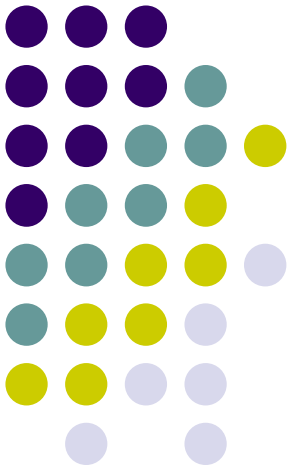
C++

February 19, 2013

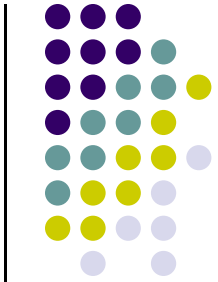
Mike Spertus

mike_spertus@symantec.com

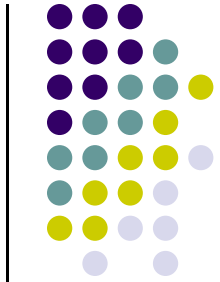
YIM: spertus



g++ 4.7 on the cluster

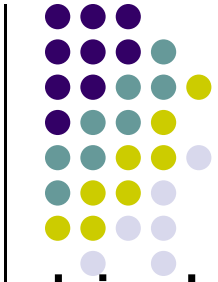


- California now has g++ 4.7
- Other machines have 4.6, which is usually OK, but as we saw last week, can cause problems.
- Either should be OK for this week
- We will try to get more machines up shortly



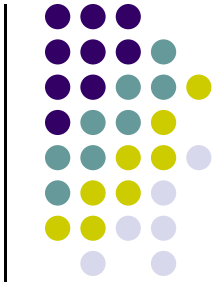
MOVE SEMANTICS

Rvalue references



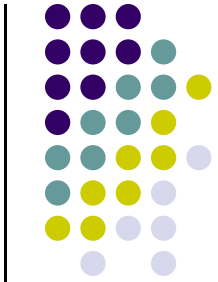
- A reference with “&&” instead of just “&” can bind to a temporary and move it elsewhere.
- Objects are often much cheaper to “move” than copy
- ```
  template<class T>
  void swap(T& a, T& b)// "perfect swap“(almost)
  {
    T tmp = move(a); // could invalidate a
    a = move(b); // could invalidate b
    b = move(tmp); // could invalidate tmp
  }
  ```
  ```

Move semantics example: putting threads into an array



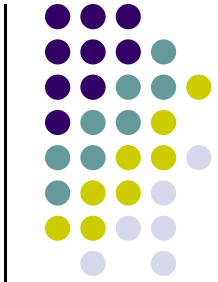
- Recall that std::threads are not copyable
- Since they are movable, we can construct a temporary thread and move it into a vector
- ```
  template<typename T> class vector {
    ...
    push_back(T &t);
    push_back(T &&t);
    ...
  };
  ```
- ```
  vector<thread> vt;
  for(int i = 0; i < 10; i++) {
    vt.push_back(thread(f, i));
  }
  ```

“Rvalue reference references”



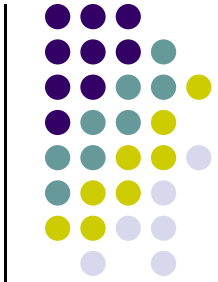
- Here are some useful references on rvalue references
- http://thbecker.net/articles/rvalue_references/section_01.html
  - What I lectured from in class
- http://blogs.msdn.com/b/vcblog/archive/2009/02/03/rvalue-references-c-0x-features-in-vc10-part-2.aspx
- http://www2.research.att.com/~bs/C++0xFAQ.html#rval

How do I make a type movable?



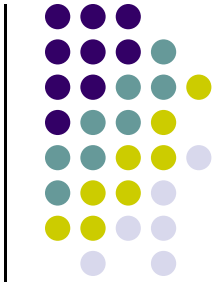
- ```
  template<class T> class vector {
    // ...
    vector(vector<T> const &); // copy constructor
    vector(vector<T> &&); // move constructor
    vector& operator=(const vector<T>&); // copy
  assignment
    vector& operator=(vector<T>&&); // move assignment };
  // note: move constructor and move assignment takes
  // non-const && they can, and usually do, write to
  // their argument
  ```
- In, C++11 all containers have move constructors, and versions of insert, push_back, etc. taking rvalue references, improving performance because they copy less
- Move constructors also allow a “non-broken auto_ptr” called unique_ptr that can be stored in an STL container and more efficient return managed objects
  ```

auto_ptr, unique_ptr and rvalue references



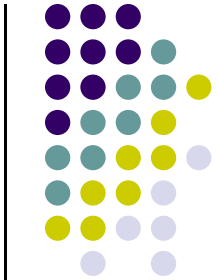
- While auto_ptr is good for RAII, it is not good for other “single-ownership” purposes
- For example, auto_ptr’s cannot be put in an STL container
- C++11 deprecates auto_ptrs and defines a new unique_ptr type that has a move constructor that moves ownership and can therefore go in an STL container

How is std::move implemented? (Very advanced)

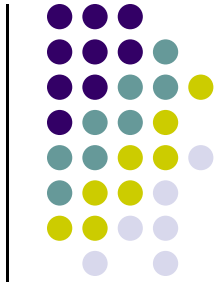


- First, we need to understand the rules for collapsing rvalue references
- T& & ≅T&
- T& && ≅T&
- T&& & ≅T&
- T&& && ≅T&&

std::move code

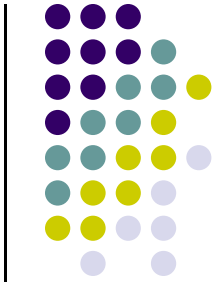


- ```
  template <class T>
  typename remove_reference<T>::type&&
  move(T&& a)
  { return a; }
  ```
- What happens in the code
  ```
  A a;
  f(move(a)); // calls f(A &&)
  ```
- For what `T` is `T&&` an `A` or `A&`?
- By the collapsing rules, we see that the only option is that `T` ≅ `A&`. (`T &&` ≅ `A& &&` ≅ `A&`)
- Now, we return a `remove_reference<A&>::type&&` ≅ `A&&`
- If you are interested, you can check that all the other cases work
  ```



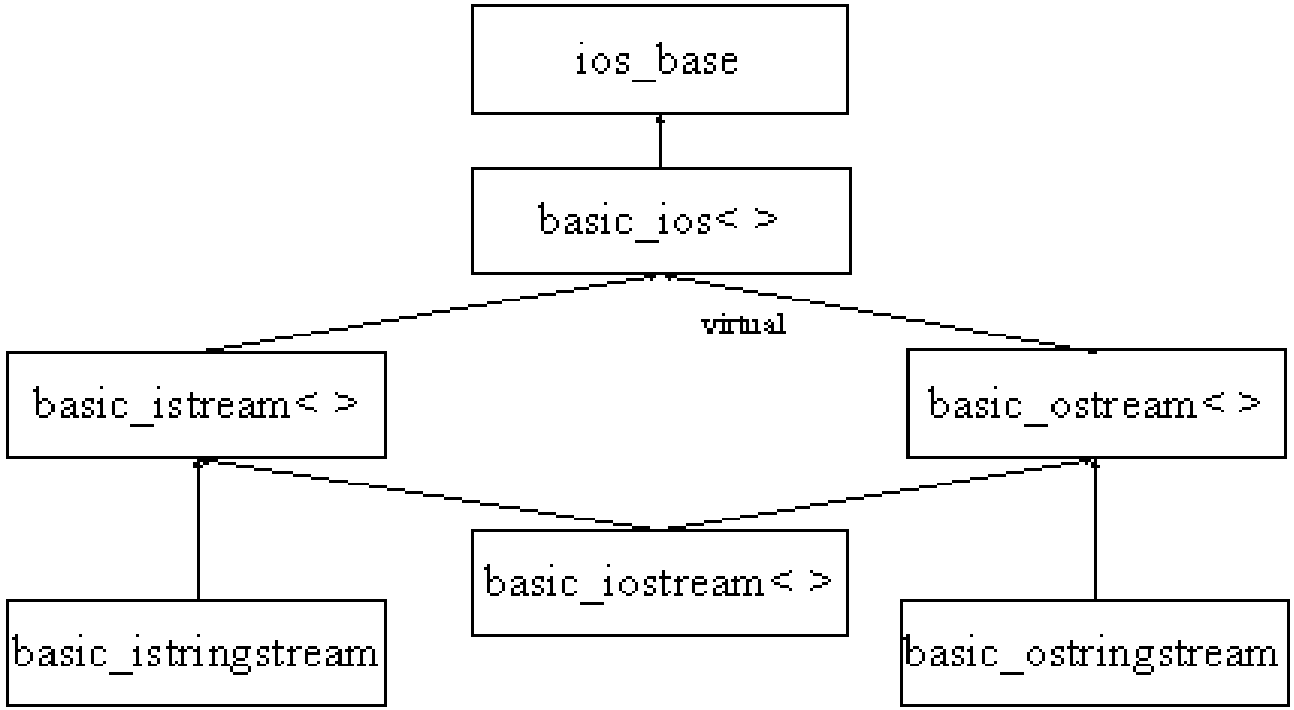
IOSTREAMS

Stream classes

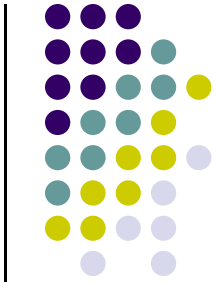


- Recall that `istream` is a typedef for `basic_istream<char>` and `wistream` is a typedef for `basic_istream<wchar_t>`
- Common stream functionality to both `basic_istream` and `basic_ostream` is contained in `basic_ios<charT, traits>` (not `basic_stream<...>`)
- All functionality that is independent of character type is in the non-template class `ios_base`
- If something is both an input and an output stream, type is `basic_iostream<charT, traits>`

A picture is worth a thousand words



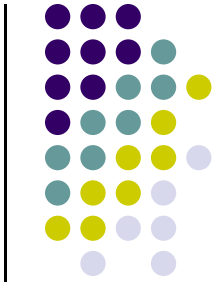
Status bits



- The state of a stream is a bitmap of type `basic_ios::iostate` of which the bits are

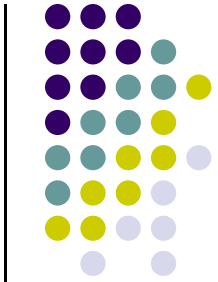
`goodbit`	Everything's OK
`eofbit`	An input operation reached the end of an input sequence
`failbit`	An operation failed to produce the desired result
`badbit`	Indicates the loss of integrity of the system

Checking flags



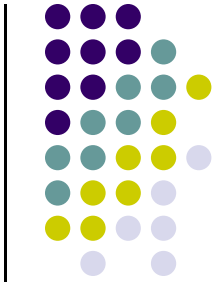
- Call one of the following methods
  - `good()`
  - `eof()`,
  - `fail()` (`failbit` or `eofbit` is set)
  - `bad()`
  - `operator!()` (Same as `fail()`)
  - `operator void *()` (Null if `fail()`)
  - `clear()` (Sets flags to good)

sync



- No less an authority than Bjarne Stroustrup has said that sync flushes all remaining input characters
- What does it really do?
  - Flushes output streams, implementation defined for input streams
- Appears to simply crash g++
- This is the correct design decision!
  - Consider a pipe

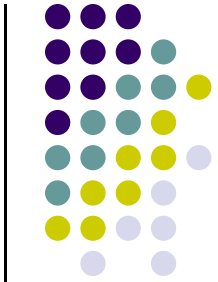
Moral



- Try not to mix formatted and unformatted I/O
- If you need to, explicitly ignore characters remaining to the end of line with

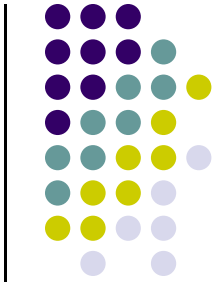
```
cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

I/O Manipulators



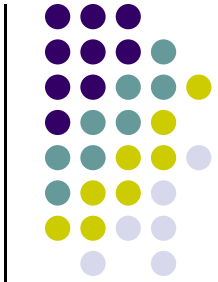
- Recall that `endl` not only inserts a `'\n'` in an `ostream`, it also flushes it.
- An object that actively manipulates a stream when inserted (or extracted) like `endl` is called an I/O manipulator.
- Some iomanipulators
  - std::endl of course
  - std::hex for hexadecimal i/o
  - std::setw to set a fixed width for the next insertion
  - std::setfill allows you to set the fill characters if the insertion is smaller than the width

Defining `endl`



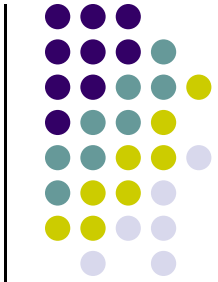
- C++ makes it easy to create an I/O manipulator.
  - Simply define it as a function that takes and returns a stream
- If we only care about `ostreams` (i.e., ordinary characters), it could be defined as follows
- ```
  ostream &
  endl(ostream &os)
  {
      os << '\n';
      os.flush();
      return os;
  }
  ```
  ```

More advanced



- Really, `endl` should work for any output stream, regardless of character type.
- Recall from slide 3, that the stream class are really templates.
- Here's how `endl` is really defined.
- ```
  template<typename charT, typename Traits>
  basic_ostream<charT, Traits> &
  endl(basic_ostream<charT, Traits> &os)
  {
      os << '\n';
      os.flush();
      return os;
  }
  ```
  ```

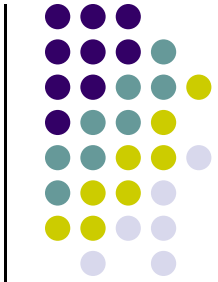
I/O manipulators with arguments



```
struct setw {
  setw(size_t s) : width(s) {}
  size_t width;
}

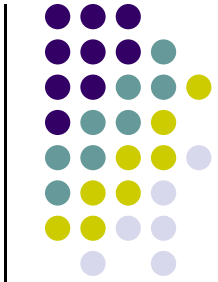
template<typename char_t, typename traits>
basic_ostream<char_t, traits> &
operator<<(basic_ostream<char_t, traits> os,
                    setw sw) { … }
```

Stream buffers



- Stream classes do formatted I/O (i.e., they let you insert and abstract arbitrary types from a character stream).
- Once it is time to actually output characters to (or input from) a device, a stream buffer is used to perform the I/O.
- Every stream has a streambuf member that can be set using `basic_ios::rdbuf`. Recall that `basic_ios` is the base class for all streams.

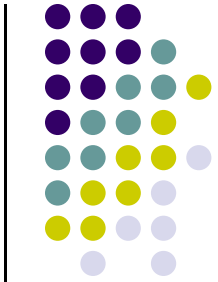
Customizing streams



- Basically, different types of streams are created by customizing a streambuf type
  - Stream types have no virtual functions
  - While stream buffers do
- However, we need to create a new stream type just to attach our stream buffer in the constructor. For example, the implementation of file streams is (logically)

```
struct ofstream : public ostream {
  ofstream() : ostream(mybuf) {…}
  … // Other constructors and methods
private:
  filebuf mybuf;
};
```

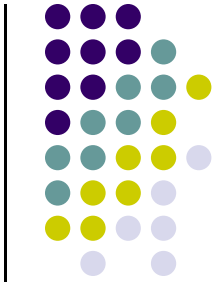
Now with templates



- Of course, since ostream is really a typedef for basic_ostream<charT, traits>, the above code would really be:

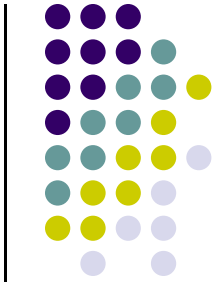
```
template
  <class charT,
   class traits=char_traits<charT> >
struct basic_ofstream
  : public basic_ostream<charT, traits> {
  basic_ofstream() : basic_ostream(mybuf) {…}
  // Other constructors and methods
private:
  filebuf mybuf;
};
```

Customizing stream buffers



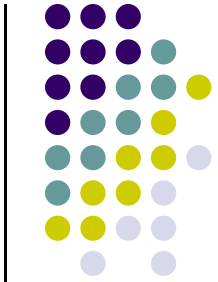
- A stream buffer can have an underlying memory buffer that can buffer characters
  - Set it with _Init() method
  - Calling _Init() with no arguments leads to unbuffered I/O
- To customize a stream buffer, we override the overflow (for output) or underflow (for input) methods
  - These are called when the buffer (if it exists) cannot buffer any more characters

Stream buffers



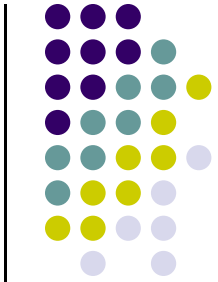
- A description of how stream buffers work and when overflow/underflow are called is at http://www.angelikalanger.com/IOStreams/Excerpt/excerpt.htm

Locales



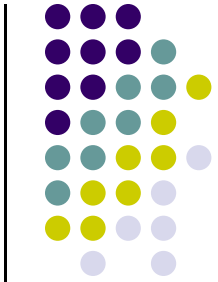
- A locale is the current “region”
  - Get the current locale by “`locale()`”
  - Get a specific locale by “`locale("German")`”
    - Except for “C”, all names are implementation-defined
  - Set a stream’s locale (if you don’t want the current one)
    - `cout.imbue(std::locale("De_DE"));`
  - Get a stream’s locale
    - `cout.getloc() // Inherits ios_base.getloc()`

Facets



- A facet is a particular area of functionality that is localized.
  - Character types (`ctype`)
  - Alphabetical order (`collate`)
  - Internationalize message catalogs (`message`)
  - case change (`ctype`)
  - Date (`time_get, time_put`)
  - Number punctuation (`num_get, num_put`)
  - Money (`moneypunct, money_get, money_put`)
  - etc.

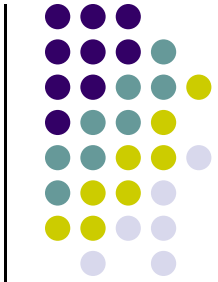
Using a facet



```
cout << "true in " << locale().name() << " is "
     << use_facet<numpunct<char> >(locale()).truename()
     << endl;
```

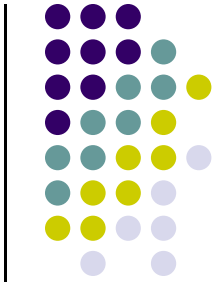
prints the current language's version of true

Sorting text



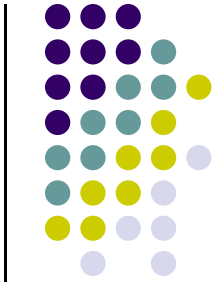
- Alphabetical order in English
- Not so obvious in some languages
  - Japanese
  - Chinese
  - Spanish!

HW 7-1



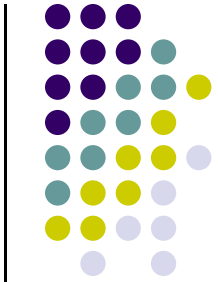
- Modify the binary tree class at http://www.cprogramming.com/tutorial/lesson18.html to be movable

Exercise 7-2



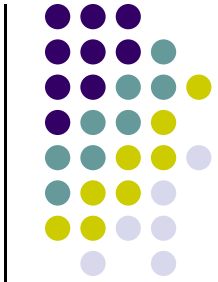
- This problem, originally given by Stuart Kurtz, will expose you to some real-world I/O challenges. Don't feel obligated to use formatted I/O. Simply using `getline`() on your input stream may be a simpler alternative. (There are also nice solutions using formatted I/O if you prefer).
- You will likely want to look up `ifstream` for doing file I/O if you aren't familiar with it already

Exercise 7-2 (Continued)



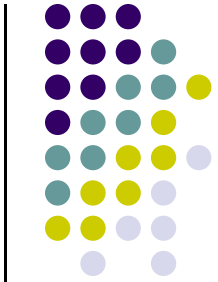
- While the political debate over the role of anthropogenic forcing factors has on global warming rages, the hurricanes in the Atlantic rage on. The destructiveness of recent year's storms has been given as evidence for global warming. A serious problem in evaluating this is the extreme variability in the number and strength of hurricanes over time. The NOAA has a good (not perfect) data set that records hurricanes since 1851. What I want you to do is write a program that will attempt to analyze the strength of each hurricane season based on NOAA data. (Continued on next slide)

Exercise 7-2 (Continued)



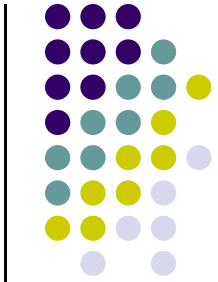
- I propose the following easily computed aggregate measure. There is a well known scale of hurricane strength -- the Saffir-Simpson scale was developed in 1969, long before the current controversies began. I want you to measure the aggregate storm activity in Saffir-Simpson days. For example, the NOAA data has four daily sustained wind speed readings for each storm, given in knots. Map each wind speed to the Saffir-Simpson scale (check Wikipedia for the precise definition). Divide each such entry by 4.0, and add it to the year's total (each entry is taken as a surrogate for 1/4 of a day's total activity). (Continued on next slide)

Exercise 7-2 (Continued)



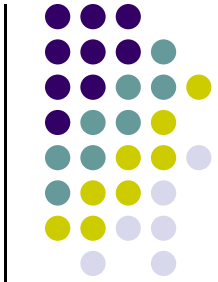
- Historical data on hurricane strength is given in the file http://www.nhc.noaa.gov/data/hurdat/hurdat_atlantic_1851-2011.txt
  - An explanation of the format of the data can be found in http://www.aoml.noaa.gov/hrd/data_sub/hurdat.html.
  - This data set is pretty typical for scientific datasets accumulated over many years -- it is an ASCII version of a punch-card set.
- Your assignment is to write a program that computes the annual Saffir-Simpson day totals for each year from the above data
  - Extra credit may be awarded for particularly nice solutions

HW 7-3 and 7-4



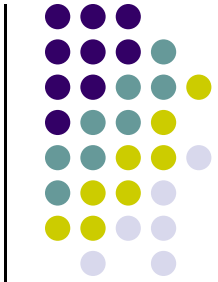
- Do one of the following 2 problems for regular credit
- Do both for extra credit

HW 7-3



- Create a new stream `IndentStream` and I/O manipulators `indent` and `unindent` with the following properties.
  - Each line output to an IndentStream is indented by a given amount.
  - Inserting indent or unindent into a stream increases or decreases the indent level by 4.

HW 7-3 (Continued)



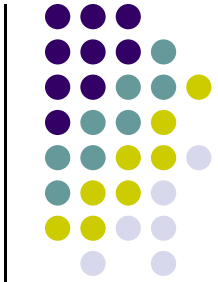
- For example,

```
IndentStream ins(cout);
ins << "int" << endl;
ins << fib(int n) {" << indent << endl;
ins << "if (n == 0) " << indent << endl;
ins << "return 0;" << unindent << endl;
ins << "if (n == 1) " << indent << endl;
ins << "return 1;" << unindent << endl;
ins << "return fib(n-2) + fib(n-1);" << unindent << endl;
ins << "}" << endl;
```

will output

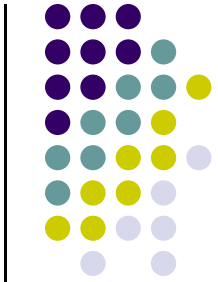
```
int
fib(int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fib(n-2) + fib(n-1);
}
```

HW 7-3 (continued)



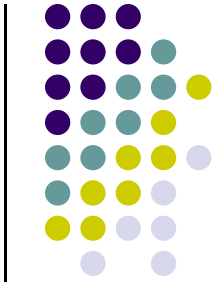
- As described in class, almost all the work will be done by creating an `IndentStreamBuf` class and redefining its `overflow` method. Just about all `IndentStream` will do is set its stream buffer to an appropriate `IndentStreamBuf` in the constructor.
- Useful link
  - http://www.angelikalanger.com/IOStreams/Excerpt/excerpt.htm

HW 7-3-extra credit



- You will receive more extra credit if your solution works for arbitrary character types (IOW, if you customize `basic_ostream` instead of just `ostream`).

HW 7-4



- Use boost::iterator's function_output_iterator to create a delimited_ostream_iterator that acts just like ostream_iterator except that the delimiter only goes between elements (and not after the final element).
- Use this to easily create an "operator<<" to print vectors in ostreams.