hierarchies of structures

- function specialization
  - dynamic types
  - function pointers
  - function pointer tables

classes...

1A polar - ~ 75% new full, ~25% low
1B ditto

2 avg >12, median ~10 — most struggling usually with
3 avg >20, median ~22 corner cases (empty list, first
4 avg ~23, median ~27 in list)

MT3 results

MT3 mean 69
median 73.5
stdev 22
max 100

Lesson in science:
beware careful with small sample sizes!
hear property... most got it.

I will hold some special office hours, will tell you when
soon, but on weekend.

but... no one came to C++ tutorial!
Review: hierarchy of structures for a "library"

```c
struct double-list-t {
    double-list-t* next;
    double-list-t* prev;
};

struct reference-t {
    double-list-t* link;
    char* author-list;
    char* title;
    int year;
};

struct book-t {
    reference-t ref;
    char* publisher;
    char* address;
    (unsigned long) -> uint64-t ISBN;
};
```

Function to print reference as citation:

```c
void print-citation (reference-t* ref) {
    ...}
```

Static `double-list-t library;` sentinel for any library
(doubly-linked cyclic list)

to print whole library...
```c
double-list-t* elt;
for (elt = library->next; &library != elt; elt = elt->next) {
    print-citation ((reference-t*)elt); /* elt may be reference, book,
    textbook, paper, etc.
    all safe because each has reference of the beginning of struct */
}
What if I want to print citations differently for books, conference papers, and textbooks? (true in practice)

Let them think?

Keep separate lists?

Not necessarily... could add a 'type' field.

(Dynamic type information - available while program executes)

... where? in our example, to double-light or reference (so its always there)

Then, in print-citation

void print-citation (reference ++ ref)

switch (ref -> type)

case TYPE_BOOK:

    break;

    case TYPE_CONF_PAPER:

        break;

    etc.

    break;
If these are long chunks of code for each type (or if we want to use elsewhere), maybe turn into functions...

Case TYPE_BOOK: /* ref is the reference */

/* code for printing book citation */
break;

void print_book_citation (book_t* book)
{

and...

Case TYPE_BOOK:
print_book_citation ((book_t*) ref);
break;

we know that this ref is in fact a book!
But if every case is just a function call, why do we need a switch statement?

- Every case calls a function.

- Every function takes a pointer to one structure as its only argument (pointer is of 'appropriate' type for that function: `book_t` for `print_book_citation`, for example).

\[ \Rightarrow \text{Add a function pointer to `reference_t` instead!} \]

```
struct reference_t {
    char *author_list;
    char *title;
    int year;
};
```

For books, we set the function pointer field to `print_book_citation`.

For structures of other types, we set the function pointer field to the code for their citation-printing functions.
Now, to print a library:

```c
double* elt;

for (elt = library.next; elt != NULL; elt = elt->next)
    reference* ref = (reference*)elt;
    ref->print-citation((ref);
    a field name;
    a function pointer
```

Note also: if we do not want to specialize
the code, we can use other types' functions
safely. [Code for any less specific type, that is: textbook can use book or reference, for example.]

For example, we can choose not to
write `print-book-citation` and instead
set the `print-citation` pointer in all `book-t`'s
to point to the function that prints
a reference- `print-ref-citation`, let's say.
How much extra space do these functions cost us?

One pointer per structure . . . per function defined.

Can we reduce?

What if we made a table/array of function pointers? Only need one per type (not per instance).

Still a pointer! But what if we have 20 such functions instead of 1?
The organization of data and function specialization that you've just seen forms the core of data & function inheritance in C++.

In 1980s, people started wondering, why not just have programmer give the hierarchy and say which functions should be charged for a subtype.

Let the compiler lay out the data, create the function tables, and so forth.

Next: modules $\rightarrow$ classes
C++ classes

intro & relation to data structures

data & function inheritance

member functions / class functions

fields / class data

access control

Summary