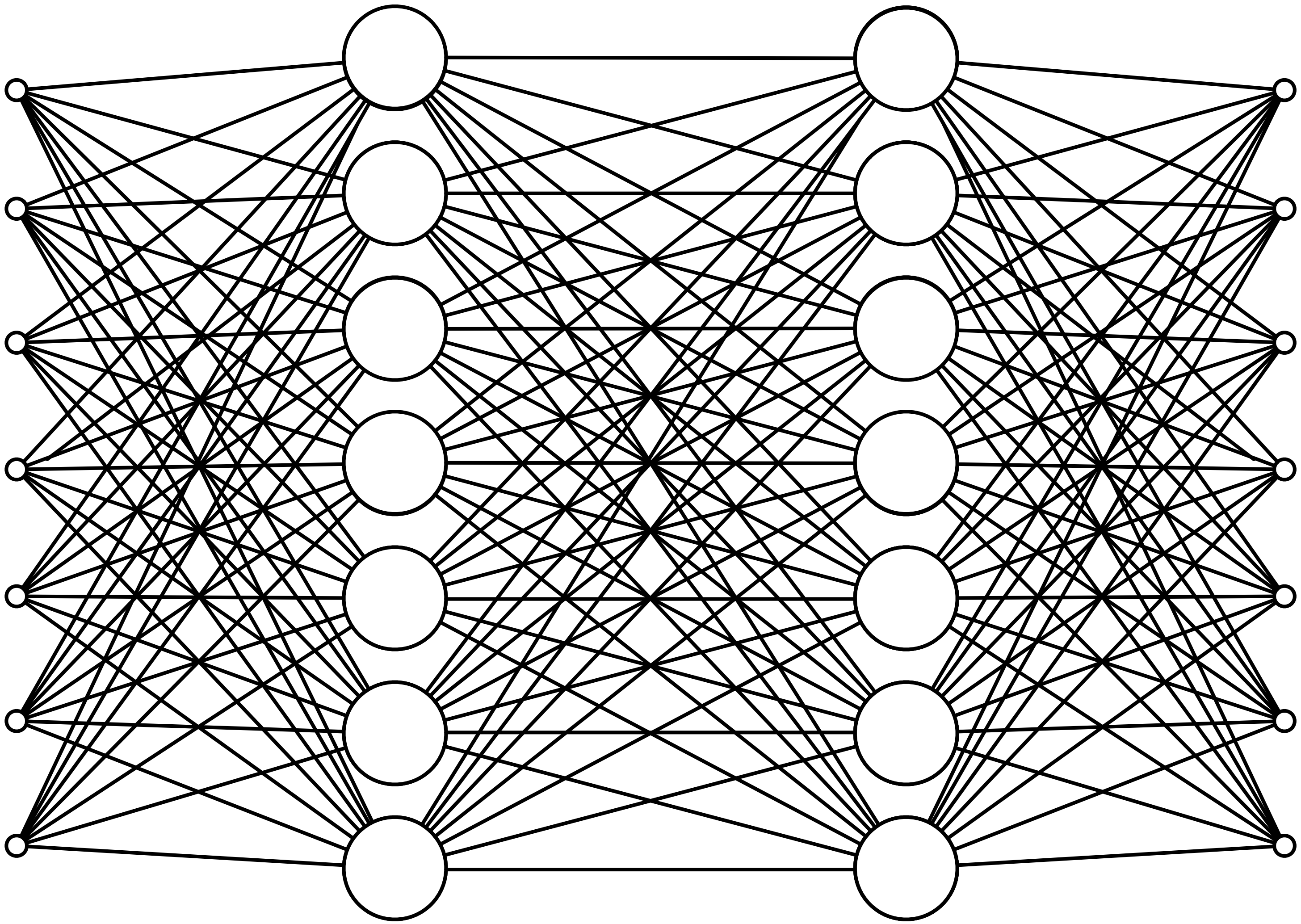
- The fundamental computation at the core of many (most/all) computational science programs is solving $Ax = b$

- Assume $x, b \in R^N$ and $A \in R^{N \times N}$

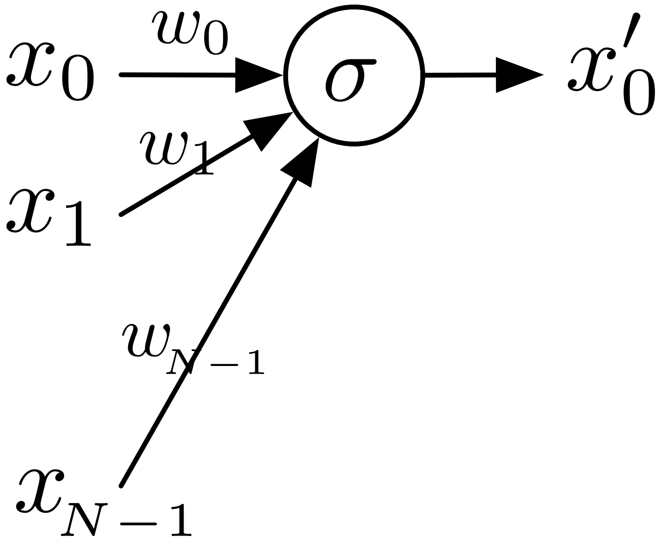
- i.e., x and b are vectors with N real elements and A is a matrix with N by N real elements

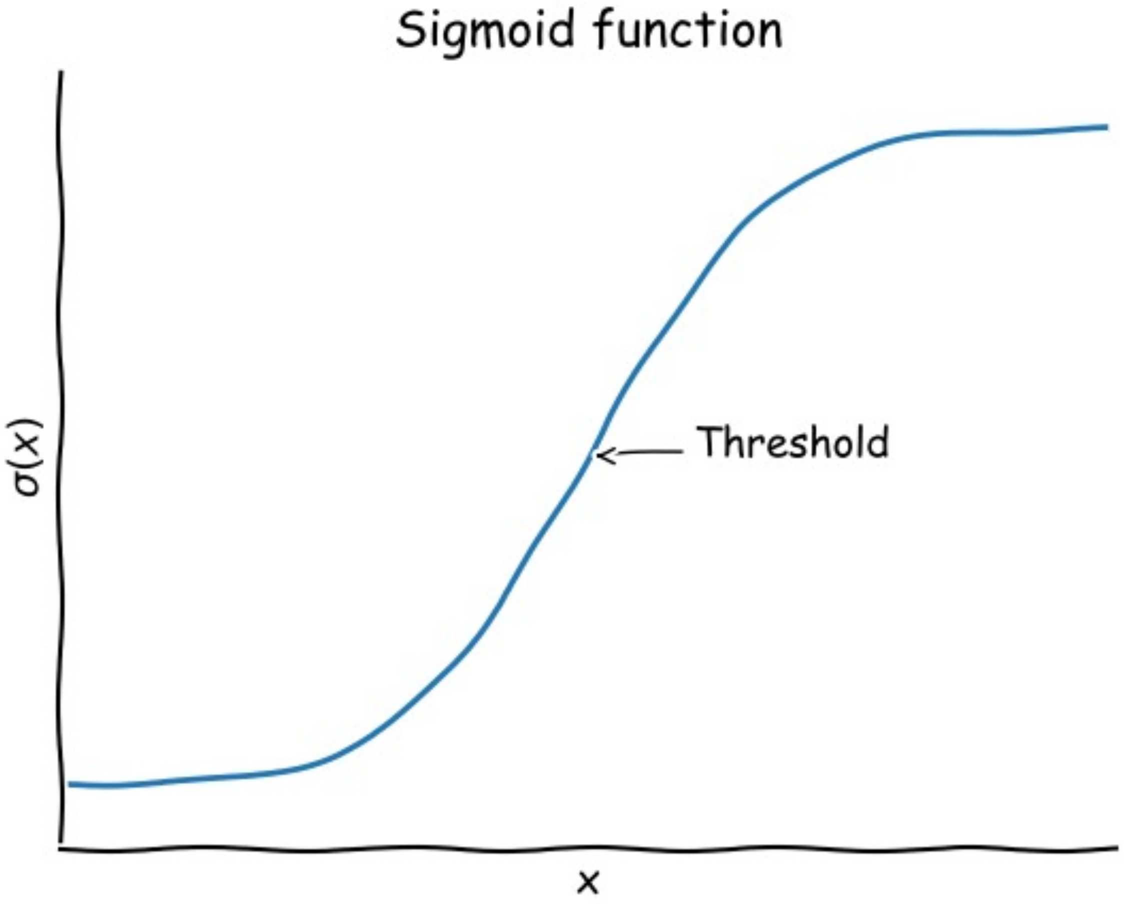
- Solution process only requires basic arithmetic operations

Neural Network

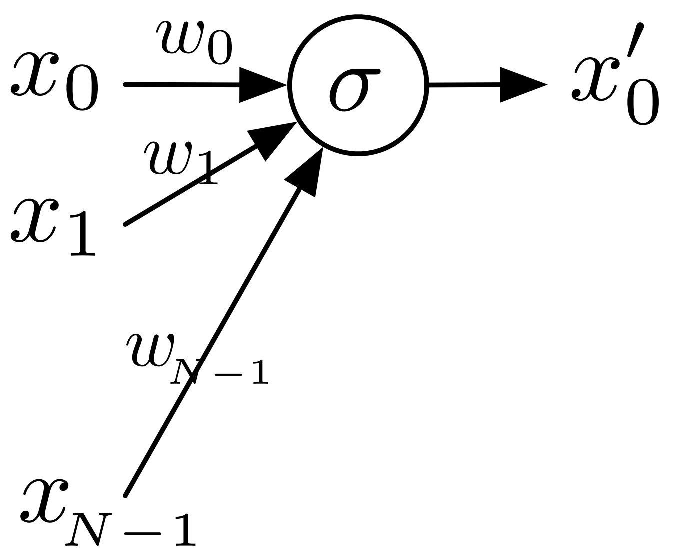


Zoom In On One "Neuron"





Zoom In On One "Neuron"

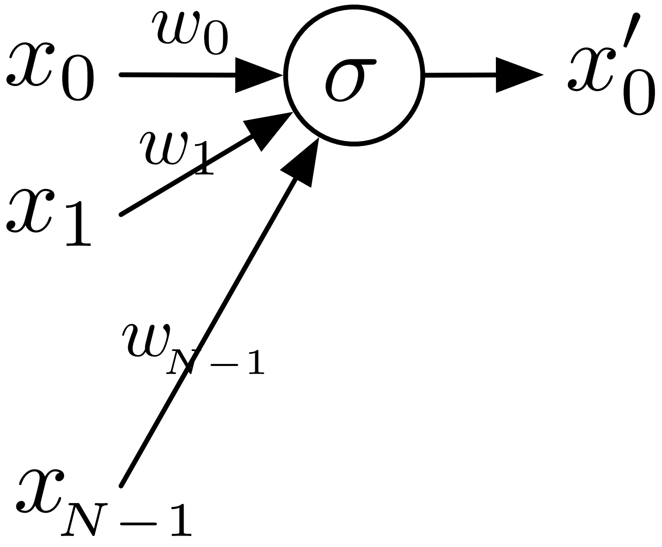


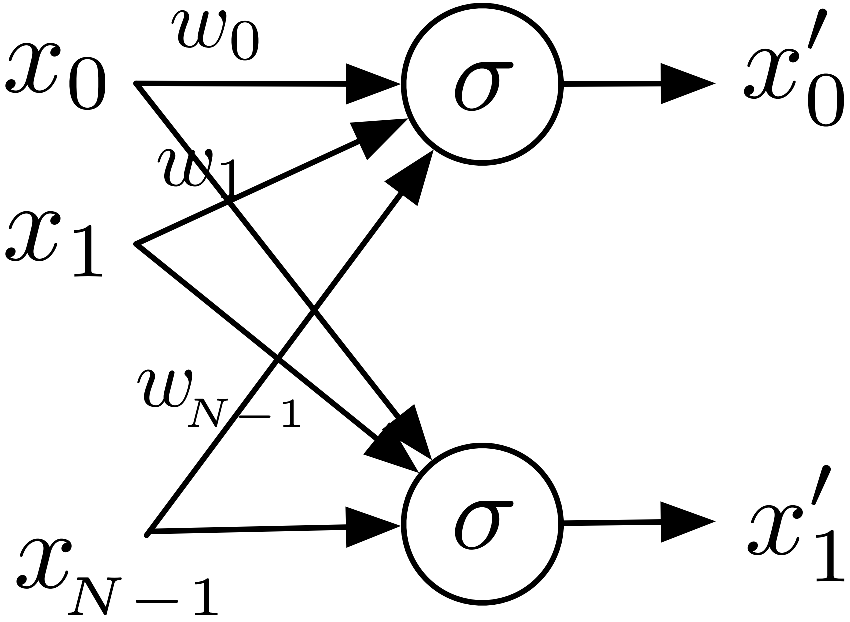
$$x_0' = \sigma(t)$$

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

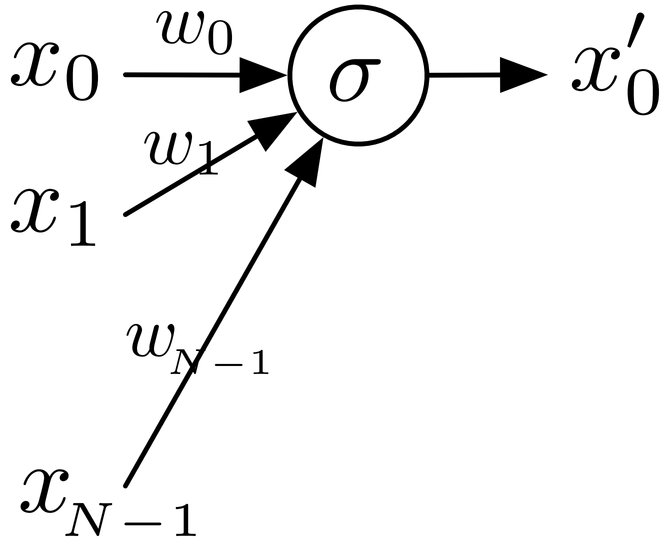
$$x_0' = \sigma(\sum_{i=0}^{N-1} w_i x_i)$$

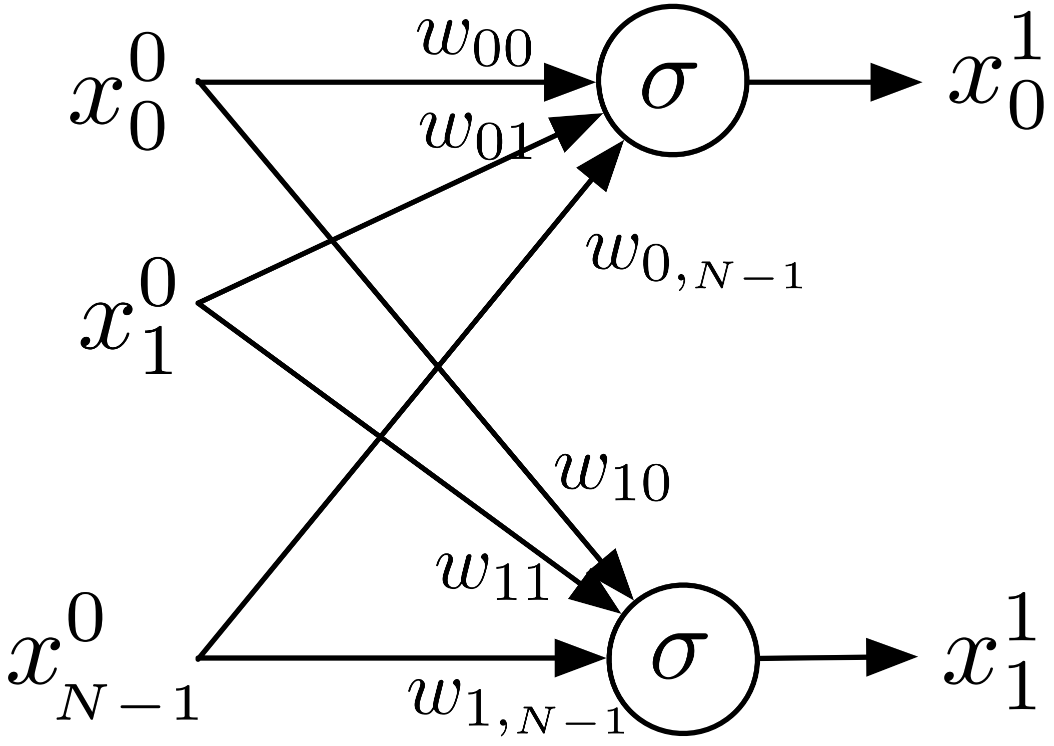
Zoom In On Two "Neurons"



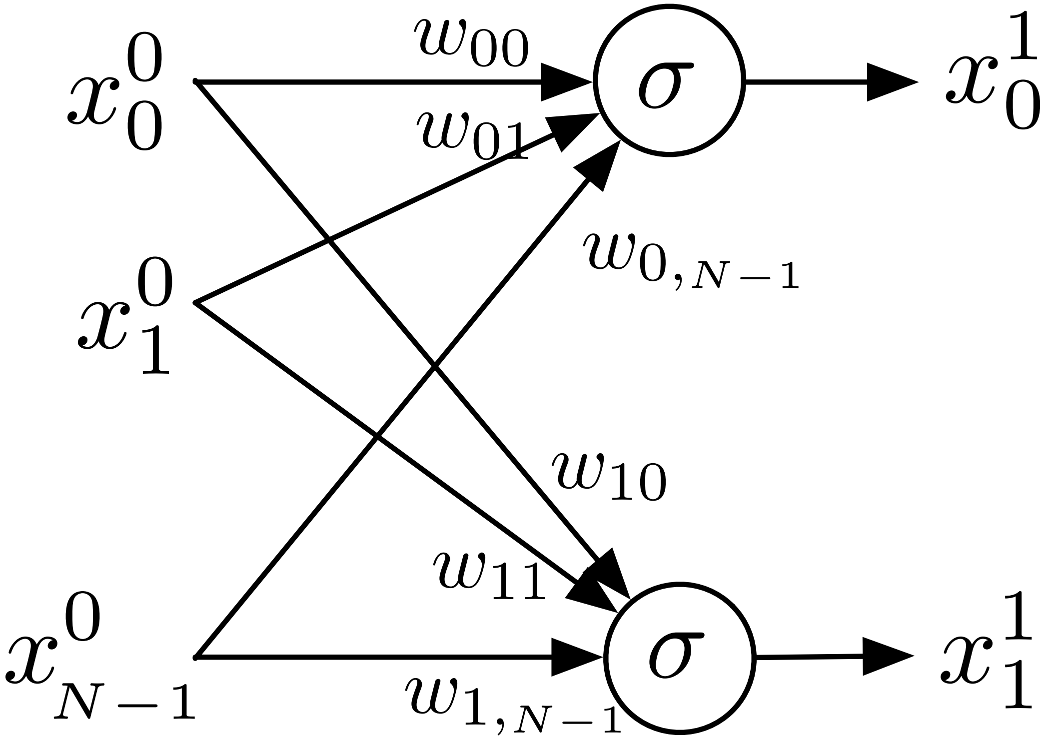


Zoom In On Two “Neurons”





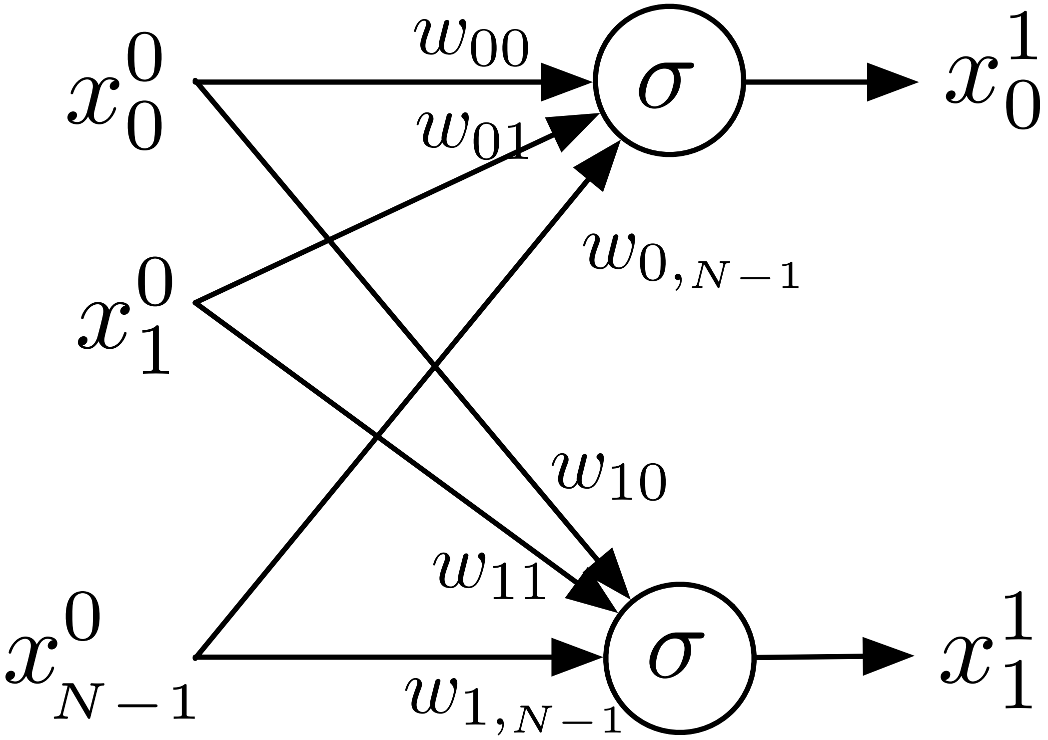
Zoom In On Two "Neurons"



$$x_0^1 = \sigma(\sum_{i=0}^{N-1} w_0 i x_i^0)$$

$$x_1^1 = \sigma(\sum_{i=0}^{N-1} w_1 i x_i^0)$$

Zoom In On Two "Neurons"



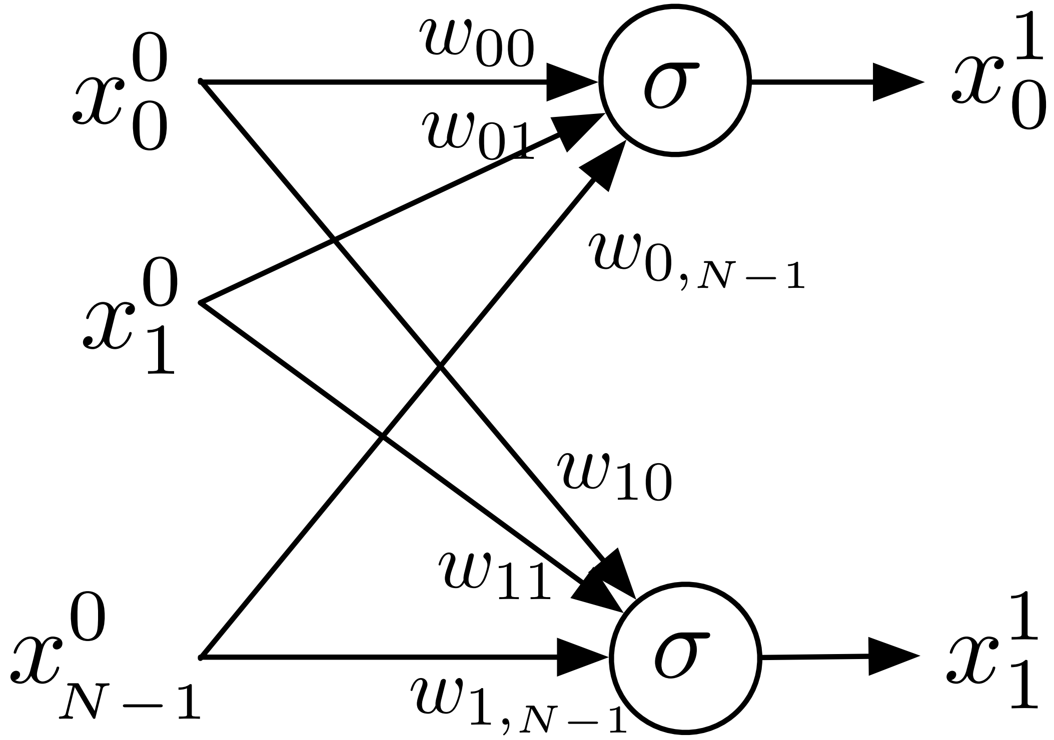
$$x_0^1 = \sigma(\sum_{i=0}^{N-1} w_{0i} x_i^0)$$

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

$$\vdots$$

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

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
   (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 ever 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 vetors is distributive with respect to scalar addition $(a + b)x = ax + by$

commutative

associative

We need to be able to add 2 vectors → vector

Identity over +

Identity over x

associative

distributive

distributive

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 ever 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 vetors 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
   (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 ever 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 vetors is distributive with respect to scalar addition $(a + b)x = ax + by$

commutative

associative

We need to be able to add 2 vectors → vector

Identity over +

Identity over x

associative

distributive

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)
```

- 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];
```

Two different functions

For same operation

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 class and Vector class

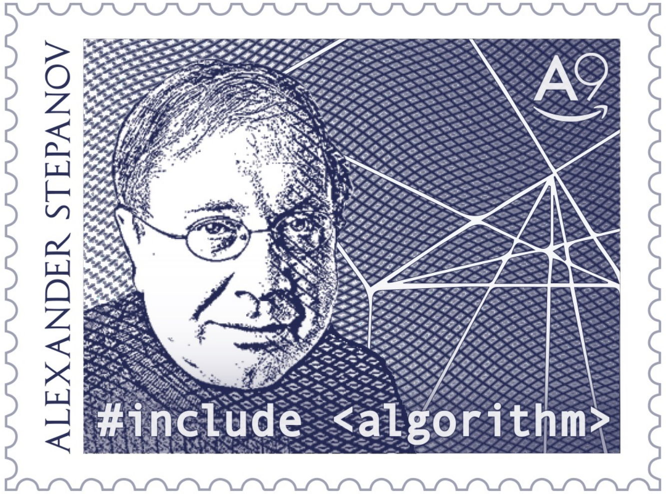
- C++ classes enable encapsulation of related data and functions
- Provide visible interfaces
- Hide implementation details

std::vector<double>

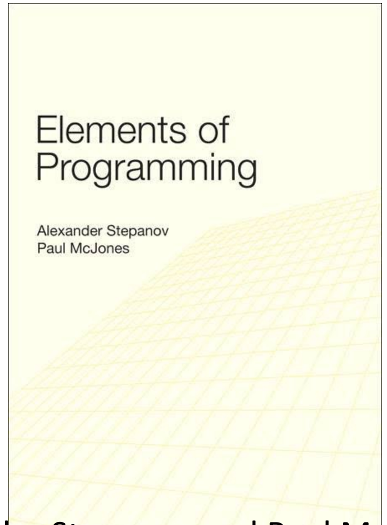
- Before rushing off to implement fancy interfaces
- Understand what we are working with
- And how hardware and software interact
- std::vector<double> will be our storage
- But its interface won't be our interface
    - We will gradually build up to complete Vector
    - And complete Matrix

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::for_each`
`std::list`	ForwardIterator	`std::sort`
`std::map`	ReverseIterator	`std::accumulate`
`std::vector`	RandomAccessIterator	`std::min_element`
`...`		`...`



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

Standard Library container

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

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

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

Standard Library container

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

iterator

iterator

Initial value

std Containers

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

```
template <typename T> class vector;
template <typename T> class dequeue;
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

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

```
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

Declares an interface

Hides implementation

```
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_;
};
```

Anatomy of a C+

```
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

Objects are variables of that type

And hides implementation

Interface specifies how to use objects

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 to what?

Store the size of the Vector

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

And when?

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

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

The constructor is a function

And it has a body

```
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_;
};
```

One option for initialization

```cpp
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 functi

```
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

Interface in Vector.hpp

```
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_;
};
```

Function declaration (implementation elsewhere)

Implementation in Vector.cpp

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

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

http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Thank you!