# CS240: Programming in C

## Lecture 11: Bit fields, unions, pointers to functions

# Structures recap

- 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

# Structure recap

- Member access
  - Direct: **`s.member`**
  - Indirect: **`s_ptr->member`**
  - Dot operator . has precedence over indirection -> : agenda.contact->name
- Use const to make a structure read-only

# Memory layout for a structure

- Data alignment: when cpu accesses the memory reads more than one byte, usually 4 bytes on a 32-bit platform.
- What is the data structure is not a multiple of 4? Padding.
- Many computer languages and computer language implementations handle data alignment automatically.

# 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) { ….
}
```

# Unions

- They can hold different type of values at different times
- Definition is similar with 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.

# 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.

# Unions: examples

```
union number {
   int    ival;
   float  fval;
   double dval;
};
```

**Union can be member of a structure**

```
struct {
   int type;
   union number {
       int    ival;
       float  fval;
       double dval;
   }value;
} n;
```

# Example:

```
#include <stdio.h>

typedef enum { INT, FLOAT, DOUBLE}
my_type;

struct my_number{
  my_type  type;
  union {
    int    ival;
    float  fval;
    double dval;
  }value;
};

void initialize_my_number(struct
my_number * n, int ival) {
  n->type = INT;
  n->value.ival = ival;
}
```

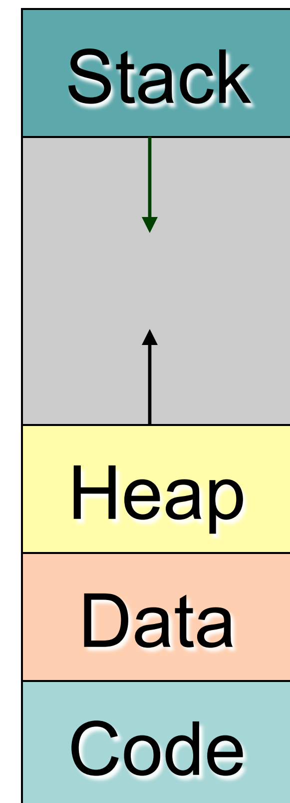
```
void print_my_number(struct
my_number n) {
  switch (n.type){
  case INT:
    printf("%d\n", n.value.ival);
    break;
  case FLOAT:
    printf("%f\n", n.value.fval);
    break;
  case DOUBLE:
    printf("%lf\n", n.value.dval);
    break;
  default:
    printf("Unknown type\n");
    break;
  }
}
```

# Example (cont.)

```
int main() {

  struct my_number i;

  initialize_my_number(&i, 12);
  print_my_number(i);

  return 0;
}
```

# Memory layout for a process

- The operating system creates a process by assigning memory and other resources
- **Stack**: keeps track of the point to which each active subroutine should return control when it finishes executing; stores variables that are local to functions
- **Heap:** dynamic memory for variables that are created with *malloc, calloc, realloc* and disposed of with *free*
- **Data**: initialized variables including global and static variables, un-initialized variables
- **Code: the program instructions to be executed**



# Java vs C Structures: Example

**Java Example:**

```
class Slot {
      int x;
      int y;
      int direction;
methods ...
}
```

**In C:**

```
struct Slot {
  int x;
  int y;
  int direction;
};
```

**What about functions ???**

# Pointers to functions

- Code resides in memory
- **Function Pointers are … pointers which point to the address of a function.**
- Function pointers are variables.
- A function pointer always points to a function with a specific prototype, i.e. same parameters and return-type!

```
int (*Function_ptr)(int, char);
```

# Why do we need function pointers?

- **Functions as arguments to other functions:** sort routine where the main mechanism for sorting is passed as a comparison function by the caller **(354)**
- **Callback Functions:** functions that are invoked when a particular event happens. Useful in networking or graphic applications (**354**)

# Working with pointers to functions

- Declaration and initialization

```
int (*Function_ptr)(int, char*) = NULL;
```

- Assignment

```
int print_error(int n, char* str){
  printf(”Error (%d): %s\n”, n, str);
  return 0;
};
Function_ptr = print_error;
Function_ptr = &print_error;
```

- Calling a function pointer

```
int ret = (*Function_ptr)(4, “Exit\n”);
```

# Working with function pointers

- Pass a function pointer as argument

```
void SomeFunc(int (*ptrFunc)(int, char*)){
 int result = (*ptrFunc)(1, “OK”);
}
SomeFunc(&print_error);
```

- **Return a function pointer**

```
typedef int(*ptrFun)(int, char*);
ptrFun GetPrint(int type){
  if(type == DETAILED)
          return &print_details;
  else
          return &print_summary;
}
```

# Pointers to functions and structures

- Function pointers can be members of structures.

```
struct Slot {
  int x;
  int y;
  int direction;
  void (*print)(Slot *s);
};
```

# Example

```c
#include <stdio.h>

typedef void (*ptrFun)(int, char*);

void print_error(int n, char* str){
  printf("Error (%d): %s\n", n, str);
}


void print_message(int n, char *str) {
  printf("Message(%d): %s\n", n, str);
}


void PassFunctionPointer(ptrFun g, int
    n, char *s) {
   (*g)(n, s);
}
```

```c
int main() {
  ptrFun f = NULL;


  f = &print_error;
  (*f)(10, "Call print_error");

  f = &print_message;
  (*f)(11, "Call print_message");

  PassFunctionPointer(&print_error,
   12, "Passing print_error");
  PassFunctionPointer(&print_message,
   13, "Passing print_message");

  return 0;
}
```

# Readings and exercises for this lecture

K&R Chapter 5.11, 6.8, 6.9

Code all the examples in the lecture.