Today's Goals

- More on Pointers
  - The dangling pointer
  - Pointer arithmetic
  - Pointers to pointers
  - **const** and pointers
- Function pointers
- Memory blocks

Section 1

The Dangling Reference Problem

- An unassigned pointer (i.e. not pointing to any memory) is known as a dangling pointer.
- Dereferencing any such pointers will surely result in a segmentation fault or bus error.
  - Always assign your pointers to **malloc**ed memory or by **&** of a variable.
  - Remember that stack variables (i.e. locals in functions) are deallocated as soon as a function returns. Therefore, avoid assigning pointers to addresses of such variables.

Section 2

Pointer Arithmetic

- These operations are only well-defined if the address involved are within a single memory block (array or **malloc**ed)
  - The sum of a pointer and an integer
  - The difference of a pointer and an integer
  - Pointer comparison
  - The difference between two pointers

Pointers to Pointers

- A variable can be modified by a function if and only if it is passed by reference/pointer.
- If the variable to be modified is a pointer itself, one must pass a pointer to pointer, i.e. one must always add an extra level of referencing.

```
int make_node(Node **new) {
  *new=(Node *)malloc(sizeof(Node));
  if (*new != NULL)
    return 1;
  else
    return 0;
}
```


Section 3

Use **const** to Protect Pointers

- We already know that keyword **const** prevents the value of a variable from being changed.
  - `const int x;`
  - `void f(const int *p);`
  - Prevents ***p** from being changed
  - `void f(int * const p);`
  - Prevents the pointer **p** itself from being changed
  - `void f(const int * const p);`

Section 4

Function Pointers

- Function pointers point to memory addresses where functions are stored.
  - `int (*fp) (void);`
  - A function pointer determines the prototype of a function, but not its implementation.
  - Any function of the identical prototype can be assign to the function pointer.
  - A function without its argument lists becomes its own pointer
  - Function pointers do not need **&** or *****

Function Pointer: Example

```
#include <stdio.h>

int main() {
  int i = 1;
      int (*fp) (const char *, ...) = printf;
      fp("i == %d\n", i);
  (*fp)("i == %d\n", i);
      return 0;
}
```

- Notice no need for **&printf** or **(*fp)**
- But I like to stick with **(*fp)**

Overriding Functions

- Also known as late-binding, this is emulated in C with function pointers.
- Together with generic pointers (**void ***), one can have typeless parameters and functions.

```
void fd (void *base, size_t n, size_t size){
  double *p = base;
  for (p = base; p < (double*) (base+(n*size)); p++) ;
}
int main() {
  double a[5] = {0, 1, 2, 3, 4};
  if (type == DOUBLE) {
    void (*f) (void *, size_t, size_t) = fd;
    (*f)(a, 5, sizeof(double));
  }
}
```


Printing of Generic Arrays

```
typedef struct {
  double x;
  double y;
} Point;
int main() {
  double a[5] = {0, 1, 2, 3, 4};
  int b[5] = {5, 6, 7, 8, 9};
  Point ps[2] = {{0.5, 0.5}, {1.5, 2.5}};

  gp(a, 5, sizeof(double));
  gp(b, 5, sizeof(int));
  gp(ps, 2, sizeof(Point));
}
```


Printing of Generic Arrays

```
void gp (void *b, size_t n, size_t size){
  char *p;
  for (p=b; p <(char*)(b+(n*size)); p+=size){
    switch (size) {
    case sizeof(double):
      printf("%.2f ", *(double*)p);
      break;
    case sizeof(int):
      printf("%d ", *(int*)p);
      break;
    case sizeof(Point):
      printf("x = %.2f ", ((Point *)p)->x);
      printf("y = %.2f ", ((Point *)p)->y);
      break;
    }}
  printf("\n"); }
```


qsort

- **qsort** and its comparison function:
  - sorts a generic array of any type
  - **stdlib.h**
  - `void qsort(void *base, size_t ne, size_t s, int (*compar) (const void *, const void *));`
  - **base** points to the start (array) element
  - **ne** is the number of elements to be sorted
  - **s** is the size of the elements (in the array)
  - **compar** is the generic comparison function supplied by user

qsort Example

```
int vs[] = {40, 10, 100, 90, 20, 25};
int comp ( const void *a,
           const void *b){
  return ( *(int*)a - *(int*)b );
}

int main () {
  qsort (vs, 6, sizeof(int), comp);
}
```

compar

```
int comp_nodes (const void *a, const void *b) {
  struct Node *n1 = a;  struct Node *n2 = b;

  if ( (n1->num < n2->num )  return -1;
  else if (n1->num > n2->num )  return 1;
  else return 0;
  /* or
  return ((struct Node *)n1)->num - ((struct Node
  *)n2)->num; */
}

qsort(nodes, 10, sizeof(struct Node), comp_nodes);
```


Simulating Methods

- Function pointers used together with **struct**
  - ```
    typedef struct _Node {
        int num;
            struct Node *prev;
        struct Node *next;
       struct Node * (*make_node) (int);
    } Node;
    ```


Other Uses of Function Pointers

- Function pointers can be used like any pointers.
  - Array of function pointers

```
void (*file_cmd[])(void) = { new_cmd,
                             open_cmd,
                             close_cmd,
                             save_cmd,
                             print_cmd,
                             exit_cmd
                               };
```


Other Uses of Function Pointers

- Function returning pointer to function
- Function that takes a **double** and returns a pointer to a function which takes a **char *** and returns an **int**:
  - Without **typedef**:

```
int (*f(double)) (char *);
```

  - With **typedef**:

```
typedef int (*Pfunc) (char *);
Pfunc f(double);
```


Section 5

Working with Memory Blocks

- Generic pointers can not be dereferenced.
- **string.h** functions to work with typeless memory blocks pointed to by **void ***.
  - `void *memcpy(void *des,void *src,size_t l);`
  - `void *memmove(void *des,const void *src,size_t l);`
  - `int memcmp(const void *p,cost void *q,size_t l);`
  - `void *memchr(const void *p, int val, size_t l);`
- These are generic versions of the corresponding string functions (**strcpy**, **strcmp** and **strchr**).
- **memmove** is used instead of **memcpy** if the two blocks overlap.

Section 6

File I/O on Memory Blocks:Binary Files

- Binary files do not have a line-oriented structure.
- They consist of blocks of objects, and the format of these objects are application as well as machine dependent.
- When opening a binary file, always append a **'b'** to the second argument, i.e.
  - **fopen("test.exe", "wb");**
  - **fopen("test.exe", "rb");**
```

Current File Position

- **`long ftell(FILE *fp);`**
- Returns the current file position, or **`-1L`**.
- **`fseek`** is used to set the file position:

```
fseek(fp, -sizeof(double), SEEK_CUR);
```

- **`ftell`**'s return value is often used by **`fseek`**:

```
long pos;
pos = ftell(fp);
/* some code */
fseek(fp, pos, SEEK_SET);
```


Read/Write a Binary File

- **`size_t fread(void *buf, size_t s, size_t ne, FILE *in);`**
- Reads up to **`ne`** elements, each of size **`s`** in bytes and stores them in **`buf`**
- **`size_t fwrite(const void *buf, size_t s, sizt_t ne, FILE *out);`**
- Similarly, writes from **`buf`** up to **`ne`** blocks, each of size **`s`**
- Number of elements (not bytes!) successfully read/written is returned

Examples

- Converting a text file of **`int`**s to binary:

```
while (fscanf(in, "%d", &i) == 1)
  if (fwrite(&i,sizeof(int),1,out)!=1){
    fprintf(stderr, "Error writing file\n");
    break;}
```

- **`fwrite`** is convenient for storing data in a file temporarily (or another program), which can later be read back with **`fread`**.

```
long pos; pos = ftell(fp);
fpos_t pos; fgetpos(fp, &pos);
fwrite(&n, sizeof(struct Node), 1, fp);
fseek(fp, pos, SEEK_SET); fsetpos(fp, &pos);
fread(&n,sizeof(n), 1, fp);
```


Summary

- Be careful with pointers!
- Function pointers can be really useful, especially when writing larger applications.
- **`fread`** and **`fwrite`** are designed to copy large chunks of memory from/to file.
- Although primarily used in binary file I/O, with a little care in address/size calculation, they can be used with text files just as well.