More advanced C

Memory allocation, Structs Files, Directories, and I/O

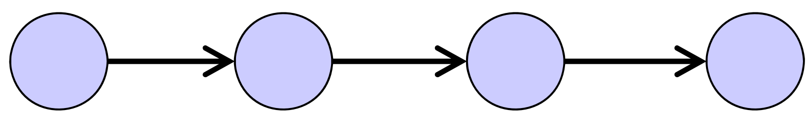
Memory Allocation

Dynamic memory allocation in a nutshell

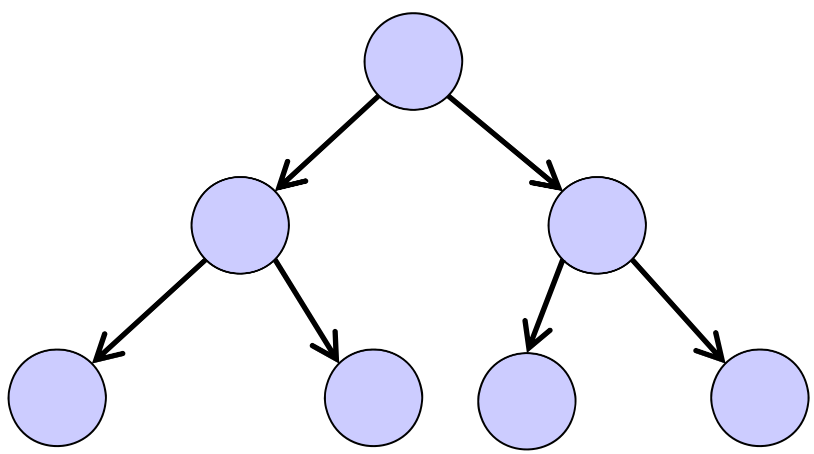
- Dynamic memory allocation = request memory and free it when done
  – New allocations are on the Heap
- It’s up the to user to keep track of that
  – Pointers keep track of addresses to such requested memory chunks
- Why do we need it?
  – To expand and shrink data structures dynamically

Dynamic data structures

- Can you think of any examples?



Lists



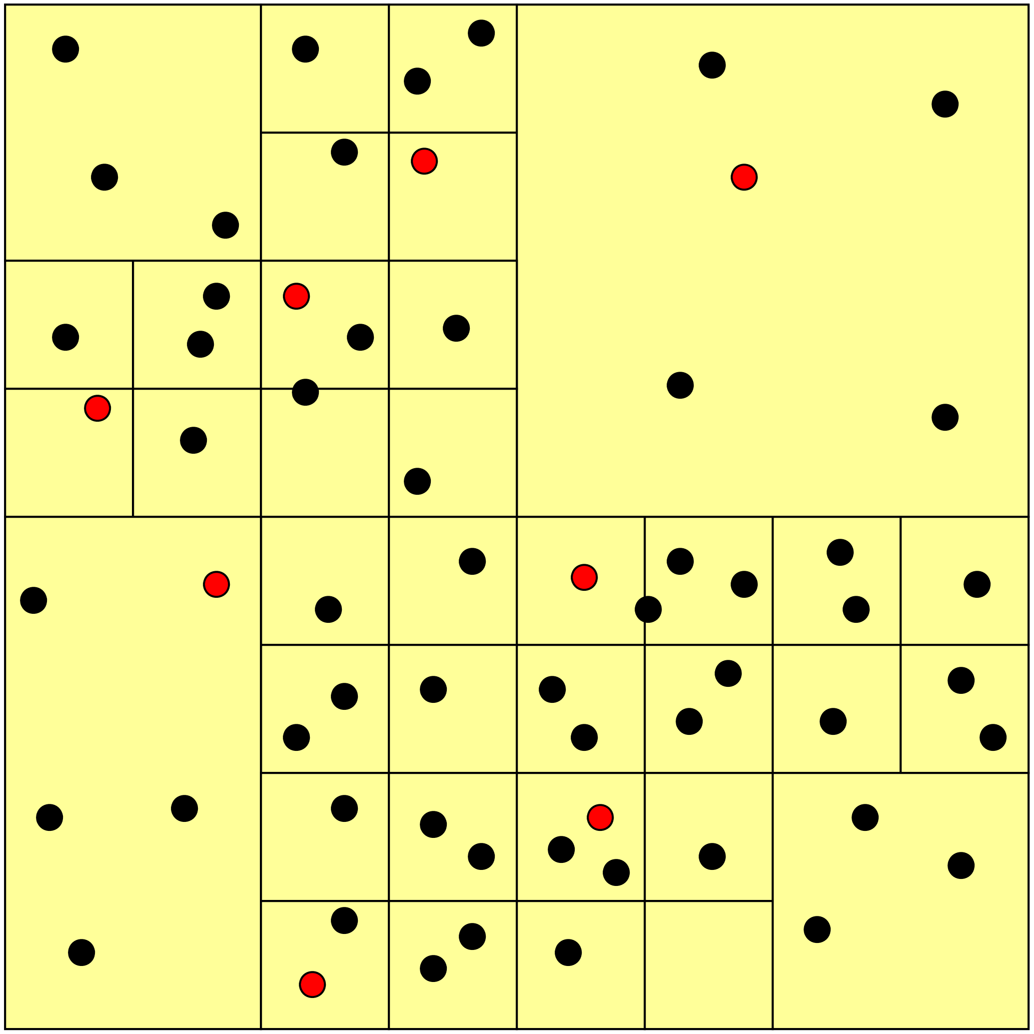
Trees

Example: multiplayer games

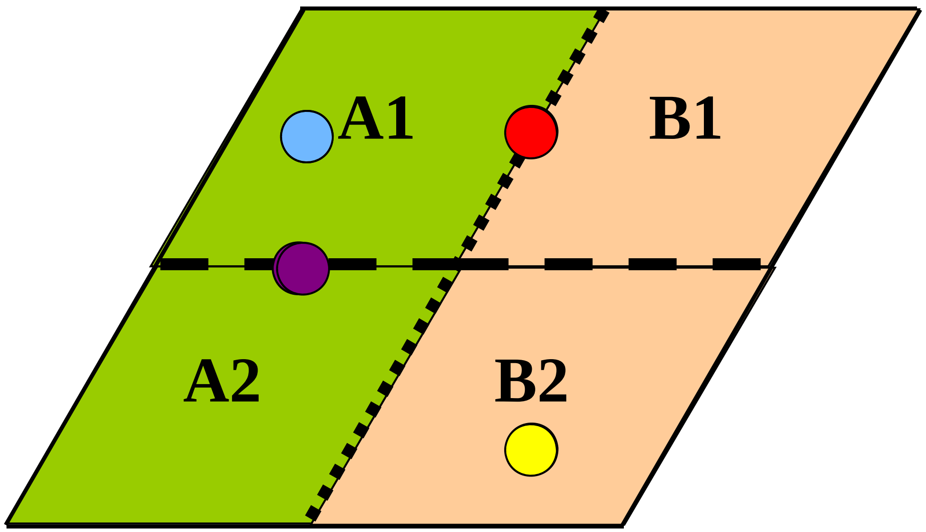
Game map representation

- Goal: fast retrieval of game objects

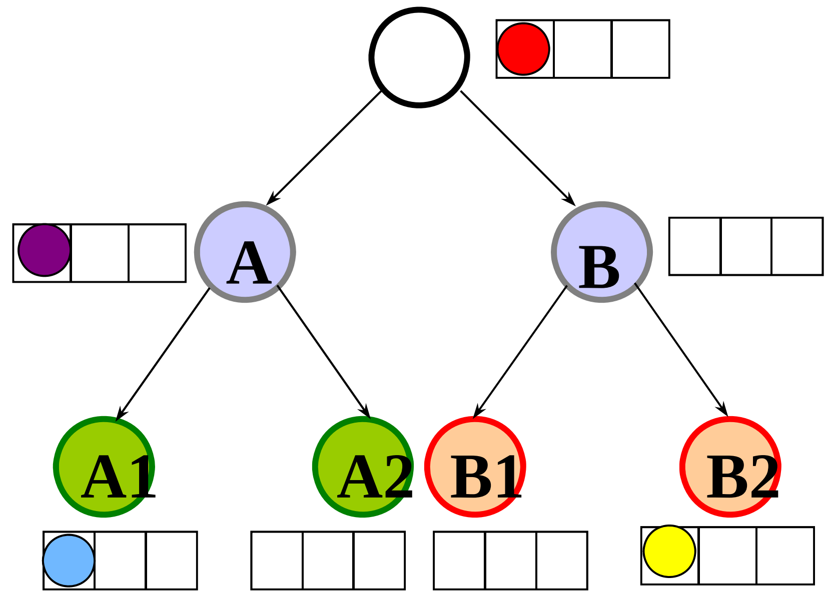
- Game map representation: tree structure



Sample gamemap tree



Game map

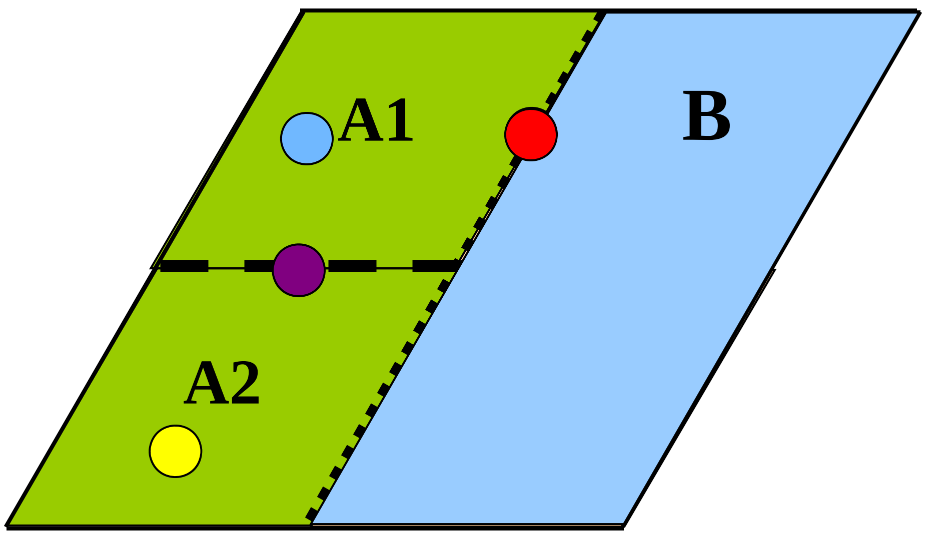


Tree representation
for fast access

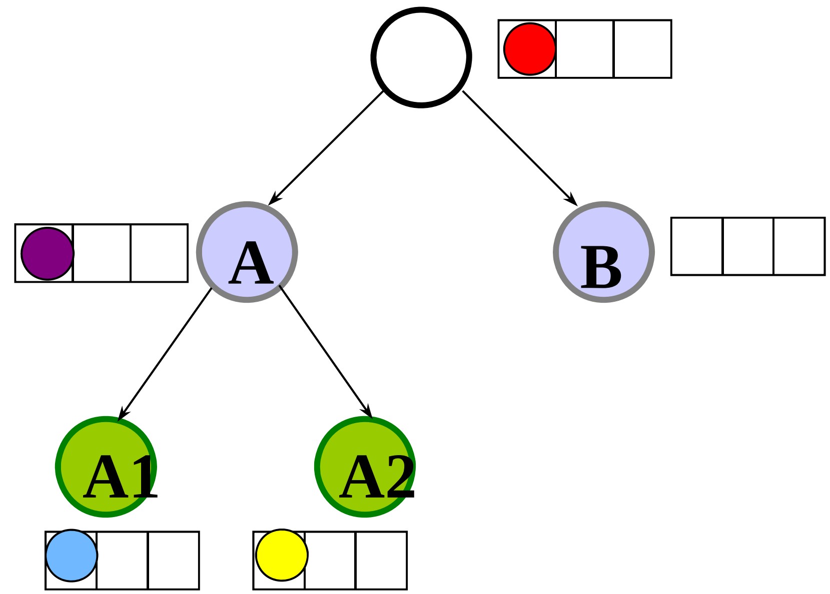
Split the game map into regions
based on number of players

Each player is stored in a list
in the corresponding tree node

Sample gamemap tree



Game map

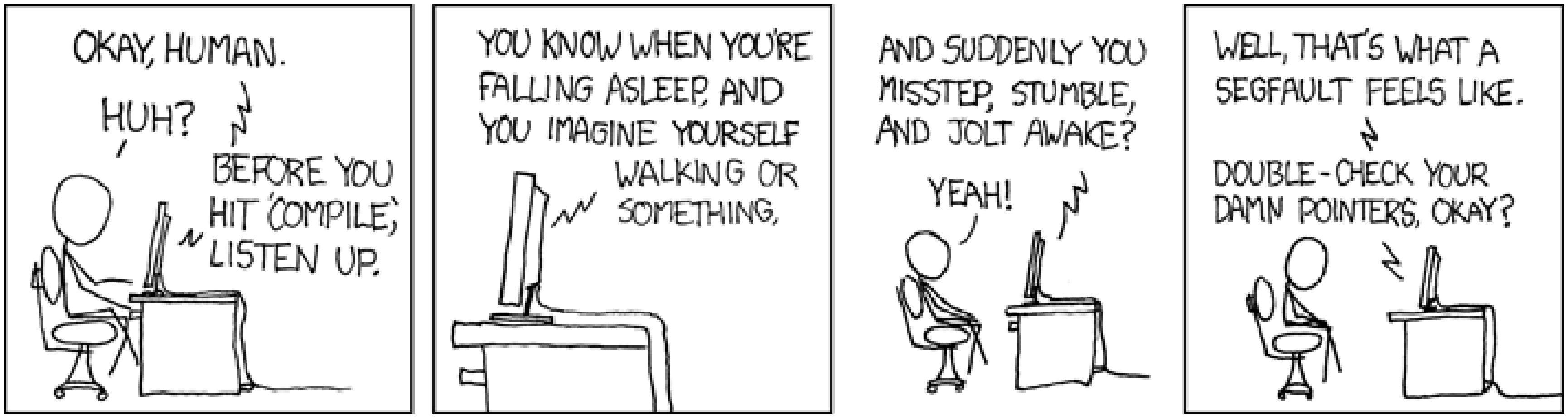


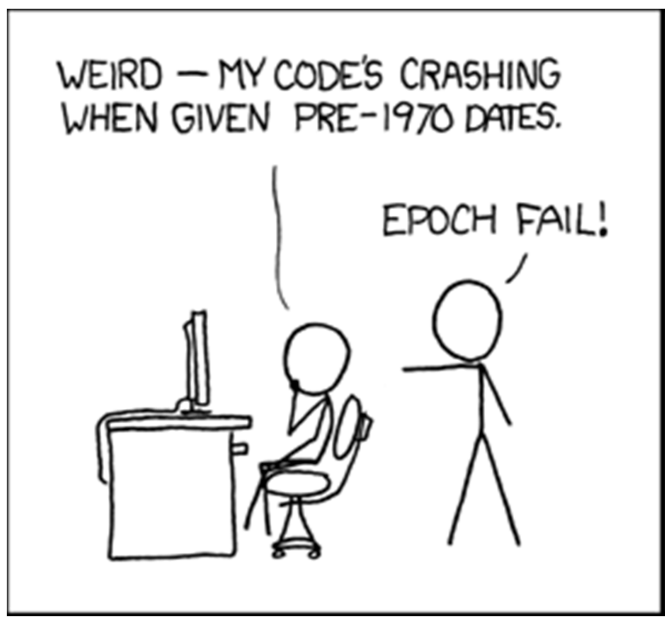
Tree representation
for fast access

Tree grows and shrinks
based on player movements

Dynamic memory allocation - summary -

- We don’t know in advance how much memory our data structure will occupy
- Need to be able expand and shrink data structures during program execution
- Mechanism to request for a dedicated new chunk of memory
  - Malloc() => returns a pointer containing the address of the new chunk!
  - Free() => relinquishes the memory chunk!


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?


WEIRD — MY CODE'S CRASHING WHEN GIVEN PRE-1970 DATES.
EPOCH FAIL!

xkcd.com

Static Allocation

- Recall: static allocation happens at compile time based on variable definitions.

```
int x = 2;
int a[4];
int *b;

int main() {}
```

```
0x804837c   main

0x804957c   init.data
0x8049588   2

0x8049684   uninit. data
0x8049688   ???
0x804968c   ???
0x8049690   ???
0x8049694   ???
0x8049698   ???
```

```
SYMBOL TABLE:
main     0x804837c .text   f9
x        0x8049588 .data   04
b        0x8049688 .bss    04
a        0x804968c .bss    10
```

Dynamic Memory Allocation

- In Java,

```
Set s; // Memory is allocated for pointer s
// Memory is allocated for object
s = new HashSet();
```

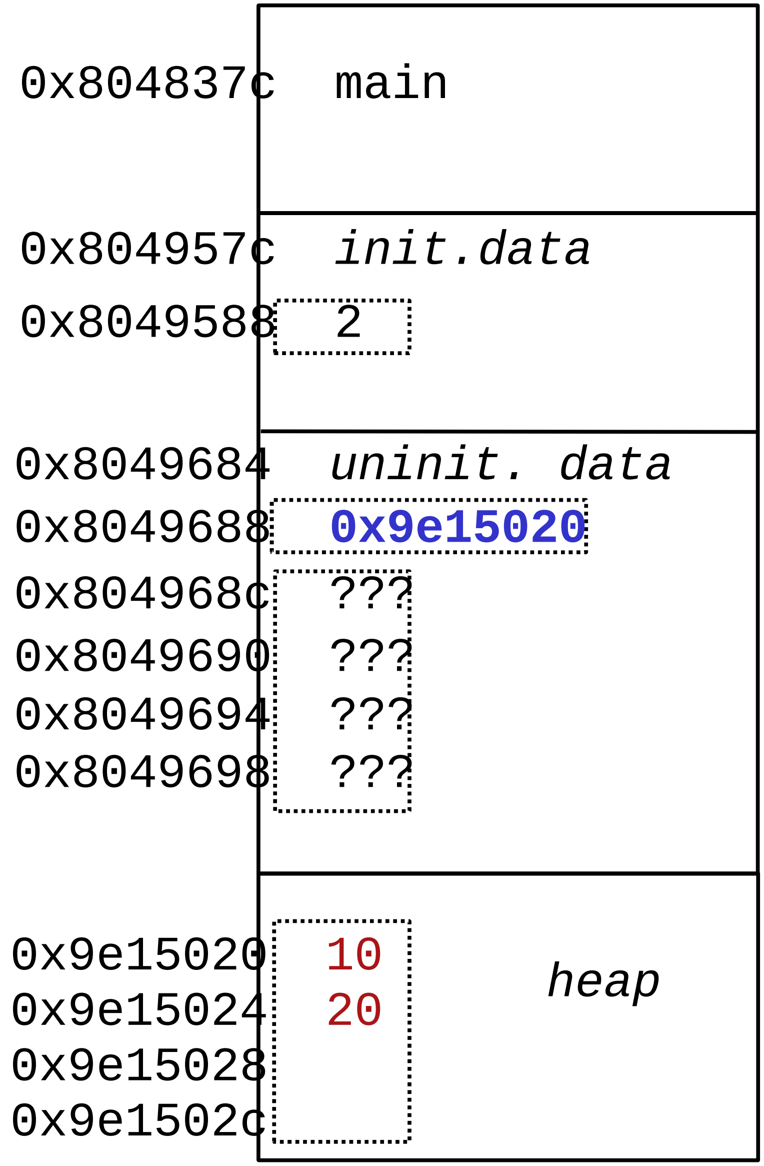
- In C,

```
int *a;  /* Memory is allocated for pointer a */
/* Memory is allocated for a to point to */
a = malloc(10 * sizeof(int));
```

Dynamic Allocation

```
int x = 2;
int a[4];
int *b;

int main() {
  b = malloc(4 *
sizeof(int));
  b[0] = 10;
  b[1] = 20;
}
```



Always check manual pages

$ man malloc

SYNOPSIS

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

DESCRIPTION

malloc() allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared.

free() frees the memory space pointed to by ptr, which must have been returned by a previous call to **malloc(), calloc()** or **realloc()**. Otherwise, or if **free(**ptr**)** has already been called before, undefined behaviour occurs. If ptr is **NULL**, no operation is performed.

malloc

```
void *malloc(size_t size);
```

- Some things you haven't seen yet:

  `void *`

  - A generic pointer type that can point to memory of any type.

  `size_t`

  - A type defined by the standard library as the type returned by `sizeof`.
  - The type is `unsigned long`.

malloc

- Can always assign a void pointer to any more specific type of pointer.

```
int *i = (int*)malloc(sizeof(int)); //
  type cast not mandatory
int *i = malloc(sizeof(int));     //
  implicit conversion
char *c = malloc(NAME_SIZE);
```

- sizeof works on types, and knows type of expressions.

```
double *d = malloc(5*sizeof(*d));
```

- Be careful to allocate the correct number of bytes.
- E.g., `int *i = malloc(1); /*wrong*/`
  - allocates 1 byte, not 1 int.

NULL pointers

- A function that returns a block of memory might fail to do so, in which case it returns a NULL pointer.
- NULL is a pre-processor variable defined in iolib.h (included from stdio.h) and other places
  - it is usually defined to be 0 (no program allocates anything at address 0x0)

De-allocating memory

```
int *a = malloc(10 * sizeof(int));
int b[10];
...
a = b;
```

- What is wrong with the last line? It compiles and runs fine.
- We have lost the pointer to the memory region allocated in the first line, so that space is now tied up until the program terminates.

⇨ Memory leak!

free()

- Before removing the last pointer to a memory region, you must explicitly deallocate it.
  - No garbage collection in C!

```
int *a = malloc(10 * sizeof(int));
int b[10];
...
free(a);
a = b;
/*No memory leak */
```

Is “a” NULL after the free statement?

→ No, `free` cannot change the value of a parameter

Dangling pointers

```
int *a = malloc(10 * sizeof(int));
...
free(a);
printf("%d\n", a[0]); /* Error */
```

- Dereferencing a pointer after the memory it refers to has been freed is called a “dangling pointer”.
- Behaviour is undefined. Might:
  - appear to work
  - bogus data
  - program crash

Dangling pointers

```
int *a = malloc(10 * sizeof(int));
...
free(a);
printf("%d\n", a[0]); /* Error */
```

- Can you re-use pointer “a” after free() though?
  - Yes, recall that memory to store the pointer is allocated by default; However, allocating memory for the location where the pointer is pointing to, that’s up to the programmer!

```
free(a); /* memory is released => a points to an invalid
                                   location */
int  i = 5;
a = &i;   /* valid, now points to the address of variable i */
printf("%d\n", *a);

a = malloc(15 * sizeof(int)); /* valid, we request new memory */
printf("%d %d\n", a[0], a[5]);
```

Arrays of pointers

- Most obvious use is to get an array of strings
  - Consider a word = an array of chars
  - A sentence = an array of words

```
#define LEN 4                   // define macro, constant
char **strs = malloc(3*sizeof(char *));  // define an array of 3
   words

for(i = 0; i < 3; i++) {
    strs[i] = malloc(LEN);   // each word itself has to be
      allocated
}
strs[0] = strncpy(strs[0], "209", LEN); // copy a word in strs[0]
strs[1] = strncpy(strs[1], "369", LEN); // copy a word in strs[1]
```

- What else can we represent?
  - A matrix : `int **a;`
    (static allocation: `int a[10][10];`)
  - An array of matrices: `int ***a;`
    (static allocation: `int a[10][10][5];`)

Copying or moving memory

```
$ man memcpy
NAME
       memcpy - copy memory area

SYNOPSIS
  #include <string.h>

  void *memcpy(void *dest, const void *src, size_t n);

DESCRIPTION
       The  memcpy()  function copies n bytes from
memory area src to memory area dest.  The memory areas
must not overlap.  Use memmove(3) if the memory areas do
overlap.
```

Copying or moving memory

- What's the difference between these 2:

```
int *p, *q, i;
p = malloc(10*sizeof(int));
q = malloc(20*sizeof(int));
for (i = 0; i < 10; i++)
   p[i] = i;
```

1) q = p; // what also happens here?

OR

2) memcpy(q, p, 10*sizeof(int));

Tips

- Use a debugger and start to figure out what valid addresses look like.
- Check return values from library functions.
- Watch out for common errors:
  - forgetting to allocate memory when a pointer is declared
  - dereferencing a pointer after it's been free'd
  - losing track of a memory block without free'ing (memory leak)!
- Remember: practice, practice, practice!

Structures

Structs

- A collection of related data items

```
struct Point {
    float x;
    float y;
};
/* Semicolon is important! */
```

Structure members, or fields

```
struct Point p;
p.x = 1.53;
p.y = 8.27;

struct Point q;
q.x = p.x;
q.y = p.y;
```

Structs

```
struct student {
    char *name;
    int age;
};

struct student s1; /*allocates space for the record */
s1.name = malloc(4*sizeof(char));/* don’t forget this! */
s1.name[0] = ‘J’;
s1.name[1] = ‘O’;
s1.name[2] = ‘E’;
```

- Pointers use ‘->’ instead of ‘.’ to refer to struct members!

```
struct student *s2;
s2 = malloc(sizeof(struct student)); /* allocate pointer */
s2->name = malloc(4*sizeof(char));/* again, don’t forget this!*/
```

- To simplify syntax – use typedef:

```
typedef struct student Student;
Student s3, *s4;
s4 = malloc(sizeof(Student));
```

Structs as arguments

```
/* Remember: pass-by-value */
void print_student(struct student s) {
   printf("Name = %s\n", s.name);
   printf(“Age  = %d\n", s.age);
}

int main() {
  struct student s1, *s2;
  ...
  print_student(s1);
  print_student(*s2);
}
```

Passing pointer or struct?

```
/* Incorrect */
void incr_age(struct student *r) {
   r.age++;
}

/* Correct */
void incr_age(struct student *r) {
   r->age++;
}
```

Concrete Example

```
int stat(const char *file_name, struct stat *buf);
```

```
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device)*/
    off_t         st_size;     /* total size, in bytes */
    blksize_t     st_blksize;  /* blocksize for filesystem I/O */
    blkcnt_t      st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

stat

- By calling the `stat` function on a filename you want to fill in the fields of the `struct stat`.
- You must pass in a pointer, and there must be space allocated!!!

```
struct stat sbuf;
if(stat("myfile", &sbuf) == -1) {
   perror("stat");
   exit(1);
}
printf("Owner = %d", sbuf.st_uid);
```

Common error

```
struct stat *sbuf;
if(stat("myfile", sbuf) == -1)
{
  perror("stat");
  exit(1);
}
```

NEXT UP

- Files, I/O
- Strings