fall 2021: computational science I

Module 6: CPP programming

history of the c++ programming language

- developed in 1979 by bjarne stroustrup at bell labs.
- it was called "c with classes" and can be viewed as an extension of the c language... with it you have the ability to create **objects**.
- 1985: c++ released to the public.
- compiler and libraries were shipped to anyone who wanted to use it... you just had to pay the shipping cost (in the era before the internet!)
- stroustrup was at texas a&m for a long time, as a professor of computer science. he now works in finance in nyc and is affiliated with columbia.

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

relationship of c and c++

the syntax for c and c++ are essentially the same.

c programs can most likely be compiled with a c++ compiler, with some exceptions.

there are additional keywords in c++ that are not available in c.

there are also certain type conversions that are not allowed in c++.

we can now have **classes** in c++. class are like a new data type, and are sort of like structs, but with classes you can define new *operators* for the corresponding objects and you have a notion of *inheritance*.

matrices $\leftarrow$ square matrices $\leftarrow$ symmetric matrices.

c++ also has new things like references, templating, function overloading, etc.

structure of a c++ program

source files with an extension `.cpp`.

header files with an extension `.h`.

some people use other extensions like `.cc` or `.hh`, it doesn't really matter.

as in c, you must have a `main` function in one of your source files that will be executed first.

header files as before contain function prototypes, structures, and also definitions of *classes*.

compilation is the same as with c source code, but you want to use a c++ compiler:

```
g++ main.cpp -o main
```

hello world!!

```
#include <cstdio>
#include <iostream>

int main(void){

   printf (" hello world with printf from a c++ program \n");

   std::cout << " hello world using iostream" << std::endl;

   return 0;

}
```

namespaces

the basic idea is to encapsulate variables, functions, pointers, and classes in a space with a single name.

these things allow us to have entities with the same names, because they "live" in different namespaces.

it is a nice way to organize large projects that may otherwise encounter naming conflicts.

a namespace is defined as follows:

```
namespace my_space{
   // entities definition or declaration
   double foo, blah;
   void print_stuff(void);
}
```

namespaces in practice

namespaces are usually defined in a separate header file, but can also be defined in source code outsize of any function.

```
namespace my_space{
   // entities definition or declaration
   double foo, blah;
   void print_stuff(void);
}
```

variables defined in namespaces are used outside of the namespace as:

```
my_space::foo = 3.14;
```

notice that function prototypes can also appear in namespaces, and their implementation will be done outside of the namespace like:

```
void my_space::print_stuff(void){
   // instructions
}
```

namespaces in practice

after including the namespace in your code or via a header file

```
namespace my_space{
   // entities definition or declaration
   double foo, blah;
   void print_stuff(void);
}
```

you can use the keyword `using`...

```
using namespace my_space;
```

after you have done this, you do not have to use the prefix "`my_space::`" to refer to the variables in there.

all standard c++ library things are defined within the `std` namespace.

c++ references... what are they?

references are a new kind of variable in c++. you should think about them as an *alias* (i.e. another name for...) to a previous defined variable.

they are declare with & following the data type.

```
// declare and initialize an integer
int ii = 5;

// declare a reference to an integer and initialize it
int& ref_ii = ii;
```

the reference `ref_ii` is an alias in the sense that if we modify its value, that change is reflected also in the value of `ii`, and vice versa.

more on references

references are not pointers. they differ from pointers in several ways:

- references must refer to an actual address in memory and cannot be `NULL`
- initialized references cannot be changes to refer to another object or variable.
- when a reference is created, it must be initialized.
- the value associated with a reference is accessed via the variable name... you **do not** need to "dereference" it with the * operator as with pointers.

some nice examples and descriptions here:

https://www.tutorialspoint.com/cplusplus/cpp_references.htm

an example using references

```
// declare and initialize an integer
int ii = 2;

// declare and initialize a reference to ii
int& ref_ii = ii

// change value, note that ref_ii will also equal 10.
ii = 10;

// change value again, note that ii will also equal 20.
ref_ii = 20;
```

references as function arguments

references are often passed as function arguments for at least two reasons:

- when passing by reference, function arguments are not copied, so it is more efficient.
- parameters, or function arguments, passed by reference, can be modified inside the function implementation, and this change is reflected outside of the scope of the function.

by the way, the `const` qualifier can be used with references if you want to make sure that you do NOT change the value.

function overloading

in c++, you can have functions with the same name but different parameter lists and these are treated as different functions:

```
void fcn_overload(int, double);
void fcn_overload(double, int);
void fcn_overload(int, int, int);
void fcn_overload(int, double, double);
void fcn_overload(int* , double*);
```

these are all considered different functions in c++ because the function naming generated in the object files depends also on the ordering and type of function arguments.

classes in c++

classes allow you to define a data structure that corresponds to a new type of variable.

this new variable is usually called an "object" of the class

distinction between classes in c++ and structures in c: classes can have functions associated with them. these functions are usually called "methods" of the class.

you will have **constructor** and **destructor** methods to build and delete objects, as well as possibly other methods that are specific to what the class is actually representing.

both the *data* contained in the class as well as the *methods* are called the class "members."

class definition

a header file contains the class declaration, or definition, like what we have below for a class called `MyVector`.

```
class MyVector {
  public:
  int length;
  double* vec;
};
```

a source file has the implementation of the methods, or functions, of the class.

an object of this class can be declared like:

```
MyVector foo;
```

the object foo can access all public members of the class using the dot operator:

```
foo.length = 20;
```

qualifiers for class members

`public`: can be accessed in the class and also by objects of the class.

`private`: can only be accessed within the class implementation.

`protected`: can only be accessed within the class implementation and also within *derived* classes.

the default qualifier is `private`.

members that are declared at `static` are shared by all objects created by the class.

for example, if we have

```
static double blah = 3.14;
```

then the member `blah` will equal `3.14` for all objects of the class.

class methods

to add a method to a class, you must do the following:

- add a function prototype to the class header file. if you want to access this function through an object *outside* of the class, it must be declared with the `public` qualifier.
- implement the function within the class's source file.

when you implement a class member in the corresponding source file, you need to add the classes name with the two colons "`::`"

so if we have a function that returns the length of a `MyVector` object, its implementation might look something like:

```
int MyVector::GetVectorLength(void){
   return length;
}
```

friend functions

let's say you have an implementation of our vector class that looks like:

```
class MyVector {
  int length;
  public:
  double* vec;
};
```

note that the member `length` is `private` and can only be accessed within the class. if you want to access it outside of the class, you need to make a `friend` function:

```
class MyVector {
  int length;
  public:
  friend int GetLength(MyVector& vec);
  double* vec;
};
```

friend functions

the implementation of the function `GetLength` might look like:

```
int GetLength(MyVector& vec){
   return vec.length;
}
```

so you can use it to get access to that `private` member!

constructors and destructors

a **constructor** function is executed when an object is created.

a **destructor** function is executed when the object goes out of scope.

the constructor function has the same name as the class, while the destructor function has a name that include the class name with a "$\sim$" in front of it.

```
// a constructor
MyVector::MyVector(){
   // instructions like memory allocation,
   // initialization, etc
}

// a destructor
MyVector::~MyVector(){
   // instructions like memory deallocation...
}
```

constructors and destructors

note that the constructor for a class can be overloaded, i.e. you can have multiple implementations depending on the data you have to set up the object.

```
class MyVector {
  public:
  int length;
  double* vec;

  // several constructor prototypes
  MyVector();
  MyVector(int nn);

  // destructor prototype
  ~MyVector();
};
```

constructors and destructors

```
// a constructor that sets a default value
// for the vector length
MyVector::MyVector(){
   length = 5; // a default value
   vec = (double*) calloc(length, sizeof(double));
}

// a constructor that allows the user to
// set the vector length value.
MyVector::MyVector (int nn){
   length = nn;
   vec = (double*) calloc(length, sizeof(double));
}

// destructor
MyVector::~MyVector (){
   if(vec != NULL){
      free(vec);
   }
}
```

pointer to a class

if you declare a pointer like

```
MyVector* ptr_vec;
```

you can then access class members with the "->" operator:

```
ptr_vec->length
```

note that the "`this`" keyword is important in c++. it refers to the pointer to the *current* object and is often used in the implementation of class methods. it can only be used by methods in the class.

overloading operators

you can "redefine" operators like multiplication, addition, "()," for objects of a given class.

the format would be as follows:

```
output_type operator operator_name(inputs){
   // instructions
}
```

for example, you define an operator that returns a writable reference to a vector element as:

```
double& operator()(int index){
   return vec[index];
}
```

and you can use this operator like:

```
MyVector vec1(4);
double d = vec1(0);
vec1(3) = 5.0;
```

overloading operators

here would be an implementation of vector addition:

```
MyVector operator+ (const MyVector& vec_in){

   MyVector vec_out(this->length);

   for(int ii = 0; ii < vec_out.length; ii++){
      vec_out.vec[ii] = this->vec[ii] + vec_in.vec[ii];
   }

   return vec_out;
}
```

the rule of three

you should go ahead and provide implementations of a destructor, copy constructor, and a copy assignment operator.

there might be unexpected behavior if you only implement some of these and rely on the default implementation for the others.

we discuss these operators and methods in the vector class example... please see the code on our class website.

more on c++ classes

we will first discuss class *inheritance*. this involves a *base* class and a class (or classes) *derived* from the base class.

why is this inheritance idea useful? the base class can contain methods and data that we would like to use in derived classes **without** having to reimplement them.

think about our example with matrices:

matrices $\leftarrow$ square matrices $\leftarrow$ symmetric matrices

we can have a base class for matrices that contains methods common to all types of matrices, while derived classes might contain additional methods specific to a given type of matrix.

derived classes

the syntax for implementing a derived class is:

```
class name_derived_class : set_inheritance name_base_class {
   // instructions
};
```

`set_inheritance` is called the "access specifier" and can be `public`, `private`, or `protected`. if not specified, it is `private` by default.

a derived class has access to the `public` and `protected` members of the base class, and the access specifier determines the possible access qualifiers for the base class' members within the derived class.

more on the access specifier

the possible values for `set_inheritance` give the following access to the member of the derived class. note again that only `public` and `protected` members are inherited.

`public`. inherited members maintain their same access.

`protected`. inherited members become `protected` in the derived class.

`private`. inherited members become `private` in the derived class.

the only data and methods *not* inherited are:

constructors, copy constructors, destructors, friend functions, and overloaded operators

constructors and derived classes

while constructor functions are not inherited in derived classes, **a constructor from the base class will still be called** when setting up the object from the derived class.

if you have a parametrized constructor, you can explicitly call it within the derived class using the following syntax:

```
Triangle::Triangle(double a, double b) : Box_2D(a, b){
   // stuff
}
```

where recall that the `Triangle` class is derived from the `Box_2D` class.

if you do not specify a constructor for the base class as above, **the default constructor for the base class will be called** as part of constructing the object for the derived class.

multiple inheritance

you can also derive a class from multiple base classes.

below, we derive a class from base classes `class1` and `class2` with access specifiers `inher1` and `inher2` respectively.

```
class derived_class : inher1 class1, inher2 class2 {

   // instructions

};
```

polymorphism

what does *polymorphism* mean in the context of computer programming? there are three different types:

- **ad hoc** polymorphism
- **parametric** polymorphism
- **subtype** polymorphism

generically, the word polymorphism means several types of "things" being accessible through a single interface.

see:

https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

ad hoc polymorphism

an example of this is function overloading. overloaded functions all have the same name, and possibly do the same thing, but they have different arguments (parameter lists) that distinguish them.

```
my_function(int, double);

my_function(double, char);
```

parametric polymorphism

imagine you want to create a class for matrices, but you want the programmer to specify the type for the elements of a matrix.

for example, elements could be either `int` or `double`.

the idea of *parametric polymorphism* is to have a single object depend on a "parameter" that might change depending on the needs of the programmer.

some programmers might only need matrices with `int` entries, while other programmers might want to deal with matrices with `double` entries, so make the element type be a "parameter" of the matrix class.

subtype polymorphism

the idea of *subtype polymorphism* arises when you have a base class and derived classes. you might have a method which has the same name in the base and derived classes, and the question is, what version of the method is called?????

consider the following example where class B is derived from class A, and they both have a method called `foo_method()`.

```
// pointer to an object of class A
A* ptr_A;

// declaration of an object of class B
B b;

// set pointer A to the memory address of object b
ptr_A = &b;

// what version of foo_method() is called here?
ptr_A->foo_method();
```

subtype polymorphism

```
// pointer to an object of class A
A* ptr_A;

// declaration of an object of class B
B b;

// set pointer A to the memory address of object b
ptr_A = &b;

// what version of foo_method() is called here?
ptr_A->foo_method();
```

if `foo_method()` is declared as a `virtual` function, `ptr_A->foo_method()` will call the implementation of this method in class B since `ptr_A` is a pointer to an object of the derived class B.

otherwise, `ptr_A->foo_method()` will call the implementation of this method from the base class A.

virtual functions

a function within a class should be declared using the keyword `virtual` if its implementation in the base class will be overwritten by a new implementation in a derived class.

the idea is that you can have a pointer to a *base* class that contains the address of an object of a *derived* class. then, when you call `virtual` methods using this pointer, the methods implemented in the derived class will be executed.

methods are declared virtual in the base class as follows:

```
class base_class {

   public:

   virtual output_type function_name(inputs);

};
```

virtual functions in c++11

from c++11 onward, there are additional qualifiers you can use with virtual functions.

- `final`... such a virtual function cannot be overwritten in a derived class.

  ```
  virtual output_type function_name(inputs) final;
  ```

- `override`... such a virtual function must be implemented in a derived class. otherwise, a compilation error results.

  ```
  virtual output_type function_name(inputs) override;
  ```

the `final` keyword can also be used to enforce that a class cannot be a base class for some derived classes:

```
class A final {
   // instructions
};
```

virtual inheritance

there can be some ambiguity if you have derived classes from derived classes. for example, consider the following:

```
class A{
public:
  int foo;
};
class B: public A{
};
class C: public A{
};
class D: public B, public C{
};
```

where class `D` is derived from both `B` and `C`.

virtual inheritance

note that the code below does not compile because it is not clear if we should use the definition of the variable `foo` from class `B` or class `C`.

```
int main(void){
  D d;
  d.foo=1;
  std::cout << "d.foo = " << d.foo << std::endl;
  return 0;
}
```

to resolve this ambiguity, we can force an object of class `D` to use the definition of `foo` from the base class. this is also done with the `virtual` keyword as:

```
class B: public virtual A{
};
class C: public virtual A{
};
class D: public B, public C{
};
```

pure virtual functions

classes can have functions that are **pure virtual**. such a function is not implemented for a class, but will be implemented in derived classes or must remain pure virtual.

objects **cannot be created** from a class that contains a `pure virtual` function... the lingo that is sometimes used here is that the class cannot be "instantiated."

the syntax is as follows:

```
virtual output_type function_name(inputs) = 0;
```

pure virtual functions

a pure virtual function can be implemented outside of the base class, in a source file, if you want to use such an implementation for derived classes.

derived class can then "implement" the pure virtual function by simply calling the implementation as follows:

```
output_type function_name(inputs){

   return base_class_name::function_name(inputs);

}
```

if `output_type` is `void`, then `return` should be omitted.

abstract classes

an abstract class:

- must contain a pure virtual function.
- might contain data members like integers or doubles.
- might contain methods and virtual functions.
- might have implementations of virtual functions *outside* of the class implementation (that will be used in derived classes).

```
class Box_2D {
   public:
   //pure virtual function
   virtual double Get_Area (void) = 0;
};
```

abstract classes serve as an "outline" for derived classes.

templating functions and classes

templating things is an example of **parametric polymorphism**.

we can template both functions and classes: the idea here is that we might want to have mutiple data types for functions or classes that are otherwise doing very similar things.

for functions, the syntax is:

```
template <typename T> output_type function_name(inputs);
```

or

```
template <class T> output_type function_name(inputs);
```

the identifier "T" can be anything you want.

templating functions and classes

you can also have several template parameters:

```
template <typename T1, typename T2>
output_type function_name(inputs);
```

here is an example of an addition function with a templated parameter:

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

this funtion can be called with integers or doubles:

```
int ii = add<int>(2.3, 3.9);
double d = add<double>(2.3, 3.9);
```

templating functions and classes

the syntax for templating classes is:

```
template <typename T>
class class_name{

   // declaration data members
   // definition method members

};
```

and you can create objects as follows:

```
class_name<int> foo;
class_name<double> blah;
```

templating functions and classes

we remark that the implementation of methods for a templated class should be done in the header file where the class is defined.

the syntax for implementing a methods of a templated class is:

```
template <typename T>
output_type class_name<T>::function_name(inputs){
   // instructions
}
```

we need to do this because the compiler needs to have direct access to the templated methods when creating an object with a given type. if the methods are implemented in another (source) file, the compiler might not be able to find them.

template parameters that are not data types

you can also have template parameters corresponding to values instead of data types:

```
template <typename T, int N>
   T add_N(T a){
   return a + N;
};
```

you can call this function like:

```
double d = add_N<double, 5>(3.1);
```

or, if you want to set the non-type parameter, it must be `const`:

```
const int n = 5;
double d = add_N<double, n>(3.1);
```

default non-type template parameters in c++11

you can have default values for template parameters that are not data types in c++11. the default value is used if a template parameter is not specified.

```
template <typename T, int N = 6>
   T add_N(T a){
   return a + N;
};
```

template specialization of classes and methods

you may want to have methods be defined or declared *depending on* the template argument used for instantiating an object of a class.

one clear application of this might be for using lapack functions. these functions have names that depend on the data type you are working with (set `http://www.netlib.org/lapack/explore-html/index.html`)

template specialization of classes and methods

suppose you have a templated class as follows:

```
template <typename T>
class foo_blah{
public:
    void print(void){
        std::cout<< "default function print" << std::endl;
    };
};
```

then you can specialize the `print` function for a given template parameter, in this case an integer, like:

```
template<>
void foo_blah<int>::print(void){
  std::cout << "hello from foo_blah<int>" << std::endl;
}
```

compiling c++ code with fortran (lapack and blas)

if you want to use some fortran functions like from lapack or blas in your c++ code, you will want to make sure that the function names are not "mangled."

recall that in c++ you can overload functions. this feature is achieved by mangling function names depending on their parameter lists.

in c, this is not possible and function names are **not** mangled. to keep certain function names not mangled when compiling c++ code, you will want to use:

```
extern "C" blah blah blah
```

where `extern "C"` precedes the function name.

dynamic memory allocation in c++

this is done with the functions `new` and `delete`.

you can still use the c functions `calloc`, `malloc`, `realloc`, and `free` in c++ code if you wish... you just need to include the `cstdlib` header file.

`new and delete` will call class constructors and destructors, and if memory issues arise, `new` will throw an exception that will stop the code.

recall that if something bad happens with memory allocation in c, a `NULL` pointer is returned and you must explicitly check for that.

dynamic memory allocation in c++

for example, if we want to allocate memory for a pointer to an object of type `foo`, then we would do:

```
foo* obj = new foo;
```

in the above example, the default constructor would be called.

if we include a parameter list as

```
foo* obj = new foo(par1, par2, par3);
```

the corresponding user-defined constructor would be called.

to call the destructor, we would do:

```
delete obj;
```

allocating memory for an array of variables

this is done using `new[]` and `delete[]`. to allocate memory for an array of ten integers, you would do something like:

```
int* ptr = new int[10];
```

and to deallocate that memory, you would do:

```
delete[] ptr;
```

when allocating an array of variables, it is not possible to call a user-defined constructor. instead, you would have to do it like this:

```
foo** ptr = new foo*[7];
for (int ii = 0; ii < 7; ii++){
   ptr[ii] = new foo(par1, par2, par3);
}
```

allocating/deallocating memory for a matrix

for example, if you want to allocate memory for a `double` array representing a matrix, stored in column major format, you could:

```
double** mat = new double*[num_columns];
for (int jj = 0; jj < num_columns; jj++){
   mat[jj] = new double[num_rows];
}
```

note that `mat[jj][ii]` corresponds to the matrix element in the `ii`*th* row and `jj`*th* column.

to deallocate, `delete` must be called in the **reverse order** from `new`:

```
for (int jj = 0; jj < num_columns; jj++){
   delete[] mat[jj];
}
delete[] mat;
```

note that the memory allocated here is *not* contiguous. how would you allocate a contiguous chunk of memory?

allocating/deallocating memory for a matrix

contiguous memory allocation would be done as follows:

```
double** mat = new double*[num_columns];
mat[0] = new double[num_columns*num_rows];
for (int jj = 0; jj < num_columns; jj++){
   mat[jj] = mat[0] + jj*num_rows;
}
```

deallocation is simpler:

```
delete[] mat[0];
delete[] mat;
```

exception handling in c++

this is a nice way to add runtime check to parts of your code. this is done with three keywords: `try`, `throw`, and `catch`.

- `try`: some code is put within a `try` block. during execution, the program checks to see if `throw` is encountered in the `try` block.
- `throw`: used to indicated a problem has occurred in the `try` block.
- `catch`: a `catch` block immediately follows a `try` block and is used for processing the problems that are "thrown."

exception handling in c++

```
#include <iostream>

int main(void){

   try{
      // code to try
      throw 13; // throw exception
   }

   catch (int e){ // catch throw of integer
      std::cout << " exception encountered :
      " << e << std::endl;
      return 1; // stop program
   }

   return 0; // no exception encountered

}
```

exception handling in c++

```
// user constructor
template <typename T>
my_matrix<T>::my_matrix(int Nrows, int Ncols){
  try{
    if (Nrows < 1 || Ncols < 1){
      throw " matrix dimensions do not make sense.";
    }
  }
  catch (char* problem){
    std::cout << " exception encountered:"
     << problem << std::endl;
  }
// more code follows below if the exception
// is not caught.
```

input/output in c++

we have been using this in examples... `std::cout` can be used to output information to the terminal. it is part of the `std` namespace and stands for "standard character output."

note that you do *not* need to include a format specifier as we had to do with `printf` in the c language. here is an example:

```
int ii = 6;
std::cout << "a random integer is " << ii << std::endl;
```

this would be equivalent to the c code:

```
int ii = 6;
printf("a random integer is %d\n", ii);
```

input/output in c++

```
int ii = 6;
std::cout << "a random integer is " << ii << std::endl;
```

note that "`std::endl`" corresponds to the endline character, so the above would be equivalent to:

```
int ii = 6;
std::cout << "a random integer is " << ii << "\n";
```

you can think about the "<<" (called the **insertion operator**) as concatenating things together (string, integers, doubles, ...), which are then "streamed" to the console specified by `std::cout`.

read from the terminal

we can also read info from the terminal, like the function `scanf` in the c language. consider the following example:

```
int foo;
std::cout << "type the value you would like for foo: ";
std::cin >> foo;
```

the above code will wait for your user input at the second line, at which point you can enter a value for `foo` in the terminal.

“>>” is called the **extraction operator**, and it is not clear what it does if you do not enter an integer or string...

read/write to a file

there are standard classes for doing file i/o in c++. these are:

- `std::ofstream`, class for writing to files.
- `std::ifstream`, class for reading to files.
- `std::fstream`, class for both reading and writing.

```
std::fstream file;
file.open(file_name, opening_mode);
// instructions
file.close();
```

read/write to a file

```
std::fstream file;
file.open(file_name, opening_mode);
// instructions
file.close();
```

the `opening_mode` can take a combination of the following values:

- `ios::in`, (input) open to read.
- `ios::out`, (ouput) open to write.
- `ios::trunc`, (truncate) if file has been opened to write, overwrite its contents.
- `ios::ate`, (at end) set stream position to end of file.
- `ios::app`, (append) if file has been opened to write, append to end of file.
- `ios::bin`, (binary) binary mode.

multiple values can be specified by separating modes with "|."

example for file i/o

.

see below how we open a file with multiple values for the `opening_mode`.

```
std::fstream blah_stream;
blah_stream.open("my_test_file.txt",
 ios::out | ios::app | ios::in);
// do some things
blah_stream.close();
```