ECE 198KL

L37 PA
17 April 2013

hierarchies of structures

review

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

~~C++~~ classes ...

reading — 190 manual "week 14"
— 2X lectures on web page (notes)

MT3 results

1A polar — ~ 75% new full, ~25% low

1B ditto

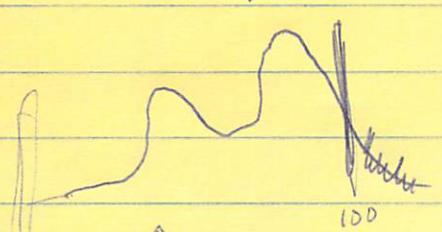
2 avg >12, median ~10 — most struggling, usually with corner cases (empty list, first in list)

3 avg >20, median ~22

4 avg ~~>0~~ 23, median ~27

MT3 mean 69
median 73.5
stdev 22
max 100

lesson in science.
be careful with small sample sizes!
(heap property... most got it)



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

but ... no one came to GPB tutorial!?

review: hierarchy of structures for a "library"

```
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;
    uint64_t ISBN;
};
```

(unsigned long long)

function to print reference as citation...

```
void print_citation(reference_t* ref);
```

```
static double_list_t library; ← sentinel for my library
                                (doubly-linked cyclic list)
```

to print whole library...

```
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
                                         at 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)

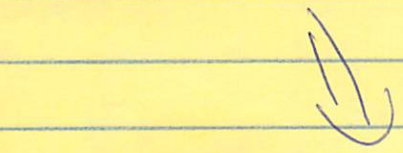
... where? in our example, to double_list_t or reference_t (so it's always there)

Then, in print_citation

```
void print_citation (reference_t* ref)
{
    switch (ref->type) {
        case TYPE_BOOK:
            ...
            break;
        case TYPE_CONF_PAPER:
            :
            break;
        etc.
    }
}
```

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...

we know that this ref is in fact a book!

```
case TYPE_BOOK:
    print_book_citation ((book_t*) ref);
    break;
```

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).

⇒ Add a function pointer to reference_t instead!

```
struct reference_t {
    double_list_t link;
    enumerated_type_t type;
    char* author_list;
    char* title;
    int year;
};
```

→ replace with

```
void (*print_citation)(reference_t* ref);
```

↙ Change to 'correct' pointer type inside the function

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---

```
double_list_t* elt;
for (elt = library.next; &library != elt; elt = elt->next) {
    reference_t* ref = (reference_t*)elt;
    ref -> print_citation (ref);
}
```

print_citation: a field name; a function pointer

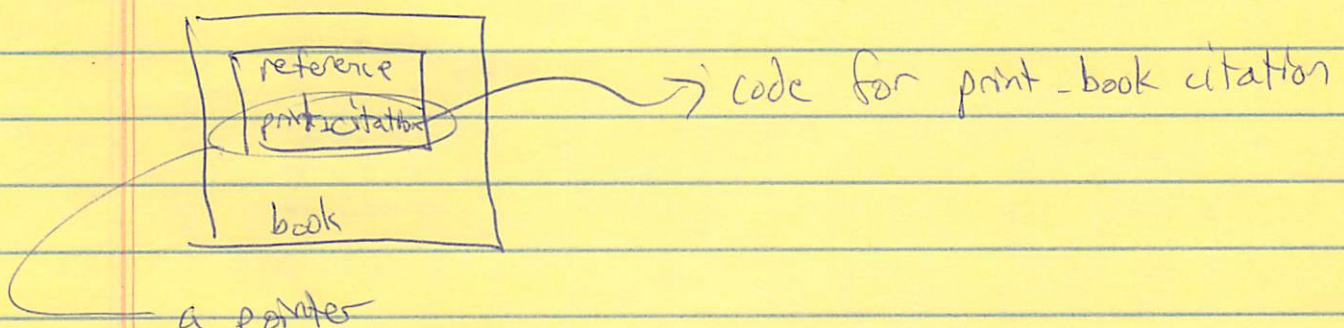
Note also: if we do not want to specialize the code, we can use other types' functions safely [code for any less specific type, that is= textbook can use book or reference, for example]

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_t (print_ref_citation, let's say).

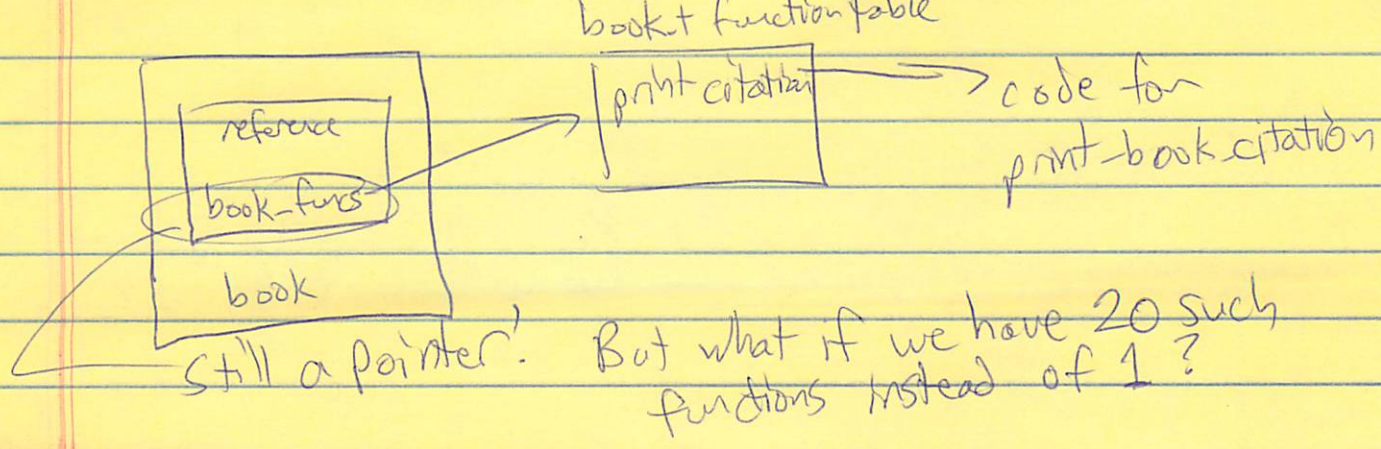
How much extra space do these functions cost us?

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).



The organization of data and function specialization that you've just seen forms the core of data & function inheritance in C++.

In 80s, people started wondering, why not just have programmer give the hierarchy and say which functions should be changed for a subtype

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

Next: modules → classes

CS110

C++ classes

- intro & relation to data structures
- data & function inheritance
- member functions / class functions
- fields / class data
- access control
- Summary