COMP20200 Unix Programming

Lecture 6

CS, University College Dublin, Ireland

Pointers

- Pointer is a memory address of something:
  - scalar variables, structures, arrays, functions
- `&variable` to get the address of `variable`
- Dereference with "*" to access or change what pointer points to.

```
int var = 20;
int* ip;
ip = &var;
*ip = *ip + 10; // var now 30
printf("Address of var variable: %p\n", &var  );
printf("Address stored in ip variable: %p\n", ip );
printf("Value of *ip variable: %d\n", *ip );
```

Why pointers?

- C abstraction of address in Unix machine
- Easy to express Unix memory access and management
- Easy to compile C programs into machine code

Use pointers:

- C uses call by value, not call by reference.
- If a function needs to return more than 1 variable, need to use either struct or pointers

```c
int main(){
  int i = 4;
  double f = 3.14;
  func(i, f);
  printf("%d, %lf\n", i, f);
  return 0;
}
void func(int i, double f){
  i++;
  f = f * 2;
}
```

prints: 4, 3.14

```c
int main(){
  int i = 4;
  double f = 3.14;
  func(&i, &f);
  printf("%d, %lf\n", i, f);
  return 0;
}
void func(int* i, double* f){
  (*i)++;
  *f = *f * 2;
}
```

prints: 5, 6.28

Use pointers:

Pointers are needed when we allocate memory in a function.

```
int main(){
  int* array;
  array = func(10);
  int i;
  for (i = 0; i < 10; i++)
    printf("%d ", array[i]);
  printf("\n");
  free(array); array = NULL;
  return 0;
}
int* func(int size){
  int* a = malloc(sizeof(int) * size);
  int i;
  for (i = 0; i < size; i++)
    a[i] = i * 2;
  return a;
}
```

output: 0 2 4 6 8

Use pointers:

- If we want to pass an array (or string) to a function.

```
void func(char* s){
   printf("%s", s);
}

int main(){
   char* str = "hello\n";

   func(str);
   return 0;
}
```

Arrays

- Name of an array is pointer to its first element
- Static allocation in stack, size known at compile time
  `int arr[10];`
- Variable-length arrays (C99) in stack, size known at runtime time
  `int arr[n];`
- Dynamic allocation in heap, size known at runtime time
  `int* arr = malloc(sizeof(int) * n);`
- Once allocated, both of above can be used the same
  `arr[3] = 15;`
- In general, by definition `e1[e2]` = `*((e1)+(e2))`

2D arrays

- No 2D arrays as such - arrays of arrays
- `int array[num_rows][num_col];`
  - Is a pointer to array (`int(*)[num_col]`)
  - 2nd row, 3rd column: `array[2][3]` or `*(*(a+2)+3)`
- "Multi-dimensional" arrays on heap
  - `int* array = malloc(sizeof(int) * rows * columns);`
    - 2nd row, 3rd column: `array[2 * columns + 3]`
  - `(int (* array)[columns]) = malloc(sizeof(int) * rows * columns);`
    - 2nd row, 3rd column: `array[2][3]`

Pointer Arithmetic and Arrays

- Pointers can point to any cell in array

```
int *ip;
int a[10];
ip = &a[3];
```

- Adding one to a pointer addresses next cell.

```
*(ip + 1) = 6;
```

  - Note, here we are adding `sizeof(<type>‘)` to address of `ip`, not just `1`.
    Try: `printf("%p %p \n", array, array+1);`
- `*(a + 5)` is the same as `a[5]` or `5[a]`
- `*ip++` References what ip points to and then points `ip` to next cell
- `*++ip` Moves `ip` to point to next cell and then reverences cell content
- `(*ip)++` Will increment cell content `ip` points to.

Function pointers

- Can be used to replace switch/if-statement
- Simplify complex logic.
- Given 2 functions:

```
double plus(double a, double b);
double minus(double a, double b);
```

Can have a function pointer `*fpt` :

```
double (*fpt)(double, double);
fpt = &plus;
```

Function pointers simple example

```
double plus(double a, double b) { return a+b; }
double minus(double a, double b) { return a-b; }
/* Using switch */
void calc_print(double a, double b, char op){
  double result;
  switch(op){
    case '+' : result = plus(a, b); break;
    case '-' : result = minus(a, b); break;
  }
  printf("%lf\n", result);
}
/* Using function pointer */
void calc_print_fp(double a, double b,
                   double (*fpt)(double, double)){
  printf("%lf\n", fpt(a, b));
}
int main(){
  calc_print(4, 5, '+');
  calc_print_fp(4, 5, &minus);
  return 0; }
```

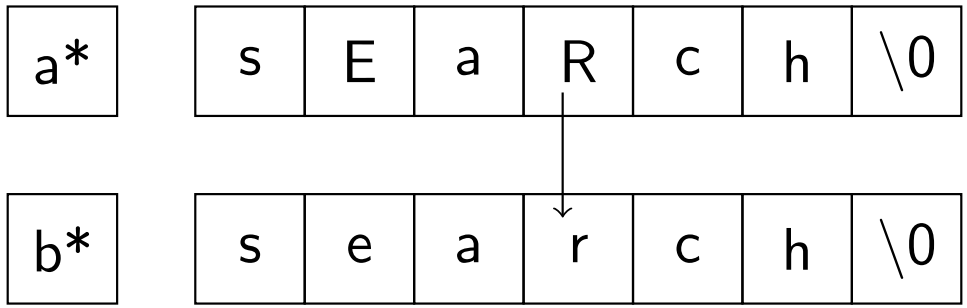
Function pointers: qsort

- qsort can sort numbers, strings, structs, etc.
- qsort developer doesn't know how user wants sort objects.
  `void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));`
- User writes own comparison function and passes function pointer.

```
int int_cmp(const void *a, const void *b) {
   const int *ia = (const int *)a; // casting pointer
   const int *ib = (const int *)b;
   return *ia - *ib;
  /* returns negative if b > a and positive if a > b */
}
int main(){
   int length = 100;
   int* numbers = (int*)malloc(sizeof(int) * length);
   qsort(numbers, length, sizeof(int), int_cmp);
   free(numbers);
}
```

Working with strings: tolower example

Want:



- First need memory allocated.
- How much?

```
char b[100];
```

- Enough? or too much?

Find lenght of string

```
#include <string.h>
int len = strlen(a);
```

Or write your own:

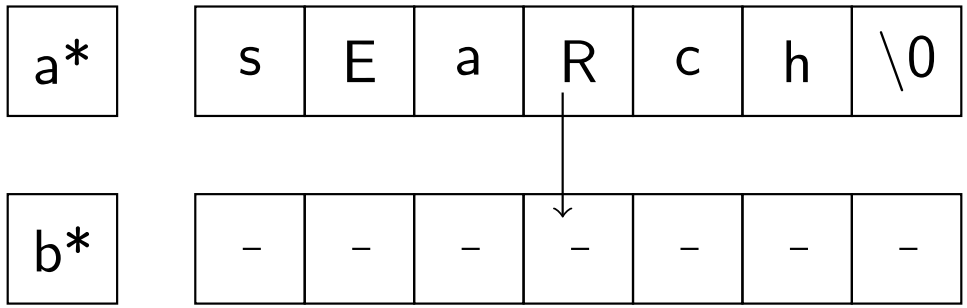
```
int mystrlen(char s[]){
  int i = 0;
  while (s[i] != '\0')
    ++i;
  return i;
}
```

Then allocate memory:

```
char* b = malloc((strlen(a) + 1) * sizeof(char));
```

Note: '+1' for the null character.

Lower each character at a time




```
char c;
int i = 0;
while((c = a[i]) != '\0'){
   b[i] = tolower(c);
   i++;
}
b[i] = '\0';
```