# CMOR 420/520
# Computational Science
# Programming in C++

# C vs C++

- C++ is (formally) an extension of C to introduce additional features.
- Invented by Bjarne Stroustrup in 1985.
  - New standards released every few years for both C and C++, but C++ has changed more significantly over its history.
- Some slightly different default behaviors, but if you program C based on the lectures in this class, your C program should run as a C++ program too

**Timeline of C language**

| Year | Informal name | Official standard |
|---|---|---|
| 1972 | first release | — |
| 1978 | K&R C | — |
| 1989, 1990 | ANSI C, C89, ISO C, C90 | ANSI X3.159-1989 ISO/IEC 9899:1990 |
| 1999 | C99, C9X | ISO/IEC 9899:1999 |
| 2011 | C11, C1X | ISO/IEC 9899:2011 |
| 2018 | C17 | ISO/IEC 9899:2018 |
| 2024 | C23, C2X | ISO/IEC 9899:2024 |
| Future | C2Y | — |

**C++ standards**

| Year | ISO/IEC Standard | Informal name |
|---|---|---|
| **1998** | 14882:1998[34] | C++98 |
| **2003** | 14882:2003[35] | C++03 |
| **2011** | 14882:2011[36] | C++11, C++0x |
| **2014** | 14882:2014[37] | C++14, C++1y |
| **2017** | 14882:2017[38] | C++17, C++1z |
| **2020** | 14882:2020[17] | C++20, C++2a |
| **TBA** | 14882:2024 | C++23, C++2b |
| **TBA** | | C++26, C++2c |

# Some caveats of C++

- Be warned that, since C++ introduces new features, these features can introduce confusing/contradictory behavior (and consequently, bugs) as well.
- I mostly know C++14; “modern” C++ includes C++17, 20, and 23.
- Some notable quotes from notable people:
  - The problem that I have with them today is that... C++ is too complicated. At the moment, it's impossible for me to write portable code […] unless I avoid all exotic features. **Whenever the C++ language designers had two competing ideas as to how they should solve some problem, they said "OK, we'll do them both".**
  - "Within C++, there is a much smaller and cleaner language struggling to get out."

# Compiling C++ code

- For C, files are “.h” and “.c”, and we use the “gcc” compiler.
- For C++, files are typically “.hpp” and “.cpp”, and we use the “g++” compiler
- Install the “g++” compiler via “sudo apt install” on Linux
  - On Mac, you should have it installed if you installed Xcode tools previously. If not, you can install it via “brew install g++”.

# C vs C++ “hello world”

```c
#include <stdio.h>

int main(void){
  printf("Hello, world.\n");
  return 0;
}
```

```cpp
#include <cstdio> // or <stdio.h>
#include <iostream>

int main(void) {
  printf("Hello world\n");
  std::cout << "Hello world" << std::endl;
}
```

- Very similar structure, but C++ can use the “iostream” package for different printing syntax. Note: “std” refers to the “**st**an**d**ard library” in C++.
  - Here, “<<” sends a string to a *stream*, which can print to terminal, print to a file, reroute to another stream, etc.
  - “std::cout” is an example of *namespacing*; the function “cout” belongs to “std”

# Exercise

- Write a ’hello_world’ program in C++

# References in C++

- C++ introduces “references” to avoid some potential issues with pointers
- A reference is a variable that behaves like a dereferenced pointer.
  - Modifying a reference is equivalent to dereferencing a pointer and modifying it.
  - Like pointer addresses, references must be initialized to a variable (e.g., they cannot be initialized to a temporary value)

```
int i;
int * ptr_i = &i;
int & ref_i = i;

*ptr_i = 6;
ref_i = 7;
```

Modifying either “ptr_i” or “ref_i” modifies “I” as well

```
// neither is allowed
int * ptr_i = &1;
int & ref_i = 1;
```

# More on references in C++

- References are basically pointer aliases. A reference and the variable it references have the same memory address.
- However, we can treat a reference as a normal variable otherwise (perform arithmetic, assign values to it, etc)
- Note: beware dangling references (similar to dangling pointers); don't return a reference to a stack-allocated variable inside a function!

```
int main(void) {

  int i;
  int * ptr_i = &i;
  int & ref_i = i;

  cout << "Address of i = " <<
    &i << endl;
  cout << "ptr_i = " <<
    ptr_i << endl;
  cout << "Address of ref_i = " <<
    &ref_i << endl;
```

```
/a.out
Address of i = 0x16f493468
ptr_i = 0x16f493468
Address of ref_i = 0x16f493468
```

# Passing references to functions

- A function can take in a reference as an input; passing in a variable then converts the variable to a reference within that function.
- “const” is often used with references to avoid modifying variables unintentionally inside a function.
- Side note: here, “using namespace std” *imports* all function names in “std” into the global namespace.

```
#include <iostream>

typedef struct{
  int x;
  int y;
} Foo;

void modify_Foo(Foo & foo){
  foo.x = 10;
  foo.y = 20;
}

using namespace std;

int main(void) {
  Foo foo = {1, 2};
  modify_Foo(foo);
  cout << "foo.x, foo.y = " <<
    foo.x << ", " << foo.y << endl;
}
```

# Function overloading

- Unlike C, C++ allows you to define two functions with the same name, but *different argument types*.
  - Can select a function based on the type of the argument, as well as the number of arguments.
- The correct function is selected at compile time (not at run time).

```
#include <iostream>
using namespace std;

int foo(int x, int y){
  return x + y;
}
int foo(int x, double y){
  return (int) (x + 2.0 * y);
}
int foo(int x, double y, int z){
  return (int) (x + 3.0 * y + z);
}

int main(void){
  cout << "foo(1, 1) = " <<
    foo(1, 1) << endl;
  cout << "foo(1, 1.0) = " <<
    foo(1, 1.0) << endl;
  cout << "foo(1, 1.0, 1) = " <<
    foo(1, 1.0, 1) << endl;
}
```

# What does C++ do for function overloading?

- The “nm” program in Linux allows one to look inside “.o” files
- We can look inside the “.o” file with function overloading to see evidence of “name mangling”
  - Overloaded functions names start with “__Z” and end with some info about the argument types

```
int foo(int x, int y){
  return x + y;
}
int foo(int x, double y){
  return (int) (x + 2.0 * y);
}
int foo(int x, double y, int z){
  return (int) (x + 3.0 * y + z);
}
```

```
nm overloading.o
```

```
0000000000000020 T __Z3fooid
0000000000000054 T __Z3fooidi
0000000000000000 T __Z3fooii
```

# Exercise

- Write a function called 'Double' that has input int
- Overload it with another 'Double' function that has input data type double
- Check that the overloading works by finding 'Double' of 2 and 4.2

- Change these functions to take references

# Beware of implicit type conversion in C++

- C++ has made IMO some questionable choices for their default behavior. Implicit type conversion is one of those.
- What do you think the following code prints out? Why is it doing this?
- Technically, the input is a “char *” type, so there’s no appropriate overloaded “foo”. However, C++ checks if the “char *” type can be converted to a compatible input type.
  - Here, C++ checks if “char *” can be converted to a “bool” before it checks if it can be converted to a “string”.

¯_(ツ)_/¯

```
#include <iostream>

using namespace std;

int foo(bool x){
  return 0;
}

int foo(string string){
  return 1;
}

int main(){
  int i = foo("hi");
  cout << "i = " << i << "\n";
}
```

implicit_conversion.cpp

# C++ classes

- The original reason C++ was developed was to introduce *classes* (or objects) to C.
- Generalization of structs, core idea in object oriented programming
- Here, “Vector” class is similar to a struct
  - New concept: the “*public*” keyword controls accessibility of the fields
  - Can also specify that class member fields are “*private*” or “*protected*”.

```
class Vector {
 public:
  int length_;
  double* vals_;
};

int main(void) {
  Vector vec;
  vec.length_ = 3;
}
```

# C++ classes: member functions

- Big difference from structs: classes can have *functions* as members.
- Note: *private* members can only be accessed by member functions of a class.
- A lot of C++ is enforcing a permission structure (who is allowed to change what)
  - Hard to do this using “const” - why?
  - Public “accessor” and “setter” functions can enforce a “read-only” condition.

```
class Point{
public:
  void set_coords(double x,
                  double y){
    x = x;
    y = y;
  }
private:
  double x;
  double y;
};


int main(void){
  Point point;
  point.set_coords(1.0, 2.0);
}
```

# More on public/private and member functions

- Member functions can be declared inside a class but *defined* externally (e.g., in a .cpp vs a .hpp file) using the "::" syntax.
- Details on keywords:
  - public: external code can access or set
  - private: only member methods can access (i.e. no external code). This is C++'s default.
  - protected: the only external code that touch fields are *subclasses* (or explicitly labeled "friend" classes or functions).

```
class Point{
public:
  void set_coords(double x,
                  double y);
private:
  double x;
  double y;
};

void Point::set_coords(double x,
                       double y){
  x = x;
  y = y;
}
```

# Exercise

- Make a class called 'person' that has private variables 'int age' and 'char* name'
- Inside the class, define and declare a member functions called 'set_age' and 'set_name' that allows for age and name to be set.
- Outside of the class, define a member function called 'older_person' that adds one to the age. Define this outside the class. Note this must be declared inside the class.

# Classes have default (overrideable) functions

- For every class, C++ defines several methods if they are not specified
  - The "*constructor*" and "*destructor*" methods are the main ones
  - The constructor is automatically called when a class "instance" is created.
  - The destructor gets called when an instance goes out of scope.
- The default constructor just initializes members of the class.
  - For the default constructor, if a class has another class instance as a member, the constructor for that class gets called.
  - The destructor has similar recursive behavior (it calls the destructor for each member of the class).

# Overriding defaults

- You can specify a custom constructor or destructor for a class “Point” by creating functions called “Point” or “~Point”.
  - You can pass in arguments to constructors (not to destructors, however)
  - The “destructor” gets called when an instance goes out of bounds and is freed.
- However, because C++ does more behind the scenes than C, overriding default constructors can cause some weirdness.

* With this constructor, ‘Point point()’ is initialized with x=0 and y=0

```
class Point{
public:
  void set_coords(double x,
                  double y){
    x = x;
    y = y;
  }
  // custom constructor
  Point(){
    x = 0.0;
    y = 0.0;
  }
  // custom destructor
  ~Point(){
    cout << "calling the "
      "custom destructor\n";
  }

private:
  double x;
  double y;
};
```

point.cpp

# More on Constructors

- You can create multiple constructors for a class. When called, the compiler will use the constructor that fits the given input.

```
class Vector{
    public:
    int length_;
    double* vals_;

    Vector(){

    };
    Vector(int l, double* val){
        length_=l;
        vals_= val;
    };

    Vector(int l, double val){
        length_=l;
        vals_= &val;
    };
};
```

# Exercise

- Make a custom constructor for the class from last time:
  - It will take as input the 'name' and 'age' variables
  - it will automatically initialize the classes variables to be the input values
- Make a custom destructor

# The “explicit” keyword

- Remember implicit conversions? They can make constructor behavior unpredictable as well. Consider the following code.
  - We shouldn’t be able to call “bar(42)” since this is an “int” argument.
  - However, since C++ does implicit type conversion, it checks if an “int” can be converted to “Foo”.
- To avoid this, we’ll often use the “explicit keyword to disallow explicit conversions for classes that we define.

```
#include <iostream>

using namespace std;

struct Foo{
  explicit Foo(int x){}
};

void bar(Foo foo){
  cout << "calling bar(Foo)\n";
}

int main() {
  bar(42);
}
```

explicit_keyword.cpp

# Weird behavior with custom constructors

```cpp
#include <iostream>
#include <cstdlib>

using namespace std;

class MyClass{
public:
  MyClass();
  MyClass(int);
  ~MyClass();

  // the `const` keyword enforces that this
  // function cannot modify this instance.
  int get_value(void) const{ return *ptr_; }
  void set_value(int val){ *ptr_ = val; }

private:
  int * ptr_;
};
```

rule_of_three.cpp

```cpp
MyClass::MyClass(){
  ptr_ = (int *) malloc(sizeof(int));
}

MyClass::MyClass(int value){
  ptr_ = (int *) malloc(sizeof(int));
  *ptr_ = value;
}

MyClass::~MyClass(){
  free(ptr_);
  ptr_ = nullptr; // same as NULL
}

int main(void){
  MyClass x;
  x.set_value(10);
  cout << "x value = " << x.get_value() << "\n";

  MyClass y(20);
  cout << "y value = " << y.get_value() << "\n";

  MyClass z = y;
}
```

# Why does this happen? Custom constructors

- C++ has *several* types of constructors and constructor-like functions that are called in different circumstances.
  - Note: setting “MyClass z = y” also calls a  “copy constructor”.
- The default copy constructor does a “shallow copy” (copies field values over from one instance to another).
  - The pointer for “y” gets copied to “z”.
  - The destructor frees memory for “y”
  - Then it tries to free memory for “z”…

Can you see what goes wrong now?

```
MyClass::MyClass(){
  ptr_ = (int *) malloc(sizeof(int));
}

MyClass::MyClass(int value){
  ptr_ = (int *) malloc(sizeof(int));
  *ptr_ = value;
}

MyClass::~MyClass(){
  free(ptr_);
  ptr_ = nullptr; // same as NULL
}

int main(void){
  MyClass x;
  x.set_value(10);
  cout << "x value = " << x.get_value() << "\n";

  MyClass y(20);
  cout << "y value = " << y.get_value() << "\n";

  MyClass z = y;
}
```

# Why does this happen? Custom constructors

- To summarize: the default “copy constructor” doesn’t do what we want it to do (it just does a “shallow copy”).
- We can define a custom copy constructor, which has a specific signature (class name + reference to class type).
- Now the old code runs! But if we introduce one more line things break again…what now?

This should be
“MyClass::MyClass(const & MyClass copy_from)”

```
MyClass::MyClass(MyClass & copy_from){
  ptr_ = (int*) malloc(sizeof(int));
  *ptr_ = copy_from.get_value();
}
```

```
int main(void){
  MyClass x;
  x.set_value(10);
  cout << "x value = " << x.get_value() << "\n";

  MyClass y(20);
  cout << "y value = " << y.get_value() << "\n";

  MyClass z = y;

  z = x;
}
```

# The copy assignment operator in C++

- The behavior “z=x” calls something other than a constructor; it calls a “copy assignment operator”.
  - Again, the default behavior just does a shallow copy of the fields.
- We can *overload* the “=” operator in C++ to copy the value that “ptr” points to rather than the pointer address itself.
  - “z = x” uses this overloaded operator!

```
int main(void){
  MyClass x;
  x.set_value(10);
  cout << "x value = " << x.get_value() << "\n";

  MyClass y(20);
  cout << "y value = " << y.get_value() << "\n";

  MyClass z = y;

  z = x;
}
```

Should be “MyClass & operator=(const MyClass & copy_from)”

```
MyClass& operator=(MyClass & copy_from){
  *ptr_ = copy_from.get_value();
  return *this;
}
```

* ‘this’ is a special c++ ptr that stores the address of the current (class) instance that invoked the function

# Three C++ constructors

- Default constructor
  - Called when initializing “Foo foo”
- Copy constructor
  - Called when initializing “Foo foo2(foo)”
- Copy assignment *operator*
  - Called when setting “foo2 = foo”
  - Technically not a constructor

Note: both copy and copy assignment constructor should have “const Foo &” arguments.

```
#include <iostream>

using namespace std;

class Foo{
public:
  Foo(){
    cout << "calling the "
      "default constructor.\n";
  }
  Foo(Foo &){
    cout << "calling the "
      "copy constructor.\n";
  }
  Foo & operator=(Foo &){
    cout << "calling the copy "
      "assignment operator\n";
    return *this;
  }
};

int main(void){
  Foo foo;
  Foo foo2(foo);
  foo2 = foo;
}
```

# More on overloading

- You overload other operators in the same way, e.g. `+=`, `-=`, `+`, `-`, etc
- 'output_type operator+ =(inputs){*operation*}'

# The rule of three

- Default behaviors in C++ are interconnected!
- Rule of three: if you implement a non-default version of the destructor, copy constructor, or copy assignment operator, then you should implement all three
- Other C++ guidelines: “rule of five” and “rule of zero”.
- Rule of thumb: if you can avoid defining default operations, then do so.

# Exercise

Using the previous exercise

- Make a copy constructor
- Overload the '=' operator

# Dynamic memory in C++

- We can use C-style memory management (malloc, calloc, realloc) in C++
- However, C++ provides “new” and “delete” memory management keywords that work better with classes.
  - “new” allocates a single variable or an array of variables (instead of “malloc”), and “delete” is used (instead of “free”).
  - If you allocate a single variable, use “delete”; if you allocate an array, use “delete[]” to free the entire array.

```
#include <iostream>

using namespace std;

int main() {

  // allocating single numbers
  double * x = new double;

  // initialize (* y) = 1.0
  double * y = new double(1.0);

  delete x;
  delete y;

  // allocating a vector
  double * ptr_x = new double[10];
  delete[] ptr_x;
}
```

# “new” and “delete”

- Why use “new” and “delete” instead of “malloc”? They incorporate custom constructors/destructors in classes.
- If you have class members who are classes, “new” will call their constructors recursively and “delete” will call their destructors recursively.
- No special syntax for “(double **)” pointers; just allocate the list of pointers e.g., “double ** A = new double * [10]”, then allocate, e.g., “A[i] = new double [8]”.

```
class A {
 public:
  A(void){
    cout << "In A's constructor\n";
  }
  A(int){
    cout << "In A's `int` constructor\n";
  }
  ~A(){
    cout << "In A's destructor\n";
  }
};

class B {
 public:
  B(void) : a() {
    cout << "In B's 1st constructor\n";
  }
  B(int) : a(1) {
    cout << "In B's 2nd constructor\n";
  }
  ~B(){
    cout << "In B's destructor\n";
  }
  A a;
};
```

new_delete_classes.cpp

# Exercise: a MyVector class

- Create a MyVector class
  - It should have private members “int length_” and “double * data_”. Please also use “new” and “delete” instead of “malloc”.
  - It should follows the *rule of three* and implement the following methods:
    - “get_length(void)”, which returns the length,
    - “get_index(int index)”, which returns the value at the “index"
    - “set_index(int index, double value)” which sets the “index” entry of the vector to “value”
    - a custom destructor

# Array formats in C++

- We have the same 3 formats in C++ as in C
- replace 'malloc(sizeof(double)*m*n)' with 'new double[m*n]'
- Still isnt contiguous in memory

```
// Approach 1: flat array, column major
double * A;
A = malloc(sizeof(double) * m * n);
for (int j = 0; j < n; ++j){
  for (int i = 0; i < m; ++i){
    A[i + j * n] = (double) (i+j);
  }
}
```

# Array formats in C++

- replace 'malloc(sizeof(double*)*m)' with 'new double*[m]'
- Have to be careful indexing

```
// Approach 2: array of arrays (row major)
double ** B;
B = malloc(sizeof(double*) * m);
for (int i = 0; i < m; ++i){
  B[i] = (double*) malloc(sizeof(double) * n);
  for (int j = 0; j < n; ++j){
    B[i][j] = (double) (i+j);
  }
}
```

- replace 'malloc(sizeof(double*)*m)' with 'new double*[m]'
- replace 'malloc(sizeof(double)*m*n)' with 'new double[m*n]'

```
// Approach 3: array of arrays (row major)
double ** C = malloc(sizeof(double*) * m);

// allocate a vector of contiguous memory
C[0] = malloc(sizeof(double) * m * n);
for (int i = 1; i < m; ++i){
  C[i] = C[0] + i * n;
}
```

# Extra Exercise

Using the previous exercise

- Make a ”MyMatrix” class with
  - private members ”int m, n” and a ”double ** data” pointer to the matrix entries.
  - a custom constructor ”MyMatrix(int m, int n)” which initializes a matrix instance.
  - Implement the necessary functions to ensure that the Rule of Three is satisfied.

# Inheritance: classes and subclasses

- A powerful feature of C++ is that you can define a “derived” class which inherits features from another “base” class.
  - This inheritance can be private (default), public, or protected as well.
- A derived class inherits from the base class
  - All public and protected variables
  - All methods *except* for constructors, destructors, assignment operators, and friend methods/operators.

```
class BaseClass{
public:
  int x;
  int get_x(void){ return x; };
private:
  int y;
};

class DerivedClass :
  public BaseClass{
public:
  // inherits int x
  // inherits get_x
  void set_x(int val){
    x = val;
  }
};
```

* Derivedclass as ‘get_x’ member function, but BaseClass does not have ‘set_x’ member function

```cpp
class Base {
public:    // visible everywhere
  int x;
protected: // visible to derived classes,
           // member functions, "friends"
  int y;
private:   // visible only to member functions
  int z;
};

class PublicDerived: public Base {
  // x is public
  // y is protected
  // z is not accessible from PublicDerived
};

class ProtectedDerived: protected Base {
  // x is protected
  // y is protected
  // z is not accessible from ProtectedDerived
};

class PrivateDerived: private Base {
  // x is private
  // y is private
  // z is not accessible from PrivateDerived
};
```

# Multiple inheritance

- You can even inherit from *multiple* classes in C++!
- Same rules apply for multiple inheritance: a derived class inherits functions from all base classes, except for constructors, destructors, assignment operators, and friend methods/operators.
- We won't use this too much in this class.

# More on inheritance

```cpp
class Base1{
    // members and vaiables
};

class Base2{
    // members and vaiables
};

class DerivedClass : public Base1, public Base2{
    //inherits memebers and variables from Base1 and Base2
};
```

# Inheritance: a concrete example

```cpp
class Shape2D {
 public:
  Shape2D(void) {}

  // note: new constructor syntax!
  Shape2D(double w, double h) : width_(w), height_(h) {}

  void SetWidth(double w){ width_ = w; }
  void SetHeight(double h){ height_ = h; }
  double GetWidth(void) const { return width_; }
  double GetHeight(void) const { return height_; }

protected:
  double width_;
  double height_;
};
```

shape_example.cpp

```cpp
class Triangle : public Shape2D{
 public:
  Triangle(double w, double h) : Shape2D(w, h)

  double GetArea(void) {
    return 0.5 * GetWidth() * GetHeight();
  }
};
```

```cpp
class Rectangle : public Shape2D {
 public:
  double GetArea(void) {
    return width_ * height_;
  }
};
```

# Good coding habits

- Have all classes and derived classes in a header file
- This is useful for codes with many .cpp files which need the class

* Alternatively, all classes can be in a .cpp file, and a header file contains ‘forward declarations’

- For large projects, there may be multiple header files, one for classes, and one for function prototypes

# Exercise

- Make a class called 'Rect' with:
  - Public int variables 'length' and 'width'
  - member function 'get_area' that takes length and width as input and returns the area
  - custom constructor 'Rect(int l, int w)' that sets values of length and width
- Make a derived class of 'Rect' called 'Square'
  - custom constructor 'Rect(int l)' that sets values of length and width

# Issues with pointers to derived classes

- We can create a pointer to a base class from an instance of a derived class
- Suppose the derived class overloads a base class function. Which one should C++ run?
- Consider the Shape2D example. What will happen when I run “print_name”, and why?
    - C++ chooses which function to run at *compile* time.

```
class Shape2D {
 public:
  void print_name(void){
    cout << "This is a Shape2D\n";
  }
```

```
class Triangle : public Shape2D{
 public:
  void print_name(void){
    cout << "This is a Triangle\n";
  }
```

```
Shape2D * x = new Triangle(1.0, 2.0);
x -> print_name();
```

# Inheritance and virtual functions

- C++ introduced *virtual functions* to allow overloaded functions in derived classes to be chosen at *run time,* rather than at *compile time.*
  - Usage: append “virtual” before the function name, e.g., “virtual void print_name(void)”.
- Suppose again that we have “Shape2D * x = new Triangle(1.0, 2.0);” and that we run “x -> print_name()”. The C++ compiler isn’t smart enough to tell that “x” is really a “Triangle” type until it dereferences it at runtime
  - If we make “print_name” a virtual function, then “Triangle::print_name” gets run instead of “Shape2D::print_name”!

# Pure virtual functions

- Suppose we want to enforce that classes have to implement certain functions (e.g., define an interface). This can be done using “pure virtual functions”
  - Regular virtual functions can have a default implementation, which can be overridden by derived classes.
  - *Pure* virtual functions are defined with “=0” and do not provide a default implementation. Instead, C++ requires that any derived class provides an implementation for any pure virtual function.

```
class Shape2D {
 public:
  virtual void print_name(void) = 0;
```

# Exercise

- Make a 'Shape' Class and edit 'Rect' and 'Square to inherit from 'Shape'. Edit 'Shape' to include
  - public definitions of length and width variables (remove these from 'Rect')
  - pure virtual member function of 'get_area'
- Make a 'Shape' pointer, and set it to a 'Rect' instance, and again for a 'Square' instance.

# Operator overloading in C++

- Have you ever wondered how Matlab is implemented? You can do “b = A * x” where “b, x” are vectors and “A” is a Matrix.
- You’ve already seen the copy assignment operator; however, you can overload several other operators in C++ as well.
- Operators commonly use the keyword “this”, which is a pointer to the current instance of this class.
  - Rule of thumb: use “this” if an overloaded operator modifies a class member.

* So ‘*this’ dereferences to the current instance of the class

```
Foo & operator=(Foo &){
  cout << "calling the copy "
    "assignment operator\n";
  return *this;
}
```

# Overloading + for a “Point”

- Consider a simple “Point” class with coordinates “x” and “y”.
- If we want to add two points together, we can overload the “+” and “-” operators.
- Adding this overload allows users to add two instances of “Point” together.

```
Point a(1.0, 2.0);
Point b(2.0, 3.0);
Point c = a + b;
```

point_overloading.cpp

```
class Point{
public:
  Point(double x, double y){
    this->x = x;
    this->y = y;
  }
  double get_x(){ return x; };
  double get_y(){ return y; };
private:
  double x;
  double y;
};
```

```
Point operator+(Point a, Point b){
  return Point(a.get_x() + b.get_x(),
               a.get_y() + b.get_y());
};

Point operator-(Point a, Point b){
  return Point(a.get_x() - b.get_x(),
               a.get_y() - b.get_y());
};
```

# Overloading * for a “Point”

- If we want to multiply an instance of a Point with a “double”, we can overload “operator *”
  - Note that C++ doesn’t know that multiplication is a symmetric operation! We have to implement both “double * Point” and “Point * double” separately.
- Note that we did not explicitly overload “operator * (int a, Point b)”; we are relying on explicit conversion from “int” to “double” here.

```
Point operator*(double a, Point b){
  return Point(a * b.get_x(),
               a * b.get_y());
};

Point operator*(Point b, double a){
  return Point(a * b.get_x(),
               a * b.get_y());
};
```

```
Point a(1.0, 2.0);
Point b(2.0, 3.0);
Point c = 2 * (a + b);
```

# Overloading += for a “Point”

- What about overloading an accumulation operator?
  - “+=” should modify a Point instance instead of creating a new copy.
- Note: “operator +=” must be a *member function* to use the “this” keyword!
- Why return “Point &”? Not needed for “a += b” to work.
  - If we return a value, we can define the overloaded “+” operator in terms of “+=”.

```
Point & Point::operator+=(Point a){
  this->x += a.get_x();
  this->y += a.get_y();
  return * this;
}
```

```
a += b;
```

```
Point operator+(Point a, Point b){
  return a += b;
};
```

# Overloading indexing for a “Point”

- C++ allows you to overload indexing, e.g., specify the behavior of “x[0]”
  - Note: multi-dimensional indexing was only added in C++ 23; we will not cover it in this class.
- Question: this implementation is *read-only*. Can we implement an overloaded indexing operation that allows both read and write?
  - Yes! Just return *references* to member fields.

```
double & Point::operator[](int index){
  if (index==0){
    return this->x;
  }else if (index==1){
    return this->y;
  }else{
    cout << "Error: accessing "
      "beyond index 2\n";
  }
}
```

# Adding “ScaledPoint” to “Point”

- Suppose we derive from “Point” a “MyScaledPoint” class whose “x, y” coordinates are just “scaling * x, scaling * y” for some “double scaling” parameter.

```
class ScaledPoint : public Point{
public:
  ScaledPoint(double scaling, Point a) :
    Point(a.get_x(), a.get_y())
  {
    this->scaling = scaling;
  };
private:
  double scaling;
};
```

- What if we wanted to overload “Point & operator +=(ScaledPoint a)”?

```
class Point{
public:
  Point(double x, double y){
    this->x = x;
    this->y = y;
  }
  double get_x(){ return x; };
  double get_y(){ return y; };

  Point & operator+=(Point a);

  void print(){
    cout << get_x() << ", " <<
      get_y() << endl;
  }
protected:
  double x;
  double y;
};
```

# We have a circular dependency

- We have a member class of “Point” which depends on “ScaledPoint”.
- However, “ScaledPoint” is a derived class of “Point”, and can’t be defined without defining “Point” as well.
- How to resolve this circular dependency issue?

```
class Point{
public:
  Point(double x, double y){
    this->x = x;
    this->y = y;
  }
  Point & operator+=(Point a);
  Point & operator+=(ScaledPoint a);
```

```
class ScaledPoint : public Point{
```

# Forward declarations

- Pretty simple fix; use a *forward declaration* of ScaledPoint.
  - We can now reference “ScaledPoint” inside “Point”.
- Why does this work?
  - Forward declarations are basically the same as function prototypes but for classes (or structs).
  - Compiler just looks to see if a function/symbol is defined, and fills in the definition during linking.

```
class ScaledPoint;

class Point{
public:
  Point(double x, double y){
    this->x = x;
    this->y = y;
  }
  Point & operator+=(Point a);
  Point & operator+=(ScaledPoint a);
```

```
class ScaledPoint : public Point{
```

`forward_declaration_point.cpp`

# “const” with functions and classes

- You’ve seen the “const” keyword in C; it behaves similarly in C++ *except for functions and classes.*
- Suppose we have the “Point” example from before. What if we try to make a “const Point”?
  - We can create a “const” point, but C++ restricts the functions that we are allowed to call.
  - In this case, the indexing operator returns a reference, so it’s not a const function!

```
int main(void){
  const Point a(1.0, 3.0);
  cout << "a[0] = " << a[0] << endl;
}
```

This gives a compile error!

```
double & Point::operator[](int index){
  if (index==0){
    return this->x;
  } else if (index==1){
    return this->y;
  } else {
    cout << "Error, out of bounds\n";
  }
}
```

# Overloading “const” and “const correctness”

- How do we fix this? C++ actually allows you to overload functions based on the “const” keyword
  - C++ enforces “const correctness”: you’re not allowed to call functions that can mutate const objects (e.g., members of const classes).
- You can have both const and non-const versions of a function (common when overloading the indexing operator).

```
const double & Point::operator[](int index) const{
  if (index==0){
    return this->x;
  } else if (index==1){
    return this->y;
  } else {
    cout << "Error, out of bounds\n";
  }
}
```

```
double Point::operator[](int index) const{
  if (index==0){
    return this->x;
  } else if (index==1){
    return this->y;
  } else {
    cout << "Error, out of bounds\n";
  }
}
```

point_const.cpp

# Exercise

- Make a class called 'Vector' that:
  - has private variables 'a', 'b', 'c'
  - Make a custom constructor that inputs values double 'a', 'b', 'c'
  - Use 'this' to set the values in a custom constructor
  - has member functions 'get_a', 'get_b', 'get_c'
- overload the $+=$, $+$ operator for vector arguments
- overload the $*$, $+=$, $+$ operator for double argument

# Object oriented programming

- **Encapsulation:** grouping related data and functions together as objects and defining an interface to those objects
- **Inheritance:** allowing code to be reused between related types
- **Polymorphism:** allowing a value to be one of several types, and determining at runtime which functions to call on it based on its type
- We've actually already discussed all of these!

# Encapsulation

- Packaging related stuff together (in C++ this is with classes)
- Do not need to know how a class works in order to use it; all we need to know about is its public methods/data (its interface).
- The practice of hiding away these details from client is called "data hiding"

- We define what objects exist and what each one knows, and then the objects send messages to each other (by calling each other's methods) to exchange information and tell each other what to do.

# Polymorphism

- Polymorphism means “many shapes.” It refers to the ability of one object to have many types.

- Example: If we have a function that expects a Shape object, we can safely pass it a Square object, because every Square is also a Shape.
- Example: References and pointers, anywhere you can use a Shape*, you can use a Square*.

# Motivation for templates in C++

- C++ allows function overloading based on the types of input arguments.
- However, the argument *types* must always be specified at compile-time.
- Can lead to unnecessary “boilerplate” code and code repetition

```
int add(int a, int b);
float add(float a, float b);
double add(double a, double b);
```

```
class IntVector;
class BoolVector;
class FloatVector;
class DoubleVector;
```

In computer programming, **boilerplate code**, or simply **boilerplate**, are sections of code that are repeated in multiple places with little to no variation. When using languages that are considered *verbose*, the programmer must write a lot of boilerplate code to accomplish only minor functionality.[1]

The need for boilerplate can be reduced through high-level mechanisms such as metaprogramming (which has the computer automatically write the needed boilerplate code or insert it at compile time), convention over configuration (which provides good

https://en.wikipedia.org/wiki/Boilerplate_code

# Template basics

- Templates introduce the *template* keyword, as well as *template parameters* (in this case, “typename T”).
- You can specify the template parameter when calling a function
- More advanced behavior: you can *specialize* behavior for specific template parameters (e.g., when “T” is a specific type)

```
#include <iostream>

using namespace std;

template <typename T> T add(T a, T b) {
  return a + b;
}

int main(void) {
  int int_val = add<int>(2, 3);
  cout << "2 + 3 = " << int_val << endl;

  double dbl_val = add<double>(2.1, 3.0);
  cout << "2.1 + 3.0 = " << dbl_val << endl;

  // this converts from int to double implicitly
  dbl_val = add<double>(2, 3.5);
  cout << "2 + 3.5 = " << dbl_val << endl;
```

# Templated classes

- You can specify template parameters for classes as well as functions.
- This mostly just requires adding the “template” keyword when you need to refer to the template parameter (either “T” or “Type” in this code)

```
template <typename Type>
class Wrapper{
public:
  Type get_value();
  void set_value(Type value){
    this -> value = value;
  };
private:
  Type value;
};

template <typename T>
T Wrapper<T>::get_value(){
  return this -> value;
}

int main(void) {
  Wrapper<int> foo;
  foo.set_value(1);
  cout << "foo.value is ";
  cout << foo.get_value() << endl;
}
```

# Some odd linker behavior

- The usage of templates is relatively simple; however, templates can behave oddly when organizing code into .hpp and .cpp files.
- Suppose we break up our template code into .hpp, .cpp files and a driver file. What happens when we try to compile them together?
- Why does this happen?

foo.hpp

```
#include <iostream>
using namespace std;

template <typename T> void foo(T x);
```

foo.cpp

```
#include "foo.hpp"

template <typename T> void foo(T x){
  cout << "Running foo()\n";
}
```

main.cpp

```
#include <iostream>
#include "foo.hpp"
using namespace std;

int main(void) {
  foo<int>(1);
}
```

# Think of templates as *patterns*, not code

- Template function definitions cannot be defined in another file; you will end up getting a compilation error. Why does this happen?
  - Recall what C++ did with function overloading; it introduced “name mangled” versions of the functions which it substitutes into code.
  - C++ does something similar with templates: it finds what specific template parameters are needed and substitutes them into the template pattern.
  - However, this means that the compiler needs to be able to see the entire class definition to create this specialized class *before the linking step* - which is when different parts of code “see” each other!

See also https://isocpp.org/wiki/faq/templates#templates-defn-vs-decl

# Organizing template code

- Alternatives to “put everything in the header file anyways”:
  - Put template definitions in a .tpp file, which is included at the end of a .hpp file (commonly recommended)
  - Instantiate specific template parameter values (e.g., int) in .cpp files (less general).
- Putting everything into header files can increase compilation time, but this is hard to avoid with templates.

foo.hpp

```
#include <iostream>
using namespace std;

template <typename T> void foo(T x);

#include "foo.tpp"
```

foo.tpp

```
template <typename T> void foo(T x){
  cout << "Running foo()\n";
}
```

Instantiate specific template parameters

Including tpp files

# Organizing templated class code

- The .tpp / .hpp strategy is the same as before.
- We can similarly define specialized instances of templated classes in .cpp files is very similar.

```cpp
#include "wrapper.hpp"

template <typename T>
T Wrapper<T>::get_value(){
  return value;
}

template class Wrapper<int>;
```

wrapper.cpp

```cpp
template <typename Type>
class Wrapper{
public:
  Type get_value();
  void set_value(Type value){
    this->value = value;
  };
private:
  Type value;
};
```

wrapper.hpp

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

using namespace std;

int main(void) {
  Wrapper<int> foo;
  foo.set_value(1);
  cout << "foo.value is ";
  cout << foo.get_value() << endl;
```

# Exercise

- Make a template for the 'MyMatrix' class from a previous exercise (or any other)
- Put this into a .tpp file
- Make appropriate changes so that it compiles

# Templated function specialization

- Template parameters don't need to be types; can be numeric values as well.
- Can enforce specialized behavior for specific types or parameter values.
- C++ also allows "partial specialization" for classes (but not for functions).

```
#include <iostream>
using namespace std;

template <typename T, int n> void foo() {
  cout << "Running foo (no specialization)\n";
  cout << "n = " << n << endl;
}

template <> void foo<int, 0>() {
  cout << "Running foo<int, 0>\n";
}

int main(void) {
  foo<double, 1>();
  foo<int, 0>();
}
```

template_specialization_function.cpp

# Templated class specialization

- Can specify default values for template parameters
- C++ also allows “partial specialization” for classes (but not for functions).

```
int main(void) {
  Foo<double, 1> x;
  x.print();
  Foo<double, 0> y;
  y.print();
}
```

```
template <typename T, int n=1> class Foo{
public:
  Foo() {
    cout << "Foo(): full specialization\n";
  }
  void print(){
    cout << "Fully specialized printing\n";
  }
};

template <typename T> class Foo<T, 0> {
 public:
  Foo(){
    cout << "Foo(): partial specialization\n";
  }
  void print();
};

template <typename T> void Foo<T, 0>::print(){
  cout << "Partially specialized printing\n";
}
```

template_specialization_class.cpp

no specialization

# Template metaprogramming

- Template specialization enables tricks like compile-time computation. A common example is a recursive definition of the factorial function.
  - The “const static” keyword makes “result” accessible from the struct, e.g., “Factorial<n-1>::result”.
- Key observation: calling “Factorial<4>” doesn’t involve any (run-time) computations!
- Note: The “constexpr” keyword in modern C++ enables the same trick without template computing.

```
template <int n> struct Factorial{
  const static int result =
    n * Factorial<n-1>::result;
};

// specialization on n=0
template <> struct Factorial<0>{
  const static int result = 1;
};

int main(void) {
  int n = Factorial<4>::result;
  cout << "n = " << n << endl;
}
```

template_metaprogramming.cpp

# The C++ standard library

- C++ provides a huge amount of functionality in the Standard Template Library (STL); we'll just cover a few useful features.
- Templated containers and utilities (e.g., iterators):
  - Vector (expandable arrays). Provides sorting, slicing, etc utilities.
  - Pair, Tuples, Array (fast fixed size containers).
  - Map (associative arrays built from Pair types)
  - Set (implementation of an unsorted mathematical set). Provides union, set difference, intersection, etc utilities.

# C++ std::vector

- The “vector” type stores a list of elements which are all a single type.
- Includes utilities for sort, insertion, swapping, reversing, etc.
- Allows for the use of iterators, which we’ll go over later in more detail.

```
int main(void){
  vector<int> x = {1, 2, 3, 4};

  cout << "x is size " << x.size();
  cout << " before `push_back`\n";

  // see also insert, pop_back,
  // erase, swap, clear, etc...
  x.push_back(4);

  cout << "x is size " << x.size();
  cout << " after `push_back`\n";

  x.resize(10);

  cout << "x is size " << x.size();
  cout << " after `resize`" << endl;

  vector<int>::iterator i;
  for (i = x.begin(); i != x.end(); ++i){
    cout << *i << " ";
  }
  cout << endl;
}
```

# C++ std::pair, std::map

- Pairs are just of two elements of arbitrary type.
- Pairs are used to create “maps” (also known as associative arrays), which let you index by an arbitrary value.
- Similar to dictionaries in Python

```
#include <utility>
```

```
pair<int, char> x;
x.first = 100;
x.second = 'G';
cout << x.first << " ";
cout << x.second << endl;
```

```
map<string, int> x;
x["this"] = 1;
x["that"] = 10;

x.insert(pair<string, int>("pair", 20));
```

# The “auto” keyword

- With templated definitions, a type can be very long and complicated
- The “auto” keyword allows you to fill in the type of a variable - if it can be inferred from a function (e.g., based on the return type).

```
auto a = make_pair<double, pair<int, char>>(1.0, x);
```

# Makefile

- We can break up compilation for c++ the same as in c
- 'g++ -c file_1.c file_2.c' returns .o files
- 'g++ -o executable file_1.o file_2.o' returns an executable

- We can also use Makefiles!

# Makefile

- Same set up as in C
- replace .c, .h file type with .cpp, .hpp

```makefile
INC_DIR = ./include
SRC_DIR = ./src
OBJ_DIR = ./obj
CC = g++
CFLAGS = -I$(INC_DIR)

DEPS = $(wildcard $(INC_DIR)/*.hpp)
SRC_FILES = $(wildcard $(SRC_DIR)/*.cpp)
OBJ_FILES = $(patsubst $(SRC_DIR)/%.cpp, $(OBJ_DIR)/%.o, $(SRC_FILES))

all: hello_make hello_make_again

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.cpp $(DEPS)
	$(CC) -c $< -o $@ $(CFLAGS)

hello_make: $(OBJ_FILES) $(DEPS)
	$(CC) -o hello_make $(OBJ_FILES) $(CFLAGS)

hello_make_again: $(OBJ_FILES) $(DEPS)
	$(CC) -o hello_make_again $(OBJ_FILES) $(CFLAGS)

clean:
	rm -f hello_make $(OBJ_FILES)
```

# Cmake

- Cmake, introduced in 2014, has become popular
- Creates Make files for you

- Uses a file called 'Cmakelists.txt'
- To use Cmake, theres a few steps to do:
  - use the command 'cmake location' to create a Makefile at the specified location
  - creates a bunch of 'junk' files at the current directory as well as a Makefile at the specified directory

# Cmake

- First create a ’build’ directory, and cd into it
- Use the command ’cmake ..’ within the build directory to create a Makefile one directory up (outside of the ’build’ directory)

# Eigen: templated linear algebra

- You don't have to write your own linear algebra library from scratch. There are several options that provide Matlab or NumPy functionality.
  - Examples: Eigen, Armadillo, Blaze, Kokkos (I've used Eigen, Kokkos)
- Keep in mind that different libraries serve different purposes. Do you need
  - Eigen/Armadillo are Matlab-like.
  - Kokkos is more scalable but a larger and more complex library.

```
#include <iostream>
#include <Eigen/Dense>

using Eigen::MatrixXd;

int main()
{
  MatrixXd m(2,2);
  m(0,0) = 3;
  m(1,0) = 2.5;
  m(0,1) = -1;
  m(1,1) = m(1,0) + m(0,1);
  std::cout << m << std::endl;
}
```

# Installing Eigen

- Eigen is a *header-only* templated linear algebra library.
- Just download the Eigen files and provide a "-I*Eigen_directory*" flag to the compiler.
- Built for reliability, but still pretty fast. Can also be configured to use LAPACK to improve performance slightly.
- https://eigen.tuxfamily.org/index.php?title=Main_Page

eigen_example.cpp

```
#include <iostream>
#include "Eigen/Dense"

using namespace std;
using Eigen::MatrixXd;
using Eigen::VectorXd;

int main(void){
  MatrixXd A = MatrixXd::Random(3, 3);
  MatrixXd B = MatrixXd::Constant(3, 3, 1.2);
  A = (A + B) * 5;
  cout << "A =" << endl << A << endl;

  VectorXd v(3);
  v << 1, 2, 3;
  cout << "A * v =" << endl << A * v << endl;

  double vAv = v.transpose() * A * v;
  cout << "v' * A * v = " << vAv << endl;

  VectorXd x = A.lu().solve(v);
  cout << "residual = " << (A * x - v).norm();
  cout << endl;
}
```

# Eigen's optimizations

- Why is Eigen fast?
  - Well-designed templated classes with appropriate specialization
  - "Expression templating"
  - In-place matrix operations (e.g., factorizations) and lazy evaluations
  - Optional LAPACK, OpenMP (threads), GPU support

```
VectorXf a(50), b(50), c(50), d(50);
...
a = 3*b + 4*c + 5*d;
```

↓

```
for(int i = 0; i < 50; ++i)
  a[i] = 3*b[i] + 4*c[i] + 5*d[i];
```