CS240: Programming in C

Lecture 10:

Review - Structures and Memory Allocation

Unions

Recap: Structures

- Holds multiple items as a unit
- Treated as scalar in C: can be returned from functions, passed to functions
- They can not be compared
- A structure can include
  - a pointer to itself, but not a member of the same structure
  - a member of another structure, the latter has to have the prototype declared before
- Allocation of memory for structure’s fields must respect alignment as dictated by the underlying architecture

Structure recap

- Member access
  - Direct: **`s.member`**
  - Indirect: **`s_ptr->member`**
  - Dot operator . has precedence over indirect access operator ->
    - What does s.t->u mean?
    - How about (s.t) -> u? Or, s.(t->u)?
    - Is there a difference between (*s).t and s->t?

Memory layout for a structure

- Data alignment: when the processor accesses the memory reads more than one byte, usually 4 bytes on a 32-bit platform.
- What if the data structure is not a multiple of 4? Padding.
- Implementations must typically handle alignment.

Manipulating Structures

- What happens when a structure is passed as an argument?

```
#include <stdio.h>

struct Foo {
  int a;
};

struct Foo foo (struct Foo b) {
  b.a = 13;
  return b;
}

int main () {
  struct Foo f;
  f.a = 100;
  foo(f);
  printf("value of structure parameter is %d\n", f.a);
}
```

What gets printed?

Example (cont)

```
#include <stdio.h>

struct Foo {
  int a;
};

struct Foo foo (struct Foo b) {
  b.a = 13;
  return b;
}

int main () {
  struct Foo f;
  f.a = 100;
  f = foo(f);
  printf("value of structure parameter is %d\n", f.a);
}
```

Example (cont)

```
#include <stdio.h>

typedef  struct {
  int a;
} Foo;

void foo (Foo * b) {
  b->a = 13;
}

int main () {
  Foo f;
  foo(&f);
  printf("value of structure parameter is %d\n", f.a);
}
```

Recall that C uses a call-by-value discipline

Use indirection to implicitly propagate effects

Bit fields

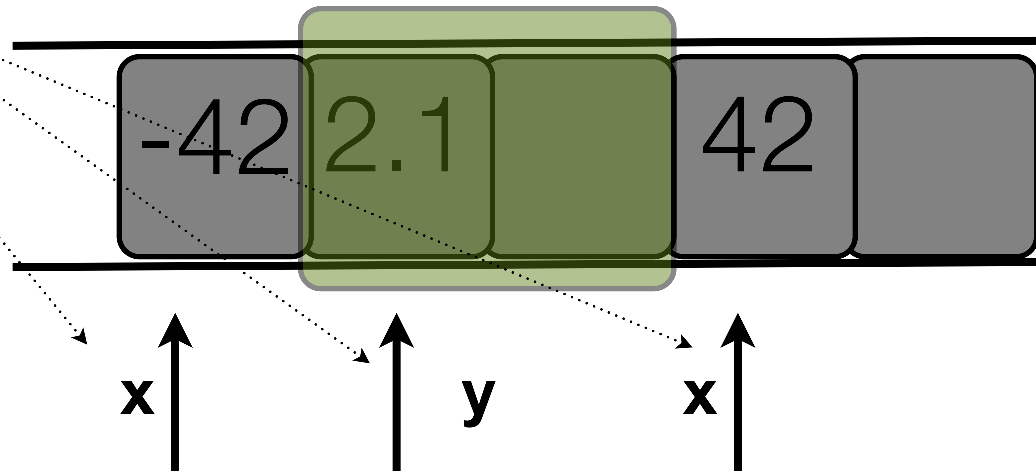
- Structure member variables can be defined in bits
- Everything about bit fields is machine-dependent

```
struct {
  unsigned int is_down : 1;
  unsigned int is_red : 1;
} flags;
flags.is_down = 1;
if (flags.is_red == 0) { ….
}
```

Memory Management

```
int main() {
 int* x; int* start;
 double* y;
 start = (int*) malloc(5);
  x = start;
 *x = -42;      x++; y=(double*) x;
 *y = 2.1;      y++; x=(int*) y;
 *x = 42;
 printf("%i\n", *start);
 printf("%i\n", start[0]);
 printf("%i\n", start[1]);
 printf("%i\n", start[2]);
 printf("%i\n", start[3]);
 printf("%i\n", start[4]);
 printf("%i\n", start[5]);
 printf("%f\n",
    *(double*)(start+1));
```

What does start represent?



Memory Management

```
char *mess = NULL;
mess = (char*) malloc(100);
...
free(mess);
...
*mess = 7;
```

What is the state of the memory pointed by mess after free? What happens if mess is accessed after free?

Unions

- They can hold different type of values at different times
- Definition is similar to a structure BUT
  - STORAGE IS SHARED between the members
  - Only one field type stored at a time
  - Programmer's responsibility to keep track of what it is stored.
- Useful for defining values that range over different types
  - Critically, the memory allocated for these types is *shared*

Unions memory layout

- All members have offset zero from the base
- Size is big enough to hold the widest member
- The alignment is appropriate for all the types in the union

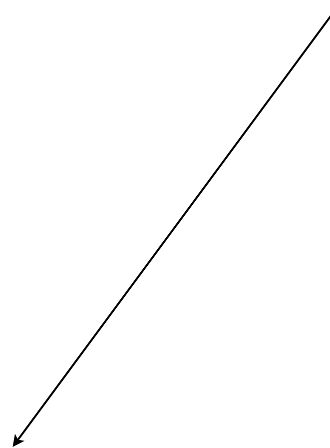
Union operations

- **Same as structures**: The same operations as the ones permitted on structures are permitted on unions:
  - Assignment,
  - Coping as a unit
  - Taking the address
  - Accessing a member
- **Initialize**: can be initialized with a value of the type of its first member.

Example

```
typedef union {
   int units;
   float kgs;
 } amount ;

typedef struct {
   char selling[15];
   float unit_price;
   int unit_type;
   amount how_much;
 } product;
```



The variable *howmuch* can be either an int or a float depending on the kind of product

The compiler allocates the memory necessary to store the largest sized type in the union (here float)

Safety

- C provides no safety guarantees that components within unions are correctly accessed

```
void foo(amount x) {
  printf("... %d\n", x.units);
}

int main () {
  product p[10];
  p[0].selling = "toys";
  p[0].unit_price = 2.0;
  p[0].unit_type = 10;
  p[0].how_much.kgs = 3.0;
  foo(p[0].how_much)
}
```

Example (cont)

```
void checkUnits(int nitems, product* store[]) {
  int i;
  for (i=0; i<nitems; i++) {
    printf("\n%s\n",store[i]->selling);
    switch (store[i]->unit_type) {
    case 1:
      printf("We have %d units for sale\n",store[i]->how_much.units);
      break;
    case 2:
      printf("We have %f kgs for sale\n", store[i]->how_much.kgs);
      break;
    }
  }
}
```

Create an array that points to different products: product * store[n] and supply this array as the argument to checkUnits