University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

Hierarchies of Structures

Bibliographies Reference Many Types of Documents

What if we have a large group of inter-related types?

For example, documents in a bibliography.

For each document, we have **a reference**.

But some **documents are papers**, while others are **books**.

Some **papers are conference papers**, while others are **journal papers**.

And some **books are textbooks**, while others may be **part of a series**.

Type Hierarchy Implies Structural Inclusion Relations

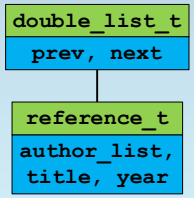
The **basic type is a `reference_t`**.

It has an **`author_list`**, a **`title`**, and a **`year`** of publication.

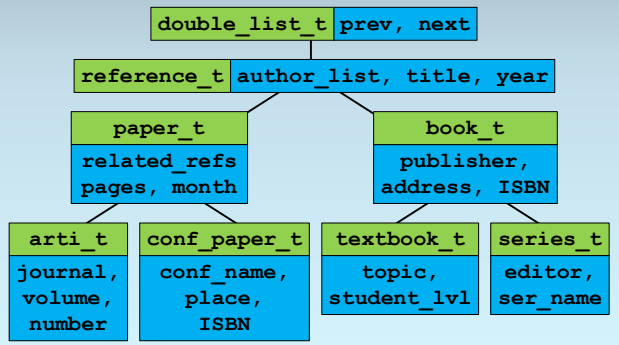
To organize references, **we include a `double_list_t`** (at the start of the **`struct`**).

The **reference is shown as a child or sub-type of the `double_list_t`**.

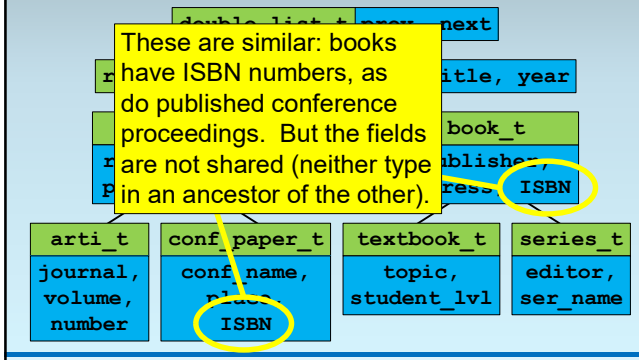
The result is called a **type hierarchy** (or, in other languages, a **class hierarchy**).



Example: A Bibliographic Type Hierarchy



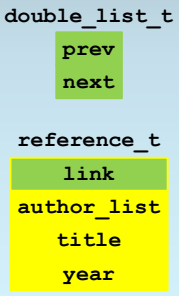
Unrelated Types Sometimes Have Similar Fields



Structures Include Their Parent Type at the Start

Omitting typedefs, we write…

```
struct double_list_t {
    double_list_t* prev;
    double_list_t* next;
};
struct reference_t {
    double_list_t link;
    char* author_list;
    char* title;
    int32_t year;
};
```



A Book's First Field is a Reference

And, for books,

```
struct book_t {
    reference_t ref;
    char* publisher;
    char* address;
    uint64_t ISBN;
};
```

Pointers Can Safely Be Cast to Ancestor Pointers

What does a book really look like in memory?

Notice that
- a **book_t*** also
- **points to a reference_t**
- **and to a double_list_t.**

It is **SAFE to cast pointers to pointers of any ancestor type.**

How Can We Print Citations for All References?

Say we have a **function**

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

that **prints a citation** for a reference.

We **want to print citations for all references in** a bibliography (a sentinel):

```
static double_list_t biblio;
```

How can we accomplish this task?

Loop Over All Elements, Cast Explicitly, and Call

Let's do it directly this time…

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

Iterate over all elements in the bibliography.

Cast forces change of type. Value is not affected for pointer casts. All elements are references.

All Elements of Bibliography Start with a Reference

Note that **a given element of `biblio` may not be a reference**.

So how is casting acceptable?

The element may also be a paper, a book, an article, a conference papers, a textbook, or a book from a series.

But **all of them start with a `reference_t`!**

(`reference_t` is the ancestor of all of the types.)

Can We Cast to Pointers of Subtypes?

So that's nice.

Unfortunately, in reality,

- citations for books are printed differently
- than citations for conference papers and
- differently than citations for journal articles.

What can we do?

Can we cast from `reference_t*` to `book_t*`?
Or to `article_t*`? (to subtypes in general?)

Cannot Know What's in Memory After a Reference

In other words,

given only a `reference_t*`,

can we safely cast to a subtype pointer?

Absolutely not!

Without additional information, we have
no way to know what's in memory after the `reference_t`.

Separating the Bibliography by Type is Cumbersome

So … keep a bibliography for each type?

No. Separate lists are unattractive.

Why bother to have a type hierarchy
- if we have to operate separately
- on every type
- for every operation?

Instead, add dynamic type information.

Record Actual Type in a New Field

```
struct reference_t {
    double_list_t link;
    char* author_list;
    char* title;
    int32_t year;
    int32_t type;
};
```

a `reference_t`?
a `paper_t`?
a `book_t`?
an `arti_t`?
a `conf_paper_t`?
a `series_t`?
a `textbook_t`?

Use `switch` to Handle Each Subtype Separately

Then, in `print_citation`…

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

Print in a distinct style for each subtype.

Perhaps Useful to Write as Separate Functions

But maybe the citation printing code
- is lengthy, or
- is useful elsewhere.

In that case, write…

```
print_paper_citation
    (paper_t* paper);
print_book_citation
    (book_t* book);
```

and so forth.

Each Case is Then Just a Function Call

Then, in **print_citation**…

```
switch (ref->type) {
    case TYPE_PAPER:
        print_paper_citation
            ((paper_t*)ref);
        break;
    // ...
}
```

Safe to cast: we know that **ref** points to a **paper_t**.

A Function Pointer Has a Specific Signature

But if every case is just a function call…

…why not just add a function pointer to reference_t instead?

One problem: the **functions don't have the same signatures**…

```
print_paper_citation
    (paper_t* paper);
print_book_citation
    (book_t* book);
```


Solution: Use a Pointer to the Known Ancestor Type

Let's
- **replace** all pointers **with reference_t***
- and **cast** to the appropriate type **in each function** (we know it's safe in each case).

For example,

```
print_book_citation
    (reference_t* ref) {
    book_t* book = (book_t*)ref;
    // ...
```


Add a Function Pointer for Printing a Citation

```
struct reference_t {
    double_list_t link;
    char* author_list;
    char* title;
    int32_t year;
    void (*print_citation)
        (reference_t* ref);
};
```


Function Pointers Allow Partial Customization

In any **`paper_t`**, the **`print_citation`** field **points to `print_paper_citation`**.

Similarly, **in** any **`book_t`**, the **`print_citation`** field **points to `print_book_citation`**.

And so forth.

Note that **we do not have to specialize**
- for example, we might choose to have
- **`print_citation`** point to **`print_book_citation`** in any **`textbook_t`**.

To Print All, Loop Over All Elements and Call

Now to print our bibliography…

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

Iterate over all elements in the bibliography.

Function called now depends on actual type of **`*ref`**.

Cost Increases if We Customize Many Functions

As you saw,
- the **code is a little cleaner**
- but **allows customization of functionality by type**.

What's the cost?

One pointer per structure variable …

… per function customized.

Can we reduce this cost?

Instead, Use One Table of Function Pointers per Type

What if, **instead**,
- **for each type**
- **we create a table** (a **`struct`**[+])
- filled **with function pointers**?

We only need **one such table per type**, NOT one table per variable.

And one pointer per variable—to the variable's type's table of function pointers.

[+]An array of pointers, but the functions have different signatures.

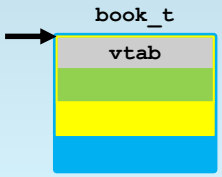
Virtual Function Table Pointer Refers to Table for Type

Let's **call the table pointer `vtab`**.[+]

And place it **at the start of the structure**, so that we can always find it.

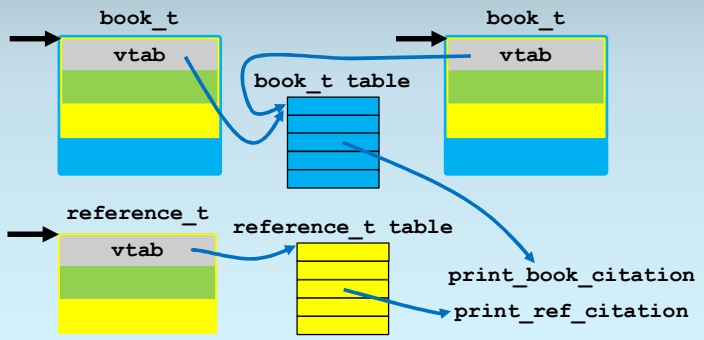
Every book's `vtab` points to the table for books.

Every reference's `vtab` points to the table for references.



[+]For "virtual function table."

Calling a Virtual Function Requires Two Memory Reads



A Virtual Function Call Costs Extra Memory Reads

Two loads followed by a call (JSRR, for example), instead of just a call.

That's the more significant cost.

You may want to try converting our bibliography printing code into **LC-3** assuming that **`vtab`** is at the start of each structure (before the **`double_list_t`**). Choose some non-zero index for **`print_citation`** in the table to make it interesting (the index must be a constant, of course).

