C++ Part 2

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Half leftover topic: Auto variables
- String class
- Vector class
- Vector class with iterators
- Defining our own classes
  - Access restrictions
  - Class vs Struct

Auto Variables

- C++ is good at inferring types for things.
- In C++ 11, we can use the auto keyword to say "Just infer a type for this."
- We could use this because we're too lazy to write the whole type

```
long func( int i, const double &j )
{
  return (long) (i * j);
}
```

```
  auto a = 25;
  auto x = 3.45;

  auto f = func;

  auto c = f( a, x );
  cout << c << endl;
```

These still have specific types. We just asked the compiler to figure them out.

It even works for function pointers.

Auto Variables

- Why use auto?
  - Maybe we’re just lazy.
  - Well, that’s about it.

  - But some type names are really long and difficult to write down.
  - And sometimes it’s hard (for us) to figure out the exact type for an expression.
    - Especially when type parameters are involved.

Strings

- C++ supports a string type.

```
#include <string>
```

- String is an object.
  - But, you can use it as if was a fundamental type
  - Including using value semantics.

```
string a = “123”;
string b = “xyz”;
string c;
c = b;
c = a + b;
```

Assignment makes a deep copy.

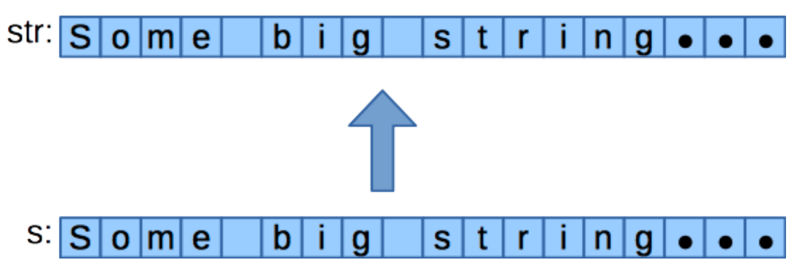
The + operator overloaded for concatenation.

Passing Strings

- C++ makes it easy to pass strings by value.
- This is great if it's what you need … if not, it can be expensive.

```
void f( string str ) {
  ...;
}
```

```
  string s = “some big string … “;
  f( s );
```

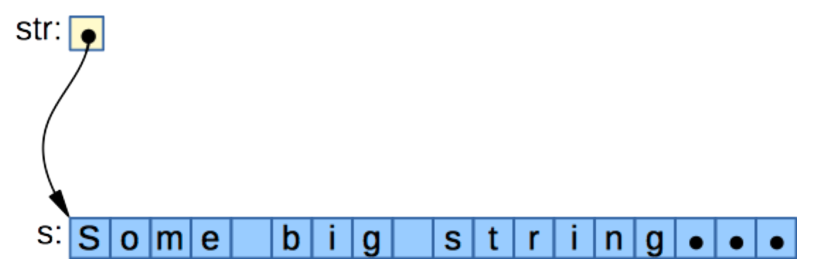


Passing Strings

- This can be a good job for reference parameters.
- … or const reference parameters.

```
void f( const string &str ) {
  ...;
}
```

```
  string s = “some big string … “;
  f( s );
```



String I/O

- The usual I/O operators work for strings.

```
cout << message << endl;
cin >> word;
```

- There are some new functions

```
getline( cin, line );
```

These will allocate more space as needed. So, less risk of buffer overflow.

Working with Strings

- Comparison and relational operators work:

```
if ( a == b )
  ...;

if ( a < b )
  ...;
```

Do these contain the same string?

Should string a be sorted before b?

- Strings have methods

```
a.length()
```

Length of the string given a value.

```
x->length()
```

Length of the string given a pointer.

- In C++, these are called member functions.

Accessing String Characters

- String have an overloaded index operator, for getting to individual characters.

```
for ( int i = 0; i < a.length(); i++ )
  cout << a[ i ];
```

- C++ strings are modifiable.

```
a[ 0 ] = 'x';
a += 'y';
```

Change a character.

Append a character.

String Member Functions

- Strings have lots of functions to help.
  - You use them like a Java method:

```
string b = str.substr( 5, 7 );
b.insert( 1, “abc” );
```

- Here are some of the more useful ones.

Function	Description
string substr(pos, len)	Create a substring of len characters, starting at pos.
size_t find(str)	Return starting index of the first occurrence of str.
insert(pos, str)	Insert str at position pos.
erase(pos, len)	Remove len characters starting at pos.

Strings vs Char Arrays

- A C++ string is an object. It's not the same thing as a character array.
- You can assign from a character array to a string:

```
string b;
b = "123";
```

- Or, you can make a string value from a char pointer, without even declaring a variable:

```
  string( "abc" )
```

This is really a C++ syntax for calling a constructor, to create a string from a char pointer.

Strings vs Char Arrays

- Inside every string, there is a null terminated char array.
- C++ strings have a method that will give it to you:

```
string str = “abc123”;
const char *p = str.c_str();
```

- To use some functions from C, we may need this

```
printf( “%s\n”, str.c_str() );
```

The STL

- C++ has the Standard Template Library (STL)
- It's a set of generic containers and algorithms, like the collections framework in Java
- We have objects that implement different types of containers:
  - Resizable arrays
  - Linked lists
  - Double-ended queues
  - Heaps
  - Balanced trees

The STL

- Each container defines member functions and operators for accessing / changing its contents
- Containers also offer *iterators*, a container-independent way to traverse the contents.
  - An iterator is a little object that works like a pointer to a value in a container.
  - In fact, in C++, iterators overload operators so we can use them like pointers.
- Many of the generic algorithms are written to work with iterators
  - So, they can work with many types of containers.

Meet vector

- We're going to learn about one STL container, vector
  - It has its own header:

```
#include <vector>
```

- It's the simplest, but it probably gets used 10 times as often as any other container.
- It's a wrapper for a resizable array, like ArrayList in Java.
- It has a type parameter, to tell it what it contains.

```
vector< int > iList;
vector< double > dList;
```

I'm an empty list of integers.

I'm the type parameter.

Adding Elements

- When you make a vector, you're calling its constructor.
- Vector has different constructors

```
vector< int > iList( 10 );
vector< short > sList( 20, -1 );
```

10 elements initialized to zero.

20 elements initialized to -1.

- After construction, the push_back() member function lets you add values to the end.

```
for ( int i = 0; i < 10; i++ )
   iList.push_back( i );
```

Iterating and Accessing Elements

- Vector has a size() member function, reporting how many elements it currently contains.
- And, an overloaded index operator, to get/set individual elements.
  - You can get elements:

```
for ( int i = 0; i < iList.size(); i++ )
   cout << iList[ i ] << endl;
```

  - Or change them:

```
for ( int i = 0; i < iList.size(); i++ )
   iList[ i ] *= 2;
```

Removing Elements

- Vectors have a pop_back function, to remove elements from the end.

```
while ( iList.size() ) {
    cout << iList[ iList.size() – 1 ] << endl;
    iList.pop_back();
}
```

- There are other member functions to:
  – Insert/remove elements anywhere
  – Get the first/last element

Vector and Value Semantics

- Vector supports value semantics

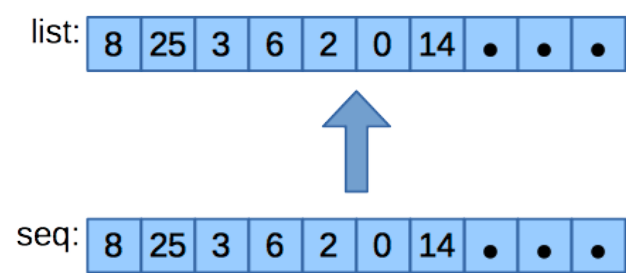
```
vector< int > otherList = iList;
```

This makes a deep copy of the whole vector.

- So, it's easy to pass copies of vectors to functions … maybe too easy.

```
void f( vector< int > list ) {
  ...;
}
```

```
  vector< int > seq;
  seq.push_back( 8 );
  ...;
  f( seq );
```



Vector and Value Semantics

- This can be another good chance to pass by reference

```
void f( vector< int > &list ) {
  ...;
}
```

I'd like to see your vector for a while.

- ... or pass by const reference

```
void f( const vector< int > &list ) {
  ...;
}
```

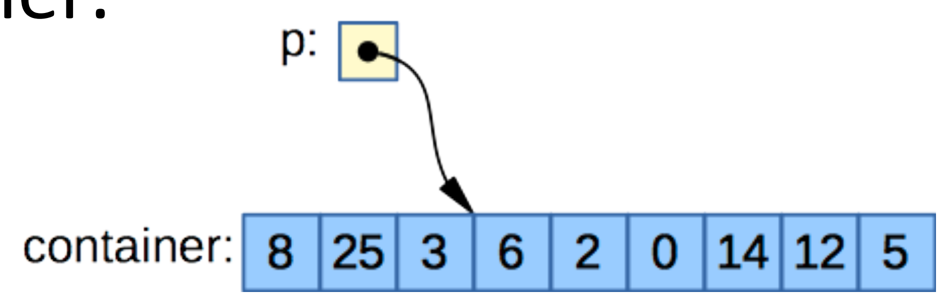
I'd just like to see your vector for a while, but I promise not to change it.

Iterators

- With vector, we can use the index operation to access individual elements:

```
list[ i ]
```

- But not every container has this. For some, we have to use iterators.
- An *iterator* is an abstraction for a pointer to an element of a container.



Getting Iterators

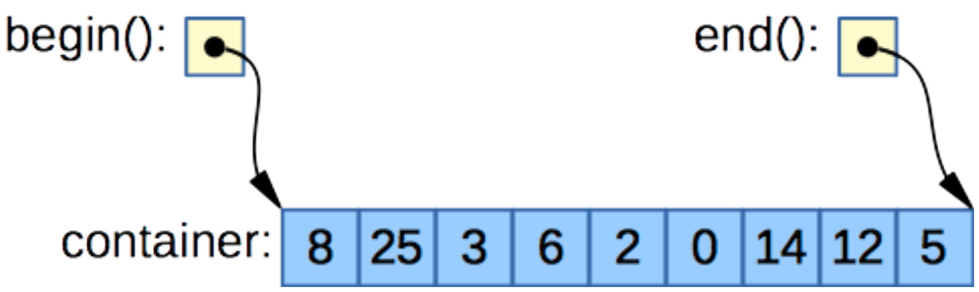
- The type of an iterator is defined inside the class it iterators over.

```
vector< int >::iterator p;
vector< double >::iterator p2;
```

I can iterate over vectors of ints.

Well, I can iterate over vectors of doubles.

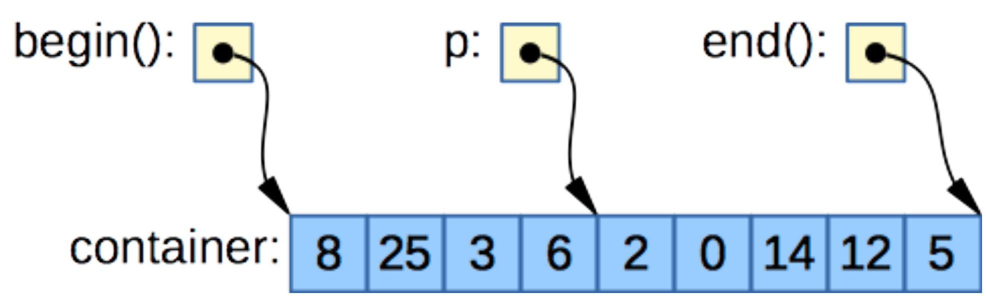
- Containers have several member functions pertaining to iterators
  - begin() : returns an iterator pointing to the first element.
  - end() : returns an iterator point one past the last element.



Using Iterators

- For an iterator, we can use pointer syntax to:
  - Move an iterator around in a container.
  - Get/modify the value it points to.
- So, this is an alternative for traversing a vector:

```
vector< int >::iterator p;
for ( p = container.begin(); p != container.end(); p++ )
   *p += 1;
```



Using Iterators

- Container and iterator types can get ugly.
  - Auto can save us some typing.

```
for (auto p = container.begin(); p != container.end(); p++)
  *p += 1;
```

- C++ 11 has a new syntax for iterating over lots of things:

```
for ( int v : container )
  cout << v << endl;
```

- We can even change the container as we go.

```
for ( int &v : container )
  v++;
```

Generic Algorithms

- There are generic algorithms that work via iterators:

```
#include <algorithm>
```

- There are simple functions:

```
reverse( container.begin(), container.end() );
```

Just give us iterators for the range of values you want us to work with.

- And more interesting ones:

```
sort( container.begin(), container.end() );
```

Classes

- C++ is object-oriented
  - We can define classes, with data and member functions
  - Classes can have constructors that automatically get called when the class is instantiated.
  - Classes can have destructors, that clean up when an instance is freed.
- A class definition looks almost like a struct. (in C++ it's almost the same thing)

First Classes

Defining a new class.

The following stuff is public, code outside the class can access it.

Every instance of this class has these fields.

A member function .

```
class Person {
public:
  string name;
  int age;

  void print() {
    cout << "Name: " << name << " age: " << age << endl;
  }
};
```

As with struct, you can start making instances here, if you want.

Making Instances

```
class Person {
public:
  string name;
  int age;

  void print() {
    cout << "Name: " << name << " age: " << age << endl;
  }
};
```

```
  Person bill;
  bill.name = "bill";
  bill.age = 22;

  Person mary = { "mary", 24 };

  bill.print();
  mary.print();
```

Great. In C++, we get a new type name, not just a tag.

If the fields are public, we can initialize them like a struct.

Calling a member function looks a lot like Java.

Keeping Secrets

- C++ offers public, protected and private access restrictions.
  - It uses a different syntax to change access
  - If we encapsulate, we're going to need public constructors and other member functions to access fields.

```
class Person {
private:
   string name;
   int age;

public:
   void print() {
      ...;
   }
};
```

The following members are private.

Now, some public members.

class vs. struct

- In C++, there's only one difference between class and struct, the default access restrictions.

```
class Person {
  string name;
  int age;

public:
  void print() {
    ...;
  }
};
```

I default to private.

But I'm public.

```
struct Person {
  void print() {
    ...;
  }

private:
  string name;
  int age;
};
```

We can both have fields and member functions.

class vs. struct

- But, there are conventions
  - Use class when you expect encapsulation
  - Use struct when you don’t.
- Classes will have constructors and lots of member functions
- Structs may not have any (but they can)