CS 294-73
Software Engineering for Scientific Computing

Lecture 3
C++: References, User-defined Types, and Templates

Moving On to C++

- Rounding out the C type system: `ref, const`
- Classes: making user-defined types.
- Templates: types that are parameterized by other types.
- The Standard Template Library.

References

- References are closely related to pointers
  - They directly reference another object of the *same* type. Consider them as another name for an object.
  - A pointer is declared using the symbol for the dereference operator. A reference is declared using the symbol for the 'address of' operator.
  - Unlike a pointer, the reference must be defined with the object to reference and it cannot be changed later.

```
#include <cstdio>
int main(int argc, char* argv[])
{
  int* ptr; // declaring a pointer to an int
  int k;
  int &ref = k;
  ptr = &k;
  k = 2;
  printf("k has the value %d and is stored at %p\n", k, &k);
  printf("ptr points to the value %d stored at %p\n", *ptr, ptr);
  printf("ref has the value %d and is stored at %p\n", ref, &ref);
}
```

References: program output

```
>./a.out
   k has the value 2 and is stored at 0x7fff513195f4
   ptr points to the value 2 stored at 0x7fff513195f4
   ref has the value 2 and is stored at 0x7fff513195f4
```

- References will be especially useful when passing arguments to functions

The const qualifier

```
const int j = 1;
```

- This means j cannot be changed later in the defining scope.
  - The const qualifier aids the programmer in understanding the code
    - j is known to be 1 everywhere in the current scope.
    - j cannot be mistakenly changed later.
  - The const qualifier is a part of the type system.

```
j = 3;                    // Error: j is const
int k = 2;
const int& ref1 = j;      // Neither values of ref1 or j can be changed
const int& ref2 = k;      // k can be changed but not ref2
int& ref3 = j;            // Error: non-const reference to const
const int *ptr1 = &j;     // Neither *ptr1 nor j can be changed
ptr1 = k;                 // But we can still change what ptr1 points to.
int *const ptr2 = &k;     // Both k and *ptr2 can be changed.
ptr2 = &j;  //Error: the address pointed to by ptr2 cannot be changed
const int *const ptr3 = &j;  // Neither location nor value can be changed
```

Using references and const

```
float vmax(const float v[], const int& length, int& location)
{
  float max = v[0]; location=0;
  for(int i=1; i<length; i++)
    {
      if(v[i] > max)
        {max = v[i]; location = i;}
    }
  return max;
}
```

- How does this help us?
  - We can immediately see that `v` is not changed, `length` is not changed, while `location` could be modified.
  - References/pointers make it efficient to pass very large objects to functions. Make them `const` to indicate if they are input only and not modified.
- Questions
  - What would it mean to return a reference?

Is Array the same as Vector ?

```
>> v = [0:2:8]
 v =
 0  2  4  6  8
```

```
float v[]  =
{0,2,4,6,8};
```

- Matlab and C can make similar looking objects
- In terms of data, they are similar, but not in functionality
- A type is both its data, and the operations that manipulate it

```
>> 2*v

 0 4 8 12 16
```

```
2*v; // won't compile

error: invalid operands
of types 'int' and
'float [5]' to binary
'operator*'
```

How About multi-dimensional arrays?

```
>> A = [1 2 0; 2 5 -1; 4 10 -1]
A =
1 2   0
2 5  -1
4 10 -1
```

```
int A[3] [4] =
{8, 6, 4, 1, 9, 3,
1, 1};
```

- In terms of data, they are similar, but not in functionality

```
>>A*A
ans =

   5  12  -2
   8  19  -4
  20  48  -9
```

```
A*A; // won't compile
printf("%d",A); // nope
printf("%d",A[2][3]);  ?
```

Making your own types

- What about a 3D array ? Fancier tensors ? What if you are not doing linear algebra…but perhaps working on one of the other motifs.
- C++ lets you build *User-Defined Types*. They are referred to as a `Class`. Classes are a combination of member data and member functions. A variable of this type is often referred to as an object.
- Classes provide a mechanism for controlling scope.
  - `Private` data / functions: There are internal representations for objects that users usually need not concern themselves with, and that if they changed you probably wouldn't use the type any differently (recall the peculiar state of a `float`)
  - `Public` data / functions: There are the functions and operations that manipulate this state and present an abstraction for a user. (arithmetic operations on `float`)
  - `Protected` data / functions: intermediate between the first two, but closer to private than to public.

Trivial example

```
class Counter {
    private:
        int m_MyCounter ;
        int m_zeroEncounters;
    public:
         //null constructor
        Counter():m_MyCounter(0),m_zeroCrossings(0) {}
        void incrementCounter() {
           if(m_MyCounter==-1) m_zeroEncounters++;
           m_MyCounter++; }
        void decrementCounter() {
           if(m_MyCounter==1) m_zeroEncounters++;
           m_MyCounter--; }
        int getCounterValue() { return m_MyCounter; } const
};
```

- The object has it's data `m_MyCounter m_zeroEncounters`.
  - Mostly data members are made private.
- This class has a public interface, what a user of `Counter` would want to be able to do with this object.

Object-Oriented Programming

- Abstractions in programming are often described in terms of *data*, and *functions* that operate on that data.
- So, put them together in the language
  - Several languages have adopted this paradigm
    - C++, Java, Python, Fortran, C#, Modula-2
  - Some had it from the start, some found a way to make it work
    - You can do OOP in *any* language, just some make it easier
    - Even the latest Matlab has introduced user-defined classes
- The main benefits that makes bundling the two ideas together desirable.
  - Encapsulation
  - Modularity
  - Inheritance (will defer discussion)

Encapsulation

- Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state.
  - In Counter, how might a public data access result in an inconsistent state ?
- A benefit of encapsulation is that it can reduce system complexity, and thus increases robustness, by allowing the developer to limit the interdependencies between software components.
  - The code presents a contract for the programmer
  - If private data has gotten messed up, you don't have to look over your whole code base to determine where the mistake occurred.
  - The compiler has enforced this contract for you.

Modularity

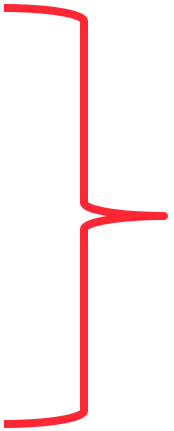
- Separation of Concerns
  - There is a small number of people that need to know how Counter is implemented.
  - There are a dozen people that need to know to use this class
  - There could be hundreds of people that just need to know this class is doing it's job.
- Improve maintainability by enforcing logical boundaries between components.
- Modules are typically incorporated into the program through interfaces. C++ makes this explicit with the `public` interface

A Simple Type: `Vector`

- So, Matlab has a vector type . Even 2nd generation languages like Fortran have powerful multi-dimensional arrays, as real types.
- So, Let's make one for us. using the new keyword `class`

A simple Vector class

```
class Vector //version one.
{
public: // ??
  ~Vector(); // Destructor.
  Vector(int a_dim); // Constructor.
  Vector(float a_start, float a_end, int a_elements);
  Vector operator=(Vector a_rhs);
  Vector operator*(float a_x);
  float  operator*(Vector a_rhs);
  float& operator[](int a_index);
  Vector operator+(Vector a_rhs);
  float norm(int ntype);
  Vector shift(int i);
private: // ??
  float *m_data;
  int m_size;
};
```



Other member functions.

Things to take notice of

- our { } scoping tokens are present here to show the compiler when we have finished defining our new type
- Inside this class scope we have declared several *member functions*
  - These functions define the abstract concept of the type (`int, float, double` are defined by what arithmetic does to them).
- There is a set of `public` functions that describe how the new type behaves
  - including how it interacts with objects of it's own type, and with other types. `operator` functions are special functions.

    ```
    float operator*(Vector a_rhs);
    ```

    - When two Vectors have the binary * operator between them, call this function, which returns a float.
  - There are a couple of odd functions that have the same name as the class
- There is a `private` section that represents how the class is implemented.
- The size of the data is known at compile time. Run-time sizing is mediated through `new / delete.`

Let's go off and use that new class

```
#include "Vec1.H"   // See us use include, has the definition
#include <cmath>  // another system include here, more libc
int main(int argc, char* argv[])
{
  int cells = 11;  float len=10;
  Vector x(0,len,cells);
  Vector u(cells);
  float dx=len/(cells-1);
  float m=0;
  for(int i=0; i<cells; m+=dx,i++)
    {
      x[i]= m; // Indexing is a member function - returns a reference.
      u[i] =sin(m/len*2*M_PI);
    }
  float nrm = u.norm(2); // calling the member function norm.
  float s = u*u; // looks symmetric, but it isn't.
 // float onevalue = (u.m_data)[0] Not allowed !!
}
```

Compile! and take stock

```
>g++ vecMain.cpp
Undefined symbols for architecture x86_64:
  "Vector::Vector(float, float, int)", referenced from:
      _main in cc7KJqdq.o
  "Vector::Vector(int)", referenced from:
      _main in cc7KJqdq.o
  "Vector::operator[](int)", referenced from:
      _main in cc7KJqdq.o
  "Vector::norm(int)", referenced from:
      _main in cc7KJqdq.o
```

- …we've skipped some steps here obviously. We told the compiler there was going to be some functions for `Vector`…but we didn't define them.

Obviously there is a bit more work to do

- Simply declaring a bunch of functions in a file `Vec1.H` wasn't the end of the job.
- `Vec1.H` is called a *declaration*, or *prototype*. We call a file with these in them a *header file* (hence the .H extension)
- What we are still needing is a *definition*. We put class and function definitions *in source files*. For C++ the convention is to use the .cpp file extension for source files.

- We need to make a `Vec1.cpp` file.

Vec1.cpp

```
#include "Vec1.H"
Vector::Vector(int a_dim)
{
  m_data = new float(a_dim);
  m_size=a_dim;
}
Vector::~Vector()
{
  delete[] m_data;
}
```

- ..and already we went and made things more complicated. Just two functions in!
  - odd named functions
  - :: token

Constructors, Destructors, Operators

- For built-in types the language defines constructors destructor and various operator functions
  - You don't get to change these functions.

```
{  //open scope
  float a; // compiler makes room for sizeof(float)
           // then float::float() called
  a = 5 //  float::operator=(int) called
}
//  'a' goes out of scope, float::~float() called
//  space for sizeof(float) reclaimed.
```

- Classes also need these functions
  - If you don't define these functions, the compiler will make them up for you. That might not be what you want to happen.

Recall from earlier

```
class Vector
{
public:
  ~Vector();
  explicit Vector(int a_dim);
  Vector(const Vector& a_rhs);
  float& operator[](int a_index);
  Vector operator+(Vector a_rhs);
  float norm(int a_ntype);
private:
  float *m_data;
  int m_size;
};
```

What would a double precision Vector look like ?

```
class DVector
{
public:
  ~DVector();
  explicit DVector(int a_dim);
  DVector(const DVector& a_rhs);
  double& operator[](int a_index);
  DVector operator+(DVector a_rhs);
  double norm(int a_ntype);
private:
  double *m_data;
  int m_size;
};
```

Need a new class name

Some types are changed

Otherwise, it's the same code
"Cut and Paste coding"

We don’t like Cut and Paste coding

- Usually you find bugs long after you have cut and pasted the same bug into several source files
- The once uniform interfaces start to diverge and users become frustrated
- Awesome new functions don’t get added to all versions of your class
- you end up with a LOT of code you have to maintain, document, and test.

Generic Programming

- Many languages have a mechanism to express a repeated pattern of code just once and let the language/runtime system/compiler deal with generating the needed variants.
- The concept pre-dates C++, but it was the introduction of templates into C++ in the early 1990's that brought the technique into wide use
  - All extant programming languages now support some form of generic programming.

Vector as a Template Class

```
template <class T> class Vector
{
public:
  ~Vector();
  explicit Vector(int a_dim);
  Vector(const Vector<T>& a_rhs);
  Vector<T> operator*(T a_x) const;
  T  operator*(const Vector<T>& a_rhs) const;
  T& operator[](int a_index);
  Vector<T> operator+(Vector<T> a_rhs);
private:
  T *m_data; // Must be able to call new / delete using T.
   int m_size;
 };
```

Using template <class T> class Vector

```
#include "Vector.H"
int main(int argc, char* argv[])
{
  int cells = 11;  float len=10;
  Vector<float> x(cells);
  Vector<float> u(cells);
  float dx=len/(cells-1);
  float m=0;
  for(int i=0; i<cells; m+=dx,i++)
    {
      x[i]= m;
      u[i] =sin((m/len)*2*M_PI);
    }
  float s = x*x;
  Vector<float> result = dx*(x+x)*(u*x)*m;
}
```

How about `double` instead of `float`

```
#include "Vector.H"
int main(int argc, char* argv[])
{
  int cells = 11;  double len=10;
  Vector<double> x(cells);
  Vector<double> u(cells);
  double dx=len/(cells-1);
  double m=0;
  for(int i=0; i<cells; m+=dx,i++)
    {
      x[i]= m;
      u[i] =sin((m/len)*2*M_PI);
    }
  double s = x*x;
  Vector<double> result = dx*(x+x)*(u*x)*m;
}
```

So where is `Vector<double>` being *defined* ?

- So far I have only shown you a *Declaration* of Vector<T>
- The compiler doesn't even know you need a `Vector<double>` until you start compiling `vecDMain.cpp`
- The compiler does not have enough information in a `Vector.cpp` file to build a `Vector.o` file.
- You can either let go of strong typing (build object files that don't know their types), or you can do what C++ does.
  - The generic *Definition* also goes into Vector.H

Template Definitions

- Your first excuse to put your definition in a header file
- Coding Standard
  - declaration at the top of the file, documented in doxygen-style
  - definition in the second part of the file
- inline functions are another example where you will put the definition in the header file.

Template Definition continued

```
  T *m_data;
  int m_size;
};
template <class T>
Vector<T> operator*(T a_x, const Vector<T>& a_y);
//  ==========Definitions ================
template <class T>
Vector<T>::Vector(int a_length)
{
  assert(a_length > 0);
  m_data = new T[a_length];
  m_size=a_length;
}
```

How can you use Vector ?

```
Vector<float> fvector;
Vector<int> ivector;
Vector<int> res = 5*ivector;
Vector<int> i2vector(fvector); //  ?
Vector<Box> bvector(5);
bvector[2] = Box(lo, hi);
bvector[2].grow(5);
Box d = bvector*bvector;    //    ?
```

Standard Template Library

- Wow, `Vector` is great. This will save me lots of coding
- In fact, so great that the `vector` class comes bundled with C++ already.
  - It is just a nice example to illustrate ideas.

- The STL has lots of stuff in it. We will use only a small subset of it here.

std::vector

```
#include <vector>
#include <cmath>
using namespace std;
int main(int argc, char* argv[])
{
  int cells = 11;  double len=10;
  vector<double> x(cells);
  vector<double> u(cells);
  double dx=len/(cells-1);
  double m=0;
  for(int i=0; i<cells; m+=dx,i++)
    {
      x[i]= m;
      u[i] =sin((m/len)*2*M_PI);
    }
   x=u;
  //double s = x*x;
  //vector<double> result = dx*(x+x)*(u*x)*m;
}
```

Without this, STL classes would have to be invoked `using std::vector<…>`, etc. See Namespace, C++ Standard Namespace

std::vector does not come with these defined

std::vector and std::string and std::cout (Stream I/O)

```
#include "VectorHelper.H"
#include <vector>
#include <string>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
  vector<string> strings;
  strings.push_back("string1");
  strings.push_back("string2");
  string extraString = strings[0] + " " +strings[1];
  strings.push_back(extraString);
  cout << strings << endl;
}
```

VectorHelper.H

```
#include <vector>
#include <iostream>
#include <cassert>
using namespace std;
template <class T>  T operator*(const vector<T>& a_1, const vector<T>& a_2)
{
  T ret = 0;
  assert(a_1.size() == a_2.size()); // what's this ?
  for(int i=0; i<a_1.size(); i++)
    {
      ret+=a_1[i]*a_2[i];
    }
  return ret;
}
template <class T> ostream& operator<<(ostream& a_stream, const vector<T>& a_vec)
{
  for(int i=0; i<a_vec.size(); i++)
    {
      a_stream << a_vec[i];
    }
  return a_stream;
}
```

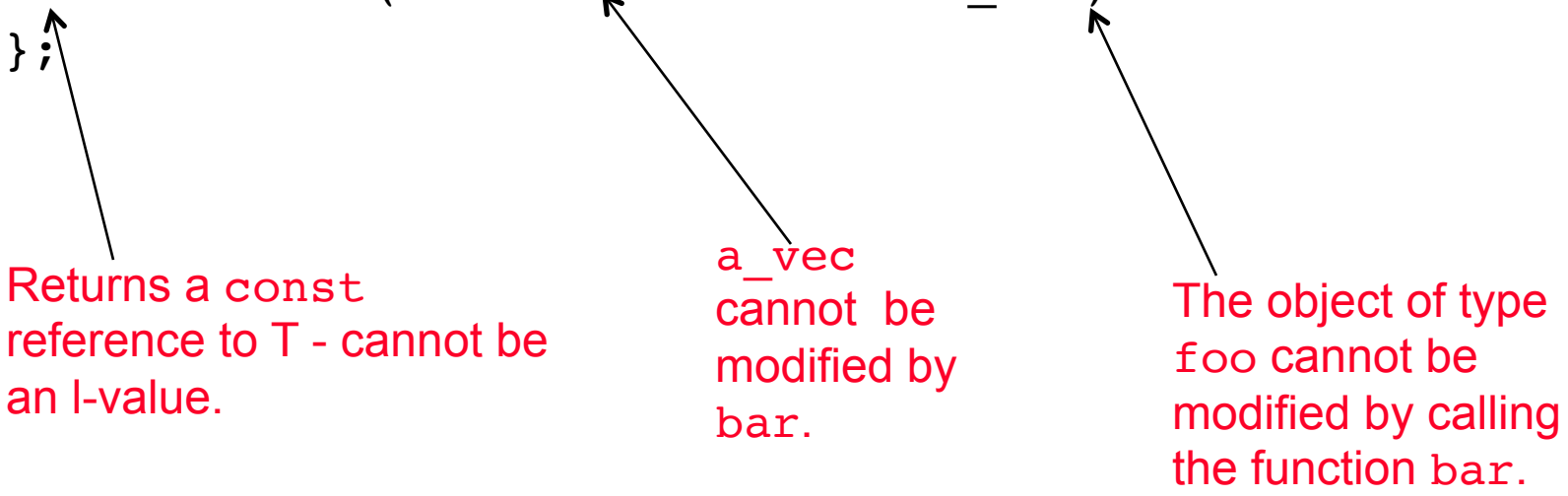
Templated functions / classes are only instantiated if they are called in a program. So it doesn't matter if operator()* is not defined on T, if you don't use it for that template class T.

and the results

```
>g++ stdstring.cpp
>./a.out
string1string2string1 string2
```

`const` and Classes

```
const T& foo::bar(const vector<int>& a_vec) const
{ … };
```




```
T& foo::bar(const vector& a_vec) const
{ … };
```

Is a different function (overloading). However, this kind of overloading can be hazardous, since the compiler will sometimes choose the wrong function. Nonetheless, there are circumstances you need both. Example: indexing.

What `const` means to a class is that it does not change the data members. What if the data members are pointers ?

Constructors and Destructors.

- Every class needs a constructor and a destructor. Constructors are called in order to create objects of a class (e.g. in declarations). Destructors are mostly called implicitly whenever an object goes out of scope.

- There is an implicitly-defined default constructor that calls the default constructor for all the data members (for POD, default construction is equivalent to declaration without any further definition).
  - `Vector<double> foo;` calls the default constructor.
  - `Vector<double> foo(m);` calls the constructor we defined.

- The destructor executes the body of the destructor, then execute the destructors of all member data. So a destructor of the form `~Vector::Vector(){};` just deletes the member data. If your member data includes a pointer to which you have applied `new`, you've just generated a memory leak. On the other hand, if your destructor is of the form `~Vector::Vector(){delete [] m_data;};` you've localized the management of memory to just getting the destructor right – when your `Vector` goes out of scope, the memory is released automatically.

Summary

- Classes – user-defined types.
  - Combination of member data and member functions.
  - Control of scope: `public / protected / private.`
  - Member data fixed-size, with run-time sizing of data mediated through pointers / memory management.
  - Power of composition: building up more complicated classes out of simpler classes.
- Template classes – types parameterized by other types, integers.
  - Template parameters completely defined at compile time. Integer parameters are integer literals (possibly defined by the C preprocessor).
  - Good for collections. Many of the operations on arrays, lists … don't depend on what the list elements are made up of.
  - Standard Template Library gives us rich collection of templated classes.
- Things we've used from the Standard Namespace: `<vector>, <iostream>, <cmath>, <cassert>, <string>`