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

ECE 220: Computer Systems & Programming

Structured Data in C

Sometimes, Knowing Which Thing is Enough

In MP6,
- we **represented** the current piece type
- **with** a small integer
- (**a bit pattern**).

Sometimes,
- **such a representation suffices**:
- in MP6, we just needed to know which piece.

ECE120 treated all representations that way.

Often Want to Group Data Together Conceptually

More frequently, **we want**
- **to record several pieces of information** about a given thing,
- and **to group these data together** conceptually.

Examples:
- LC-3 instructions encode several fields.
- MP2 and MP3 used "events" (a name, a set of days, and a time or set of times).

How Does One Describe a Book?

Imagine that you want to
- track your personal library
- as an app on your phone.

What do you want to know about a book?

author	length (in pages)
title	price
ISBN	edition

Can Also Define Operations on a Group of Data

In addition to grouping information,
- **we associate operations (functions)**
- **with such a grouping**.

For example, in MP2 and MP3, you wrote
- event insertion into a schedule,
- selecting a possible hour for an event, and
- event deletion from a schedule (for popping the stack in MP3).

What Can One Do with a Book?

Given information about a book, we can…
- print a citation
- find an author in a list of authors
- compare with online prices
- check whether we have the latest edition
- find other books by the same author
- …

Abstract Definition of a Data Structure

In the abstract, a **data structure** is

1. A **logical grouping of several pieces of data**, and
2. Some **operations that manipulate those data**.

One Can Build Data Structures with Arrays

Technically,
- you know enough
- to use data structures in **C**.

How? Use an array for each field.

`author[42]` corresponds to **`title[42]`**, **`price[42]`**, and so forth.

As any good Fortran programmer will tell you, that's all you need, so get to work!

C Allows Programmers to Define Structures

Letting the compiler

- **know about the grouping**
- is **far more convenient**
- and **less error-prone**.

For that purpose, **C allows programmers to define structures**.

Let's see how a book might look as a **C** structure.

Definition of a C Structure Representing a Book

```
struct book_t {
    char     author[50];
    char     title[100];
    uint64_t isbn;
    int32_t  pages;
    double   price;
    int32_t  edition;
    // and any other fields we want
};
```


A Structure Definition Defines a Structure Type

A `struct` **definition**

- **does not create instances** of the `struct`.
- Instead, it **defines a type**.
- In our example, the new type is **`struct book_t`**.

Then we can **declare variables**…

```
struct book_t book;
```

…**in the same way** as with other types.

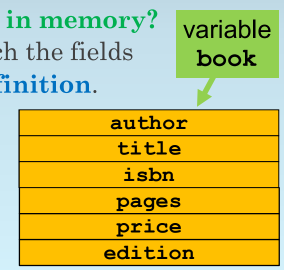
Fields in Memory Ordered as in Struct Definition

```
struct book_t book;
```

How is book laid out in memory?

- **In the order** in which the fields
- are **listed in the definition**.

(Fields are not shown to scale here. Each takes an appropriate number of memory locations.)



`sizeof (expr)` Evaluates to Number of Bytes Needed

```
struct book_t book;
```

When you need
- the size of a structure,
- use the **`sizeof ()`** operator
- with a variable or an expression.

For example,

```
sizeof (book)
```

evaluates to the **number of bytes occupied by the variable `book`** (a `struct book_t`).

Pitfall: Using `sizeof` with a Type

You will see code using a type with **`sizeof`**.

For example,
- **`sizeof (struct book_t)`** in place of
- **`sizeof (book)`**.

This code will work correctly…

…until someone changes the type of `book`.

Just hope that they remember to change the type used with **`sizeof`**, too.

Pitfall: "Calculating" Sizes

You **may want to calculate a size** yourself.

Avoid doing so if possible:
- **sizes change** from ISA to ISA,
- and sometimes from OS to OS,
- or even from compiler to compiler.

Compilers must guarantee aligned accesses
- and thus **sometimes insert padding**
- between fields or at the end of a **`struct`**.

Most ISAs Impose Alignment Requirements

What is an alignment requirement?

Most **ISAs** (with byte-addressable memory) **require that**
- **loads and stores of N bytes**
- **use addresses that are multiples of N**.

For example,
- trying to load a 32-bit value (4B)
- from address 0x20000001 (= 1 mod 4)
- causes a program to crash.

Compilers Must Produce Working Assembly Code

Even **ISAs that**
- **do not require aligned accesses**
- **execute unaligned accesses slowly** (sometimes as much as ~100× slower).

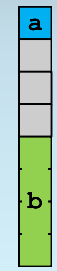
Compilers must produce working code.

Thus **compilers align**
- **fields to their size** (for primitive types),
- and **structures to the maximum alignment needed by any field**.

A Padding Example

Consider:

```
struct one_t {
    int8_t a;
    int32_t b;
};
```



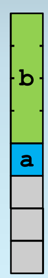
A `one_t` **must be 4-byte aligned** because of `b`.

After `a`, a compiler
- inserts **3 bytes of padding**
- so that `b` is aligned properly.

Changing Order May or May Not Affect Size

Consider:

```
struct one_t {
    int32_t b;
    int8_t a;
};
```



What if we change the order?

Same result: 8 bytes.

(Arrays of `one_t`s must have proper alignment, too.)

Field Access Operator . Accesses a Structure's Fields

The **C operator for field access is**

. (a period).

For example, given

```
struct book_t book;
```

we can write

```
book.author   // the author field
book.title    // the title field
```


Fields of a Structure are Just Like Other Variables

Fields act
- **like any other variable**
- **of the field's type.**

With our book example,

`book.pages` has type `int32_t`,

`book.price` has type `double`, and

`book.author` has type `char*` (the `author` field is an array of characters, so the field name has type `char*`).

Structure Types Must By Default Include "struct"

By default,
- the **name of a structure type** in **C**
- **must include** the keyword **`struct`**.

For example:

```
struct book_t a_book, another_book;
```


Structure Assignment Copies All Bits

```
struct book_t a_book, another_book;
// … some code to fill in a_book

// What does this assignment do?
another_book = a_book;
```

Copies all bits from a_book into another_book.

Pass Pointers to Structures, not Structures, as Arguments

```
struct book_t a_book, another_book;
// … some code to fill in a_book
another_book = a_book;

// Why pass a structure's address?
my_book_printer (&another_book);
```

To avoid copying the entire structure onto the stack.

Call-by-Value Demands Copies of Structure Arguments

If you pass a structure to a **C** function,
- **call-by-value semantics demand**
- that the **compiler make a copy** of the structure.

Every function call must make a new copy.

Structures can be large.

Doing so is rarely acceptable.*

*A complex number composed of two floating-point numbers is an example of a possible exception.

Let's Define a Stack Structure to Solve a Problem

Let's do an example. Let's **develop**
- a **stack structure** and
- some **operations on a stack**,
- then use the stack to solve a problem.

Our stack structure?

`struct stack_t`

The task:
- **read input line by line,**
- **then print it out in reverse.**

Compiler Must Be Able to Know a `struct`'s Size

```
struct stack_t {
    // 500 lines of up to 200 chars
    char data[500][200];
    int32_t top;
};
```

Why only 200 characters per line?

And why only 500 lines?

Fields must have known size.

Fields Can Have Pointer Types

But …

wait a minute …

a pointer has known size, too!

Later, we will learn how to allocate memory dynamically.

For now, we have to pick values, so
- at most 500 lines, and
- at most 200 characters per line.

`top` Field Indicates Which Elements Are Meaningful

```
struct stack_t {
    // 500 lines of up to 200 chars
    char data[500][200];
    int32_t top;
};
```

`top` holds index of data element on **top of the stack**, so
- when stack is **empty, `top` is 500**, and
- when stack is **full, `top` is 0**.

The `fgets` Function Reads a Line from a Stream

To read lines from the keyboard,
- we will **use an input routine**
- **from C's standard library**:

```
char* fgets (char* s, int size,
             FILE* stream);
```

The **`fgets`** function
- **reads** up to (**`size – 1`**) characters or **until the end of a line** (whichever comes first)
- **into array `s`** and
- **returns `s` on success, or `NULL` on failure**.

Use `fgets` to Read from the Keyboard

```
char* fgets (char* s, int size,
             FILE* stream);
```

For now,*
- ignore the **`stream`** argument, for which
- we use **`stdin`** to read from the keyboard.

*We will study I/O in a few weeks.

The `strcpy` Function Copies a String

We will also use **a standard C library function that copies strings**:

```
char* strcpy (char* dest,
              const char* src);
```

`strcpy`
- **copies the string from `src`**
- **into the array at `dest`**.
- The destination must have enough space! (No checking can be done by the function.)

Begin by Initializing the Stack

Let's write the code.

```
int main ()
{
    char buf[200];
    struct stack_t stack;

    stack.top = 500;
```

- `char buf[200];` — a buffer to store one line
- `struct stack_t stack;` — a stack
- `stack.top = 500;` — Initialize stack to empty.

Read from Keyboard Until Stack Full or Input Ends

```
while (0 < stack.top &&
     NULL != fgets
         (buf, 200, stdin)) {
    strcpy (stack.data[--stack.top],
            buf);
}
```

- `0 < stack.top` — Stack not full?
- `NULL != fgets (buf, 200, stdin)` — Got line from keyboard?
- `strcpy` — Copy from **buf** into stack.
- `--stack.top` — Decrement, then use index.

Logical AND Shortcutting Prevents Read with Full Stack

```
while (0 < stack.top &&
     NULL != fgets
         (buf, 200, stdin)) {
    strcpy (stack.data[--stack.top],
            buf);
}
```

- `&&` — Important: If the stack is full, no line is requested (**fgets** is not called).

Print a Line, Pop, and Repeat Until Stack is Empty

```
while (500 > stack.top) {
    printf ("%s",
        stack.data[stack.top++]);
}
return 0;
} // end of main
```

- `500 > stack.top` — Stack not empty?
- `printf` — Print one line (includes LF).
- `stack.top++` — Use index, then increment.

Data Structures Should Hide Their Implementations

The code works, but doesn't exhibit good style.

A **good data structure**
- **allows other code to use the structure**
- **and operations** defined **on the structure**
- **without knowing** details of the structures's **implementation**.

Such a structure illustrates "**information hiding**" (Parnas, 1972).

Why is Information Hiding Useful?

Example: choice of 500-line limit

Why shouldn't users know?

Imagine that 100 programs use our stack.

Then we change from 500 to 1,000 lines.

Now we **need to** find and **update**
- stack initialization, and
- any checks for stack empty
- in **100 programs**!

Remember: Pass Pointers, Not Structures

Instead, we can write functions
- to initialize a **stack_t** and
- to check whether a **stack_t** is empty.

Let's start with the second.

How about…

```
int32_t stack_empty
     (struct stack_t s);  ?
```

Our struct is ~100kB! Don't force compiler to make a copy!

A Function to Check Whether a **stack_t** is Empty

```
// Returns 1 if stack is empty, or
// 0 if stack is not empty.

int32_t stack_empty
     (const struct stack_t* s)
{
     return (500 == (*s).top);
}
```

pointer to a **struct stack_t**

Parentheses required; . has precedence over *

Use the `->` Operator to Access Fields after Dereferencing

One more operator:
- **->**
- **dereference and access a field**

Rather than writing

`(*s).top` ,

we can write

`s->top` .

The two expressions are equivalent.

Revised Function to Check Whether a `stack_t` is Empty

```
// Returns 1 if stack is empty, or
// 0 if stack is not empty.


int32_t stack_empty
     (const struct stack_t* s)
{
     return (500 == s->top);
}
```

Use the -> operator.

A Function to Initialize a `stack_t`

Notice the human naming convention: the **`stack_`** prefix tells programmers that the function deals with a **`stack_t`**.

```
void stack_init (struct stack_t* s)
{
     s->top = 500;
}
```


What Other Operations Do We Want for `stack_t`?

What other operations might we write for our stack?
- **Check whether a `stack_t` is full,**
- **push a string onto a `stack_t`, and**
- **pop a string from a `stack_t`.**

The first is easy.

For push/pop, we need to make choices.

A Function to Check Whether a `stack_t` is Full

```
// Returns 1 if stack is full, or
// 0 if stack is not full.

int32_t stack_full
     (const struct stack_t* s)
{
     return (0 == s->top);
}
```


Information Hiding and Performance Sometimes at Odds

How do we push a string without exposing details of the implementation?

For example,
- should we make a copy of the string, or
- just copy the pointer passed in?

Caller or callee
- must ensure that string does not disappear after it is pushed,
- but which one? Copying twice is wasteful.

Let's retain our current design, so **stack_push must make a copy**.

How Can We Handle Long Strings? Fail…

What should happen if caller passes a string longer than 199 characters?
- Fail? A valid choice, but not so useful.
- Copy the first 199? Also valid, but may not be what the user wants.
- We have no other choice with the current implementation!

We will go with failure for simplicity.

A Function to Push a String Onto a `stack_t`

```
// Returns 1 on success,
// or 0 on failure.
int32_t stack_push (struct stack_t* s,
                    const char* str)
{
     int32_t i;
     char*   write;
     if (stack_full (s)) {
         return 0;
     }
```

the stack

the string

for copying string

No space on stack? Fail.

Use a `char*` to Point to Stack Element to be Written

Decrement, then use index.

Copy to this element on stack.

```
write = s->data[--s->top];
```


Loop Until End of String or Out of Space

Loop until end of str.

```
for (i = 0; '\0' != *str; i++) {
    if (199 == i) {
        s->top++;
        return 0;
    }

    *write++ = *str++;
}
```

If **`str`** is too long, restore stack top and fail.

Copy a character and advance pointers.

End the String and Return Success

Write NUL to end of string.

```
    *write = '\0';
    return 1;
}
```

Push has succeeded.

Must Also Copy in `stack_pop`

What about popping a string?

We make a copy in **`stack_push`**.

After **`stack_pop`** returns,
- the copy is no longer on the stack,
- thus a call to **`stack_push`** will overwrite it
- so we should not return a pointer to the copy.

The implication?

`stack_pop` must also make a copy.

How Can We Handle Long Strings? Fill Array…

Caller to `stack_pop`

- **must provide a space** (an array) **for copy**.
- For safety, **must also pass length** of array.

What should happen if caller passes an array shorter than the stored string?

- Fail? But their code pushed the string!
- Fill the array and add a NUL? Maybe the best choice in this case.

We will go with filling the array.

A Function to Pop a String from a `stack_t`

```
// Returns 1 on success,
// or 0 on failure.
int32_t stack_pop (struct stack_t* s,
    char* buf, int32_t len)
{
    int32_t i;
    char*   read;
    if (stack_empty (s)) {
        return 0;
    }
```

the stack

the length

the array

Stack is empty? Fail.

for copying

Copy String into Buffer Provided by Caller

Copy from element on top of stack.

```
read = s->data[s->top];
for (i = 1;
     len > i && '\0' != *read;
     i++) {
    *buf++ = *read++;
}
```

Copy a character and advance pointers.

Loop (**`len – 1`**) times or until end of string.

Finish the String, Pop the Element, and Return Success

Write NUL to end of string.

```
    *buf = '\0';
    s->top++;
    return 1;
}
```

Pop copied element from stack.

Pop has succeeded.

Begin by Initializing the Stack

Now we can rewrite our code
(**new parts in blue**).

```
int main ()
{
    char buf[200];
    struct stack_t stack;

    init_stack (&stack);
```


Read from Keyboard Until Stack Full or Input Ends

Check to avoid **fgets** with a full stack.

```
while (!stack_full (&stack) &&
      NULL != fgets
          (buf, 200, stdin)) {
    if (!stack_push (&stack, buf)) {
        break;
    }
}
```

If push fails, stop reading input.

Print a Line, Pop, and Repeat Until Stack is Empty

Stack not empty?

```
while (!stack_empty (&stack)) {
    if (!stack_pop (&stack, buf,
                    200)) {
        break;
    }
    printf ("%s", buf);
}
return 0;
} // end of main
```

If stack pop fails, give up.

Print one line (includes LF).

