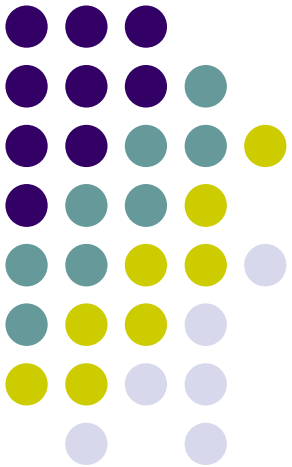
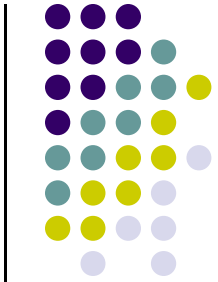
C++ Best Practices

Mike Spertus

mike_spertus@symantec.com

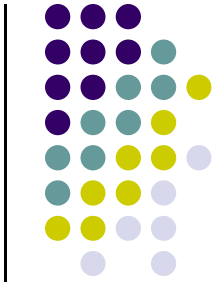


Coming Attractions



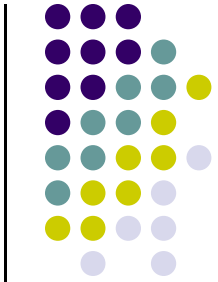
- The C++ committee is meeting in Chicago from Sept 23 to Sept 30
- You will learn a lot by coming
- We may learn a lot by having you there

Always put headers in a namespace



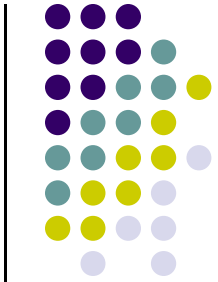
- Also use an “#ifndef …” to guard against multiple inclusions
- ```
  #ifndef FOO_H
  #  define FOO_H
  namespace cspp51044 {
  int f();
  ...
  }
  #endif
  ```
  ```

Never “use” a namespace in a header



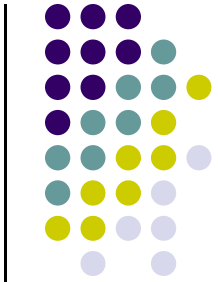
- Leaks entire namespace to any file that includes the header.
- E.g., when in a header file, say
  `using std::accumulate`
  instead of
  `using namespace std;`
  or just explicitly call `std::accumulate` without a `using` statement at all
- When in a “.cpp” file, choose whichever you prefer.

Put include guards in your header



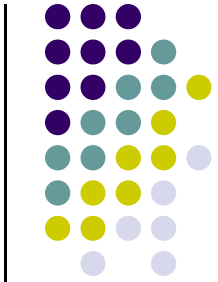
- #ifndef FOO_H
  #define FOO_H
  …
  #endif

Prefer C++-style casts to C style casts



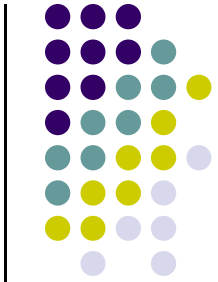
- A *a = (A *)&b; // bad
- A *a = dynamic_cast<A *>(&b);

Put const and volatile after type names



- “int const” is better and more consistent than “const int”
- Bjarne Stroustrup disagrees
- However, Dan Sachs’ ACCU “Truthiness” keynote argues this is the only rational conclusion one can reach, as it is both more logical and studies show that is leads to fewer buts.

Prefer C++-style casts to C style casts -- Rationale



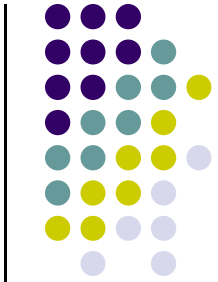
- Let's look at two cases where they differ

```
struct X {...};
struct Y {...};
X *xp = new X;
Y *yp = (X *)(xp); // Nonsense
yp = dynamic_cast<Y *>(xp); // 0 shows cast failed

struct Z : public X {...};
struct W : public X {...};
struct A : public Z, public W {...};
W *wp = new A;
X *xp = wp; // OK. Inheritance
A *ap = (A *)xp; // Oops! Points in middle of A
ap = dynamic_cast<A *>(xp); // Adjusts for
                            // multiple inheritance
```

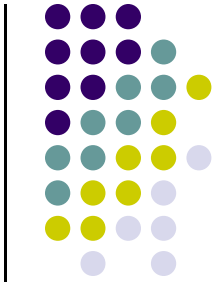
- In both cases, C++-style casts are better when they disagree

Define symmetric binary operators as global functions



- Don’t use the member form of `operator+()`
  - Because both arguments should be treated the same
- However, do define `operator+=()` as a member
  - We don’t want to += to assign to a compiler-generated temporary

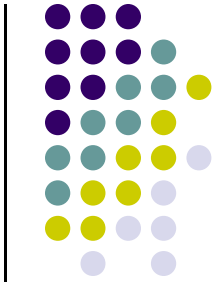
Think about types inferred by templates



- What does this print?

```
double dp[] = { 0.1, 0.2, 0.3 };
cout << accumulate(dp, dp + 3, 0);
```

Think about types inferred by templates

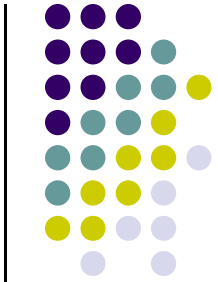


- If you're accumulating doubles with `std::accumulate` use an initial value of `0` instead of `0`
  - Or you'll accumulate integers
- E.g.,

```
double dp[] = { 0.1, 0.2, 0.3 };
cout << accumulate(dp, dp + 3, 0);
```

(surprisingly) prints 0

Beware of Dependent base classes

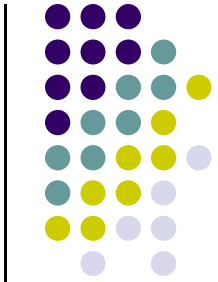


- What does the following print?

```
#include <iostream>
using namespace std;

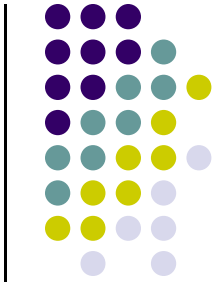
int f() { return 0; }
template<class T>
struct C : public T {
    C() { cout << f() << endl; }
};
struct A {
    int f() { return 1; }
};
int main()
{
    C<A> c;
}
```

Dependent base classes: Surprising answer



- Microsoft Visual C++ prints 1
- g++ prints 0
- g++ is correct
- T is a “dependent base class”
  - A base class that depends on the template parameter
- Symbols are not looked up in dependent base classes, so templates are not surprised by unexpected inheritance

Correct use of dependent base classes



- To see symbols in a dependent base class, reference it explicitly:

```
template<class T>
struct C : public T {
    C() { cout << T::f() << endl; }
};
```

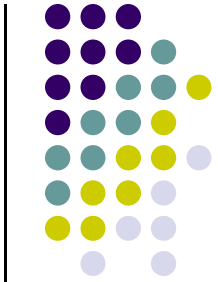
- Alternatively

```
template<class T>
struct C : public T {
    using T::f;
    C() { cout << f() << endl; }
};
```

- If you want the global symbol:

```
template<class T>
struct C : public T {
    C() { cout << ::f() << endl; }
};
```

Watch out for method hiding

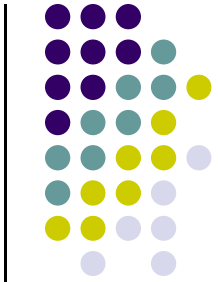


```
struct B {
  void f(bool i) { cout << "bool" << endl; }
};

struct D : public B {
  // Fix with "using B::f"
  void f(int b) { cout << "int" << endl; }
};

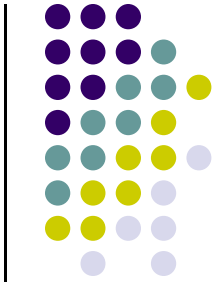
int main()
{
  D d;
  d.f(true); // Prints "int"
}
```

Use const appropriately



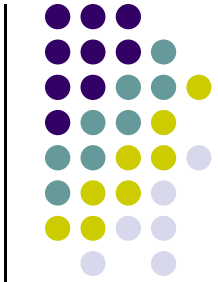
- Const methods should be const
- Const & arguments should be const
- The “const” keyword should go after the type
- ```
  class A {
  public:
    void f(int const &i) const;
  };
  ```
  ```

Use const appropriately-rationale



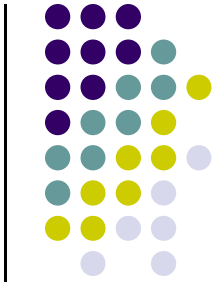
- Ignoring const is no longer an option
- int seven() { return 7; }
  void pr_int(int &i) { cout << i; }
  void pr_int_const(int const &I;
  pr_int(seven()); // Error on newer compilers
  pr_int_const(seven()); // OK
- Putting const on right prevents ambiguity
  - `const int *` looks like a constant "`int *`" but isn't
  - `int const *` could only mean one thing
  - Studies show programmers make fewer mistakes with this rule

Don’t slice objects



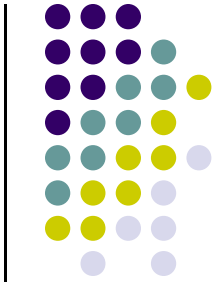
- D inherits from B
- ```
  D d;
  B b = d; // Almost certainly wrong
  ```
  ```

Use virtual destructors when you inherit



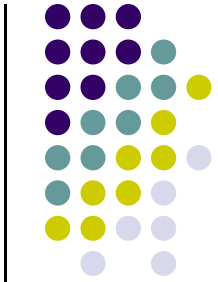
- ```
  class A {
  public:
    // virtual ~A() {}
  };
  class B : public A {
  public:
    ~B() { ... }
  };
  A *ap = new B;
  delete ap; // Doesn't call B's dest
  ```
  ```

Prefer templates to macros



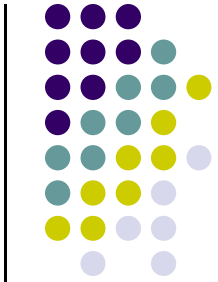
- e.g., `min` should be a template but Microsoft Visual C++ defines it as a macro

Don’t make tricky assumptions about order of evaluation



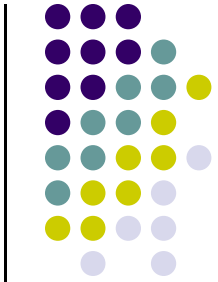
```
struct S {
  S(int i) : a(i), b(i++) {
    f(i,i++) // Undefined behavior
  }
  int b;
  int a;
};
```

Remember that primitive types have trivial constructors



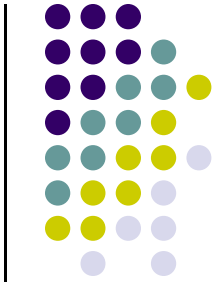
```
void
f()
{
     int i;
     cout << i; // i contains garbage
}
```

Don’t return a reference/pointer to a local variable



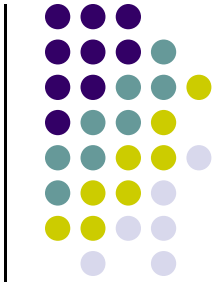
- ```
  int &
  f()
  {
      int i = 3;
      return i; // Bad!
  }
  ```
  ```

Best practice—Prefer range member functions to their single-element counterparts



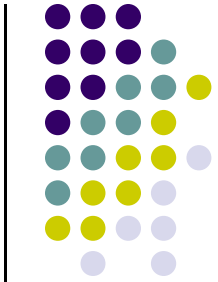
- Item 5 of Meyer's Effective STL
- Given two vectors, v1 and v2, what's the easiest way to make v1's contents be the same as the second half of v2's?
  - Don't worry whether v2 has an odd number of elements

Worst (but common)



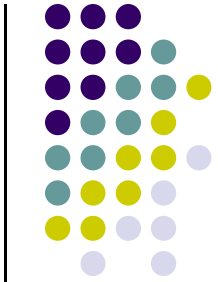
- ```
  vector<Widget> v1, v2
  ...
  for(vector<Widget>::const_iterator ci
        = v2.begin() + v2.size()/2;
      ci != v2.end();
      ++ci) {
    v1.push_back(*ci);
  }
  ```
  ```

Better



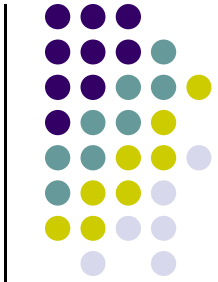
- ```
  copy(v2.begin() + v2.size()/2,
       v2.end(),
       back_inserter(v1));
  ```
  ```

Even better



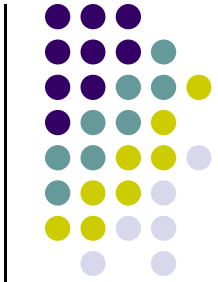
- ```
  v1.insert
      (v1.end(),
       v2.begin() + v2.size()/2,
       v2.end());
  ```
  ```

Best



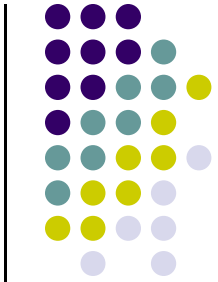
- ```
  v1.assign(v2.begin() + v2.size()/2,
            v2.end());
  ```
  ```

Best Practice: Prefer empty() to size() == 0



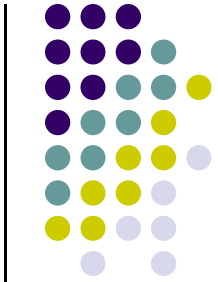
- Suppose `l` is a `list<int>`
- Which is better?
  - `if(l.empty()) { ... }`
  - `if(l.size() == 0) { ... }`
- Prefer the `l.empty()`
- Calculating `size()` can take a long time
- Effective STL Item 4

Recall the difference between virtual and non-virtual



- Review slides 11-15 of lecture 2

Final



- Open book
- Open notes
- You can look at posted sample files, lecture notes, your past HW submissions and the standard
  - You will definitely want to have ready access to the best practice list above
- Do not use a compiler
- Do not use any other resources or google for answers to questions