University of Arizona

CSc 340

Foundations of Computer Systems

Christian Collberg

April 3, 2001

C Structures and Unions

Structure Definition

- A structure is a collection of named data items. Each data item is called a field of the structure. Structures are often called records in other programming languages.
- Here is an example of a C structure that holds a date:

```
struct {
   int      day;
   int      month;
   int      year;
} today;
```

Slide 17–1

Structure Definition. . .

- This structure definition defines the variable today, a structure containing the fields day, month, and year.
- Individual fields of a structure variable are accessed using the '.' operator:

```
today.day = 13;
today.month = 4;
today.year = 2000;
```

Slide 17–2

Structure Definition. . .

- A structure variable can also be defined by first declaring the structure format, then defining the variable:

```
struct date {
   int      day;
   int      month;
   int      year;
};
struct date today;
```

- The structure format has the tag (name) `date`. This is subsequently used to define the variable today.

Slide 17–3

Structure Definition...

- Pointers to structures are very popular in C code:

```
struct date today, *ptr;
ptr = &today.
(*ptr).day = 13;
...
```

- Because of this, shorthand is available for the ⌜`(*ptr).`⌝ construct. ⌜`ptr->`⌝ is equivalent to ⌜`(*ptr).`⌝:

```
struct date today, *ptr;
ptr = &today.
ptr->day = 13;
 ...
```

Slide 17–4

Structure Definition...

- A structure declaration can contain a reference to its own (incomplete) type:

```
struct  info {
   struct info *next;
   int           foo;
};
```

Slide 17–5

Structure operations

- Older C compilers only allow ⌜`.`⌝ and ⌜`->`⌝ on structures. Newer compilers also allow them to be assigned, passed as parameters, and returned by functions. Keep in mind that C is call-by-value, so structure parameters are copied. Because of this, structure pointers are usually used.
- Structures cannot be compared via ⌜`==`⌝. Structures may contain padding to align the fields properly. This may cause different ”garbage” bits in the structure that have the same field values. You have to compare structures field-by-field.

Slide 17–6

Nested Structures

- C allows structures to be nested, although the nesting can't be recursive (obviously):

```
struct {
   char  name[20];
   struct date birthday;
} person, *ptr;
strcpy(person.name, "John");
person.birthday.day = 21;
ptr = &person;
ptr->birthday.month = 11;
ptr->birthday.year = 1965;
```

Slide 17–7

Typedef

- Using the type struct date to define a date structure variable is a bit verbose. The C typedef operation allows you to provide a synonym for an existing type name.

```
typedef int word;
```

defines word to be a synonym for int. Thus:

```
word  foo;
```

defines a variable foo of type int.

Slide 17–8

Typedef...

- It is important to remember that `typedef` does not define a new type, it merely defines a synonym for an existing type. The types are the same, and C won't complain if you interchange them:

```
typedef int word;
word  foo;
int   bar = 10;
foo = bar;
```

Slide 17–9

Unions

- A union is a data type that stores several variables, perhaps of different types, in the same memory. Because the same memory is used, only one variable at a time can hold a value.

```
union {
   char    msg[20];
   int     total;
   short   tax;
} info;
strcpy(info.msg, "hello");
info.tax = 10;
info.total = 1000;
/* Only info.total is valid
   at this point */
```

Slide 17–10

Unions...

- Unions have similar syntax to structures, but don't forget that only one field at a time is valid.
- The `union` variable info contains three variables, `msg`, `total`, and `tax`. `sizeof(info)` is the maximum size of these variables, or 20 bytes.
- Unions can be nested, and can contain structures, arrays, etc.

Slide 17–11

Unions...

- An auxiliary variable is often used to remember what is stored in a union – often this variable and the union are stored in a structure:

```
  struct {
     int    type;
     union {
        char   msg[20];
        int    total;
        short  tax;
     } info;
 } foo;
 foo.type = 0;
 strcpy(foo.info.msg, "hello");
 foo.type = 1;
 foo.info.total = 10;
```

Slide 17–12

Stack.h

```
typedef enum {FALSE,TRUE} boolean;

#define MAXSTACK 10

typedef struct {
  float x, y;
} Point;

typedef Point StackEntry;
typedef struct {
   int top;
   StackEntry entry[MAXSTACK];
} Stack;

/* function prototypes */
void CreateStack(Stack *);
boolean StackEmpty(Stack *);
boolean StackFull(Stack *s);
void Push(StackEntry, Stack *);
void Pop(StackEntry *, Stack *);
int StackSize(Stack *s);
void Error(char *);
```

Slide 17–13

Stack.c

```
#include "<stdio.h>"
#include "stack.h"

// CreateStack: initialize the stack to be empty
void CreateStack(Stack *s) {s->top = 0;}

// StackEmpty: returns 1 if the stack is empty
boolean StackEmpty(Stack *s) {return s->top <=0;}

// StackFull: returns 1 if the stack is full
boolean StackFull(Stack *s) {return s->top >= MAXSTACK;}

// StackSize: return # of items in the stack
int StackSize(Stack *s){return s->top;}
```

Slide 17–14

```
// Push: push an item onto the stack
void Push(StackEntry item, Stack *s) {
  if (StackFull(s)) Error("Stack is full");
  else          s->entry[s->top++] = item;
  }

// Pop: pop an item from the stack
void Pop(StackEntry *item, Stack *s) {
  if (StackEmpty(s)) Error("Stack is empty");
  else          *item = s->entry[--s->top];
  }

// Error: display an error message
void Error(char *message) {
  fprintf(stderr,"Error: %s\n", message);
  exit(1);
  }
```

Slide 17–15

StackTest.c

```
#include "stack.h"
int main () {
  Stack S, *P;
  Point x1, x2;

  P = &S;
  CreateStack(P);
  printf("size = %d\n", StackSize(P));

  x1.x = 0; x1.y = 0; x2.x = 1; x2.y = 1;
  Push(x1, P);
  printf("size = %d\n", StackSize(P));

  Push(x2, P);
  printf("size = %d\n", StackSize(P));

  Pop(&x1,  P);
  printf("size = %d\n", StackSize(P));

  Pop(&x1, P);
  printf("size = %d\n", StackSize(P));
}
```