Computer Systems Principles

C Structures
Announcements

• Mid-term (Th, Feb 25) exam on paper, open book, close notes
Announcements

• Quiz 4 and HW3 are out
Learning Objectives

• To learn and apply C structures
• To understand a little about alignment
• To understand and apply C typedef
C Structures

• Essential
  – For building up interesting data structures

• Definition
  – A C structure is a collection of one or more variables, typically of different types, grouped together under a single name for convenient handling
  – Kind of like a Java class with public instance variables and no methods
C struct

- Defines a new type
  - A new kind of data type that the compiler regards as a unit or aggregate of variables/types.

- Example:

```c
struct Date {
    int day;
    int month;
    int year;
};
```
C struct

• Defines a new type
  – A new kind of data type that the compiler regards as a unit or aggregate of variables/types.

• Example:

```c
struct Date {
    int day;
    int month;
    int year;
};
```
C struct

• **Defines a new type**
  – A new kind of data type that the compiler regards as a unit or aggregate of variables/types.

• **Example:**

```c
struct Date {
    int day;
    int month;
    int year;
};
```

A struct is named as a whole while individual members are named using field identifiers.
More **struct** Examples

- **Examples:**

```c
struct Date {
    int day;
    int month;
    int year;
};

struct Employee{
    char ename[20];
    int ssn;
    float salary;
    struct Date doj;
};
```

Members can be of different types (primitive, array, or struct)
More struct Examples

• Examples:

```c
struct Date {
    int day;
    int month;
    int year;
};

struct Employee {
    char ename[20];
    int ssn;
    float salary;
    struct Date doj;
};

struct Employee[3];
```
Declaring a `struct` Variable

• Declaration of a variable of struct type:

```c
<struct type> <identifier list>;
```

• Example:

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};
struct StudentRecord student1;
```
Declaring a `struct` Variable

- Declaration of a variable of struct type:
  
  `<struct type> <identifier list>;`

- Example:
  
  ```
  struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
  };
  struct StudentRecord student1;
  ... = student1.name
  student1.gpa = ...
  ```
sizeof(struct StudentRecord) =?

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};
```

38 bytes
student-01.c example

• Let us compile this example
  – What did you notice about the output of this program?
  – Is the size of this struct the same as we predicted?
  – Why is this or is this not the case?
student-01.c example

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};
```

```
25
  
4
  
1
  
8
  

name  id  gen  gpa

38 bytes
```

sizeof(struct StudentRecord) = 48
Data Allocation and Alignment

• **Data Allocation**
  – Each variable definition is allocated bytes in memory according the type of that variable
  – e.g., char = 1 byte, int = 4 bytes, double = 8 bytes
  – This is allocated in a special place in memory known as the stack.

• **Data Alignment**
  – Alignment helps the memory store data in a structured manner
    • e.g. 2-byte shorts must start on an even address
  – Machines are more efficient if allocated data is accessed in a structured manner
Data Alignment

- All types except char doesn’t normally start at an arbitrary address.
- A 1-byte char can start on any byte address
- A 2-byte short must start on an even address
- A 4-byte int/float must start on an address divisible by 4
- A 8-byte long/double must start on address divisible by 8
Alignment in Memory

Each box is a byte
Alignment in Memory

Each box is a byte
and has a location.

Memory is very much like a giant character array!
Alignment in Memory

So, how do we allocate memory for the variable declarations below?

```c
long a;
char b;
int c;
```
Alignment in Memory

So, how do we allocate memory for the variable declarations below?

```
long a;  // 8 + 1 + 4 = 13 bytes
char b;
int c;
```

8 + 1 + 4 = 13 bytes ??
Alignment in Memory

So, how do we allocate memory for the variable declarations below?

```c
long a; 8 + 1 + 4 = 13 bytes ??
char b;
int c;
```
So, how do we allocate memory for the variable declarations below?

```c
long a; char b; int c;
```

8 + 1 + 4 = 13 bytes ??
So, how do we allocate memory for the variable declarations below?

```c
long a;
char b;
int c;
```

8 + 1 + 4 = 13 bytes ??
Alignment in Memory

So, how do we allocate memory for the variable declarations below?

```
long a;
char b;
int c;
```

$8 + 1 + 4 = 13$ bytes ??
So, how do we allocate memory for the variable declarations below?

```c
long a;
char b;
int c;
```
So, how do we allocate memory for the variable declarations below?

```c
long a;
char b;
int c;
```
So, how do we allocate memory for the variable declarations below?

long a;  8 + 1 + 3+ 4 = 16 bytes
char b;
int c;
Alignment in Memory

What if c is a 2-byte short?

```c
long a;
char b;
short c;
```
Alignment in Memory

What if c is a 2-byte short?

```
long a;
char b;
short c;
```

8 + 1 + 2 = 12 bytes
Alignment in Memory

What if c is a 8-byte long?

```c
long a;
char b;
long c;
```
What if c is a 8-byte long?

long a; 8 + 1 + 7 + 8 = 24 bytes
char b;
long c;
What is the size of padding in bytes between `a` and `c` on a 64-bit machine?

```c
char b;
long a;
int c;
```

A. 0  
B. 1  
C. 2  
D. 3
i-clicker question

What is the size of padding in bytes between \( b \) and \( a \) on a 64-bit machine??

```c
char b;
long a;
int c;
```

A. 0  
B. Determined by the address of \( b \)  
C. 7  
D. 3
Reduce memory size

- We can reduce used memory size by reorder declaration statements

```c
char b;
long a;
int c;
```

```c
long a;
int c;
char b;
```

\[1 + 8 + 4 + ? = 13 + ?\]  \[8 + 4 + 1 = 13\]

*bytes*
Alignment in Memory

• There is *no* internal padding within a primitive type array
• But **there can be** internal padding within a struct
• **There can be** internal padding within struct array.
Structure alignment

• Structure alignment requirements:
  – Within a struct, each member has different alignment requirements.
  – The structure as a whole has the alignment of its widest scalar member
Structure alignment

• Structure alignment requirements:
  – Within a struct, each member has different alignment requirements.
  – The structure as a whole have the alignment of its widest scalar member

• Find memory layout
  – Step 1: Align each member as if these member has been separately declared
  – Step 2: Pad at the end so the structure has the alignment of its widest scalar member
What's the memory layout of struct foo?

```
struct foo {
    long a;
    char b;
    int c;
};
```
What’s the memory layout of struct foo?
Step 1: Align each member as if these member has been separately declared

```c
struct foo {
    long a;
    char b;
    int c;
};
```
What's the memory layout of struct foo?
Step 1: Align each member as if these member has been separately declared
Step 2: Pad at the end so the structure has the alignment of its widest scalar member

```c
struct foo {
    long a;
    char b;
    int c;
};
```
What’s the memory layout of struct bar?

```c
struct bar{
    char b;
    long a;
    int c;
};
```
What's the memory layout of struct bar?
Step 1: Align each member as if these member has been separately declared

```c
struct bar{
    char b;
    long a;
    int c;
};
```
What’s the memory layout of struct bar?
Step 1: Align each member as if these member has been separately declared
Step 2: Pad at the end so the structure has the alignment of its widest scalar member

```c
struct bar{
    char b;
    long a;
    int c;
};
```
What’s the memory layout of struct `bar`?

Step 1: Align each member as if these member has been separately declared

Step 2: Pad at the end so the structure has the alignment of its widest scalar member

```c
struct bar{
  char b;
  long a;
  int c;
};
```
i-clicker question

What is the size of the following structure on a 64-bit machine?

```c
struct baz {
    long a;
    char c;
};
```

A. 9  
B. 24  
C. 20  
D. 16
What is the size of the following structure on a 64-bit machine?

```c
struct qux {
    char a;
    char b;
    char c;
    char d;
    char e;
};
```

A. 10  
B. 20  
C. 5   
D. 8
student-01.c example

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};
```

sizeof(struct StudentRecord) = 48
What’s the memory layout of struct `StudentRecord`?

Step 1: Align each member as if these member has been separately declared:

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};
```

```c
struct StudentRecord student1;
```
Alignment in Memory

What's the memory layout of struct `StudentRecord`? Step 1: Align each member as if these member has been separately declared

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};

struct StudentRecord student1;
```
Alignment in Memory

What’s the memory layout of struct `StudentRecord`?
Step 1: Align each member as if these member has been separately declared

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};

struct StudentRecord student1;
```
What’s the memory layout of struct `StudentRecord`? Step 1: Align each member as if these member has been separately declared.

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};

struct StudentRecord student1;
```
What’s the memory layout of struct StudentRecord?
Step 1: Align each member as if these member has been separately declared

```c
struct StudentRecord {  
    char name[25];  
    int id;  
    char gender;  
    double gpa;  
};

struct StudentRecord student1;
```
Alignment in Memory

What’s the memory layout of struct `StudentRecord`?
Step 1: Align each member as if these member has been separately declared

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};
```

```c
struct StudentRecord student1;
```
Alignment in Memory

What’s the memory layout of struct `StudentRecord`?
Step 1: Align each member as if these member has been separately declared
Step 2: Pad at the end so the structure has the alignment of its widest scalar member

```c
struct StudentRecord {  
    char name[25];
    int id;
    char gender;
    double gpa;
};

struct StudentRecord student1;
```
student-01.c example

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};
```

```c
sizeof(struct StudentRecord) = 48
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

name | id | gen | gpa

38 bytes
student-01offsetof.c

• Explore the memory layout of a struct by yourself:
  – offsetof (type,member)
Structure Initialization

• There are four ways to initialize a struct
  – Positional initialization
  – Named initialization
  – Copy initialization
  – Initialize individual fields
Positional Initialization

**Positional initialization** allows you to provide the values for each of the fields based on the position of each structure member:

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};

struct StudentRecord student1 = {
    "John Doe", 1234567, 'M', 3.95
};

Not a recommended way!
```
Positional Initialization

**Positional initialization** allows you to provide the values for each of the fields based on the position of each structure member:

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};

struct StudentRecord student1 = {
    "John Doe", 1234567, 'M', 3.95
};
```

Not a recommended way!
Named Initialization

Named initialization allows you to provide the values for each of the fields based on the name of each structure member:

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};

struct StudentRecord student1 = {
    .id = 1234567,
    .gpa = 3.95,
    .gender = 'M',
    .name = "Harry Potter"
};
```
Copy Initialization

**Copy initialization** allows you to initialize a structure by assigning an existing structure:

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};
```

```c
struct StudentRecord student1 = {
    .id = 1234567,
    .gpa = 3.95,
    .gender = 'M',
    .name = "Harry Potter"
};
```

```c
struct StudentRecord student2 = student1;
```
Field Initialization

Field initialization allows you to initialize a structure by assigning to its fields:

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};

struct StudentRecord student1;

student1.id = 1234567;
student1.gender = 'M';
student1.gpa = 3.95;
```
Field Initialization

Field initialization allows you to initialize a structure by assigning to its fields:

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};

struct StudentRecord student1;

student1.id = 1234567;
student1.gender = ‘M’;
student1.gpa = 3.95;
student1.name = “Harry Potter”;
```

What about this one?
student-05.c example

• Let us compile this example
  – What problems do we encounter with this example?
  – Why can’t we assign a string to a character array?
    • Arrays are not modifiable values, that is, you can’t reassign them to “point” to different locations in memory.
Field Initialization

Field initialization allows you to initialize a structure by assigning to its fields:

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};
```

```c
struct StudentRecord student1;

student1.id = 1234567;
student1.gender = 'M';
student1.gpa = 3.95;
student1.name = "Harry Potter";
```

So, how do we fix this?
strncpy

• **Copying Strings**
  – `#include <string.h>`
    A library for manipulating C strings
  
  – To assign a new string value to a C string (e.g., character array) you must use the `strncpy` function to *copy* the bytes into the array.
Field Initialization

Field initialization allows you to initialize a structure by assigning to its fields:

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};
```

```c
class StudentRecord student1;
student1.id = 1234567;
student1.gender = 'M';
student1.gpa = 3.95;
strncpy(student1.name, "Harry Potter", 25);
```

We use the `strncpy` function!
**Field Initialization**

**Field initialization** allows you to initialize a structure by assigning to its fields:

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};

struct StudentRecord student1;

student1.id = 1234567;
student1.gender = 'M';
student1.gpa = 3.95;
strncpy(student1.name, "Harry Potter", 25);
```

Be careful about strncpy because of buffer overflow!

Size of the destination

We use the `strncpy` function!
**strlen**

- **Calculate the length of a string**
  - `#include <string.h>`
    - A library for manipulating C strings
  - Return an unsigned integer
  - Example:
    - `strlen("Revenant") == 8`
void f(char[] bar) {
    char c[12];
    strncpy(c, bar, strlen(bar));
}

Buffer overflow example