Advanced C++

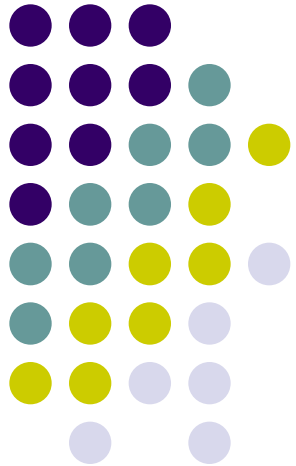
February 26, 2012

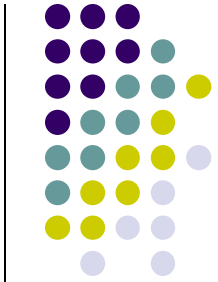
Paul Bossi lecturing for Mike Spertus

paullbossi@gmail.com

mike_spertus@symantec.com

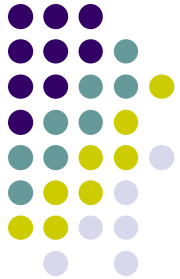
YIM: spertus





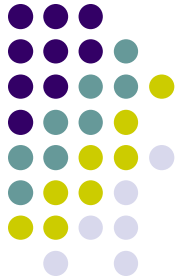
CONST, VOLATILE, AND MUTABLE

Const – Two Divergent Uses



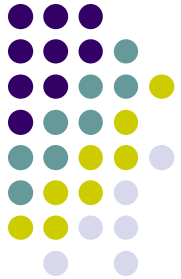
- Read-Only
  - Copy constructor example:
  - ```
    struct X {
      X( const X& x ) { … }
    };
    X x;
    X x2(x);
    ```
  - The object referenced by “x” can be modified freely in the outer context, but the copy constructor gives a contract: “I will treat this as read-only.”
- Immutable?
  - `const int MAGICNUMBER = 5;`
  - MAGICNUMBER is *intended to be* unmodifiable and immutable, and similar to a #define in C.
  - Is it?
    ```

Integers, const and const_cast



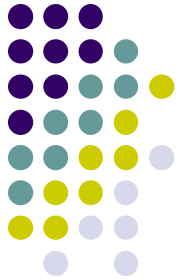
- Paradoxical example with const_cast?:
  - ```
    const int MAGICNUMBER = 42;
    const int* pci = &MAGICNUMBER;
    int* pi = const_cast<int*>(pci);
    *pi = 5;
    cout << MAGICNUMBER << endl;  // prints 42
    cout << *pci << endl;  // prints 5
    ```
- Technically, the *pi = 5 is undefined behavior.
- The rest is perfectly fine.
- Both g++ 4.7 and Visual Studio 2012 compile it all and print the values shown above.
    ```

Integers and const (cont.)



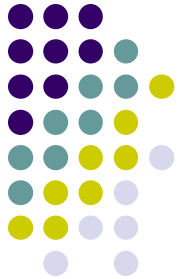
- The keyword const means “read-only” and does not mean immutable. Consider:
  - ```
    const std::string MAGICSTRING = “forty-two”;
    const int MAGICNUMBER = 42;
    const int& i = MAGICNUMBER; // hmm...
    ```
  - ```
    const string& str = MAGICSTRING; // why not?
    ```
- Semantically, these two are equivalent, and while MAGICSTRING must take up space in your program, you expect MAGICNUMBER to take up none. Taking the address of MAGICSTRING makes some sort of sense, while taking the address of MAGICNUMBER should not make much sense, but they share the identical semantic use of “const.”
- The compiler must allow the const int& i = MAGICNUMBER line, just like it must allow the more sensible const string& str = MAGICSTRING.
- Mostly, you don’t need to worry, the compiler generally does the right thing and optimizes away space usage of global const ints – but you may want to check exactly what your compiler does!

Integers and const – enum



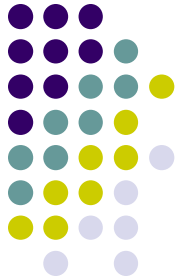
- A better solution than const for integers?
- There is no way to take the address of an enum constant and it can be absolutely relied on not to have space allocated for it in your program.
  - `enum { MAGICNUMBER = 42, };`

const_cast



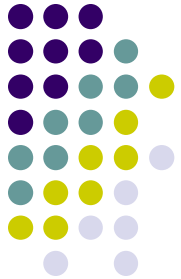
- Removes read-only from a pointer or reference.
- ```
  struct X {
    X( const X& x ) : m_iCount(0) {
      X& xr = const_cast<X&>(x);
      xr.m_iCount++;  // modify x
    }
   private:
    int m_iCopyCount;
  };
  ```
  ```

Mutable



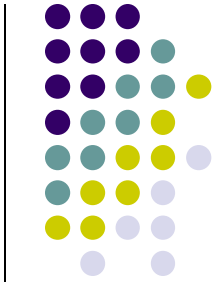
- Applies to member variables
- Means even in an instance of a class that is const, this member can be changed.
- ```
  class X {
   mutable m_iCount;
   X( const X& x ) : m_iCount(0) {
    x.m_iCount++; // const_cast not needed
   }
  };
  ```
  ```

Volatile



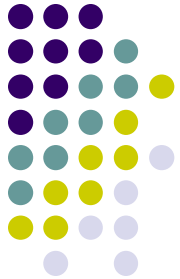
- Prohibits the compiler from making optimizing assumptions about variable changes.
- Means “this variable may change outside of any influence of the code the compiler is looking at!”
- Example:
- ```
  volatile int i = 5;
  int j = i * 10;  // can't assume i is 5
  ```
- Compiler is told that when this variable is accessed, it needs to be fetched every time from memory, even if the compiler thinks it hasn’t changed.
- Useful when a separate thread may be changing a value, for example.
  ```

Volatile and const_cast

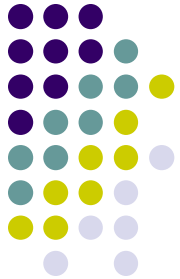


- Like const, volatile can be cast away, and you again use const_cast to do this.

Volatile (cont.)

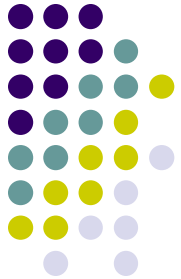


- Volatile was invented for built-in types like int, but can also be used as a member function qualifier just like const.
- Excellent reading – interesting article by Andrei Alexandrescu on a use of volatile member functions to make the compiler check thread safety for you:
- http://www.drdobbs.com/cpp/volatile-the-multithreaded-programmers-b/184403766



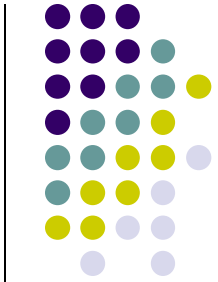
SIMPLE C++11

C++11 can be Powerful



- We've spent a lot of time on some big C++11 features
  - Concurrency
  - Lambdas
  - Move semantics
- These are powerful and important
  - But also complicated and take time and effort to learn

But also can be Simple



- In addition to powerful features like those on the last slide, C++11 adds many features that make things just “work the way you would expect things would be.”
- We’ll cover a bunch of these this lecture
- For a more complete list, see Bjarne’s C++11 FAQ at http://www.stroustrup.com/C++11FAQ.html
- Or the C++11 standard!

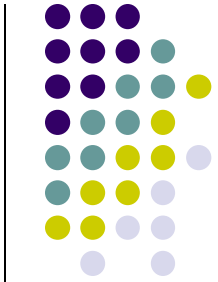
Simple C++11, I don't believe it

- OK. Here's how simple it gets
- In C++98, the following is illegal
  - `vector<vector<int>> v;`
  - The problem is that "**>>**" is interpreted as the right shift operator rather than two greater than signs
  - Have to put in an extra space like `vector<vector<int> >` instead
- In C++11, it just works
- Nothing to learn, the code just works better
- More info at Right Angle Brackets (revision 2)

Auto and decltype

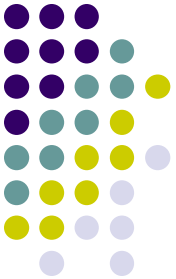
- We've covered these some, but they are simple.
- Declaring an object to have type `auto` means the type is deduced from the type of the expression
- ```
  void f(vector<int> v) {
   // No longer have to declare it
    // as having type  vector<int>::iterator
    for(auto it = v.begin(); it != v.end; it++) {:
      // Can't even come up with a type in C++98
      auto f = bind(multiplies<int>, _1, _1);
      ...
    }
  }
  ```
- If you just want the type of an expression, say `decltype(expression)`
- More info at Deducing the type of a variable from its initializer expression (revision 4)
  ```

Ranged for loop



- In C++11, we can make the for loop on the last slide even simpler:
  - `for(int x : v) { ... }`
- Or alternatively:
  - `for(auto x : v) { ... }`
- More info at references at http://www.stroustrup.com/C++11FAQ.html#for

Override controls



- We can recognize that Java has some good ideas:
- ```
  struct A {
    void f() const;
    void g() final;
  };
  ```
- ```
  struct B : public A {
    void f() override; // Error!
    void f() const override; // OK
    void g(); // Error. No overriding final
    void g() override; // Same error
  };
  ```
- More info at references given at http://www.stroustrup.com/C++11FAQ.html#override

Initializer lists

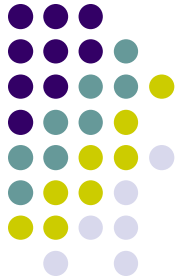
- In C/C++, it is easy to initialize an array

```
int iarr[] = { 1, 1, 2, 3, 5};
```

- How do we initialize a vector?
  - In C++98, the following is illegal, but it works as expected in C++11
    - `vector<int> = {1, 1, 2, 3, 5};`
  - Just as easy as arrays!
  - In the bad old days of C++98, we had to say
    -

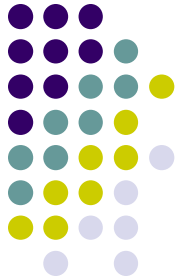
```
int iarr[] = {1, 1, 2, 3, 5};
vector<int>
  v(iarr, iarr+sizeof(iarr)/sizeof(int);
```

Using initializer lists



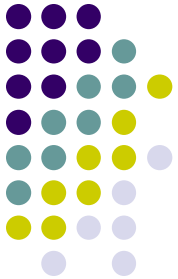
- In C++11, the following are all OK (just like you'd expect)
  - `vector<int> v = {1, 1, 2, 3, 5};`
  -
```
// Same as above but avoids the
// "most vexing parse"
vector<int> v{1, 1, 2, 3, 5};
// Now with a std::map
 std::map<std::string,int> anim = {
  {"bear",4}, {"cassowary",2}};
```

How do initializer lists work?



- This isn’t “simple C++” but its worth mentioning here
- The <initializer_list> header defines a std::initializer_list template
- A list of braced objects is effectively a literal of type std::initializer_list<E>
  initializer_list<int> a = {1, 2, 3, 10}; // OK
- Can iterate the elements of an initializer list
  cout<<accumulate(a.begin(), a.end(), 0); // 16
- More info at Uniform initialization design choices (revision 2)

How can I give my own classes list initialization?



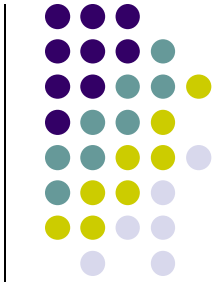
```
#include <initializer_list>
#include <numeric>

struct MyClass {
  MyClass(std::initializer_list<int> l){
    std::accumulate(l.begin(), l.end(), sum);
  }
  int sum;
};
void f()
{
  MyClass mc{1, 2, 3, 10};
  cout << mc.sum << endl; // Prints 16
}
```

Uniform Initialization Syntax

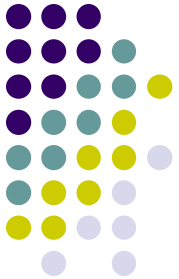
- Related to initializer lists, C++ uniform initialization syntax simplifies confusing issues that can arise from the use of constructors.
- Consider:
- ```
  ifstream f( "blah.dat" );
  vector<int> v(
    istream_iterator<int>( f ),
    istream_iterator<int>()
    );
  ```
- Problem: v is taken to be a function declaration rather than a vector instance! The arguments are taken to be function pointer types!
- This is an example of the “most vexing parse” in C++.
  ```

Uniform Initialization Syntax (cont.)



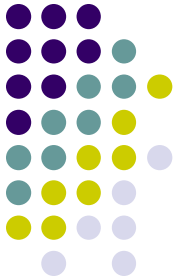
- Solution - uniform initialization syntax:
- ```
  ifstream f{ "blah.dat" };
  vector<int> v{
    istream_iterator<int>( f ),
    istream_iterator<int>()
    };
  ```
- The above now compiles as intended, as an instance of a vector.
- Notice the syntax is also similar to vector initialization based on initializer_lists discussed above:
- ```
  vector<int> v{ 1, 2, 3, 4, 5 };
  ```

Uniform Initialization Syntax (cont.)



- “But that won’t work for my class will it?”
- Yes, the same syntax works for all classes:
- ```
  struct X {
    X( int i ) : m_i(i) {}
    int m_i;
  };
  X x{5};
  X x2 = {5};
  ```
  ```

Member initializers

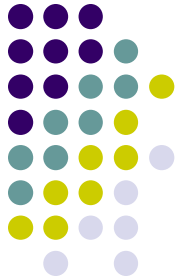


- In C++98, the following is illegal

```
struct A {
  A() : i(7) { cout << "Constructing A"; }
  ... // Many more constructors
  int i = 7;
  vector<int> v{1, 1, 5};
  fstream fs("foo.dat");
};
```

- In C++11, it just works
- In the old days of C++98, each constructor had to initialize each member even if that member always was initialized the same way
- More info at Non-static data member initializers

Inheriting constructors

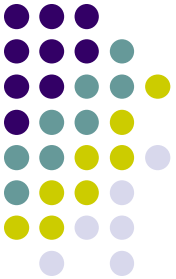


- When you derive one class from another, it gets all the members of the base class except the constructors, even though you may want them as well (“wrapper classes” often do).
- In C++11, you can have it all

```
struct A {
  A(int i);
  A(double d);
};
struct B : public A {
  using A::A; // Now B can be
  // constructed from a double or int
};
```

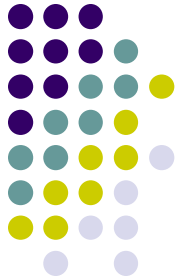
- More info at Inheriting Constructors (revision 5)

Inheriting constructors and member initializers



- Inheriting constructors and member initializers have a nice synergy
- You can inherit constructors and use member initializers to initialize any new fields
- ```
  struct X {
    X(int i_) : i(i_) {}
    int i;
  };
  struct Y : public X {
    using X::X; // Inherit constructors
    string s = "foo";
  };
  void f() {
    Y y(7);
    cout << y.i << y.s; // Prints 7foo
  }
  ```
  ```

Delegating constructors

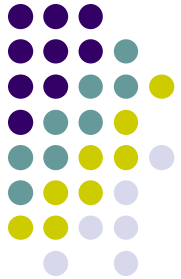


- Often one constructor is a special case of another
- In C++98, the following is illegal

```
struct A {
  A(int i) { … // Do a lot }
  A(string s) : A(s.size()) {}
};
```

- In C++11, it just works
- More info at Delegating Constructors (revision 3)

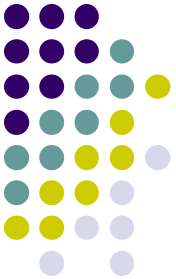
Delegating constructors and thread-safe classes



- Writing thread-safe copy constructors is difficult because it is difficult to lock the source

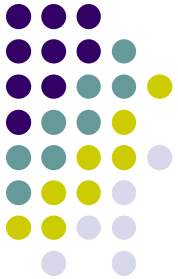
```
struct A {
  A(A const &a) : i(a.i) {
    // Too late, because we read
    // a.i before we locked
    lock_guard<mutex> l(a.mtx);
  }
  int &i;
  mutex mtx
};
```

Solution by delegating copy constructor



- ```
  struct A {
    A(A const &a)
      : A(a, lock_guard<mutex>(a.mtx)) {}
   private:
    A(A const &a, const lock_guard<mutex>& )
     : i(a.i) {
      lock_guard<mutex> l(a.mtx);
    }
    int &i;
    mutex mtx
  };
  ```
- See http://www.justsoftwaresolutions.co.uk/threading/thread-safe-copy-constructors.html for details
  ```

Template alias



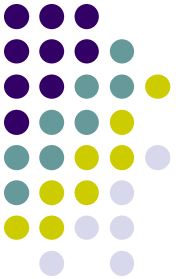
- Regular typedefs are nice:
  - typedef std::map<std::string,int> map_t;
- What about templated typedefs?
- The following is illegal in C++98

```
template<typename T>
typedef
  std::map<T, T>
  MapWithKeyValueSameType;
MapWithKeyValueSameType<int> mi;
```

- It's also illegal in C++11! But you can say

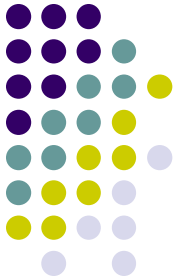
```
template<typename T>
using MapWithKeyValueSameType
  = std::map<T, T>;
MapWithKeyValueSameType<int> mi;
```

C++98 workaround



- If you need a “template typedef” and your compiler doesn’t yet support template aliases
- ```
  template<typename T>
  struct KeyValueSameType {
     typedef std::map<T, T> MapType;
  }
  KeyValueSameType<int>::MapType mi;
  ```
- Ugh!
  ```

Deleted methods



- Making a class non-copyable in C++98

```
struct X {
private:
  X( const X& ); // private & no impl
  X& operator=( const X& ); // ditto
};
```

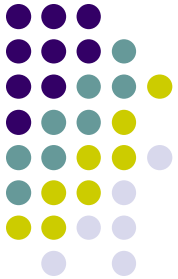
If you accidentally invoke the copy constructor in the implementation of X (where private methods are still visible), you will get a link-time error.

- In C++11: compile-time error every time, self-documenting - better all around:

```
struct X {
  X( const X& ) = delete;             // better
  X& operator=( const X& ) = delete;
};
```

- More info:
  http://www.stroustrup.com/C++11FAQ.html#default

Strong enumerations



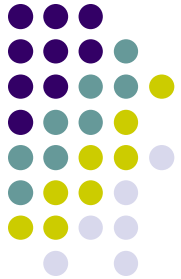
- Traditional enumerations break type safety by implicitly converting to ints and leak outside their scope

```
enum alert { red, yellow };
int i = red; // Allowed but dangerous
```

- Fixed in strong “enum class” enumerations

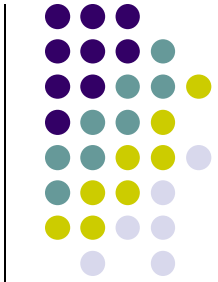
```
enum class alert { red, yellow };
alert a = red; // Error. Needs scope
alert a = alert::red; // OK
int i = alert::red; // Error.
                    // Incompatible types
```

C++11 language features



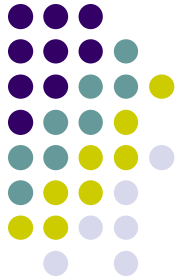
- Combined with concurrency, lambdas, and move semantics from last quarter, you now know nearly all of the new C++11 language features
- There is one big language feature remaining
  - variadics
  - We will spend several weeks on variadics this quarter
- Next week: A look at the new C++11 libraries

Homework 8-1



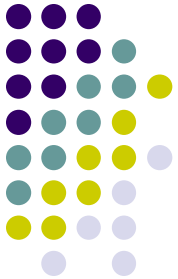
- Write some valid code with all of the features from this week and let us know which ones your compiler accepts
  - >>
  - auto and decltype
  - Ranged for
  - Override controls
  - Initializer lists
    - Regular credit for initializing a standard library container. Extra credit if you also give your own class an initializer list constructor
  - Member initializers
  - Inheriting constructors
  - Delegating constructors
  - Deleted methods
  - Template aliases
  - Strong enumerations

HW 8-2—Extra Credit



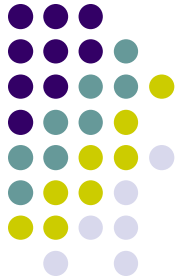
- There is a whopper of a mistake in the following article about constructor delegation and threadsafe constructors
  - http://www.justsoftwaresolutions.co.uk/threading/thread-safe-copy-constructors.html
- What is the mistake in article?
  - Please don’t read the comments after the article until you have submitted this assignment

Homework 8-3



- Write a class that keeps track of how many times data has been read.
- Do two implementations: one using const_cast and one using mutable.
- Describe which version you liked doing.

HW 8-4—Extra Credit



- Create two programs that show substantially different runtime behavior while having code that is identical except for the use of the volatile qualifier. An example of a “substantial difference” would be that one program hangs while the other exits normally.