CS 33

Introduction to C
Part 2

Some of this lecture is based on material prepared by Pascal Van Hentenryck.

Function Definitions

```
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**

Note the use of the comma in the initialization part of the for loop: the initialization part may have multiple parts separated by commas, each executed in turn.

Compiling It

```
$ gcc -o fact fact.c
$ ./fact
120
```


Function Definitions

```
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;
}
```

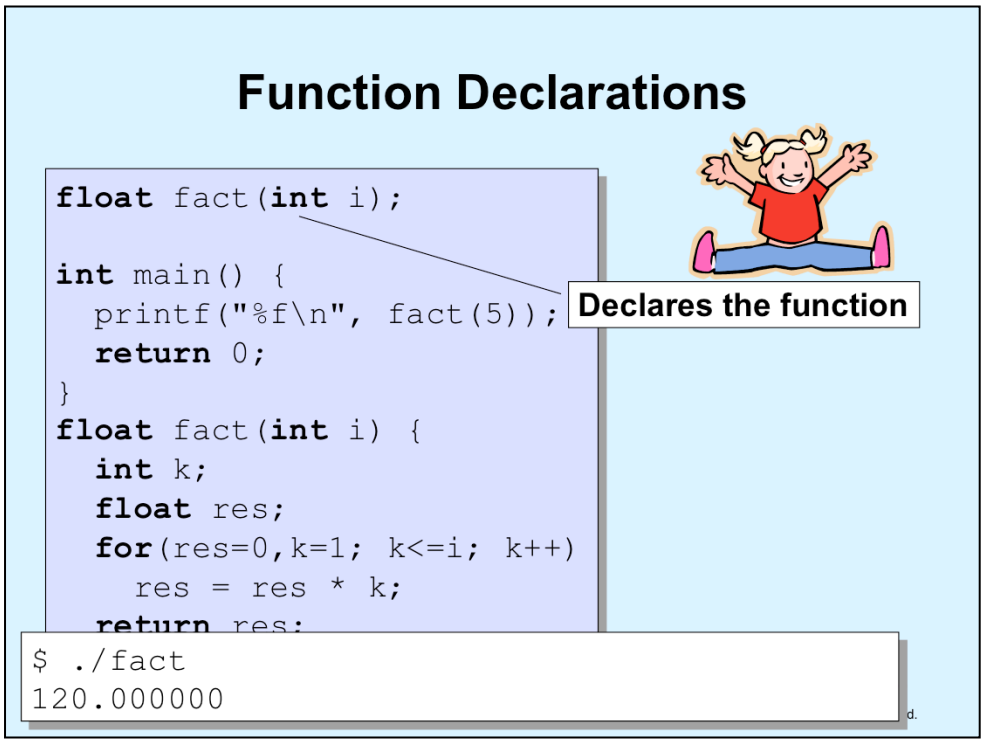

Note that not only has the definition of *main* been placed before the definition of *fact*, but that *fact* has been changed so that it now returns a *float* rather than an *int*.

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
```


If a function, such as *fact*, is encountered by the compiler before it has encountered a declaration or definition for it, the compiler assumes that the function returns an int. This rather arbitrary decision is part of the language for "backwards-compatibility" reasons — so that programs written in older versions of C still compile on newer (post-1988) compilers.



Here we have a declaration of *fact* before its definition. (If the two are different, gcc will complain.)

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


```
for (;;)
  printf("C does not have methods!\n");
```

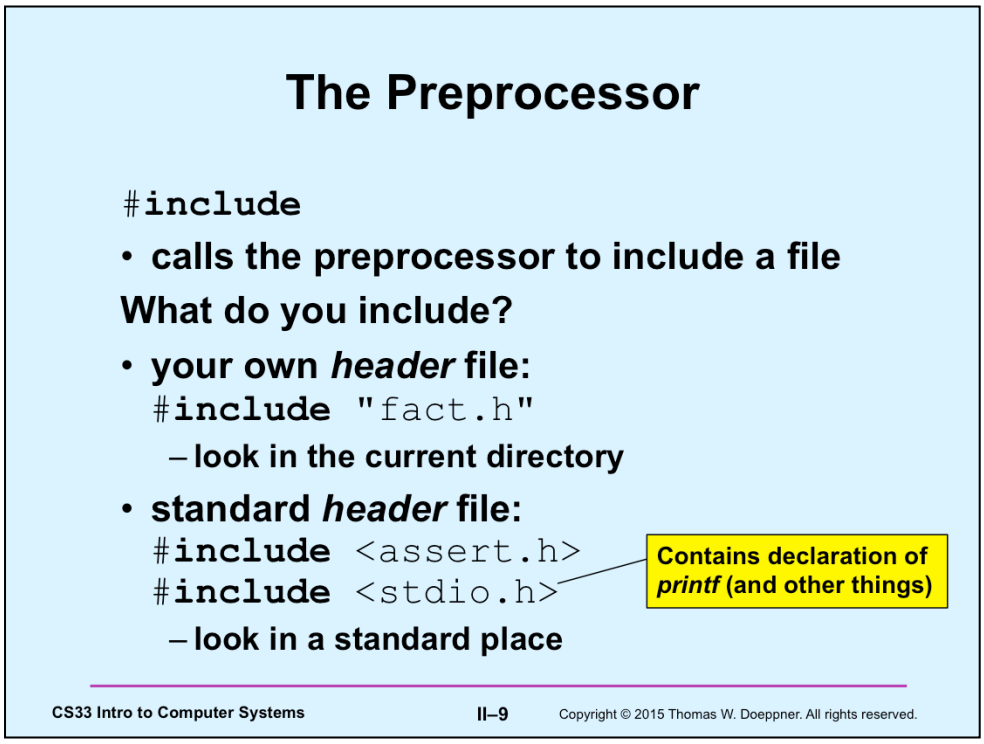
Function Declarations

fact.h

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

fact.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 rules for the distinction between using double quotes and angle brackets are a bit vague. What's shown here is common practice, which should be the rule.

Note that the preprocessor directives (such as #*include*) must start in the first column.

Note that one must include *stdio.h* if using *printf* (and some other routines) in a program.

On most Unix systems (including Linux and OS X), the "standard place" for header files is the directory /usr/include.

#define

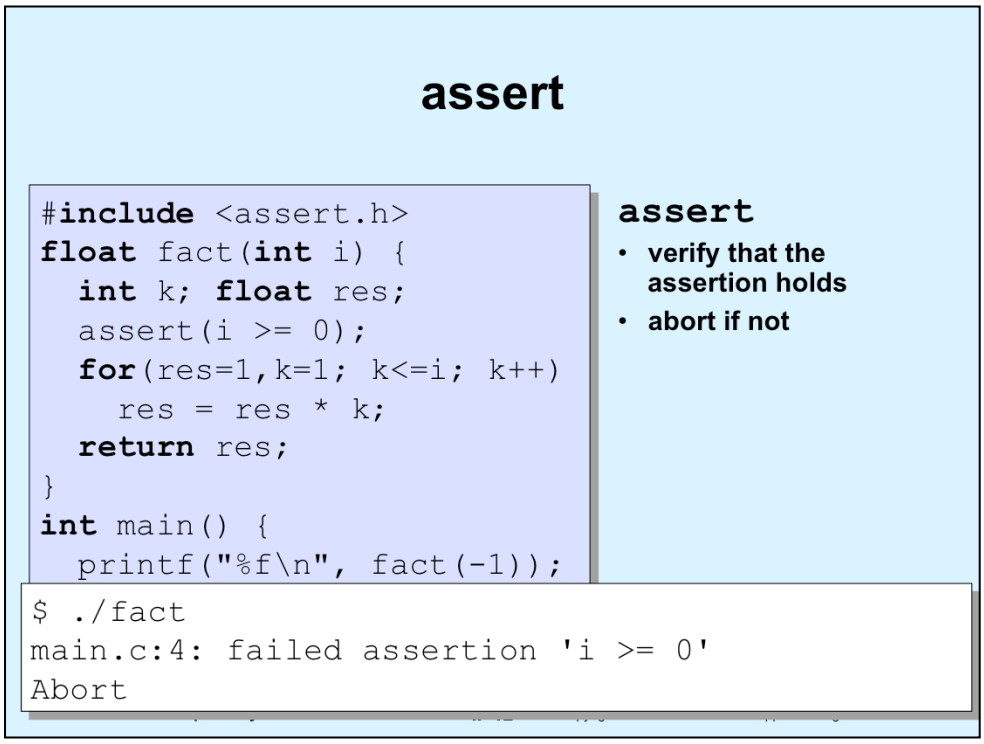
```
#define SIZE 100
int main() {
   int i;
   int a[SIZE];
}
```

#define

- **defines a substitution**
- **applied to the program by the preprocessor**

#define

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

The assert statement is actually implemented as a macro (using #define). One can "turn off" asserts by defining (using #define) NDEBUG. For example,

```
#include <assert.h>
...
#define NDEBUG
...
assert(i>=0);
```

In this case, the assert will not be executed, since NDEBUG is defined. Note that one also can define items such as NDEBUG on the command line for gcc using the –D flag. For example,

```
gcc –o prog prog.c –DNDEBUG
```

Has the same effect as having "#define NDEBUG" as the first line of prog.c.

Parameter passing

Passing arrays to a function

```
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

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


Selection Sort

```
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 */
}
```


Note that C uses the same syntax as Java does for conditional (if) statements. In addition to relational operators such as “==”, “!=”, “<”, “>”, “<=”, and “>=”, there are the conditional operators “&&” and “||” (“logical and” and “logical or”, respectively).

Swapping

Write a function to swap two ints

```
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

```
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
```

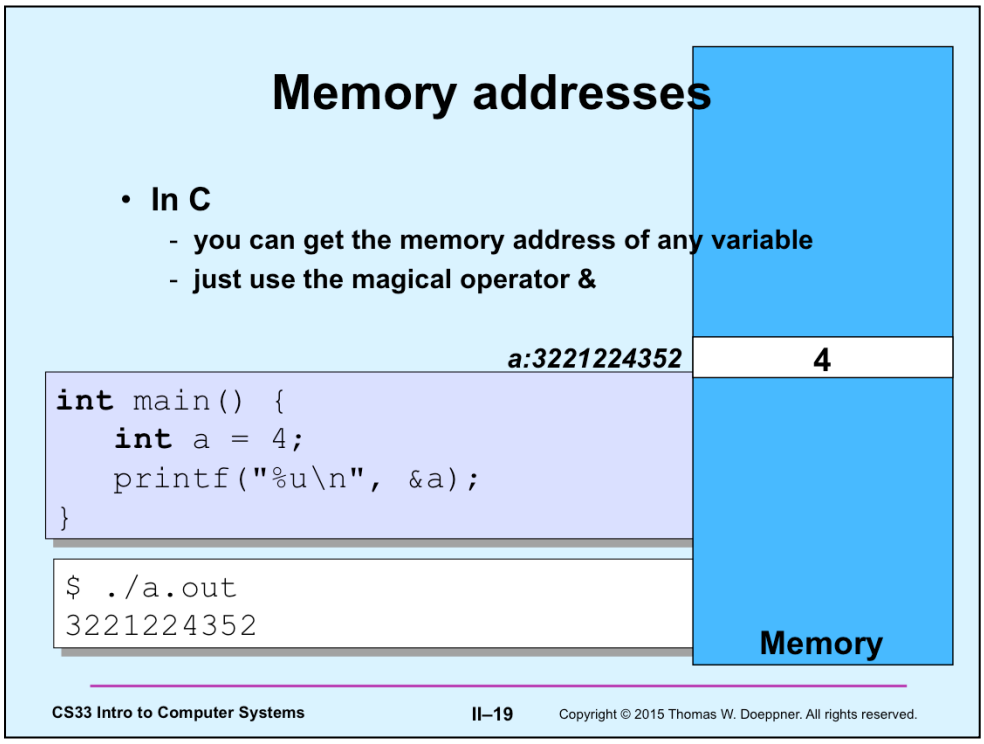

Why “pass by value”?

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

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

```
$ ./a.out
3
```


Note, this has been fixed in Fortran, and, since C passes parameters by value, this has never been a problem in C.



The "%u" format code in printf means to interpret the item being printed as being unsigned. We'll explain this concept more thoroughly in an upcoming lecture. What's being printed is an address, which can't be negative.

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

p is a pointer to an int

if you follow p, you find an int

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

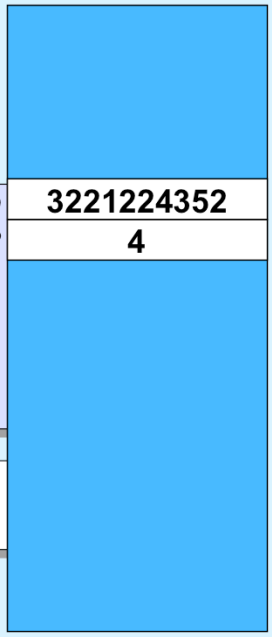
p takes the address of a

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

C Pointers

```
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?

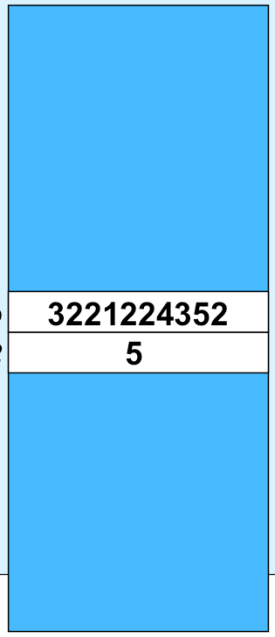
```
void swap(int *i, int *j) {
   int *tmp;
   tmp = j; j = i; i = tmp;
}
int main() {
   int a = 4;
   int b = 8;
   swap(&a, &b);
   printf("a:%d  b:%d\n", a, b);
}
```

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

C Pointers

- **Dereferencing pointers**
  - **accessing/modifying the value pointed to by a pointer**

```
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

```
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
```


Note that "*p" and "a" refer to the same thing after p is assigned the address of a.

"x+=y" means the same as "x = x+y". Similarly, there are −=, *=, and /= operators.

Swapping

```
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
```

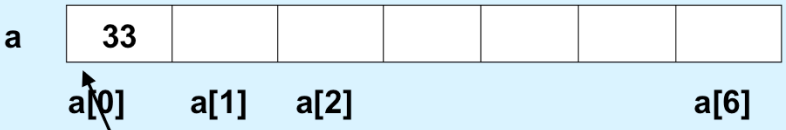

Quiz 1

```
int doubleit(int *p) {
   *p = 2*(*p);
   return *p;
}
int main() {
   int a = 3;
   int b;
   b = doubleit(&a);
   printf("%d\n", a*b);
}
```

What's printed?

a) 0
b) 12
c) 18
d) 36

Pointers and Arrays




```
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

a[0] a[1] a[2] a[6]


```
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**

a[0] a[1] a[2] a[6]

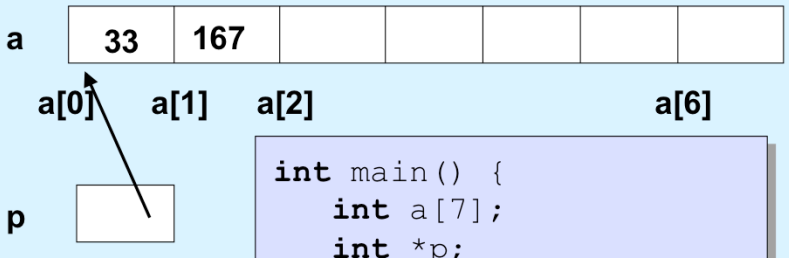

```
int main() {
   int a[7];
   int *p;
   p = &a[0];
}
```

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




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

Pointers and Arrays

`p = &a[0];` **can also be written as** `p = a;`

`a[i];` **really is** `*(a+i)`

- **This makes sense, yet is weird and confusing ...**
  - `p` **is of type** `int *`
    - **it can be assigned to**

```
int *q;
p = q;
```

  - `a` **sort of behaves like an** `int *`
    - **but it can't be assigned to**

~~`a = q;`~~

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 *literal* that represents the value 2
  - can't do

```
2 = x + y;
```


Literals and Procedures

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

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

initialized with a copy of the argument

Arrays and Procedures

initialized with a copy of the argument

```
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;
}
```


Note that the argument to proc is not the entire array, but the pointer to its first element. Thus *a* is initialized by copying into it this pointer.

Equivalently ...

```
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;
}
```

No need for array size, since all that's used is pointer to first element

Note that one could include the size of the array ("`int proc(int a[50], int nelements)`"), but the size would be ignored, since it's not relevant: arrays don't know how big they are. Thus the *nelements* argument is very important.

Quiz 2

```
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