# AMATH 483/583
# High Performance Scientific Computing

## Lecture 4:
## Data Abstraction, Classes and Objects, class Vector

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

# Overview

- Recap of Lecture 3
  - Compilation
  - Program organization
  - Header files, source files
  - make
- class Vector

# HPC in the News

Cal Tech Assistant Professor (and former MIT CS PhD student) Katie Bouman with 5PB data used to image the black hole.

# 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


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

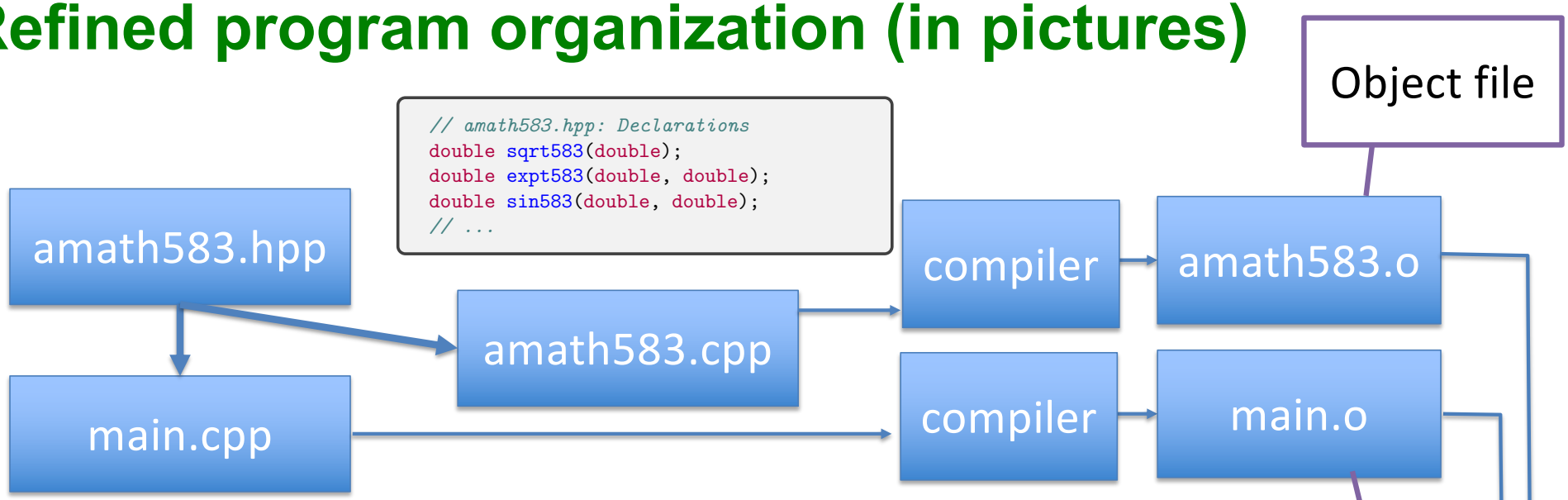
Team Meetings
Mondays 5:30PM-8:00PM

# Procedural Abstraction: 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

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

# Refined program organization (in pictures)




```hpp
// 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

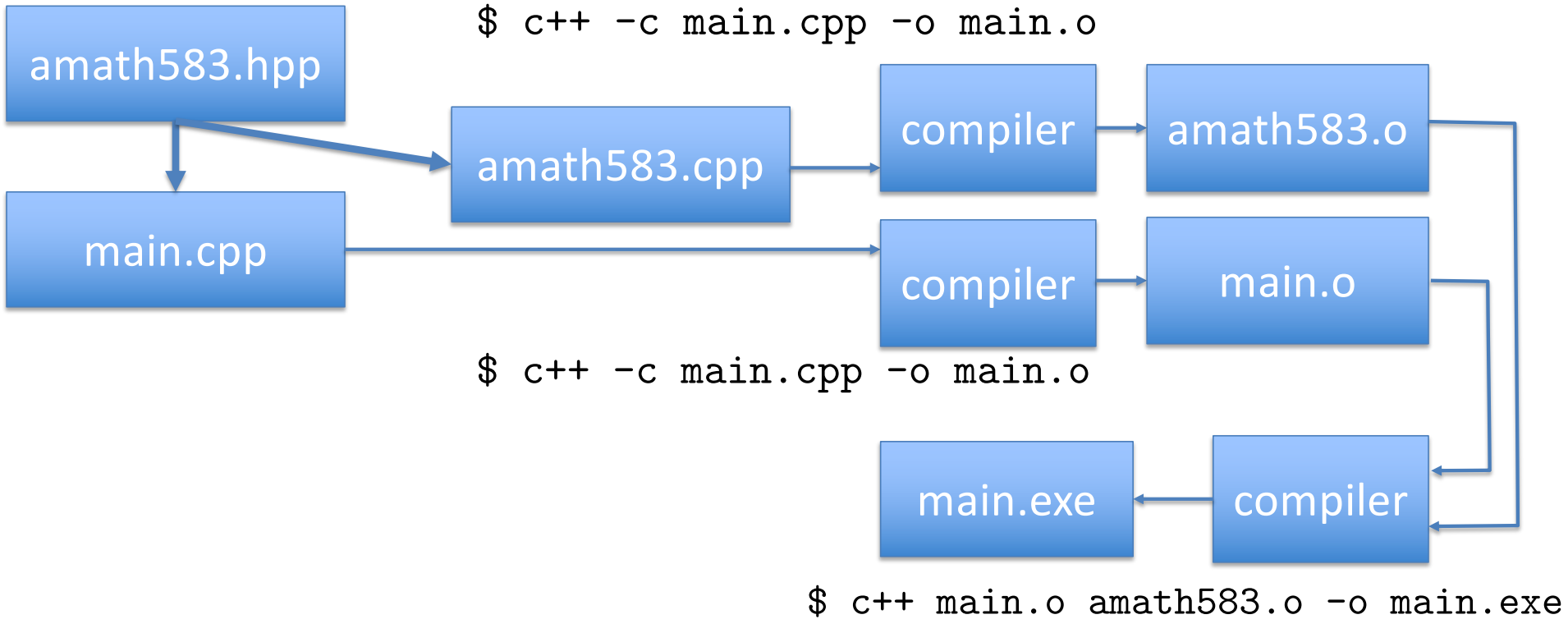
Tell the compiler to generate object

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

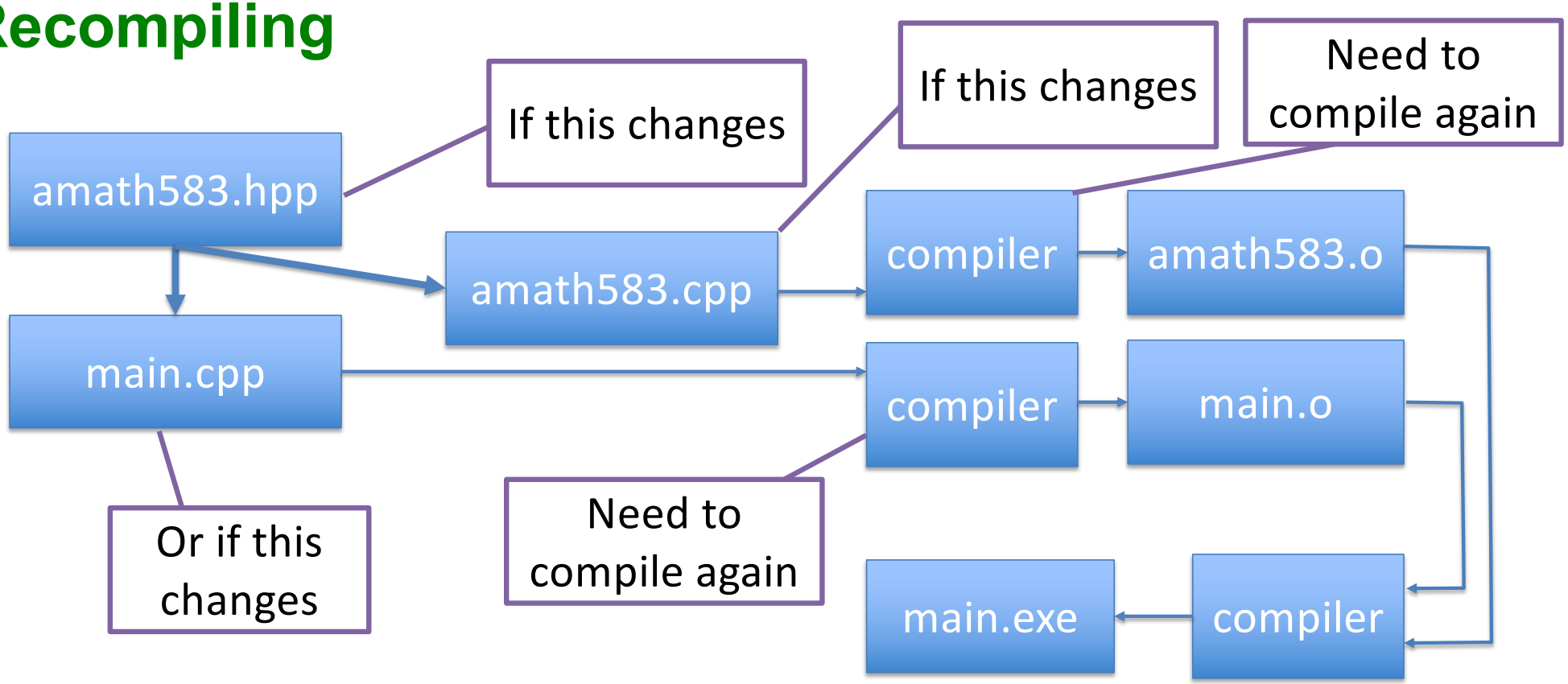
Tell the compiler name of the object

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

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

# Multistage compilation (pictorially)



# 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 → recompile main.exe
- If amath583.o is newer than main.exe → recompile main.exe
- If main.cpp is newer than main.o → recompile main.o
- If amath583.cpp is newer than amath583.o → recompile amath583.o
- If amath583.hpp is newer than main.o → recompile main.o

# 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

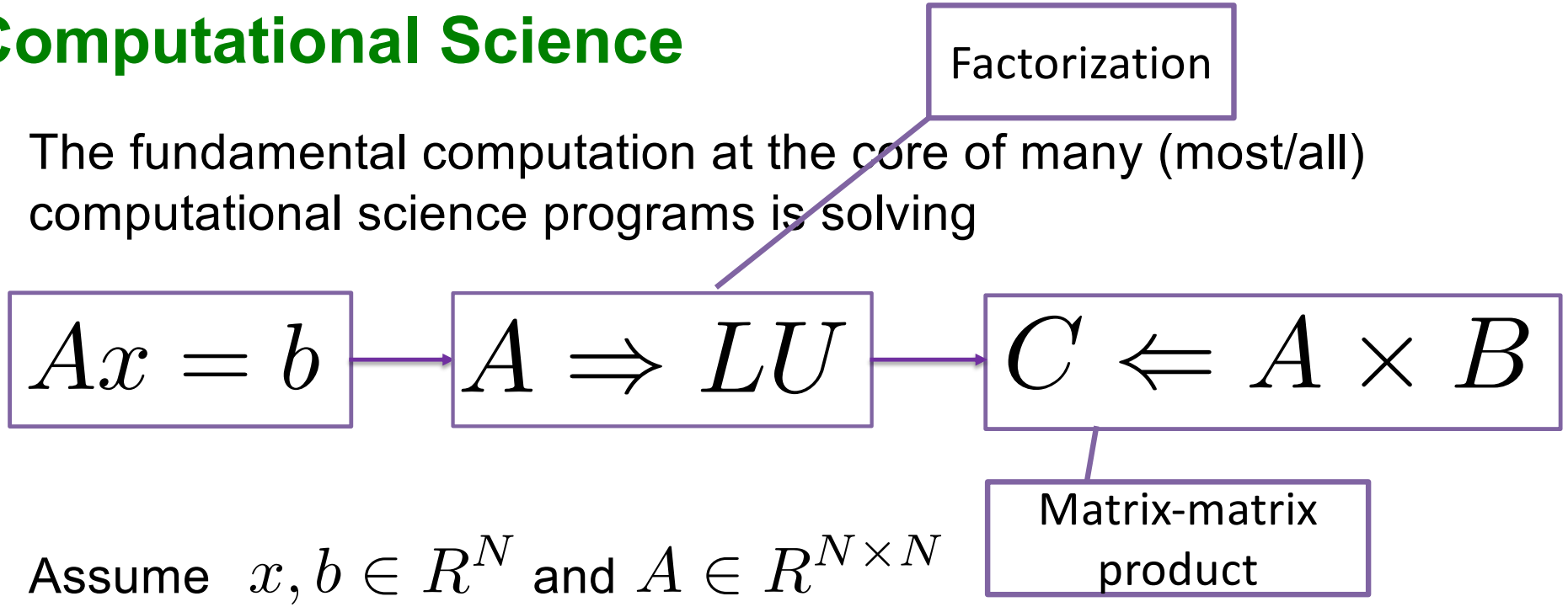
$$Ax = b$$

Find x that satisfies this

A problem we can solve

# Computational Science

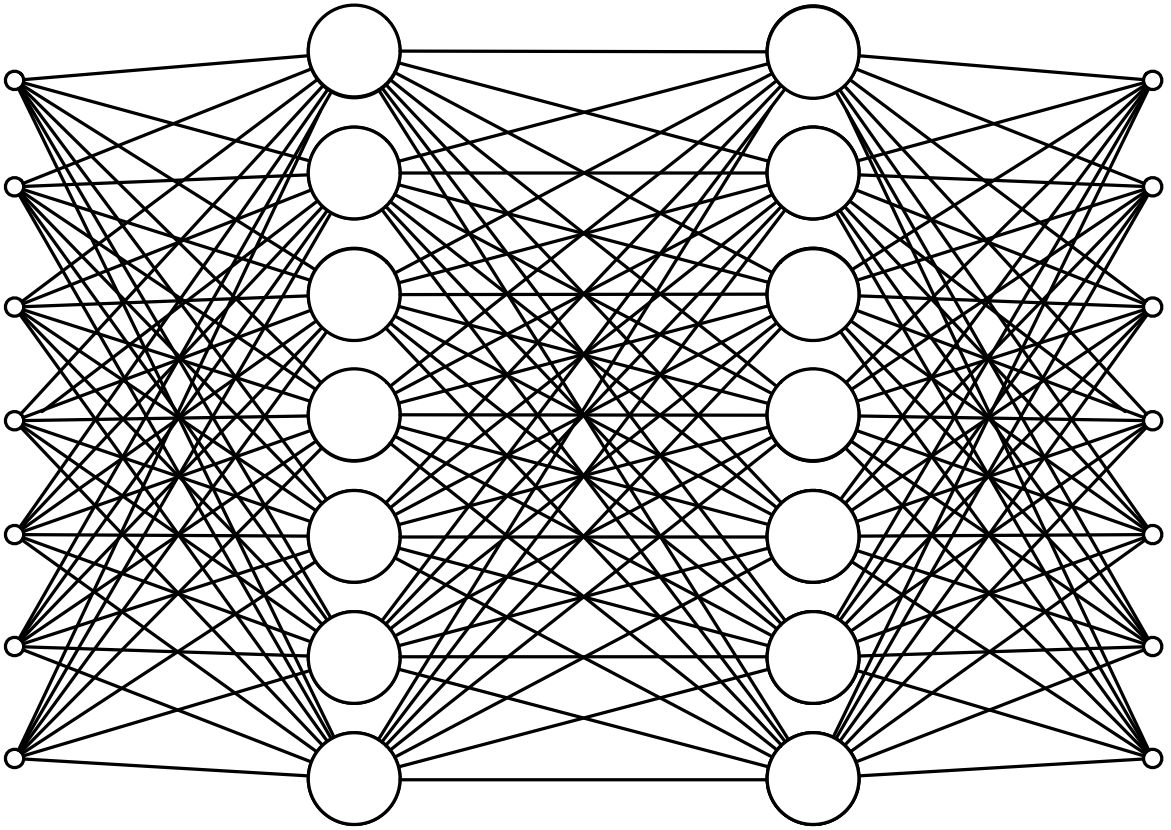
- The fundamental computation at the core of many (most/all) computational science programs is solving



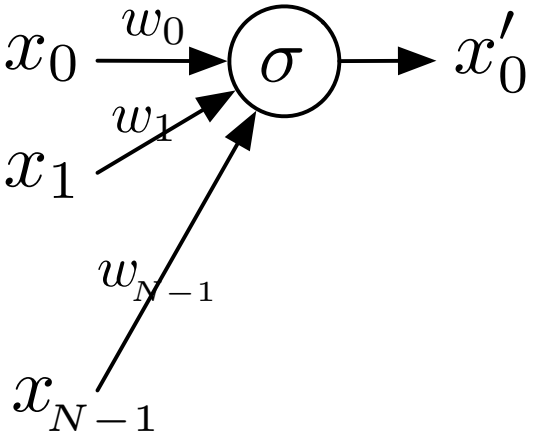

- Assume $x, b \in R^N$ and $A \in R^{N \times N}$

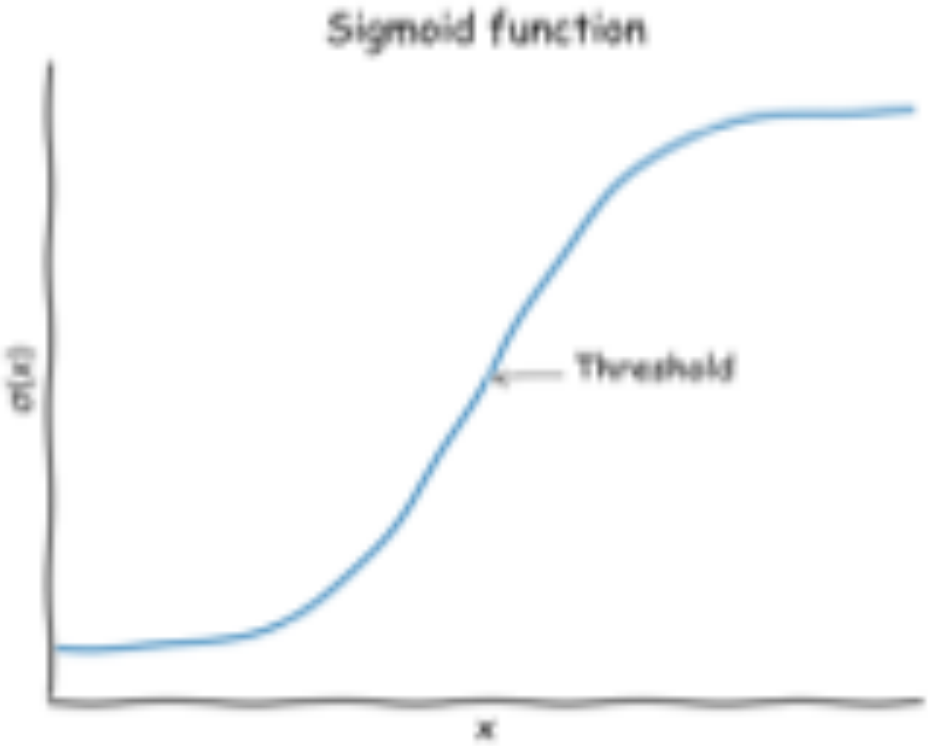
- Solution process only requires basic arithmetic operations

# Neural Network



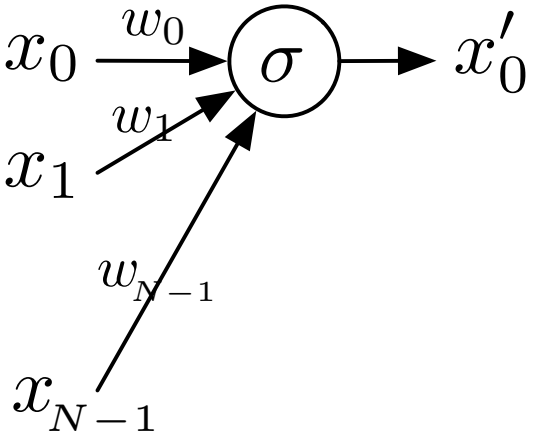
# Zoom In On One ”Neuron”



# Zoom In On One "Neuron"

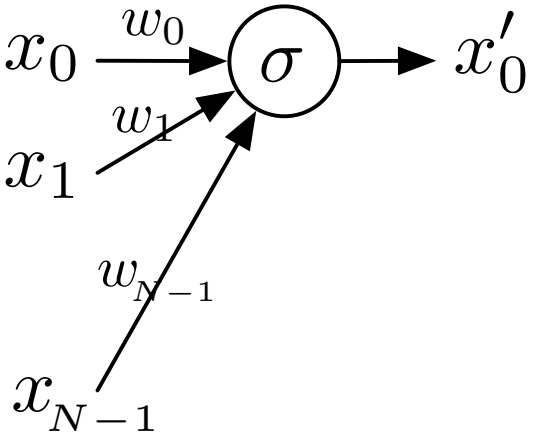
$$x_0' = \sigma(t)$$

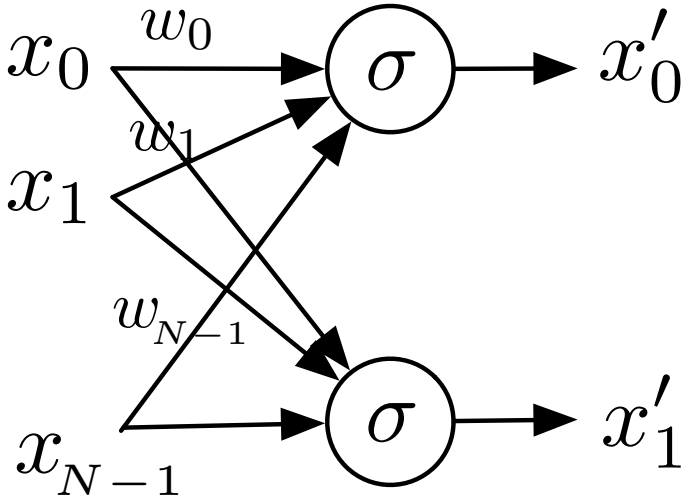



$$t = w_0 x_0 + w_1 x_1 + \cdots + w_{n-1} x_{n-1}$$
$$= \sum_{i=0}^{N-1} w_i x_i$$

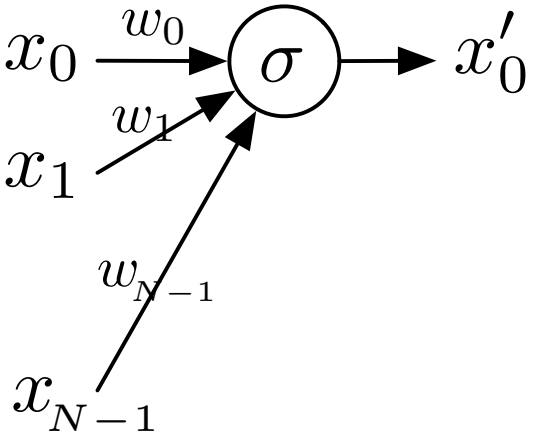
$$x_0' = \sigma\left(\sum_{i=0}^{N-1} w_i x_i\right)$$

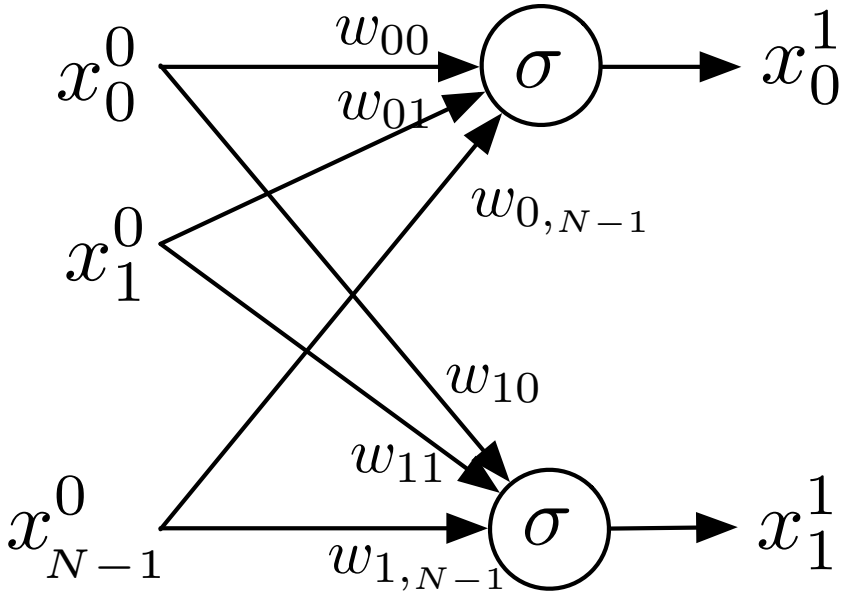
# Zoom In On Two "Neurons"



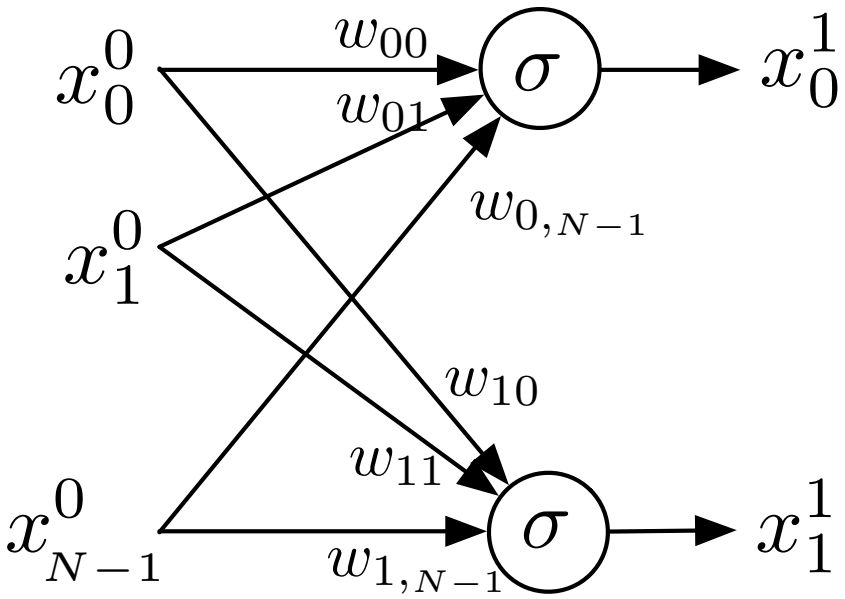



# Zoom In On Two "Neurons"



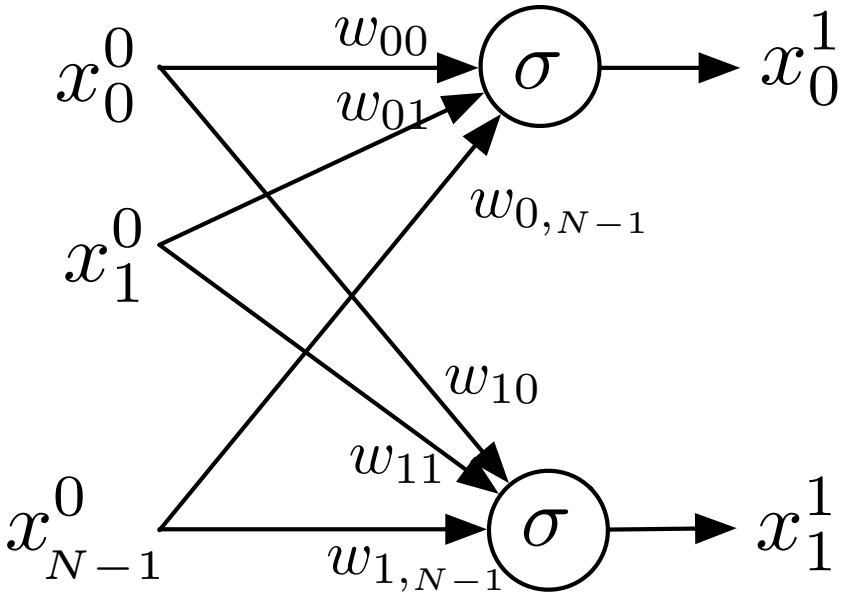
# Zoom In On Two "Neurons"




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

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

# Zoom In On Two "Neurons"



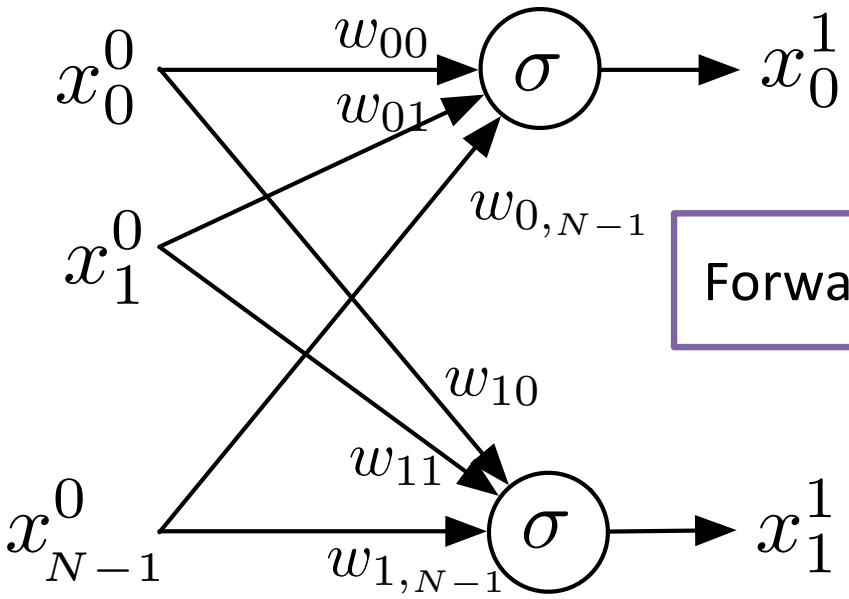

$$x_0^1 = \sigma\left(\sum_{i=0}^{N-1} w_{0i} x_i^0\right)$$

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

$$\vdots$$

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

# Zoom In On Two "Neurons"




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

Forward

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

vector | matrix | vector

# Zoom In On Two "Neurons"

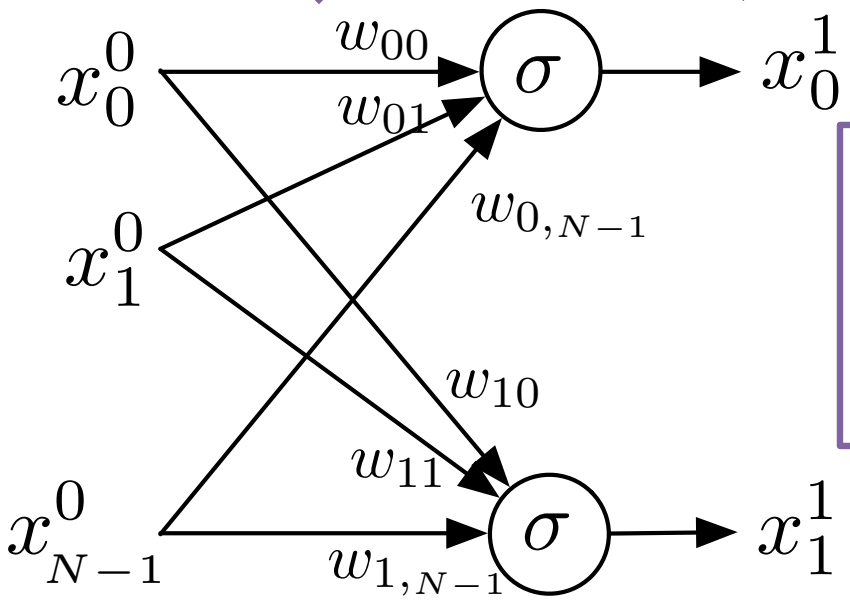
When this changes?

How does this change

More later

**Back Propagation**

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




Will still be based on vector and matrix abstractions and operations

We want to find "best" W

Need to compute gradient

# 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

# 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

**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, $x + y = y + x$
   e, $x + (y + z) = (x + y) + z$
   nique vector 0 (called the origin) such that $x + 0 = x$ for ever vector $x$, and
   $V$ 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
   scalars is associative $a(bx) = (ab)x$, and
   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

C++ does not have an n-tuple type with these properties

Identity over +

Create our own

Identity over x

associative

distributive

# Classes

- First principles: Abstraction, simplicity, consistent specification
- Domain: Scientific computing
- Domain abstractions: Matrices and vectors
- Programming abstractions: Matrix and Vector

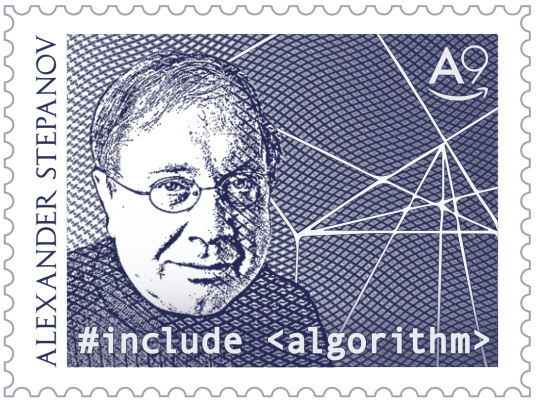
- C++ classes enable encapsulation of related data and functions
- User-defined types
- Provides visible interface
- Hides implementation

# std::vector<double>

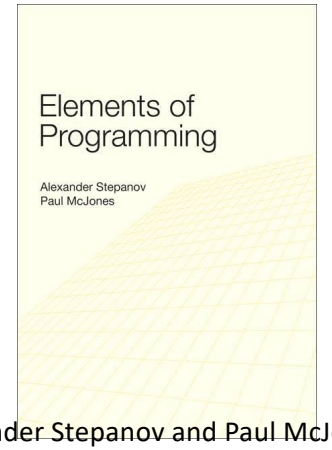
- Before rushing off to implement fancy interfaces
- Understand what we are working with
- And how hardware and software interact
- std::vector<double> will be our storage
- But its interface won't be our interface
  - Doesn't have associated arithmetic operations
  - We will gradually build up to complete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



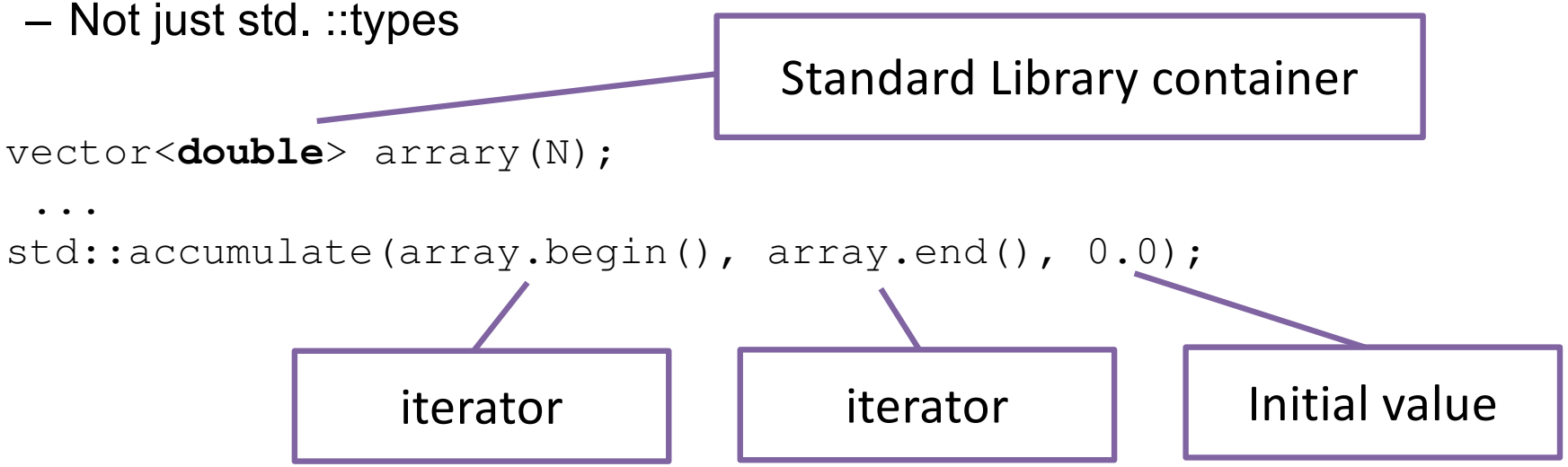

| | | |
|---|---|---|
| `std::set` | | `std::for_each` |
| `std::list` | ForwardIterator | `std::sort` |
| `std::map` | ReverseIterator | `std::accumulate` |
| `std::vector` | RandomAccessIterator | `std::min_element` |
| ... | | ... |


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



# std Containers

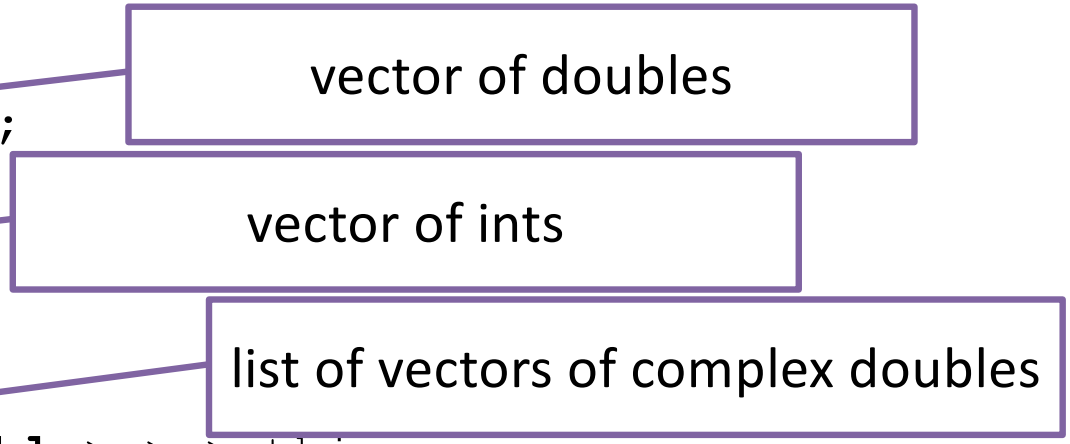
- Note that all containers have ***same*** interface
- (Actually a hierarchy, we'll come back to this)
- We will primarily be focusing on vector

| 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

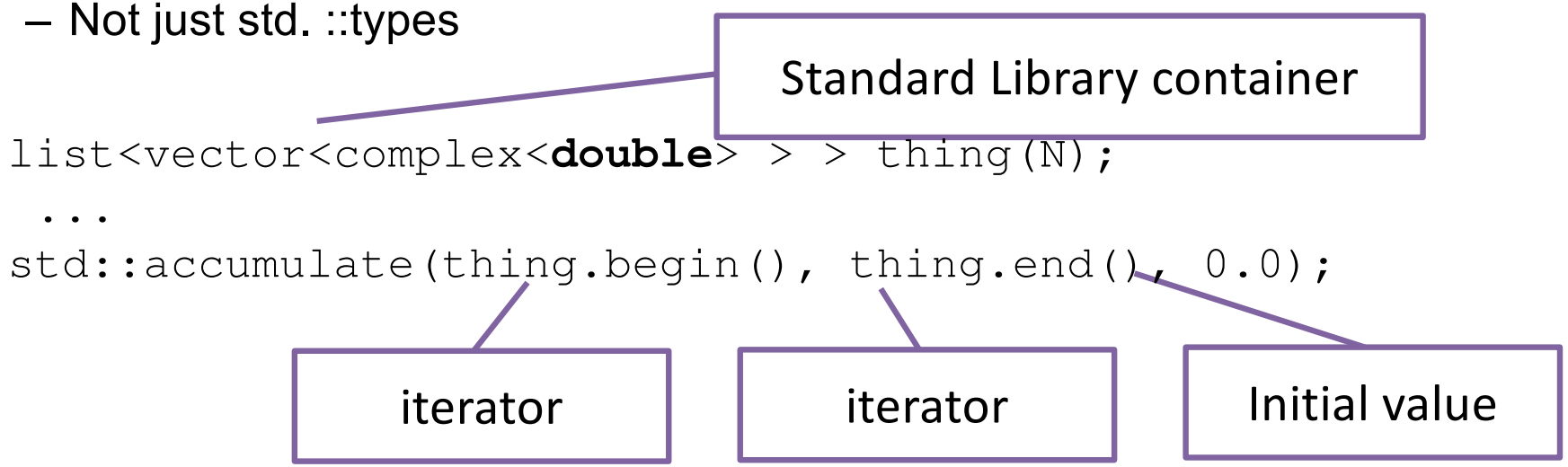
```
vector<double> array(N);

vector<int> array(N);

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




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

# Generic Programming

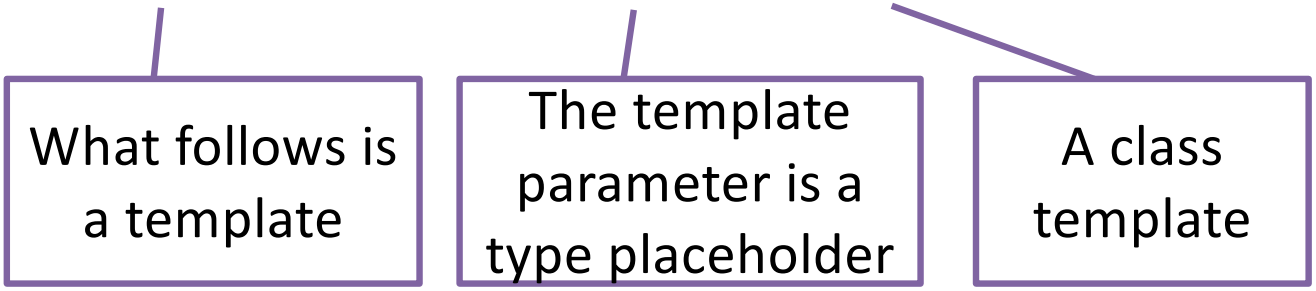
- Algorithms are ***generic*** (parametrically polymorphic)
- Algorithms can be used on ***any*** type that meets algorithmic reqts
  - Valid expressions, associated types
  - Not just std. ::types



# 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;
```




- 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

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

```
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
- User-defined types
- 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”

Declare (construct) a Vector with num_rows elements

```
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;
}
```

Access elements with subscript (index)

# Anatomy of a C++ class

Declares an interface

Hides implementation

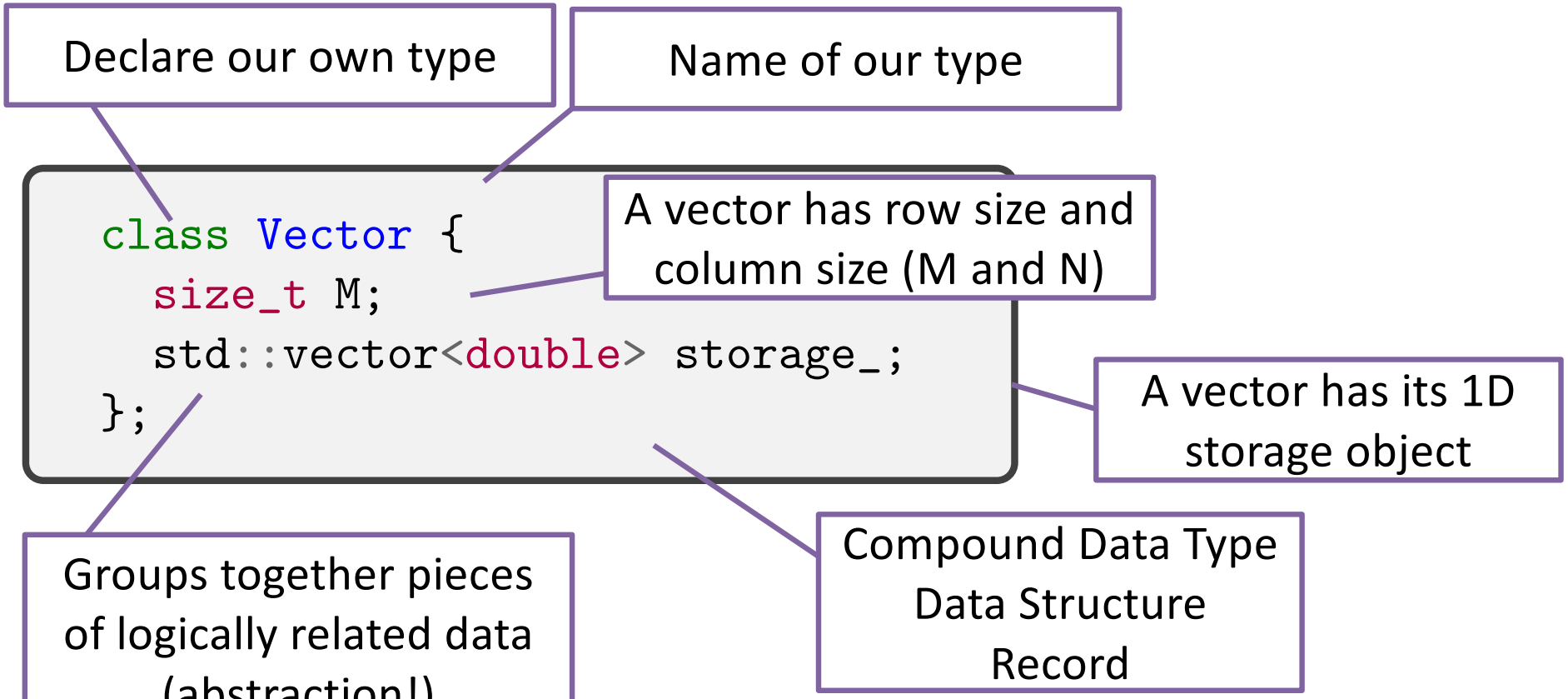
```
class Vector {
public:
  Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

  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_;
};
```

# 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

# Anatomy of Classes and Structs in C++



# Anatomy of Classes and Stru

A class is a formula for what an object will be

If I declare something to be of type Vector, I have ***instantiated*** an ***object*** of type Vector

A vector has a number of rows (M)

```
class Vector {
  size_t M;
  std::vector<double> storage_;
};
```

```
Vector A;   size_t M;
            std::vector<double> storage_;
```

Each Vector contains its ***own*** data: its own M and its own storage_

```
Vector B;   size_t M;
            std::vector<double> storage_;
```

A vector has its 1D storage object

Any Vector has its size and data bound together as a single entity (***object***)

Each Vector contains its own data: M, and storage_

# Classes and Structs in C++ (Usage)

```
class Vector {
  size_t M;
  std::vector<double> storage_;
};
```

```
Vector x;
```
size_t M;
std::vector<double> storage_;

```
Vector y;
```
size_t M;
std::vector<double> storage_;

Dot means evaluate the M belonging to x

Write to it

Acts just like a size_t

```
size_t foo = x.M;
```

```
size_t foo = x.M;
y.M = 42;
x.storage_[27] = 3.14;
```

Vector object x

Data Member M

Read from it

# Aside (Hygiene)

```
#include <vector>

class Vector {
  size_t M;
  std::vector<double> storage_;
};
```

Include declarations

Fully qualified type

Using the std::vector class

Recall core guideline: No "using" statements in header files

Hygiene for code you are sharing with others

# Member Functions

- Bundling together related data is deeper than just putting them together into a single object for convenience
- There are also ***invariants*** that need to be maintained
- So we can't just let the user do whatever they want to the data
- (And, again, we want to hide implementation from interface)

```
class Vector {
  size_t M;
  std::vector<double> storage_;
};
```

# Invariants

- For example

```
class Vector {
  size_t M;
  std::vector<double> storage_;
};
```

Should always be positive

And never change (?)

Size must ***always*** be M

- Things we can do with this interface that make no sense `Vector x;`

```
size_t len = x.storage_.size();
```

x is a vector, size() has no meaning

```
x.M = x.M - 1;
```

Can't arbitrarily change vector dimension

# Member Functions: Interface vs Implementation

```
class Vector {
  size_t num_rows();

  size_t M;
  std::vector<double> storage_;
};
```

Member functions also bundled with class

Return number of rows of vector

Can still access these

```
Vector x;

size_t foo = x.num_rows();

x.num_rows() = 5;

size_t bar = num_rows(x);
```

Call the member function num_rows on object x

Returns a value in this case (see class definition)

Need to invoke as member

# Interface vs Implementation

```
class Vector {
public:
  Vector(size_t M) : nu                    ows_)

  size_t num_rows() c

private:
  size_t
  std::vector<double> storage_;
};
```

Anything public can be accessed ***outside*** the scope of the class

Anything private can only be accessed ***inside*** the scope of the class

```
Vector x;

size_t foo = x.num_rows_;

size_t bar = x.num_rows();
```

Cannot access private data

Can call public member function

More Hygiene: ***Never*** make member data public

# Interface and Implementation

- Convention: Interface in .hpp and Implementation in .cpp
- (One pair per class)

Vector.hpp

```
#include <vector>

class Vector {
public:
  Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

  size_t num_rows() const;

private:
  size_t                num_rows_;
  std::vector<double> storage_;
};
```

Declare member function num_rows()

Vector.cpp

```
#include "Vector.hpp"

size_t Vector::num_rows() {
  return rum_rows_;
}
```

Vector scope

Access private data

Implementation

# Interface and Implementation

- For short functions, you can put implementation in the header
- (Necessary for class and function templates)

Vector.hpp

```
#include <vector>

class Vector {
public:
  Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

  size_t num_rows() const  { return num_rows; }

private:
  size_t              num_rows_;
  std::vector<double> storage_;
};
```

# The Vector class so Far

- Encapsulates vector data
- Member data for dimensions (rows) and for storing elements
- Member function to get number of rows
- Separate interface and implementation via public / private

- Three more things:
  - How to bring a Vector into being (“constructors”)
  - Function for getting vector data
  - Function for setting vector data
- Bonus: Assignment and operator()

# Constructors

- The C++ compiler “knows” about built-in types
- When a variable of a built-in type is declared, the compiler just needs to allocate space for it
- C++ classes are user-defined
- Compiler can do its best (default constructor), but usually we need to do more to create a well-defined object

- For example, a well-defined vector should be given its (positive) dimension ***when it is created***. (And the data initialized.)

# Constructors

Built-in type, compiler allocates known amount of space

Default constructor is invoked when variable is declared with no arguments

```
int x = 42;

Vector x;

Vector x(27);

std::cout << "x is " << x.num_rows() << " in length." << std::cout;
```

Compiler creates x with ***default constructor***

Compiler creates x with specific constructor

In this case, creates a 27 element Vector

# Declaring Constructors

```
#include <vector>

class Vector {
public:
  Vector();
  Vector(size_t M);

  size_t num_rows() const  { r

private:
  size_t                  num_rows_;
  std::vector<double> storage_;
};
```

A constructor is defined using the name of the class

And then the arguments

Can be ***overloaded*** (different functions distinguished by argument types)

Where have we already seen overloading?

# Defining Constructors

Vector.hpp

```
#include <vector>

class Vector {
public:
  Vector();
  Vector(size_t M);

  size_t num_rows() const  { return num_rows; }

private:
  size_t                  num_rows_;
  std::vector<double> storage_;
};
```

Vector.cpp

```
#include "Vector.hpp"

Vector::Vector(size_t M) {
  num_rows_ = M;
  storage_ = std::vector<double>(num_rows);
}

Vector::Vector() {
  num_rows_ = 1;
  storage_ = std::vector<double>(num_rows_);
}
```

# Defining Constructors

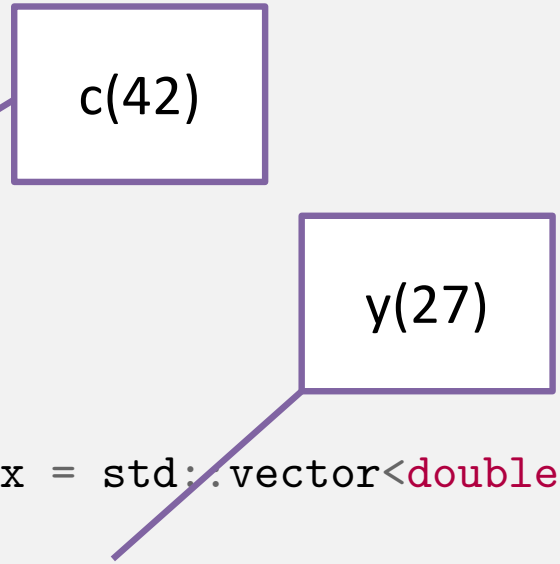
Vector.hpp

```
#include <vector>

class Vector {
public:
  Vector() {
    num_rows_ = 1;
    storage_  = std::vector<double>(num_rows);
  }
  Vector(size_t M) {
    num_rows_ = M;
    storage_  = std::vector<double>(num_rows);
  }

  size_t num_rows() const { return num_rows; }

private:
  size_t              num_rows_;
  std::vector<double> storage_;
};
```

# Initialization

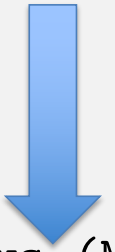
- We have said that variables should always be initialized
- Different syntaxes

```
int a = 42;

int b = int(42);

int c(42);

int d = { 42 };

std::vector<double> x = std::vector<double>(27);

std::vector<double> y(27);
```



# Defining Constructors

```
#include <vector>

class Vector {
public:
  Vector(size_t M) : num_rows_(M), storage_(num_rows) {}

  size_t num_rows() const { return num_rows; }

private:
  size_t                 num_rows_;
  std::vector<double> storage_;
};
```



Vector.hpp

Initialization syntax
Introduce with :
Construct data members

Omit default constructor (why?)

# Accessors

```
#include <vector>

class Vector {
public:
  double get(size_t i) {
    return storage_[i];
  }

private:
  size_t                num_rows_;
  std::vector<double> storage_;
};
```

Return it ***by value*** (copy)

Look up the value at location i

# Accessors

```
#include <vector>

class Vector {
public:
  double get(size_t i) {
    return storage_[i];
  }

  void set(size_t i, double val) {
    storage_[i] = val;
  }

private:
  size_t                  nu
  std::vector<double> st
};
```

lvalue vs rvalue

Pass ***by value***

Assign the element at location to i to value val

Look up location i

# Accessors

- Example – make a Vector of all ones

```
Vector x(10);

for (size_t i = 0; i < x.num_rows(); ++i) {
  x.set(i, 1.0)
}
```

Really want to say
`x(i) = 1.0;`

- Not a very natural syntax
- Asymmetric for get and set – mathematically we say x(i) regardless

# operator Functions

- C++ has special function names for functions with operator syntax
- Suppose I want to be able to write an expression to add two vectors

```
Vector x(5), y(5), z(5);

z = x + y;
```

```
for (size_t i = 0; i < x.num_rows(); ++i) {
  double tmp = x.get(i) + y.get(i);
  z.set(i, tmp);
}
```

This says to add the vectors

Which would you rather read?

# operator Functions

```
#include <vector>

class Vector {
public:
  Vector add(const Vector& y);

private:
  size_t                  num_rows_;
  std::vector<double> storage_;
};
```

And returns a Vector

Member function add()

Takes another Vector as an argument

Member function add()

Takes another Vector as an argument

```
Vector x(5), y(5), z(5);

z = x.add(y);
```

And returns a Vector

Want z = x + y;

# Before

```
#include <vector>

class Vector {
public:
  Vector add(const Vector& y);

private:
  size_t                  num_rows_;
  std::vector<double> storage_;
};
```

And returns a Vector

Member function add()

Takes another Vector as an argument

Member function add()

Takes another Vector as an argument

```
Vector x(5), y(5), z(5);

z = x.add(y);
```

And returns a Vector

Want z = x + y;

# After

```
#include <vector>

class Vector {
public:
  Vector operator+(const Vector& y);

private:
  size_t                num_rows_;
  std::vector<double> storage_;
};
```

And returns a Vector

Member function operator+()

Takes another Vector as an argument

Member operator+

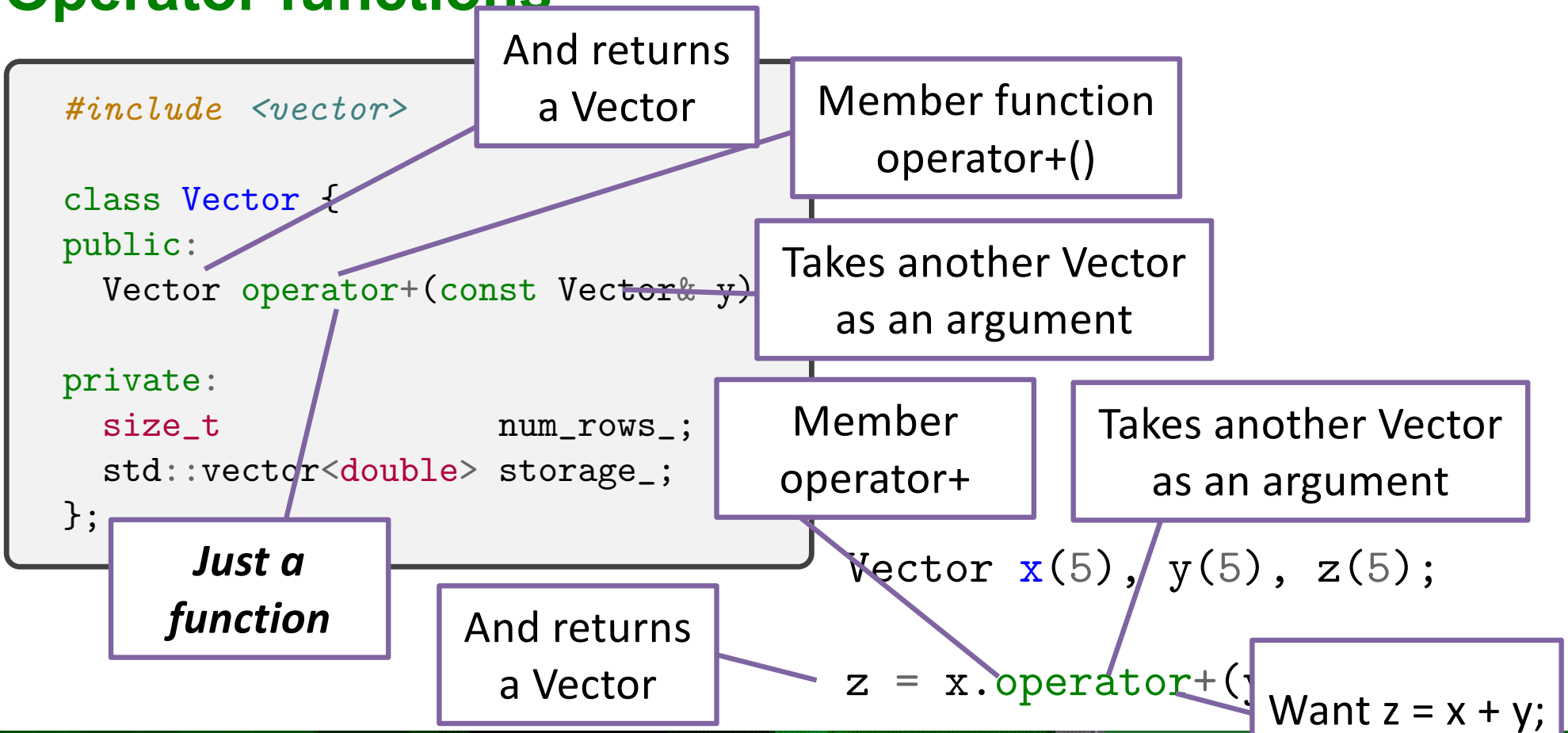
Takes another Vector as an argument

```
Vector x(5), y(5), z(5);

z = x.operator+(
```

And returns a Vector

Want z = x + y;

# Operator functions



# operator Functions

- Time out!

There is a leap coming, and you need to be here to make that leap

- Make sure you understand two things
- The way we defined the member function add()
  - Like any member function
- All we did was ***change the name*** from “add” to “operator+”
- operator+ is just a member function

- Explain this to a classmate, a friend, yourself, someone on line to make sure you understand this

# operator Functions

- C++ has a special magic syntax with operator functions

```
#include <vector>

class Vector {
public:
  Vector operator+(const Vector& y);

private:
  size_t              num_rows_;
  std::vector<double> storage_;
};
```

We've defined the member function named operator+

We invoke a member function like this

We can write it like this!

```
Vector x(5), y(5), z(5);

z = x.operator+(y);
```

```
Vector x(5), y(5), z(5);

z = x + y;
```

Still calls operator+()s

# operator Functions

- C++ has a special magic syntax with operator functions

```
#include <vector>

class Vector {
public:
  Vector operator+(const Vector& y);

private:
  size_t              num_rows_;
  std::vector<double> storage_;
};
```

***One*** argument passed in here

We invoke a member function like this, with ***one*** argument

And, the operator we will look at next is a little more confusing

***Two*** operands here

```
Vector x(5), y(5), z(5);

z = x.operator+(y);
```

```
Vector x

z = x + y;
```

# Before

```
#include <vector>

class Vector {
public:
  Vector operator+(const Vector& y);

private:
  size_t                num_rows_;
  std::vector<double> storage_;
};
```

# After

```
#include <vector>

class Vector {
public:
  double operator()(size_t i);

private:
  size_t                num_rows_;
  std::vector<double> storage_;
};
```

# operator Functions

- The next operator isn't a binary operator between two objects

```
class Vector {
public:
  double operator()(size_t i);

private:
  size_t
  std::vector<double> storage_;
};
```

The first parens are part of the function name

i is a function parameter

This member function is called "operator()"

Invoke the member function operator() with argument 3

Invoke the member function operator() with argument 3

```
Vector x(5);
double foo = x.operator()(3);
```

```
Vector x(5);
double foo = x(3);
```

# What Should operator() return?

```
class Vector {
public:
  double operator()(size_t i);

private:
  size_t                  num_rows_;
  std::vector<double> storage_;
};
```

Returns a value

Return by value is like pass by value – it's a temporary copy

But we want to do both!

So we can do this

```
Vector x(5);
double foo = x(3);
```

rvalue

But not this

```
Vector x(5);
x(3) = 0.0;
```

rvalue

# Before

```
class Vector {
public:
  double operator()(size_t i);

private:
  size_t                  num_rows_;
  std::vector<double> storage_;
};
```

# After

```
class Vector {
public:
  double& operator()(size_t i);

private:
  size_t                  num_rows_;
  std::vector<double> storage_;
};
```

# What Should operator() return?

```
class Vector
public:
  double& operator()(size_t i);

private:
  size_t                  num_rows_;
  std::vector<double>     storage_;
};
```

Return a ***reference*** to internal member data

So a reference to member data is not to something temporary

```
Vector x(5);
```

When we create ("instantiate") an object, its member data live as long as the object does

# What Should operator() return?

```
class Vector
public:
  double& operator()(size_t i);

private:
  size_t                  num_rows_;
  std::vector<double> storage_;
};
```

Return a ***reference*** to internal member data

```
Vector x(5);

double foo = x(3);
x(2) = 0.0;
```

Can assign to internal data through the reference

Can read from internal data through the reference

```
Vector x(5);
```

# Interface and Implementation

Vector.hpp

```
#include <vector>

class Vector {
public:
  double& operator()(size_t i);

private:
  size_t                num_rows_;
  std::vector<double> storage_;
};
```

Vector.cpp

```
#include "Vector.hpp"

double& Vector::operator()(size_t i) {
  return storage_[i];
}
```

# Interface and Implementation

Vector.hpp

```
#include <vector>

class Vector {
public:
  double& operator()(size_t i) {
    return storage_[i];
  }

private:
  size_t                num_rows_;
  std::vector<double> storage_;
};
```

# All Together

Vector.hpp

```
#include <vector>

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_;
};
```

# Reprise operator+()

```
#include <vector>

class Vector {
public:
  Vector operator+(const Vector& y);

private:
  size_t                num_rows_;
  std::vector<double> storage_;
};
```

# Reprise operator+()

C.4: Make a function a member only if it needs direct access to the representation of a class

```
#include <vector>

class Vector {
public:
  Vector operator+(const Vector& y) {
    Vector z(num_rows_);
    for (size_t i = 0; i < num_rows_; ++i) {
      z.storage_[i] = storage_[i] + y.storage[i];
    }
  }

private:
  size_t              num_rows_;
  std::vector<double> storage_;
};
```

Does this need to be a member?

Data for z

Data for "x"

Data for y

# All Together

Vector.hpp

```
#include <vector>

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_;
};
```

Can access via operator()

Don't need access to internals

Amath583.cpp

```
#include "Vector.hpp"

Vector operator+(const Vector& x, const Vector& y) {
  Vector z(x.num_rows());
  for (size_t i = 0; i < z.num_rows(); ++i) {
    z(i) = x(i) + y(i);
  }
}
```

Return a Vector

Take args by const reference

Nicely symmetric

# All Together

Vector.hpp

```cpp
#include <vector>

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_;
};
```

Amath583.hpp

```cpp
#include "Vector.hpp"

Vector operator+(const Vector& x, const Vector& y);
```

Amath583.cpp

```cpp
#include "Vector.hpp"
#include "amath583.hpp"

Vector operator+(const Vector& x, const Vector& y) {
  Vector z(x.num_rows());
  for (size_t i = 0; i < z.num_rows(); ++i) {
    z(i) = x(i) + y(i);
  }
}
```

# All Together

Vector.hpp

```cpp
#include <vector>

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_;
};
```

Amath583.hpp

```cpp
#include "Vector.hpp"

Vector operator+(const Vector& x, const Vector& y);
```

Amath583.cpp

```cpp
#include "Vector.hpp"
#include "amath583.hpp"

Vector operator+(const Vector& x, const Vector& y) {
  Vector z(x.num_rows());
  for (size_t i = 0; i < z.num_rows(); ++i) {
    z(i) = x(i) + y(i);
  }
}
```

# Not quite finished

```
#include "Vector.hpp"

int main() {

  Vector x(100), y(100), z(100), w(100);

  z = x + y;

  return 0;
}
```

```
% c++ constness.cpp
constness.cpp:20:12: error: no matching function for call to object of type 'const Vector'
    z(i) = x(i) + y(i);
           ^
constness.cpp:7:11: note: candidate function not viable: 'this' argument has type
      'const Vector', but method is not marked const
  double& operator()(size_t i) { return storage_[i]; }
          ^
constness.cpp:20:19: error: no matching function for call to object of type 'const Vector'
    z(i) = x(i) + y(i);
                  ^
constness.cpp:7:11: note: candidate function not viable: 'this' argument has type
      'const Vector', but method is not marked const
  double& operator()(size_t i) { return storage_[i]; }
          ^
2 errors generated.
```

# Constness

Vector.hpp

```
#include <vector>

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_;
};
```

x and y are defined to be const

Amath583.hpp

```
#in

Vector operator+(const Vector& x, const Vector& y);
```

"this" is not const

Amath583.cpp

```
#include "Vector.hpp"
#include "amath583.hpp"

Vector operator+(const Vector& x, const Vector& y) {
  Vector z(x.num_rows());
  for (size_t i = 0; i < z.num_rows(); ++i) {
    z(i) = x(i) + y(i);
  }
}
```

# Overloading

```
void foo(size_t i) {
  std::cout << "foo(size_t i)" << std::endl;
}

void foo(double d) {
  std::cout << "foo(double d)" << std::endl;
}
```

Takes a size_t

Takes a double

```
int main() {

  size_t a = 0;
  double b = 0.0;

  foo(a);
  foo(b);

  return 0;
}
```

```
% ./a.out
foo(size_t i)
foo(double d)
```

# Overloading

```
void foo(size_t i) {
  std::cout << "void foo(size_t i)" << std::endl;
}

size_t foo(size_t i) {
  std::cout << "size_t foo(size_t i)" << std::endl;
}
```

Returns void

Returns size_t

```
% c++ overload.cpp
overload.cpp:7:8: error: functions that differ only in their return type cannot be overloaded
size_t foo(size_t i) {
~~~~~~ ^
overload.cpp:3:6: note: previous definition is here
void foo(size_t i) {
~~~~ ^
```

```
int main() {

  size_t a = 0;
  size_t b = 0;

  foo(a);
  double c = foo(a);

  return 0;
}
```

Have to pick the function then call it

# No overloading on return values

```
size_t foo(size_t i) {
  std::cout << "size_t foo(size_t i)" << std::

  return i;
}


int main() {

  size_t a = 0;

  foo(a);
  size_t b = foo(a);
  double c = foo(a);

  return 0;
}
```

What happens to the return value is not the concern of the function

Ignore return value

Assign to size_t

Assign to double

# Constness

```
double parens(double& x, size_t i) {
  std::cout << "called non const parens" << std::endl;
  double y = x;
  // .. some things
  return y;
}
```

x is a ref

```
int main() {

  double x = 5.0;
  double y = parens(x);

  const double z = 5.0;
  double w = parens(z);

  double a = parens(5.0);
  double b = parens(x + y);

  const double c = parens(x + y + z + 5.0);

  return 0;
}
```

```
c++ const3.cpp
const3.cpp:27:14: error: no matching function for call to 'parens'
  double w = parens(z, 27);
             ^~~~~
const3.cpp:13:8: note: candidate function not viable: 1st argument ('const double') would lose const
      qualifier
double parens(double& x, size_t i) {
       ^
```

```
const3.cpp:29:14: error: no matching function for call to 'parens'
  double a = parens(5.0, 27);
             ^~~~~
const3.cpp:13:8: note: candidate function not viable: expects an l-value for 1st argument
double parens(double& x, size_t i) {
       ^
```

Not okay

```
const3.cpp:32:20: error: no matching function for call to 'parens'
  const double c = parens(x + y + 5.0, 27);
                   ^~~~~
const3.cpp:13:8: note: candidate function not viable: expects an l-value for 1st argument
double parens(double& x, size_t i) {
       ^
```

Not okay

# Constness

```
double parens(const double& x, size_t i) {
  std::cout << "called const parens" << std::endl;
  double y = x;
  // .. some things
  return y;
}
```

x is a const ref

```
int main() {

  double x = 5.0;
  double y = parens(x);

  const double z = 5.0;
  double w = parens(z);

  double a = parens(5.0);
  double b = parens(x + y);

  const double c = parens(x + y + z + 5.0);

  return 0;
}
```

okay

okay

okay

okay

```
./a.out
called const parens
called const parens
called const parens
called const parens
called const parens
```

# Constness

x is a const ref

```
double parens(const double& x, size_t i) {
  std::cout << "called const parens" << std::endl;
  double y = x;
  // .. some things
  return y;
}
```

x is a ref

```
double parens(double& x, size_t i) {
  std::cout << "called non const parens" << std::endl;
  double y = x;
  // .. some things
  return y;
}
```

```
int main() {

  double x = 5.0;
  double y = parens(x);

  const double z = 5.0;
  double w = parens(z);

  double a = parens(5.0);
  double b = parens(x + y);

  const double c = parens(x + y + z + 5.0);

  return 0;
}
```

x is lvalue

z marked const

5.0 is an rvalue

x + y is an rvalue

```
./a.out
called non const parens
called const parens
called const parens
called const parens
called const parens
```

# Why not always pass const reference?

```
double parens(const double& x, size_t i) {
  std::cout << "called const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

Return double

```
int main() {
  double y = 0.5;
  double p = 3.14;

  double x = 5.0;
  parens(x, 27) = p;

  const double z = 5.0;
  parens(z, 27) = p;

  parens(5.0, 27) = p;
  parens(x + y, 27) = p;

  return 0;
}
```

```
c++ const4.cpp
const4.cpp:23:17: error: expression is not assignable
  parens(x, 27) = p;
  ~~~~~~~~~~~~~ ^
const4.cpp:26:17: error: expression is not assignable
  parens(z, 27) = p;
  ~~~~~~~~~~~~~ ^
const4.cpp:28:19: error: expression is not assignable
  parens(5.0, 27) = p;
  ~~~~~~~~~~~~~~~ ^
const4.cpp:29:21: error: expression is not assignable
  parens(x + y, 27) = p;
  ~~~~~~~~~~~~~~~~~ ^
```

# Before

```
double parens(const double& x, size_t i) {
  std::cout << "called const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

# After

```
double& parens(const double& x, size_t i) {
  std::cout << "called const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

# Why not always pass const reference?

```
double& parens(const double& x, size_t i) {
  std::cout << "called const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

But x is const

Return ref to double

Can't return const

```
int main() {
  double y = 0.5;
  double p = 3.14;

  double x = 5.0;
  parens(x, 27) = p;

  const double z = 5.0;
  parens(z, 27) = p;

  parens(5.0, 27) = p;
  parens(x + y, 27) = p;

  return 0;
}
```

```
c++ const5.cpp
const5.cpp:9:10: error: binding value of type 'const double' to reference to type 'double' drops
      'const' qualifier
  return x;
         ^
```

# Before

```
double& parens(const double& x, size_t i) {
  std::cout << "called const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

# After

```
const double& parens(const double& x, size_t i) {
  std::cout << "called const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

# Why not always pass const reference?

```
const double& parens(const double& x, size_t i) {
  std::cout << "called const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

```
int main() {
  double y = 0.5;
  double p = 3.14;

  double x = 5.0;
  parens(x, 27) = p;

  const double z = 5.0;
  parens(z, 27) = p;

  parens(5.0, 27) = p;
  parens(x + y, 27) = p;

  return 0;
}
```

```
c++ const5.cpp
const5.cpp:26:17: error: cannot assign to return value because function 'parens' returns a const value
  parens(x, 27) = p;
  ~~~~~~~~~~~~~ ^
const5.cpp:5:7: note: function 'parens' which returns const-qualified type 'const double &' declared
      here
const double& parens(const double& x, size_t i) {
      ^~~~~~~
const5.cpp:29:17: error: cannot assign to return value because function 'parens' returns a const value
  parens(z, 27) = p;
  ~~~~~~~~~~~~~ ^
const5.cpp:5:7: note: function 'parens' which returns const-qualified type 'const double &' declared
      here
const double& parens(const double& x, size_t i) {
      ^~~~~~~
const5.cpp:31:19: error: cannot assign to return value because function 'parens' returns a const value
  parens(5.0, 27) = p;
  ~~~~~~~~~~~~~~~ ^
const5.cpp:5:7: note: function 'parens' which returns const-qualified type 'const double &' declared
      here
const double& parens(const double& x, size_t i) {
      ^~~~~~~
const5.cpp:32:21: error: cannot assign to return value because function 'parens' returns a const value
  parens(x + y, 27) = p;
  ~~~~~~~~~~~~~~~~~ ^
const5.cpp:5:7: note: function 'parens' which returns const-qualified type 'const double &' declared
      here
const double& parens(const double& x, size_t i) {
```

# Before

```
double& parens(const double& x, size_t i) {
  std::cout << "called const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

# After

```
double& parens(double& x, size_t i) {
  std::cout << "called const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

# How about no const at all?

```
double& parens(double& x, size_t i) {
  std::cout << "called const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

```
int main() {
  double y = 0.5;
  double p = 3.14;

  double x = 5.0;
  parens(x, 27) = p;

  const double z = 5.0;
  parens(z, 27) = p;

  parens(5.0, 27) = p;
  parens(x + y, 27) = p;

  return 0;
}
```

```
c++ const5.cpp
const5.cpp:30:3: error: no matching function for call to 'parens'
  parens(z, 27) = p;
  ^~~~~~
const5.cpp:14:9: note: candidate function not viable: 1st argument ('const double') would lose const
      qualifier
double& parens(double& x, size_t i) {
        ^
const5.cpp:32:3: error: no matching function for call to 'parens'
  parens(5.0, 27) = p;
  ^~~~~~
const5.cpp:14:9: note: candidate function not viable: expects an l-value for 1st argument
double& parens(double& x, size_t i) {
        ^
const5.cpp:33:3: error: no matching function for call to 'parens'
  parens(x + y, 27) = p;
  ^~~~~~
const5.cpp:14:9: note: candidate function not viable: expects an l-value for 1st argument
double& parens(double& x, size_t i) {
        ^
```

# How about no const at all?

```
int main() {
  double y = 0.5;
  double p = 3.14;

  double x = 5.0;
  parens(x, 27) = p;

  const double z = 5.0;
  parens(z, 27) = p;

  parens(5.0, 27) = p;
  parens(x + y, 27) = p;

  return 0;
}
```

This makes sense

This ***should*** be an error

This ***should*** be an error

This ***should*** be an error

# More sensible

```
int main() {
  double y = 0.5;
  double p = 3.14;

  double x = 5.0;
  parens(x, 27) = p;

  const double z = 5.0;
  double q = parens(z, 27);

  double r = parens(5.0, 27);
  double s = parens(x + y, 27);

  return 0;
}
```

This makes sense

This makes sense

This makes sense

This makes sense

# More sensible

```
double& parens(double& x, size_t i) {
  std::cout << "called non const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

```
int main() {
  double y = 0.5;
  double p = 3.14;

  double x = 5.0;
  parens(x, 27) = p;

  const double z = 5.0;
  double q = parens(z, 27);

  double r = parens(5.0, 27);
  double s = parens(x + y, 27);

  return 0;
}
```

```
c++ const6.cpp
const6.cpp:30:14: error: no matching function for call to 'parens'
  double q = parens(z, 27);
             ^~~~~~
const6.cpp:14:9: note: candidate function not viable: 1st argument ('const double') would lose const
      qualifier
double& parens(double& x, size_t i) {
        ^
const6.cpp:32:14: error: no matching function for call to 'parens'
  double r = parens(5.0, 27);
             ^~~~~~
const6.cpp:14:9: note: candidate function not viable: expects an l-value for 1st argument
double& parens(double& x, size_t i) {
        ^
const6.cpp:33:14: error: no matching function for call to 'parens'
  double s = parens(x + y, 27);
             ^~~~~~
const6.cpp:14:9: note: candidate function not viable: expects an l-value for 1st argument
double& parens(double& x, size_t i) {
        ^
```

Oops, need to be const

Going in circles?

# More sensible

```
const double& parens(const double& x, size_t i) {
  std::cout << "called non const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

```
int main() {
  double y = 0.5;
  double p = 3.14;

  double x = 5.0;
  parens(x, 27) = p;

  const double z = 5.0;
  double q = parens(z, 27);

  double r = parens(5.0, 27);
  double s = parens(x + y, 27);

  return 0;
}
```

```
c++ const6.cpp
const6.cpp:27:17: error: cannot assign to return value because function 'parens' returns a const value
  parens(x, 27) = p;
  ~~~~~~~~~~~~~ ^
const6.cpp:6:7: note: function 'parens' which returns const-qualified type 'const double &' declared
      here
const double& parens(const double& x, size_t i) {
      ^~~~~~
```

Oops, need to be non const

Going in circles?

# Overloading to the rescue

```
const double& parens(const double& x, siz
  std::cout << "called non const parens"
  double y = x;
  // .. some things
  return x;
}
```

const

```
double& parens(double& x, size_t i) {
  std::cout << "called non const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

Not const

```
int main() {
  double y = 0.5;
  double p = 3.14;

  double x = 5.0;
  parens(x, 27) = p;

  const double z = 5.0;
  double q = parens(z, 27);

  double r = parens(5.0, 27);
  double s = parens(x + y, 27);

  return 0;
}
```

const

Not const

```
./a.out
called non const parens
called const parens
called const parens
called const parens
```

# What does this have to do with operator()

```
const double& parens(const double& x, siz
  std::cout << "called non const parens"
  double y = x;
  // .. some things
  return x;
}
```

const

const

```
double& parens(double& x, size_t i) {
  std::cout << "called non const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

Not const

Not const

```
class Vector {
public:
  Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

  double& operator()(size_t i) { return storage_[i]; }

private:
  size_t                num_rows_;
  std::vector<double> storage_;
};
```

Where is the const or non-const thing to overload on?

# What does this have to do with operator()

```
const double& parens(const double& x, siz
  std::cout << "called non const parens"
  double y = x;
  // .. some things
  return x;
}
```

```
double& parens(double& x, size_t i) {
  std::cout << "called non const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

const

const

Not const

Not const

```
class Vector
public:
  Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

        double& operator()(size_t i) { return storage_[i]; }
  const double& operator()(size_t i) { return storage_[i]; }

private:
  si                um_rows_;
  st                orage_;
};
```

Only differing by return type

Where is the const or non-const thing to overload on?

# There is a secret argument

```
const double& parens(const double& x, siz
  std::cout << "called non const parens"
  double y = x;
  // .. some things
  return x;
}
```

```
double& parens(double& x, size_t i) {
  std::cout << "called non const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

const

const

Not const

Not const

```
class Vector
public:
  Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

        double& operator()(size_t i) { return storage_[i]; }
  const double& operator()(size_t i) { return storage_[i]; }

                        num_rows_;
  std::vector<double> storage_;
};
```

Called “this”

There is a secret argument

There is a secret argument

# There is a secret argument

```
const double& parens(const double& x, siz
  std::cout << "called non const parens"
  double y = x;
  // .. some things
  return x;
}
```

```
double& parens(double& x, size_t i) {
  std::cout << "called non const parens" << std::endl;
  double y = x;
  // .. some things
  return x;
}
```

const

const

Not const

Not const

```
class Vector {
public:
  Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

        double& operator()(Vector *this, size_t i) { return storage_[i]; }
  const double& operator()(Vector *this, size_t i) { return storage_[i]; }

private:
  size_t              num_rows_;
  std::vector<double> storage_;
};
```

How would we fix our const problem?

# Before

```
class Vector {
public:
  Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

        double& operator()(Vector *this, size_t i) { return storage_[i]; }
  const double& operator()(Vector *this, size_t i) { return storage_[i]; }

private:
  size_t              num_rows_;
  std::vector<double> storage_;
};
```

# After

```
class Vector {
public:
  Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

        double& operator()(Vector *this, size_t i) { return storage_[i]; }
  const double& operator()(const Vector *this, size_t i) { return storage_[i]; }

private:
  size_t              num_rows_;
  std::vector<double> storage_;
};
```

# After After

```
class Vector {
public:
  Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

        double& operator()(size_t i)       { return storage_[i]; }
  const double& operator()(size_t i) const { return storage_[i]; }

private:
  size_t              num_rows_;
  std::vector<double> storage_;
};
```

const “this”

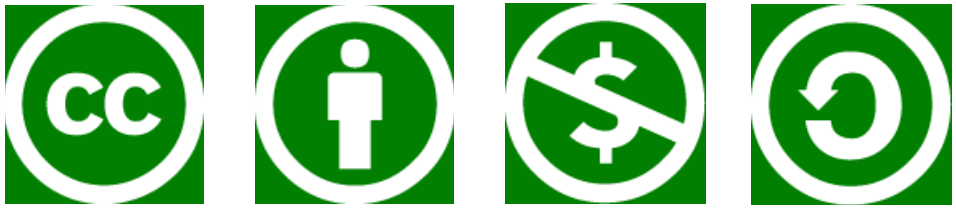
# Finally

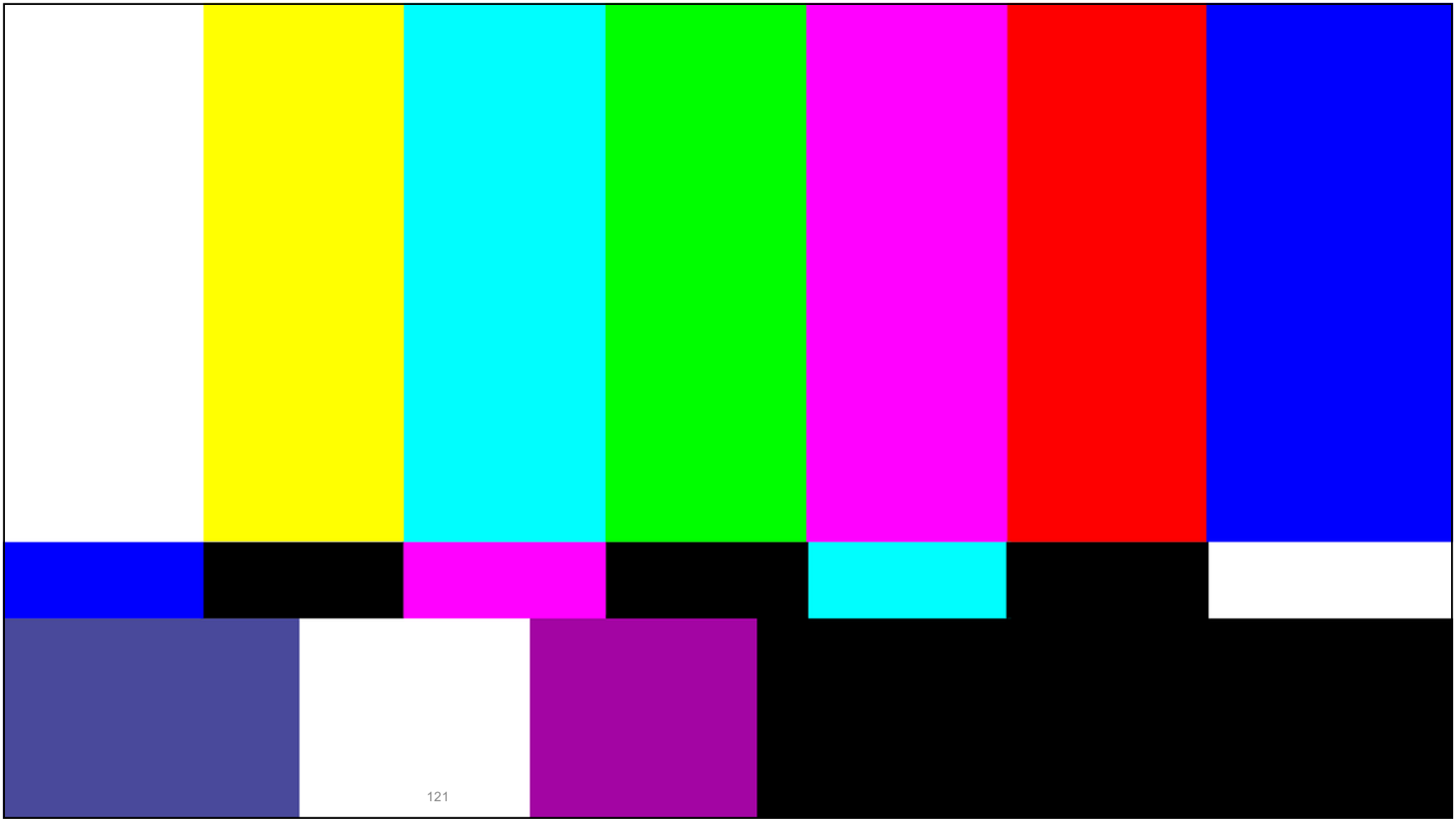
```
#include <vector>

class Vector {
public:
  Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

        double& operator()(size_t i)       { return storage_[i]; }
  const double& operator()(size_t i) const { return storage_[i]; }

  size_t num_rows() { return num_rows_; }

private:
  size_t              num_rows_;
  std::vector<double> storage_;
};
```

# 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!




121