Lecture Topics

• C++ exceptions implementation
• overloading & references
  – purpose, pros, & cons
  – matching an overloaded call
  – pitfalls

Administrivia

• none today
[REVIEW]

- exceptions in place of assertions...

```c
void foo(int n_bytes)
{
    ASSERT(0 < n_bytes);

    /* what if n_bytes < 0 ? */
    if (n_bytes < 0) {return;}
}
```

- two problems
  - if statement NEVER tested during debugging
  - (and this example actually has a bug!)
  - choose between (yikes!)
    - writing untested code
    - not thinking about effect at all
    - shipping code with assertions (nice messages for end user!)
- instead, design module to use exceptions
  - always present
  - module user can handle at any level of code
  - or can ignore, and condition will terminate program

[END OF REVIEW]

- NetLink example
  - one base class:
    ```
    class NetLinkException : public std::exception {
    }
    ```
  - three derived classes (init failed, bad args, inactive netlink)
  - code throws only derived class exceptions
Implementation

- based on experimentation with g++ 3.4.4 on Cygwin

- C++ lines
  
  ```
  #include <exception>
  class ALPHA : public std::exception {}
  throw ALPHA ();
  ```

- C++ header file location for gcc
  - `/usr/include/c++/<version>/`
  - `<version> = 4.1.2` on lab machines

- resulting x86 assembly (names cleaned up, unmangled, etc.)

  ```
  ; Allocate 4-byte exception instance (ALPHA).
  movl $4, (%esp)
  call allocate_exception

  ; Call ALPHA's (default) constructor on new memory.
  movl %eax, %ebx
  movl %ebx, (%esp)
  call ALPHA_constructor

  ; Call throw function (new exception, type info,
  ; destructor function)
  movl $ALPHA_destructor, 8(%esp)
  movl $ALPHA_type_info, 4(%esp)
  movl %ebx, (%esp)
  call throw

  ; The call does NOT return!
  ```
• throw function
  – uses type information to walk up through each stack frame
  – find appropriate catch (any superclass)
  – destructor is necessary because exception must be discarded after catch
  – unless re-thrown with "throw;", but eventually necessary
  – for each (C++) stack frame
    • also calls destructors
    • static table used to map exception point to destructors to be called

• catching (C++ lines)

```cpp
try {
    // something that can throw an ALPHA
} catch (ALPHA& a) {
    // be sure to use a reference
}
```

• adding catch statements to a function
  – creates
    • block of static data, and
    • pointer to block on the stack
  – to help throw identify
    • which types of exceptions are being caught
    • and in what order
  – throw routine uses this info
    • to calculate an index
    • then goes to a switch statement (assembly equivalent)
    • to find the right catch block
• note: exception within throw causes abort() to be called; destructors should not generate exceptions (they can use/catch exceptions internally)
Overloading

- Stroustrup Sec. 3.6 and Ch. 11

- recall examples
  - 190: don’t do it
  - 391: don’t do it except to virtualize functions
  - NetLink: three constructors for one class!

- original goal of overloading
  - support “natural” use of operators for user-defined types
  - canonical example: complex numbers

- let P and Q be complex numbers and calculate \( R = P^2 + Q^2 \)
  - in C, taking some small liberties with the stack…
    \[
    R = \text{complex\_add}\ (\text{complex\_multiply}\ (P, P), \\
    \text{complex\_multiply}\ (Q, Q));
    \]
  - in C++,
    \[
    R = P * P + Q * Q;
    \]

- also expect to fit in with “natural” conversions from integer, double, etc.
  - complex * int
  - complex * double
  - int * complex
  - double * complex
  - etc.

- define all such functions? absurd…
• instead
  – create new implicit casts
  – use friend functions for symmetry

• example

```cpp
class complex {
    complex (int real_part);
    complex (double real_part);
    friend complex operator+ (const complex& a,
                               const complex& b);
    friend complex operator* (const complex& a,
                               const complex& b);
};
```

• a few things to notice
  – single-argument constructor creates implicit cast path
    • complex P = 4; // P = 4 + 0i -- why not 0 + 4i?
    • to prevent implicit casts, use keyword “explicit”:
      ```cpp
      explicit complex (int real_part);
      ```
    • in case above, compiler will still use int to double to complex
      implicit path without warnings…
  – trailing ampersand indicates a reference type
    • implementation equivalent to pointer
    • syntactic use and rules slightly different (discussed later)
  – return type is the whole structure!
    • can’t use reference (pointer) to local variable inside function
    • copy is returned on the stack
  – all of these functions can be inlined
    • including friend functions
    • but some copying hard to optimize away
  – arguments to operators are constants (casts do not work otherwise)
References

• Stroustrup Sec. 3.7

• want syntactically equivalent yet efficient forms for user-defined types
  – syntax
    • recall: \( R = P \times P + Q \times Q; \)
    • not: \( R = *(&P \times &P + &Q \times &Q); \)
  – but class instance may be quite large
    • avoid copying to stack all the time
    • avoid returning on stack if possible

• reference
  – pointer implementation (identical!)
  – syntactically equivalent to base type
  – possible ambiguity with reference-to-reference assignment
    • copy pointer or copy contents?
    • to avoid, C++ disallows changes to reference value
      (i.e., to a new pointer) after initialization
    • assignment thus results in copy of contents

• references allow
  – redefine argument semantics on a per-argument basis.
  – can use “pass by reference” instead of “pass by value!”

• my take
  – as Stroustrup says elsewhere, any language can be misused
  – bad idea to rewrite language in a way that obscures intent
• simple example:
  – Where is “a” initialized?
  – Yes, the compiler can tell and warn you.
    
```c
int a;
foo (a);
```

• This loop seems to hang. Can you help?
  ```c
while (42 != i) {
  foo (i);
  x = bar (i);
  zap (i, x);
}
```

• worst part in my view
  – can change argument style (e.g., `int` to `int&`)
  – with no compiler warnings (not a problem when variable is passed)

• preprocessor macros
  – also support this variation
  – one reason that some people dislike them

• solutions? either
  – pick function names that make it obvious which arguments
    might change value (???)
  – or just mark arguments in the code instead

• What about arguments that don’t change?
  – most of your data is class instances
  – don’t want copied onto the stack
  – but a little clunky to write “&” everywhere

• but use `const` with reference arguments!
  - copying struct to stack can be useful, but leave decision to callee
  - `const` came from C++ (wasn’t in early C)
  - stronger in C++ : compiler can actually treat as constants when local to
    compilation unit (e.g., use in array sizes, case statements)