

Lecture Topics

- matching an overloaded call
- pitfalls of overloading & conversions
- overloading miscellany
- variable declarations

Administrivia

- Lab #1 handout & discussion

Matching

- How different do two definitions of a function really have to be for the compiler to distinguish them?
- How does the compiler decide which function you meant to call?
- C++ allows for extremely minor distinctions; use at your own risk.
- For example, C's default type conversions are not assumed:
 - char/short to int
 - float to double
 - thus the following operators are different

```
operator+= (int i);  
operator+= (char c);
```
- also allows overloaded variants based on other implicit conversions
 - signed to unsigned
 - non-const to const
- selecting between overloaded matches
 - the basics: pick the “most derived” class
 - multiple args, multiple inheritance, so not always unique
- original ambiguity resolution was by order of declaration (yikes!)

“Better Matching”

- low points
 - No one I’ve asked has ever remembered these rules, even people whose primary computer language is C++.
 - I can’t even make sense of the rules when I read them... (p. 228); to wit, Stroustrup just said (p. 225) that he wanted to differentiate const from non-const args, and in the rules he says that such conversions don’t count (and are thus ambiguous, making them illegal to ever use...); I can only guess that such oddities are the result of the slight simplification he mentions...
 - My first attempt to create a pitfall example using IBM’s online version of the rules also failed; gcc is either more strict or I mis-read them.
- keep these in mind
 - when you think that you’ve come up with something “cool” (i.e., subtle) using overloading...
 - likely to be hard to recognize, understand
- BUT: less complicated than I remember (I remember something about counting args being converted; maybe in the ARM?)
- why is matching challenging? for starters,
 - C’s implicit conversions are NOT acyclic (“most derived?”)
 - but Stroustrup wanted to get rid of implicit narrowing anyway
- rule: pick lowest numbered match, which must be unique (or causes error)
 - 1: no conversions (non-const to const, array name to pointer, etc.)
 - 2: integral promotions (widening/sign removal)
 - 3: standard conversions (int to double, derived* to base*, etc.)
 - 4: user-defined conversions (single-arg. constructors)
 - 5: ellipsis (...)
- [See ARM for more precise version]
- For >1 argument, matched function must be at least as good in all arguments and better in at least one argument.

Pitfalls of Overloading and Conversions

- here's a real danger that can be hard to foresee
 - two “natural” interpretations of one set of types...
 - watch out!
 - instead, make up new names for BOTH options
 - similar to need for “explicit” keyword, but no easy solution
- “...minimizing surprises caused by implicit conversions is inherently difficult...” Doug McIlroy, as quoted by Str, p. 227
- Consider the following
 - class MyObject
 - friend function


```
MyObject operator+ (MyObject& a, MyObject& b);
```
 - `MyObject x;`
 - What does “`MyObject y = x + 42;`” do?
- Does answer depend on which of the following are defined?


```
MyObject (int num); // conversion from int to MyObject
operator int ();    // conversion from MyObject to int
```

 - What if they're both defined?
 - What happens if I change my answer (e.g., create the constructor after using the code for a while)?
- you need both functions to compile
 - when both defined:
 - convert x to int, add, then convert sum to MyObject
- Why isn't this ambiguous?
 - Compiler can't use constructor on 42
 - because operator reference argument is non-const! Oops!
- when you add const


```
friend operator+ (const MyObject& a, const MyObject& b);
```

 - having both constructor and cast operator creates ambiguity
 - having only constructor works fine (opposite order as before...)
- so: forgetting const changed both legal options and their meanings...