

Structures

- structures are like sets in Mathematics, can hold different things together

- declaration:

```
struct MonteCarloSpecs {  
    int n_iters;  
    float time_in_secs;  
    float prop_burn_in;  
};
```

- usage:

```
struct MonteCarloSpecs mcs1;
```

- access the “members” of the structure:

```
mcs1.n_iters = 100;  
mcs1.time_in_secs = 5.0;  
mcs1.prop_burn_in = 0.1;
```

Pointers to Structures

- consider:

```
struct MonteCarloSpecs *mcs1_ptr = NULL;  
mcs1_ptr = &mcs1;
```

- SO *mcs1_ptr is same as mcs1
- access the “members” of the pointer to the structure:

```
(*mcs1_ptr).n_iters = 100;  
(*mcs1_ptr).time_in_secs = 5.0;  
(*mcs1_ptr).prop_burn_in = 0.1;
```

- a shorthand for the above operation:

```
mcs1_ptr->n_iters = 100;  
mcs1_ptr->time_in_secs = 5.0;  
mcs1_ptr->prop_burn_in = 0.1;
```

Typedefs

- short hand: you could use `MonteCarloSpecs mcs1;` if you did

```
struct MonteCarloSpecs {
    int n_iters;
    float time_in_secs;
    float prop_burn_in;
};
typedef struct MonteCarloSpecs MonteCarloSpecs;
```

- so `typedef` defines a new type much like the built-in types `int`, `double` etc.
- a compact version for the above is:

```
typedef struct MonteCarloSpecs {
    int n_iters;
    float time_in_secs;
    float prop_burn_in;
} MonteCarloSpecs;
```

Self Referential Structures I

- a canonical example of this concept is one of *singly-linked list*:

```
typedef struct SLList {  
    void *data;  
    struct SLList *next;  
} SLList;
```

- one could also say: this is called forward declaration of a structure and is preferred over the previous one because sometimes, forward declarations goes into a separate file

```
typedef struct SLList SLList;  
struct SLList {  
    void *data;  
    SLList *next;  
};
```

Self Referential Structures II

- there is a more-or-less advanced and partially written program on our website in file `prog13.c` on this topic, it is more or less a complete “library”, please please please read it to check your understanding of C so far
- if you have questions while reading thats a good sign, if it reads Greek to you its time to revise stuff covered so far!

Function Pointers I

- a function is stored at address, we can use a pointer to refer to a function
- first we have to create a type for for the pointer to function:

```
typedef int (*IntIntCompare) (const void *, const void *);
```

- here IntIntCompare is a defined type which is a pointer to a function which returns an int and takes two const void * arguments

Function Pointers II

- using IntIntCompare:

```
int
int_int_cmp (const void *ptr1, const void *ptr2)
{
    int *num1_ptr = (int *) ptr1;
    int *num2_ptr = (int *) ptr2;
    int num1 = *num1_ptr, num2 = *num2_ptr;

    if (num1 < num2)
        return -1;
    else if (num1 > num2)
        return 1;
    else
        return 0;
}
```

- note, `int_int_cmp` conforms to the prototype of a `IntIntCompare` type function

Function Pointers III

- now we can do the following:

```
IntIntCompare cmp;  
cmp = &int_int_cmp;
```

- how to use a function pointer?

```
printf("Compare ival1 and ival2: %d\n",  
      (*cmp>(&ival1, &ival2));  
printf("Compare ival1 and ival2: %d\n",  
      (*cmp>(&ival1, &ival1));  
printf("Compare ival1 and ival2: %d\n",  
      (*cmp>(&ival2, &ival1));
```

do not use `*cmp(&ival1, &ival2)`, its *wrong!*

- the above is same as:

```
printf("Compare ival1 and ival2: %d\n",  
      int_int_cmp(&ival1, &ival2));  
printf("Compare ival1 and ival2: %d\n",  
      int_int_cmp(&ival1, &ival1));  
printf("Compare ival1 and ival2: %d\n",  
      int_int_cmp(&ival2, &ival1));
```

Function Pointers IV

- note the following also works:

```
IntIntCompare cmp;  
cmp = int_int_cmp;  
  
printf("Compare ival1 and ival2: %d\n",  
      cmp(&ival1, &ival2));  
printf("Compare ival1 and ival2: %d\n",  
      cmp(&ival1, &ival1));  
printf("Compare ival1 and ival2: %d\n",  
      cmp(&ival2, &ival1));
```

- you may come across code like this but avoid this practice, some compilers give warnings

Function Pointers V

- note the following:

```
void qsort(void *base, size_t nmem, size_t size,  
          int (*compar)(const void *, const void *));
```

- note IntIntCompare has the same type as

```
int (*compar)(const void *, const void *)
```

- so we could do:

```
int  
main (int argc, char **argv)  
{  
    int ival_arr[] = { 3, 2, 1, 5, 2 };  
    IntIntCompare cmp;  
  
    cmp = &int_int_cmp;  
    qsort(ival_arr, SMALL_LEN, sizeof(int), cmp);  
}
```

Function Pointers VI

- note we could have achieved the same result using:

```
int
main (int argc, char **argv)
{
    int ival_arr[] = { 3, 2, 1, 5, 2 };

    qsort(ival_arr, SMALL_LEN, sizeof(int), int_int_cmp);
}
```

- so why use a function pointer at all?
 - if you don't know what function (among a list of functions) to call until runtime you use a function pointer: this avoids re-compilation of the whole program (pretty esoteric, I know, promise will make it clear when we discuss a large MCMC example later)
 - functions pointers can also replace a lot of `switch` or `if` statements in an elegant fashion (for a reading on this consult the *Introductory Example or How to Replace a Switch-Statement* section on the reading list)

Code Files

prog4.c

prog5.c

prog13.c