CS 33

Introduction to C
Part 2
Function Definitions

```c
int fact(int i) {
    int k;
    int res;
    for(res=1, k=1; k<=i; k++)
        res = res * k;
    return res;
}

int main() {
    printf("%f\n", fact(5));
    return 0;
}
```

- **main**
  - is just another function
  - starts the program

All functions
- have a return type
Compiling It

$ gcc -o fact fact.c
$ ./fact
120
Function Definitions

```c
int main() {
    printf("%f\n", fact(5));
    return 0;
}

float fact(int i) {
    int k;
    float res;
    for(res=1, k=1; k<=i; k++)
        res = res * k;
    return res;
}
```
Function Definitions

$ gcc -o fact fact.c
main.c:27: warning: type mismatch with previous implicit declaration
main.c:23: warning: previous implicit declaration of 'fact'
main.c:27: warning: 'fact' was previously implicitly declared to return 'int'

$ ./fact
1079902208
Function Declarations

```c
float fact(int i);

int main() {
    printf("%f\n", fact(5));
    return 0;
}

float fact(int i) {
    int k;
    float res;
    for(res=0,k=1; k<=i; k++)
        res = res * k;
    return res;
}
```

Declares the function

```bash
$ ./fact
120.000000
```
Methods

- C has functions
- Java has methods
  - methods implicitly refer to objects
  - C doesn’t have objects
- Don’t use the “M” word
  - TAs will laugh at you
Function Declarations

**fact.h**

```c
float fact(int i);
```

**fact.c**

```c
#include "fact.h"
int main() {
    printf("%f\n", fact(5));
    return 0;
}
float fact(int i) {
    int k; float res;
    for(res=1,k=1; k<=i; k++)
        res = res * k;
    return res;
}
```
The Preprocessor

`#include`

- calls the preprocessor to include a file

What do you include?

- your own *header* file:
  
  `#include "fact.h"`
  
  - look in the current directory

- standard *header* file:
  
  `#include <assert.h>`
  `#include <stdio.h>`
  
  - look in a standard place

Contains declaration of `printf` (and other things)
#define

```c
#define SIZE 100
int main() {
    int i;
    int a[SIZE];
}
```

- The `#define` directive defines a substitution
- This substitution is applied to the program by the preprocessor
#define

```
#define forever for(;;)
int main() {
    int i;
    forever {
        printf("hello world\n");
    }
}
```
assert

```c
#include <assert.h>
float fact(int i) {
    int k; float res;
    assert(i >= 0);
    for(res=1, k=1; k<=i; k++)
        res = res * k;
    return res;
}
int main() {
    printf("%f\n", fact(-1));
}
```

$ ./fact
main.c:4: failed assertion 'i >= 0'
Abort
Parameter passing

Passing arrays to a function

```c
int average(int a[], int s) {
    int i; int sum;
    for(i=0,sum=0; i<s; i++)
        sum += a[i];
    return sum/s;
}
int main() {
    int a[100];
    ...
    printf("%d\n",average(a,100));
}
```

- Note that I need to pass the size of the array
- This array has no idea how big it is
Swapping

Write a function to swap two entries of an array

```c
void swap(int a[], int i, int j) {
    int tmp;
    tmp = a[j];
    a[j] = a[i];
    a[i] = tmp;
}
```
Selection Sort

```c
void selectsort(int array[], int length)
{
    int i, j, min;
    for (i = 0; i < length; ++i)
    {
        /* find the index of the smallest item from i onward */
        min = i;
        for (j = i; j < length; ++j)
            if (array[j] < array[min])
                min = j;
        /* swap the smallest item with the i-th item */
        swap(array, i, min);
    }
    /* at the end of each iteration, the first i slots have the i smallest items */
}
```
Swapping

Write a function to swap two ints

```c
void swap(int i, int j) {
}

int main() {
    int a = 4;
    int b = 8;
    swap(a, b);
    printf("a:%d b:%d", a, b);
}
```

Parameters are passed by value
Swapping

Write a function to swap two ints

```c
void swap(int i, int j) {
    int tmp;
    tmp = j; j = i; i = tmp;
}

int main() {
    int a = 4;
    int b = 8;
    swap(a, b);
    printf("a:%d  b:%d", a, b);
}
```

```
$ ./a.out
a:4  b:8
```

Darn!
Why “pass by value”?

• Fortran, for example, passes parameters “by reference”
• Early implementations had the following problem (shown with C syntax):

```c
int main() {
    function(2);
    printf("%d\n", 2);
}
void function(int x) {
    x = 3;
}
```

$ ./a.out

3
Memory addresses

- In C
  - you can get the memory address of any variable
  - just use the magical operator &

```c
int main() {
    int a = 4;
    printf("%u\n", &a);
}
```

```
$ ./a.out
3221224352
```

```
a:3221224352
```
C Pointers

• What is a C pointer?
  – a variable that holds an address

• Pointers in C are “typed” (remember the promises)
  – pointer to an int
  – pointer to a char
  – pointer to a float
  – pointer to <whatever you can define>

• C has a syntax to declare pointer types
  – things start to get complicated …
C Pointers

```c
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%u\n", p);
}
```

$p$ is a pointer to an int

if you follow $p$, you find an int

$p$ takes the address of $a$

```
$ ./a.out
3221224352
```
C Pointers

```c
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%u\n", p);
}
```

```
$ ./a.out
3221224352
```

Can you guess what &p is?
C Pointers

• Pointers are typed
  – the type of the objects they point to is known
  – there is one exception (see later)

• Pointers are first-class citizens
  – they can be passed to functions
  – they can be stored in arrays and other data structures
  – they can be returned by functions
Swapping

What does this do?

```c
void swap(int *i, int *j) {
    int *tmp;
    tmp = j; j = i; i = tmp;
}

int main() {
    int a = 4;
    int b = 8;
    swap(&a, &b);
    printf("a:%d  b:%d\n", a, b);
}
```

Damn!

```
$ ./a.out
a:4  b:8
```
C Pointers

• Dereferencing pointers
  – accessing/modifying the value pointed to by a pointer

```c
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%d\n", *p);
    *p = *p + 1;
    printf("%d\n", *p);
}
```

$ ./a.out
4
5
Dereferencing C Pointers

```c
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%d\n", *p);
    *p = *p + 1;
    *p += 3;
    printf("%d\n", a);
}
```

```
$ ./a.out
4
8
```
Swapping

```c
void swap(int *i, int *j) {
    int tmp;
    tmp = *j; *j = *i; *i = tmp;
}

int main() {
    int a = 4;
    int b = 8;
    swap(&a, &b);
    printf("a:%d  b:%d\n", a, b);
}
```

$ ./a.out
a:8  b:4
Hooray!
int doubleit(int *p) {
    *p = 2*(p);
    return *p;
}

int main() {
    int a = 3;
    int b;
    b = doubleit(&a);
    printf("%d\n", a*b);
}
Pointers and Arrays

```c
int main() {
    int a[7];
    int *p;
    p = &a[0];
    *p = 33;
}
```
Pointer Arithmetic

Pointers can be incremented/decremented
– what this does to the pointer depends on its type

```c
int main() {
    int a[7];
    int *p;
    p = &a[0];
    *p = 33;
    *(p+1) = 167;
}
```
Pointer Arithmetic

Pointers can be incremented/decremented
– what this does to the pointer depends on its type

```c
int main() {
    int a[7];
    int *p;
    p = &a[0];
}
```

Now \(p\) and \(a\) have the same value
Pointer Arithmetic

Pointers can be incremented/decremented
– what this does to the pointer depends on its type

```c
int main() {
    int a[7];
    int *p;
    p = a;
    *p = 33;
    *p = 33;
    p[1] = 167;
}
```
Pointers and Arrays

\[ p = \&a[0]; \]

- This makes sense, yet is weird and confusing ...
  - \( p \) is of type \( \text{int} \ast \)
    - it can be assigned to
      \[ \text{int} \ast q; \]
      \[ p = q; \]
  - \( a \) sort of behaves like an \( \text{int} \ast \)
    - but it can’t be assigned to
      \[ a = q; \]

\[ p = a; \]

\[ a[i]; \]

really is \( * (a+i) \)
Pointers and Arrays

• An array name represents a pointer to the first element of the array
• Just like a literal represents its associated value
  – in:
    \[
    x = y + 2;
    \]
    » “2” is a \textit{literal} that represents the value 2
  – can’t do
    \[
    2 = x + y;
    \]
Literals and Procedures

```c
int proc(int x) {
    x = x + 4;
    return x * 2;
}

int main() {
    int result = proc(2);
    printf("%d\n", result);
    return 0;
}
```

initialized with a copy of the argument
Arrays and Procedures

```c
int proc(int *a, int nelements) {
    // sizeof(a) == sizeof(int *)
    int i;
    for (i=0; i<nelements-1; i++)
        a[i+1] += a[i];
    return a[nelements-1];
}

int main() {
    int array[50] = ... ;
    // sizeof(array) == 50*sizeof(int)
    printf("result = %d\n", proc(array, 50));
    return 0;
}
```

int proc(int a[], int nelements) {
  // sizeof(a) == sizeof(int *)
  ... 
}

int main() {
  int array[50] = ... ;
  // sizeof(array) == 50*sizeof(int)
  printf("result = %d\n", proc(array, 50));
  return 0;
}
Quiz 2

```c
int proc(int a[], int nelements) {
    int b[5] = {0, 1, 2, 3, 4};
    a = b;
    return a[1];
}

int main() {
    int array[50];
    printf("result = %d\n", proc(array, 50));
    return 0;
}
```

This program prints:

- a) 0
- b) 1
- c) 2
- d) nothing: it doesn’t compile because of a syntax error