Lecture Topics

- overloading
  - matching an overloaded call
  - pitfalls of overloading & conversions
  - miscellany
  - Thurs: new and delete

Administrivia

- PS1
  - find teams and let me know groups (for EWS sharing purposes)
  - get copy of PIN and get examples running (catch function calls, catch instructions)

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!)

[example left in notes; skip in lecture; details may not make sense without detailed matching algorithm in next lecture]

Stealing a Call

- I did have to jump through more hoops than I expected to create this example.

- Of course, gcc seemed to be identifying more things as “ambiguous” than suggested in the (dated) book we’re using or in the (up-to-date?) docs on the IBM web pages.

```
class ALPHA {};
class BETA {
    public:
        operator int () {return 42;}
    friend check (const BETA& obj, int num);
}
class GAMMA : public ALPHA, public BETA {};
```

- consider a call of the form...

```
GAMMA g, h;
check (g, h);
```

- Such calls go to the unique function **`check`** defined in BETA.

- What happens if I add the following to class ALPHA?

```
friend check (const ALPHA& obj, const ALPHA& num);
```

- answer: calls are silently transferred to the new function

- Here's a less contrived example that illustrates the potential danger of using "convenient" implicit conversions.

```
class BETA {
    public:
        operator int () {return 42;}
}
```

- Once you've created the implicit conversion, it's possible to pass a BETA to any function that takes an **`int`**.

- So people write some code with the implicit conversion.

- Now someone else comes along and decides to create a function that takes a BETA and happens to have the same name as a function that takes an **`int`** in place of the BETA

- result
  - new function *probably* needs to be friend of class
  - calls are stolen

- still a little contrived
  - single argument functions are likely to be member functions, which won't have this problem
  - more arguments are less likely to match exactly by chance

Pitfalls of Overloading and Conversions

- here's a real danger that can be hard to foresee
  - [example] cropRectangle (int, int, int, int)—two points or point + dims?
  - 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…

- note that const/non-const matching can have value
  - preserve const through function calls…

```
class ALPHA {
    public:
        WIDGET& getWidget ();
        const WIDGET& getWidget () const;
};
```

"Better Matching"

- the hazards of matching
  - No one I've asked has ever remembered these rules, even people whose primary computer language is C++.
  - when you think that you've come up with something "cool" (i.e., subtle) using overloading…
  - likely to be hard to recognize, understand

- a couple of asides [not for board]
  - 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.
  - 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
  - yet g++ [4.1.2] still allows narrowing, even for matching

- 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.

- a simple call stealing case… [more complex examples in Lec. 7 notes]

```
int func (char arg); // original function

int answer = func (42); // code calls original function

int func (int arg); // new function added later

// call shown is "stolen" silently
```

Overloading Miscellany

- consider overloading array syntax (**`operator []`**)

- Did you think of overloading reads, writes, or both?
  - **`X[i] = X[j];`**
  - left side is an L-value
  - right side is some data type stored in X at index j

- implementation
  - right side probably pretty easy (look up and return)
  - if X is a complicated data structure, left side may be slower/harder
  - Can you define one function (**`operator []`**) that works?
  - not really
    - should there be two versions of **`operator []`**?
    - or find a workaround?

- example workaround (see Str. Sec. 3.7.1)
  - use an extra data structure to hack it
  - given class ALPHA that stores objects of class BETA
  - create helper class ALPHA_REF containing ALPHA* and integer
  - operator[] returns new ALPHA_REF
  - ALPHA_REF has two operators
    - cast operator to BETA (do the actual lookup)
    - assignment operator from BETA (do an insertion)
  - now X1[i] = X2[j] becomes…

```
X1.operator[] (i).operator=(X2.operator[](j).operator BETA ())
```

- not all operators can be overloaded
  - member access (“.”)
  - pointer to member function invocation (“.*”)
  - conditional expressions (?:)
  - scope identification (::)

- overloading can break C’s duality
  - pointer-like objects and array-like objects not necessarily equal
  - pointer vs. array
    - **`array[10]`**
    - **`*(array+10)`**
  - pointer dereference
    - **`inst->member`**
    - **`(*inst).member`**
    - **`inst[0].member`**
    - not possible to change definitions equivalently because “.” can’t be overloaded

[STOPPED HERE]

- copying vs. constructing
  - What’s the difference between the two assignments below?

```
ALPHA a;
ALPHA b = a; // copy constructor
b = a;       // assignment
```

  - declaration has no “old version”
    - may need work to destroy previous version
    - e.g., rehash instance in a lookup table
  - these two are **NOT** equivalent in C++
    - default version is memberwise copy for both
    - overriding one does **NOT** catch the other (other version will use default copy)
    - compiler will **NOT** warn you