Structures and Pointers
Structures

A *structure* is a complex data type

- Defined by the programmer
- Keeps together pertinent information of an object
- Contains simple data types or other complex data types
- Similar to a class in C++ or Java, but without methods
Structures

Example from graphics: a point has two coordinates

```c
struct point {
    double x;
    double y;
};
```

x and y are called members of struct point.

Since a structure is a data type, you can declare variables:

```c
struct point p1, p2;
```

What is the size of struct point? 16
Accessing structures

```c
struct point {  
    double x;  
    double y;  
};  
struct point p1;
```

Use the “.” operator on structure objects to obtain members

```c
p1.x = 10;  
p1.y = 20;
```

Accessing structures via pointer

```c
struct point *pp = &p1;  
double d = (*pp).x;
```

Use the “->” operator on structure pointers to obtain members

```c
d = (*pp).x;  
d = pp->x;
```

Initializing structures like other variables:

```c
struct point p1 = {320, 200};  
Equivalent to: p1.x = 320; p1.y = 200;
```
More structures

Structures can contain other structures as members:

```c
struct rectangle {
    struct point pt1;
    struct point pt2;
};
```

What is the size of a struct rectangle? 32

Structures can be arguments of functions

Passed by value like most other data types

Compare to arrays

```c
int ptinrect(struct point p, struct rectangle r) {
    return (p.x >= r.pt1.x) && (p.x < r.pt2.x)
    && (p.y >= r.pt1.y) && (p.y < r.pt2.y);
}
```

What does this function return?

1 (=TRUE) when point p is in rectangle r, otherwise 0 (=FALSE)
Operations on structures

Legal operations
- Copy a structure (assignment equivalent to memcpy)
- Get its address
- Access its members

Illegal operations
- Compare content of structures in their entirety
- Must compare individual parts

Structure operator precedences
- “.” and “->” higher than other operators
- \( *p.x \) is the same as \( *(p.x) \)
- \( ++p->x \) is the same as \( ++(p->x) \)
C typedef

C allows us to declare new datatypes using “typedef” keyword

The thing being named is then a data type, rather than a variable

typedef int Length;

Length sideA; // may be more intuitive than “int sideA;”

Often used when working with structs

```c
struct Point {
    double x;
    double y;
} a;
```

```c
typedef struct Point {
    double x;
    double y;
} MyPoint;

MyPoint a;
```

Equivalent

(but must not have multiple definitions of “struct Point”)

C typedef

typedef struct Point {
    double x;
    double y;
} MyPoint;

struct Point a;
MyPoint a; } equivalent

typedef MyPoint * PointPtr;

struct Point * p;
MyPoint * p; } equivalent

PointPtr p;
C typedef

Common to use the same name.

typedef struct Point {
    double x;
    double y;
} Point;

struct Point a;
Point a;  \{ equivalent
May need to declare names before defining names.

typedef struct tnode Treenode;
typedef Treenode * Treeptr;
struct tnode {
    char   *word;
    int     count;
    Treeptr left;
    Treeptr right;
};

Treenode td;
Self-referential structures

A structure can contain members that are pointers to the same struct (i.e. nodes in linked lists)

```c
typedef struct listNode *NodePtr;
struct listNode {
    char * word;
    int count;
    NodePtr next;
};
```

```
typedef struct listNode *NodePtr;
struct listNode {
    char * word;
    int count;
    NodePtr next;
};
```

```
typedef struct listNode *NodePtr;
struct listNode {
    char * word;
    int count;
    NodePtr next;
};
```
Self-referential structures

Declared via typedef structs and pointers

What does this code do?

typedef struct listNode *NodePtr;
typedef struct listNode {
    char  * word;
    int     count;
    NodePtr next;
} Node;

static NodePtr head = NULL;  // The head of a list

NodePtr p;
while (...) {
    // Allocate a new node
    p = (NodePtr) malloc(sizeof(Node));
    // Initialize it
    p->word = ...;
    p->count = ...;
    // Add to front of the list
    p->next  = Head;
    head = p;
}
Structures in assembly

Concept
- Contiguously-allocated region of memory
- Members may be of different types
- Accessed statically, code generated at compile-time

```c
struct rec {
    int i;
    int a[3];
    int *p;
};
```

Accessing Structure Member
```c
void set_i(struct rec *r, int val) {
    r->i = val;
}
```

Memory Layout

```

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>
```

Assembly
```
# %esi = val
# %rdi = r
movl %esi,(%rdi)  # Mem[r] = val
```
Example

```c
def struct rec {
    int i;
    int a[3];
    int *p;
};

int * find_a (struct rec *r, int indx) {
    return &r->a[indx];
}
```

```asm
# %rdi = r
# %esi = indx
leal 0(%esi,4),%rax  # 4*indx
leal 4(%rax,%rdi),%rax # r+4*indx+4
```
Practice Problem

What are the offsets (in bytes) of the following fields?

- `p`: __________
- `s.x`: __________
- `s.y`: __________
- `next`: __________

How many total bytes does the structure require? __________

Fill in the missing expressions:

```c
void sp_init(struct prob *sp) {
    sp->s.x = __________;
    sp->p = __________;
    sp->next = __________;
}
```

```assembly
sp_init:
    movl 12(%rdi), %eax
    movl %eax, 8(%rdi)
    leaq 8(%rdi), %rax
    movq %rax, (%rdi)
    movq %rdi, 16(%rdi)
    ret
```
Practice Problem

What are the offsets (in bytes) of the following fields?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0</td>
</tr>
<tr>
<td>s.x</td>
<td>8</td>
</tr>
<tr>
<td>s.y</td>
<td>12</td>
</tr>
<tr>
<td>next</td>
<td>16</td>
</tr>
</tbody>
</table>

How many total bytes does the structure require?

24

Fill in the missing expressions:

```c
void sp_init(struct prob *sp) {
    sp->s.x = sp->s.y;
    sp->p = &(sp->s.x);
    sp->next = sp;
}
```

```assembly
sp_init:
    movl 12(%rdi), %eax
    movl %eax, 8(%rdi)
    leaq 8(%rdi), %rax
    movq %rax, (%rdi)
    movq %rdi, 16(%rdi)
    ret
```
Aligning Structures

Structures and their members should be aligned at specific offsets in memory

Goal: Align data so that it does not cross alignment boundaries and cache line boundaries
Alignment of structure members

Mostly matches the size of the data type

- **char** is 1 byte
  - Can be aligned arbitrarily

- **short** is 2 bytes
  - Member must be aligned on even addresses
    - (i.e. starting address of short must be divisible by 2)

- **int** and **float** are 4 bytes
  - Member must be aligned to addresses divisible by 4

- **long**, **double** and **pointers** are 8 bytes
  - Member must be aligned to addresses divisible by 8
Alignment Principles

**Aligned Data**
- Primitive data type requires $K$ bytes
- Address must be multiple of $K$
- Required on some machines; advised on x86-64

**Motivation for Aligning Data**
- Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory trickier when datum spans 2 pages

**Compiler**
- Inserts gaps in structure to ensure correct alignment of fields
Specific Cases of Alignment (x86-64)

1 byte: `char`, ...
   no restrictions on address

2 bytes: `short`, ...
   lowest 1 bit of address must be 0₂

4 bytes: `int`, `float`, ...
   lowest 2 bits of address must be 00₂

8 bytes: `double`, `long`, `char` *, ...
   lowest 3 bits of address must be 000₂

16 bytes: `long double` (GCC on Linux)
   lowest 4 bits of address must be 0000₂
Alignment within Structures

Each member must satisfy its own alignment requirement

Overall structure must also satisfy an alignment requirement “K”
- $K =$ Largest alignment of any element
- Initial address must be multiple of $K$
- Structure length must be multiple of $K$
  - For arrays of structures

Questions:
- What is $K$ for $S1$?
- What is the size of $S1$?
- Draw $S1$ and the alignment of elements within it

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Satisfying Alignment with Structures

Within structure:

Must satisfy each element’s alignment requirement

Overall structure placement

Each structure has alignment requirement $K$

$K = \text{Largest alignment of any element}$

Initial address & structure length must be multiples of $K$

Example:

$K = 8$, due to double element

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Meeting Overall Alignment Requirement

For largest alignment requirement $K$

Overall structure must be multiple of $K$

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```
Arrays of Structures

Overall structure length is multiple of K

Satisfy alignment requirement for every element

```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Accessing Array Elements

Compute array offset 12*indx $\text{sizeof}(S3)$, including alignment spacers

Element $j$ is at offset 8 within structure

Assembler gives offset $a+8$

Resolved during linking

```
short get_j(int indx)
{
    return a[indx].j;
}
```

```asm
# %rdi = indx
leaq (%rdi,%rdi,2),%rax # 3*indx
movzwl a+8(%rax,4),%eax
```
Saving Space

Put large data types first

Effect (K=4)

```
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

```
struct S5 {
    int i;
    char c;
    char d;
} *p;
```

<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

| i | c | d | 2 bytes |
Practice problem

- What is the size of this structure?
  24 bytes (w=6, c=12, j=2)

- Write assembly instructions to load a.c[1] into %eax

Start with

```assembly
movq  $a,%rbx
movl  12(%rbx),%eax
```
Practice problem

For each of the following structures determine:

The offset of each field?

The total size of the structure?

The alignment requirement?

```
struct P1 {int i; char c; int j; char d;};
  0, 4, 8, 12  : 16 bytes : 4

struct P2 {int i; char c; char d; int j;};
  0, 4, 5, 8  : 12 bytes : 4

struct P3 {short w[3]; char c[3];};
  0, 6  : 10 bytes : 2

struct P4 {short w[3]; char *c[3];};
  0, 8  : 40 bytes : 8
```
Exercise

```c
struct point {
    double x;
    double y
};

struct octagon {
    // An array can be an element of a structure ...
    struct point points[8];
} A[34];

struct octagon *r = A;
```

What is the size of a struct octagon? 16*8 = 128

What is the difference between the address r and the address A? 128*8 = 1024
Structures can be nameless

A **variable** called `x`, whose type is this structure, which has no name:

- Can not declare other variables of same type
- Can not pass in function parameters

```c
struct {
    char *key;
    int v[22];
} x;
```

A **data type** called `MyStruct`, which is this structure, which otherwise has no name

- Can use type ‘`MyStruct`’ to declare additional variables
- Can use type ‘`MyStruct`’ in function parameters

```c
typedef struct {
    char *key;
    int v[22];
} MyStruct;
```
Structures can be assigned (i.e., copied)

```c
struct MyStruct {
    char *key;
    int v[22];
};
typedef struct MyStruct MyStruct;
main() {
    MyStruct x, y;
    int i;
    // initialize x
    x.key = "hello";
    for(i=0; i<22; i++)
        x.v[i] = i;
    // structure assignment
    y = x;
    // print y
    printf("y.key = %s,  y.v[11] = %d\n", y.key, y.v[11]);
}
```

Note: Arrays can not be assigned, but as part of a structure they can.
Unions

A union is a variable that may hold objects of different types and sizes. Like a structure, but with all the members on top of each other. The size of the union is the maximum of the size of the individual datatypes.

```c
union U1 {
    char c;
    int i[2];
    double v;
} *up;
```
What’s the size of $u$?
What exactly does $u$ contain after these three lines of code?
typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f) {
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}

- Get direct access to bit representation of float
- bit2float generates float with given bit pattern
  - NOT the same as (float) u
- float2bit generates bit pattern from float
  - NOT the same as (unsigned) f
Bit Fields

If you have multiple Boolean variables...

- Bit fields can be packed together into a single byte or word.
- Saves memory space
- Used in device drivers

Example: The system call to open a file:

```c
int fd = open("filename", O_CREAT|O_WRONLY|O_TRUNC);
```

- Second argument is an integer.
- Uses bit fields to specify options.

  - **O_CREAT** = create the file if it does not exist
  - **O_WRONLY** = open it write-only; no reading allowed
  - **O_TRUNC** = reduces its size to zero if it already exists
Implementing Bit Fields

You can use an integer and create bit fields using bitwise operators:

- 32 bit-field flags in a single integer

Using #defines

```c
#define A 0x01
#define B 0x02
#define C 0x04
#define D 0x08
```

- Note that they are powers of two corresponding to bit positions

Using an “enum”

- Constant declarations (i.e. like #define, but values are generated if not specified by programmer)

```c
enum { A = 01, B = 02, C = 04, D = 08 };
```

Example:

```c
int flags;
flags |= A | B;
```
Bit field implementation via structs

Use bit width specification in combination with struct

Give names to 1-bit members

```c
struct {
    unsigned int is_keyword : 1;
    unsigned int isExtern : 1;
    unsigned int isStatic : 1;
};
```

Data structure with three members, each one bit wide

What is the size of the struct? 4 bytes
Pointers to Functions
Pointers

Central to C (but not other languages)

Gives programmer access to underlying data details
(via physical address)
Allows great flexibility; You can write very efficient code

Major concepts so far

- Every pointer has a type
- Every pointer has a value (which is a memory address)
- Pointers created via the “&” operator
- Dereferecened with the “*” operator
- Arrays and pointers are closely related

Next up…

Pointers can also point to functions
Function pointers

Pointers can point to locations of data
Pointers can also point to code locations

Function pointers

You can store and pass references to code
Each has an associated type
* The type the function returns

Some uses

- Dynamic “late-binding” of functions
  * Dynamically “set” a random number generator
  * Replace large switch statements for implementing dynamic event handlers
    » Example: dynamically setting behavior of GUI buttons

- Emulating “virtual functions” and polymorphism from OOP
  * qsort() with user-supplied callback function for comparison
    » man qsort
  * Operating on lists of elements
    » multiplication, addition, min/max, etc.
Function pointers

Example declaration

```c
int (*func)(char *);
```

- `func` is a pointer to a function taking a `char *` argument, returning an `int`
- How is this different from
  ```c
  int *func(char *)
  ```

Using a pointer to a function:

```c
int foo(char *); // foo: function returning an int
int (*bar)(char *); // bar: pointer to a fn returning an int
bar = foo; // Now the pointer is initialized
x = bar(p); // Call the function
```
#include <stdio.h>
void print_even (int i) { printf ("Even %d\n",i); }
void print_odd (int i) { printf ("Odd %d\n",i); }

int main(int argc, char **argv) {
    void (*fp) (int);
    int i = argc;
    if (!(argc%2))
        fp=print_even;
    else
        fp=print_odd;
    fp(i);
}

% a.out a
Even 2
% a.out a b
Odd 3
typedefs with function pointers

Same as with other data types

```c
int (*func)(char *);
```

The named thing, `func`, is a pointer to a function returning an int.

```c
typedef int (*func)(char *);
```

The named thing, `func`, is a data type: a pointer to function returning an int.
A Dispatch Table using Func. Ptrs.

// For each command, we should execute the corresponding operation
int doEcho(char*) {...}
int doExit(char*) {...}
int doHelp(char*) {...}
int setPrompt(char*) {...}

// Define type of pointers to operations
typedef int (*FuncPtr)(char*);

typedef struct {
    char*    name;
    FuncPtr  op_to_do;
} func_t;

// Set up dispatch table
func_t func_table[] = {
    { "echo",   doEcho },
    { "exit",   doExit },
    { "quit",   doExit },
    { "help",   doHelp },
    { "prompt", setPrompt },
};

// Determine the number of entries in the table
#define cntFuncs (sizeof(func_table) / sizeof(func_table[0]))

// find the function and dispatch it
for (i = 0; i < cntFuncs; i++) {
    if (strcmp(command,func_table[i].name)==0){
        done = func_table[i].op_to_do(argument);
        break;
    }
}

if (i == cntFuncs)
    printf("invalid command\n");
Complicated Declarations

C’s use of () and * makes declarations involving pointers and functions extremely difficult

Helpful rules

- * has lower precedence than ()
- Work from the inside-out

Consult K&R Chapter 5.12 for complicated declarations

dc1 program to parse a declaration
C pointer declarations

- `int *p`  
  p is a pointer to int
- `int *p[13]`  
  p is an array[13] of pointer to int
- `int *(p[13])`  
  p is an array[13] of pointer to int
- `int **p`  
  p is a pointer to a pointer to an int
- `int *f()`  
  f is a function returning a pointer to int
- `int (*f)()`  
  f is a pointer to a function returning int
Practice

What kind of things are these?

- `int *func(char*)`: function that takes `char*` as arg and returns an `int*`
- `int (*func)(char*)`: pointer to a fn taking `char*` as arg and returns an `int`
C pointer declarations

Read these from the “inside” out.

```c
int (*(*f())[13])();  // f is a function returning ptr to an array[13]
                      // of pointers to functions returning int

int (*(*x[3])())[5];  // x is an array[3] of pointers to functions
                      // returning pointers to array[5] of ints

char (*(*x())[][])();  // x is a function returning a pointer to
                      // an array of pointers to functions
                      // returning char
```