Meet C++

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- I/O in C++
- Overloading
- Default parameters
- Namespaces
- References
- Template intro
- Auto variables
- C++ string type

Meet C++

- C++ is just like C
  - With a bunch of extra stuff
  - Most C programs will compile fine under C++
- C++ adds object-orientation to C
- Along with many other features
  - Some of which eventually found their way back into C.
- C++ tends to round out the language
  - Lets you define new types that behave just like built-in types.
  - Let's hear from its creator, Bjarne Stroustrup

Compiling with C++

- As with C, the C++ standard continues to be refined
  - We'll be targeting the 2011 standard, C++11
- We'll need a different compiler, with a new option for this standard:

```
$ g++ -Wall -std=c++0x hello.cpp -o hello
```

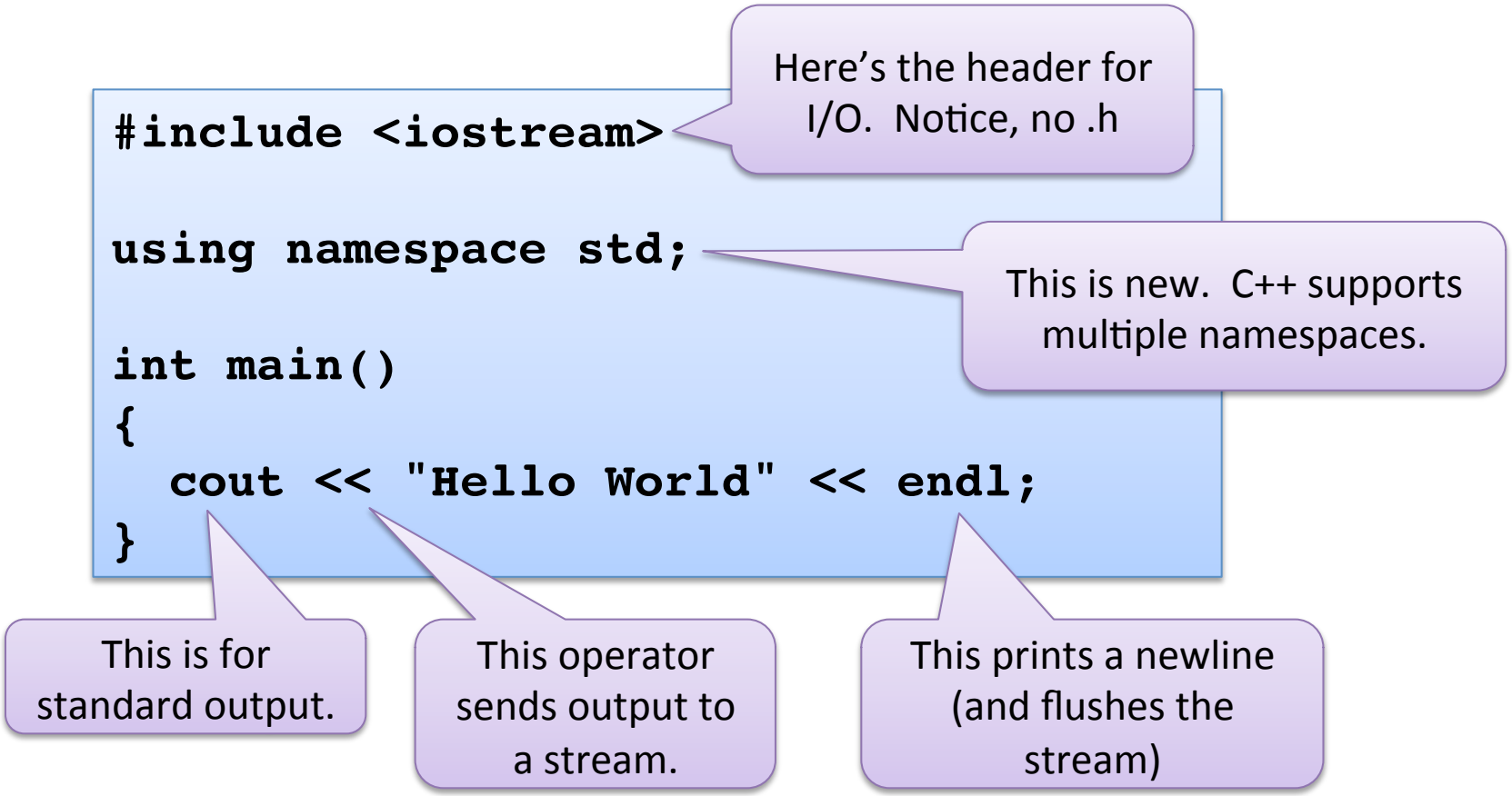
I'm your C++ compiler.

I'm the standard we'll be programming to.

This is an older option, but we have an older compiler.

Hello C++

- Here's our first C++ program, traditional.



Hello C

- The whole C standard library is still there.

```
#include <cstdio>

using namespace std;

int main()
{
  printf( "Hello world!\n" );
}
```

The header names got a make-over.

But there's our good friend, printf().

Using C Headers

- The C headers are still there, but you have to convert the names to a new naming scheme.
  - Remove the .h at the end.

```
#include <stdio.h>
```

  - Put a ‘c’ in front.

```
#include <cstdio>
```

- It’s kind of like pig Latin … only backward.

I/O In C++

- From the iostream header, we get three new objects.
  - **`cin`** : for reading input
  - **`cout`** : for writing output
  - **`cerr`** : for writing to stderr
- The left and right shift operators are overloaded for performing I/O
  - They read/write appropriately for their parameter type.
  - They associate left-to-right; you can stack them up like this.

If i is an int, print an int.

If f is a float, print a float.

```
cout << i << " " << f << " " << word << endl;
```

```
cin >> i >> f >> word;
```

These skip whitespace.

If word is a char pointer, read a word.

Are We Good?

- C++ streams have a status you can check.
  - Just by evaluating the stream itself.
  - It evaluates to true if the last read was successful.
- So, this gives us a common way to read values until we run out of inputs.

```
cin >> val;
while ( cin ) {
  ... // do something with val.
  cin >> val;
}
```

Are We Good?

- Or, the following evaluates to the stream itself:

```
cin >> val
```

I read a value and evaluate to cin.

- That's why we can stack up uses of >> in one big expression.

```
cin >> a >> b >> c >> d;
```

- We can use this to write even shorter code to read a list of values until we hit EOF.

```
while ( cin >> val ) {
   ... // do something with val.
}
```

Reading and Writing Files

- The same techniques let us work with files.
- We need a new header.

```
#include <fstream>
```

- This lets us make new streams:

```
ifstream input( "in.txt" );
```

```
ofstream output( "out.txt" );
```

- We can use these just like cin/cout.

```
while ( input >> val )
  output << ( val + 1 ) << endl;
```

- These are objects. Their files get closed when they go out of scope.

Overloading

- C++ lets us overload functions … and operators
- We can have multiple functions with the same name
  - The parameter types decide which function you get.
- The standard operators can do different things
  - Like with **<<** and **>>** for doing I/O.
  - The operand types decide which version of the operator you get
  - You can define new things for the usual operators to do when you use them with types you define.

Default Parameters

- C++ lets us specify default parameter values

```
void repeat( char const *str = "Good Evening",
             int count = 10 )
{
  for ( int i = 0; i < count; i++ )
    cout << str << endl;
}
```

The caller has to see this, so it could go in a header.

- These values are filled if the caller omits parameters:

```
repeat( "Good Morning", 5 );
repeat( "Good Afternoon" );
repeat();
```

So, it looks like we have multiple versions of the same function.

Namespaces in C++

- C++ is made for building larger programs.
  - Types, functions and global variables don't need to compete for the same namespace
  - Standard language features are defined in the std namespace.
- The scope resolution operator :: lets us specify the namespace/scope for an identifier.

```
#include <iostream>

int main()
{
  std::cout << "Hello World" << std::endl;
}
```

using namespace std;
This let us use names like cout without specifying their namespace.

Without it, we have to say the namespace every time.

Namespaces in C++

- We define our own namespaces.

```
int val = 3;

void f() {
  printf( "val: %d\n", val );
}

namespace MyNamespace {
  int val = -10;

  void f() {
    printf( "val: %d\n", val );
    printf( "global: %d\n", ::val );
  }
}
```

This will print 3.

This will print -10.

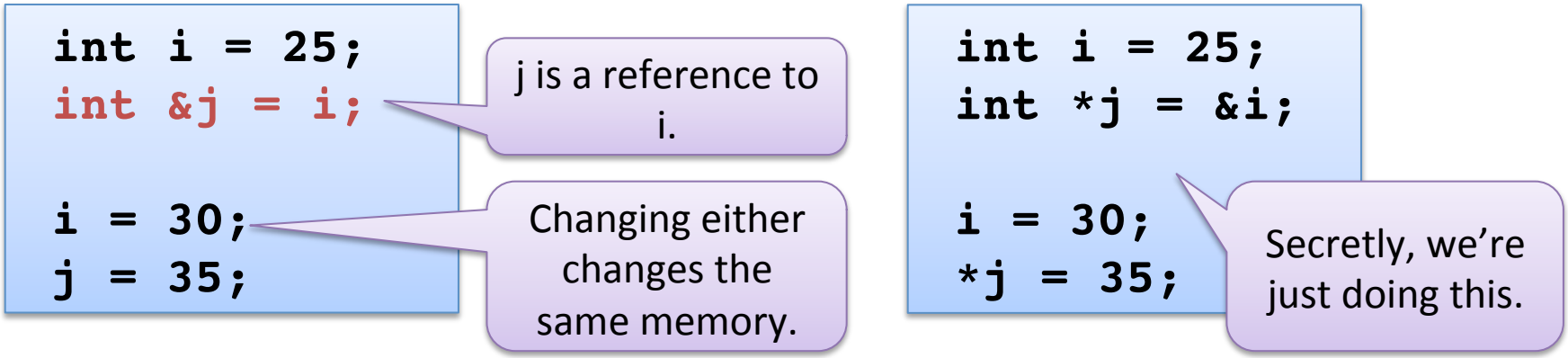
We can use the scope resolution operator to say which val we want.

```
  f();
  MyNamespace::f();
```

Same here.

References

- We have a new type operator, **&**, for defining references.
  - It's a way to get another name for existing storage.
  - References must be told what they refer to at declaration time.



  - Really, it's just syntactic sugar for pointer syntax.
  - Why is this useful?

Using References

- References simplify pass-by-address syntax.
  - We can define a function that takes references.

```
void exchange( int &a, int &b )
{
   int temp = a;
   a = b;
   b = temp;
}
```

In here, we don't need the pointer dereferencing syntax. It's automatic.

  - The caller gets pass-by-address without any special syntax.

```
   exchange( x, y );
```

I'm passing the ability to modify these variables.

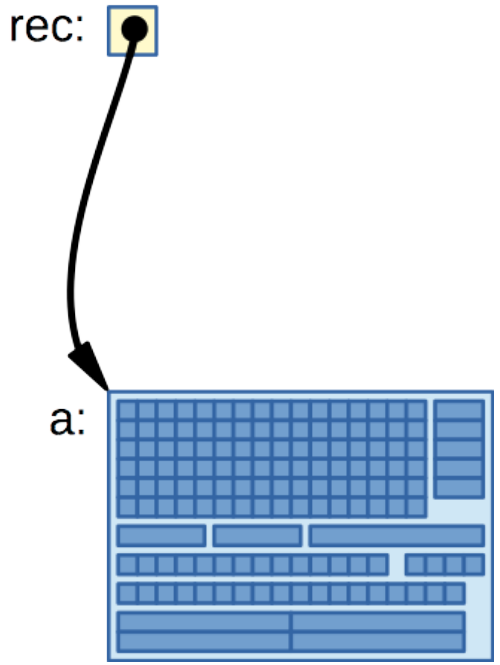
Using References

- References let us avoid copying objects, without any special effort from the caller.
  - Good. Some objects can't or shouldn't be copied.

```
int f( const struct BigRecord &rec )
{
  int sum = rec.x + rec.y;
  return sum;
}
```

```
  struct BigRecord a;
  ...
  v = f( a );
```

I don't need to think too much here.



Using References

- References let us return values that are understood to be modifiable.
  - Containers use this feature a lot, to let client code use and modify the contents.

```
int &lookup( char *name )
{
   ...;
}
```

mary	phillip	johnny	bob
42	25	8	17

Imagine we're returning a reference to a field in some structure.

```
   int x = lookup( "mary" );
   lookup( "bob" ) = 35;
```

You can just use the value returned.

Or, you can use the reference to change a value.

Using References

- Even within a single function, references can be convenient.
  - We can make a short, temporary name for accessing something complex.

```
int &x = some[ i ].complex->thing[ j ];
...
cout << x << endl;
x = 18;
```

I'm not a copy of this thing. I'm a shortcut to its memory location.

Templates

- C++ has a template mechanism
  - It's kind of like generics in Java
  - Except, C++ instantiates different versions of the template code for each type parameter it sees

```
template< typename T >
void exchange( T &a, T &b )
{
  T temp = a;
  a = b;
  b = temp;
}
```

We're saying how to write a function, for some type T.

```
int a = 25, b = 7;
double x = 3.45, y = 6.78;
exchange< int >( a, b );
exchange( x, y );
```

When you call the function, you can fill in the value of the type parameter.

Or not, the compiler may be able to infer it.

More about Templates

- You can define functions and structs/etc. with templates
  - The standard library in C++ uses this a lot.
  - Kind of like what Java does with the collections framework.
  - In fact, we called the previousfunction exchange() because there was already a template swap() function.
- A template declaration covers just the next definition
- You can have multiple type parameters for the same definition.

```
template< typename T, typename V >
void someFunction( T &a, V *b )
{
  ...;
}
```

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.

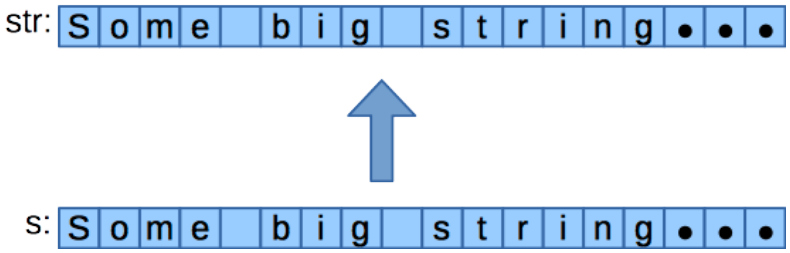
The + operator overloaded for concatenation.

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 );
```

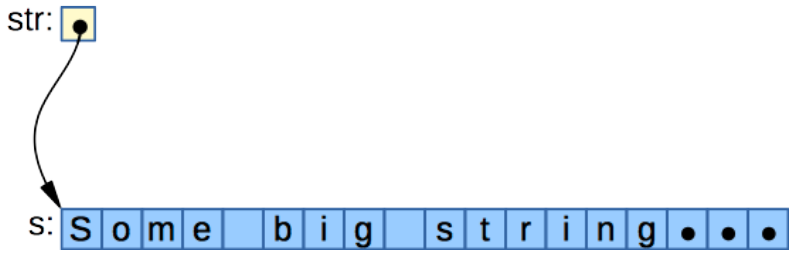


Passing Strings

- 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 ] += 1;
```

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

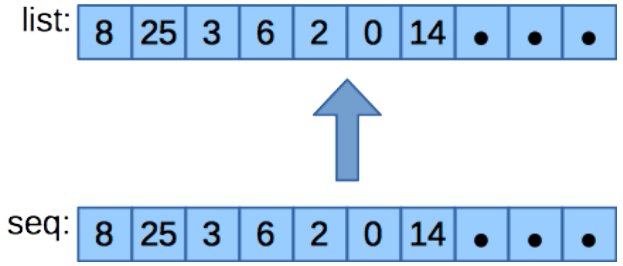
```
vector< int > otherList = iList;
```

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.