Lecture Topics

- overloading
  - pitfalls of overloading & conversions
  - matching an overloaded call
  - miscellany
  - new & delete
- variable declarations
- extensibility: philosophy vs. reality

Administrivia

- check Lab #1 progress
  - see web board post
  - ok to finish by next Tuesday?

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…

"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++ 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

- 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

Overloading New and Delete

- Str. Ch. 10 discusses memory management in detail

- both operators can be overloaded
  - overloading of **`new`** is fairly flexible
  - can overload **`delete`**, but not all versions can be reached
  - when overriding default memory management
    - include C++ standard header
    - **`#include <new>`**

- overloading operator **`new`**
  - **`std::size_t`** argument (implicit in calls)
    - must appear as first argument in all operator **`new`** signatures
    - holds number of bytes needed when called
  - array and instance allocation are distinct even by default
    - signatures
      ```
      void* operator new (std::size_t size);
      void* operator new[] (std::size_t size);
      ```
    - for a class ALPHA:
      ```
      ALPHA* a;
      a = new ALPHA (1, 2, 3); // operator new
      a = new ALPHA[10];       // operator new[]
      ```
    - note: array allocation requires a constructor with no arguments
  - can extend either/both versions with arbitary arguments
    - for example
      ```
      operator new (std::size_t size, int region_id);
      ```
    - other arguments are then passed to new as follows
      ```
      a = new(42) ALPHA (17, "potato");
      ```

- overloading operator **new** (cont'd)
  - one use of extra arguments: placement
    - want some control over location of "dynamic" allocation
    - e.g., sometimes necessary to locate instances in DMA-accessible memory (low physical addresses)
  - default placement
    - locate at a specific place (provided as argument to new)
    - `void* operator new (std::size_t size, void* p) {return p;}`

- exception handling
  - header file **<new>** also defines an exception for allocation failure
    - **std::bad_alloc**
    - derived from **std::exception**
  - versions of **new** discussed so far can generate exceptions

```
void* operator new (std::size_t size) throw (std::bad_alloc);
void* operator new[] (std::size_t size) throw (std::bad_alloc);
```

  - notation implies that no other exceptions are thrown

  - default versions also exist that return NULL instead
    - header file **<new>** defines structure and static variable to allow pseudo-argument

```
void* operator new (std::size_t size, const std::nothrow_t&)
    throw ();
```

    - notation here implies that no exceptions are thrown
    - to invoke this form, use something like

```
a = new(std::nothrow) ALPHA (42, “no exceptions”);
```

[skip most of this slide in lecture; leave in notes]

- exception handling before exceptions
  - some of the C++ support pre-dates the exception-handling mechanisms
  - this support is **not thread-safe**, and you should generally avoid it

  - default operator **`new`** behavior
    - try to allocate
    - on failure, check whether a handler has been registered for failures
    - if so, call it and try again
    - if not, throw an exception

  - you can register a handler with
    ```
    new_handler set_new_handler (new_handler) throw ();
    ```
    - **`new_handler`** is a pointer to a function that
      - takes no arguments
      - returns nothing (void)
    - **`set_new_handler`** returns the previous handler pointer

  - note that the default behavior keeps calling the handler
    - if handler can't fix the allocation problem
    - e.g., by garbage collection
    - it must throw an exception
    - to avoid an infinite loop

  - again
    - this mechanism **IS NOT THREAD-SAFE**
    - your program has one global variable for the handler pointer

- overloading operator **`delete`**
  - **`gcc`** will let you define many versions
    - but only two versions are usable
    - non-array
      ```
      void operator delete (void* p);
      delete a;
      ```
    - array
      ```
      void operator delete[] (void* p);
      delete[] a;
      ```
    - note
      - compiler **does not check** that you used the “correct” version
      - it does not remember which version of **`new`** you used
      - it allows either version of **`delete`** without warning

  - the book rambles for a while on rationale
    - tries to establish the difficulty for programmers
      to track “types” of allocations and use proper deallocations
    - instead, have **`new`** record type and **`delete`** make use of it
    - except that the array version allows exactly that type of oddity

  - what’s the real reason?
    - probably the fact that the following are synonymous
      - **`delete(a)`**
      - **`delete a`**
    - similar to people writing return (42);
    - and thus lots of code might break to support
      argument-passing to delete