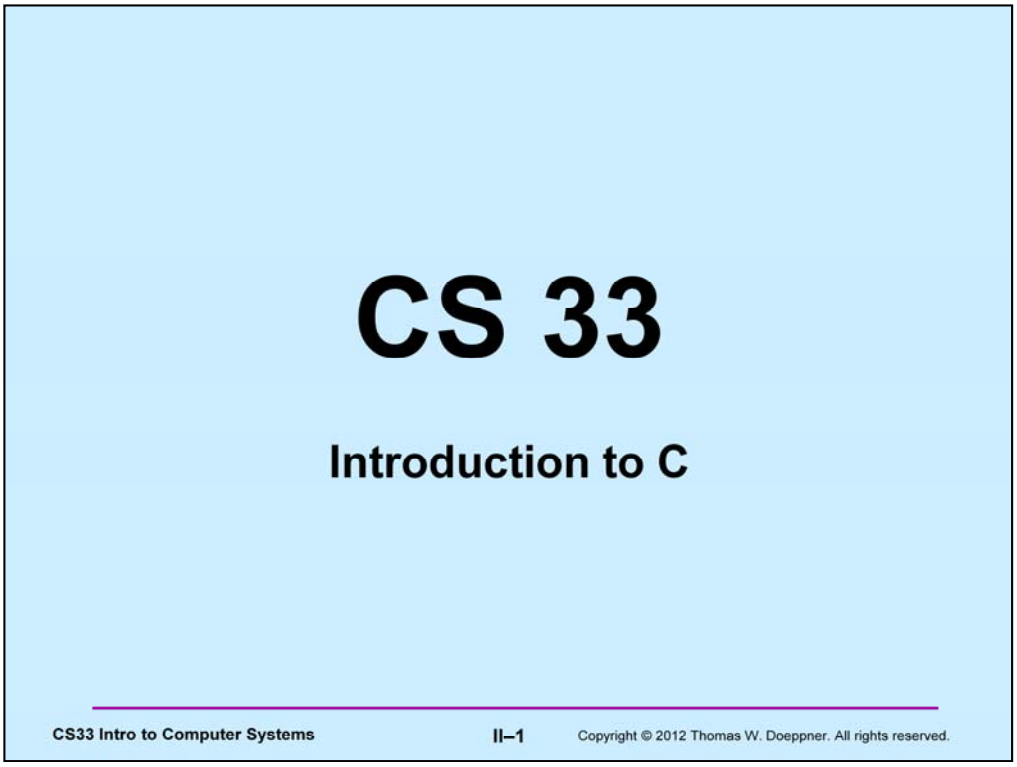
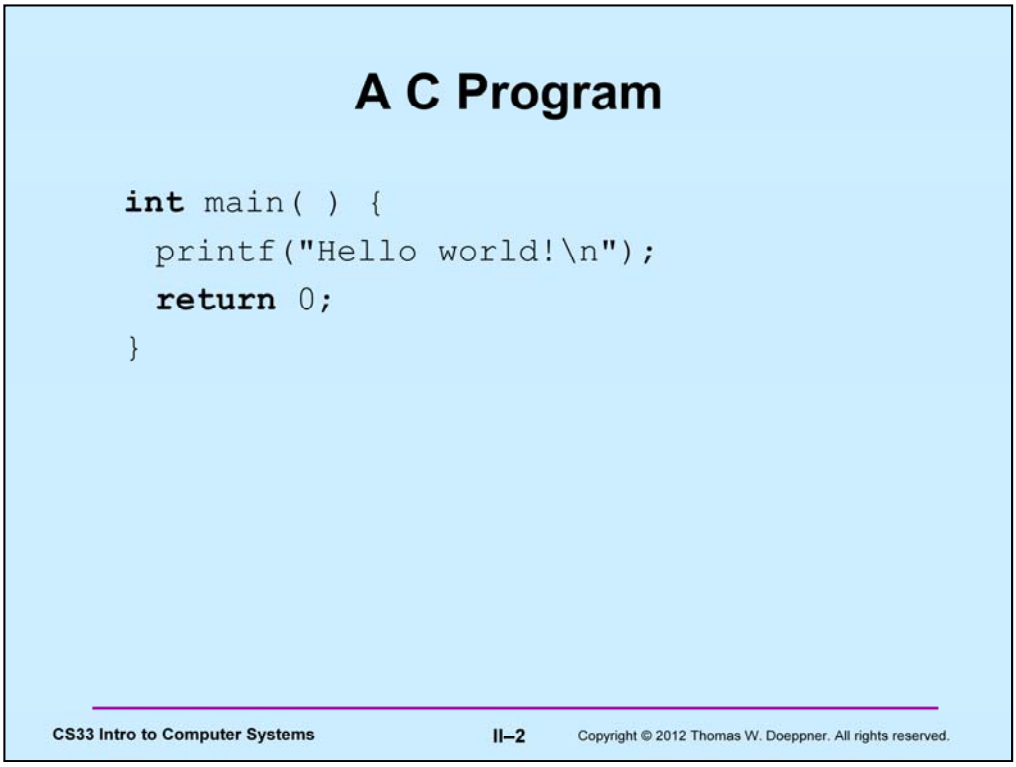
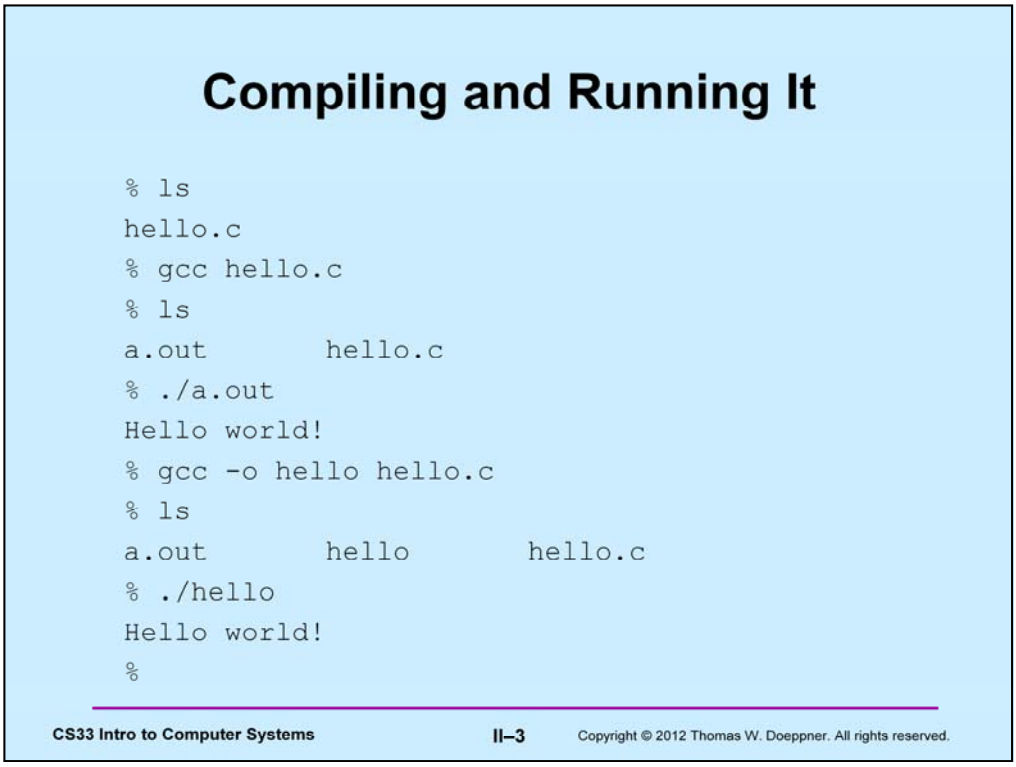


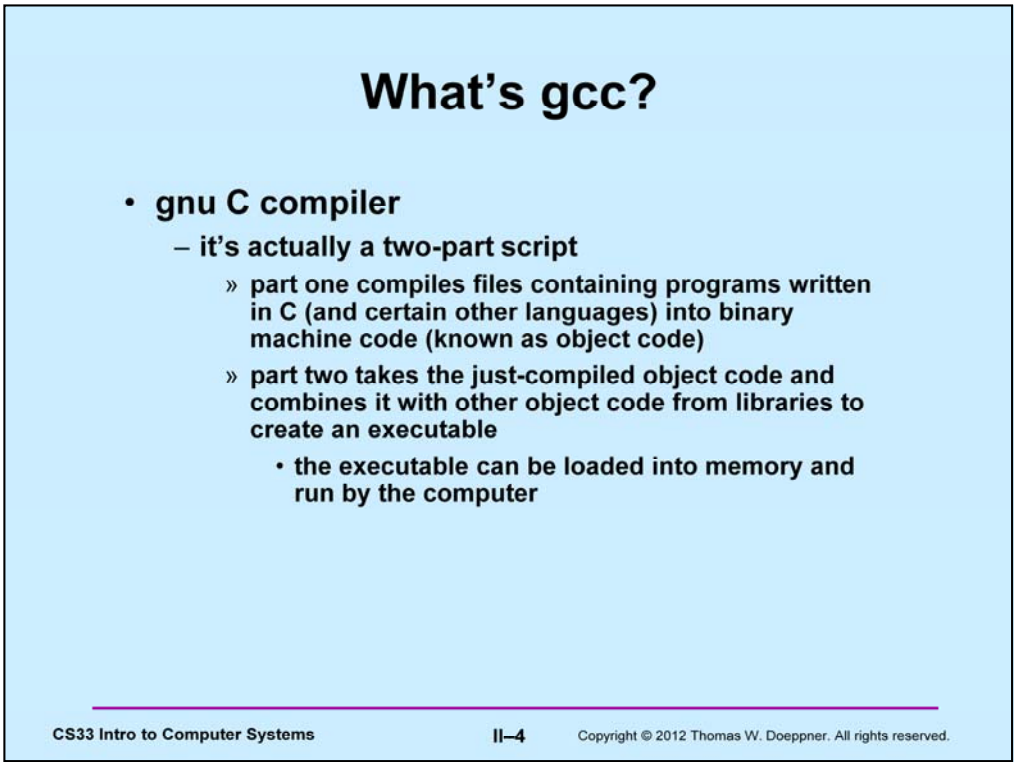
Some of this lecture is based on material prepared by Pascal Van Hentenryck.



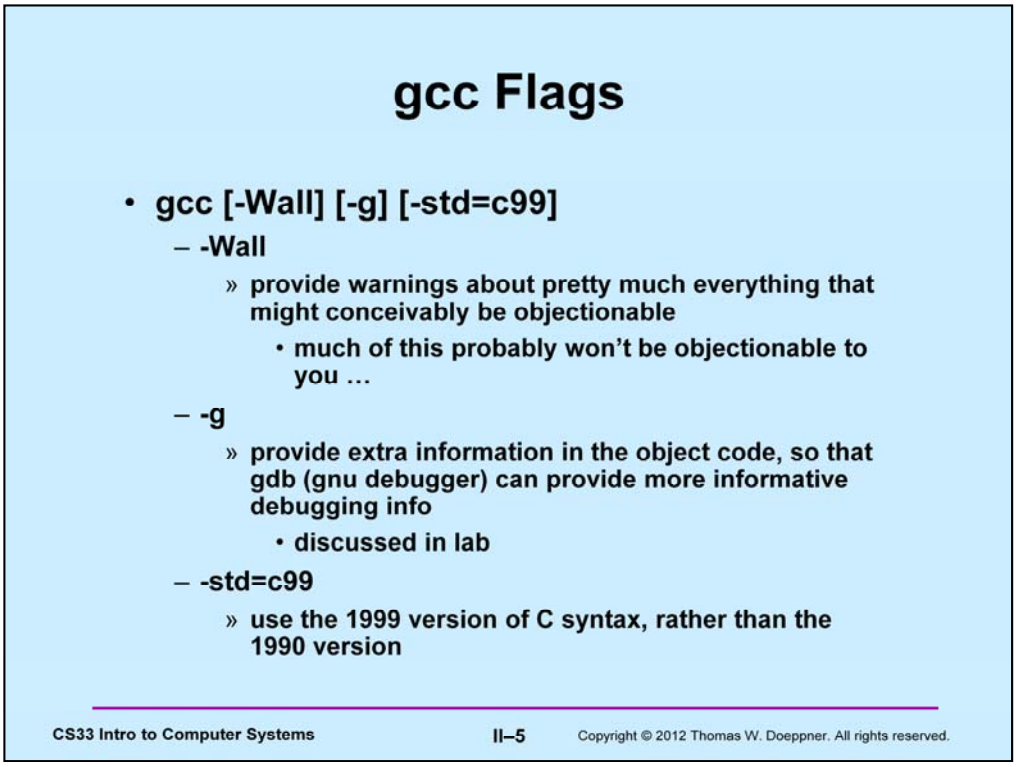
Following K&R, this is everyone's first C program. Note that C programs start in a procedure called *main*, which is a function returning an integer. This integer is interpreted as an error code, where 0 means no errors and anything else is some sort of indication of a problem. We'll see later how we can pass arguments to main.



gcc (the Gnu C compiler), as do other C compilers, calls its output "a.out" by default. (This is supposed to mean the output of the assembler, since the original C compilers compiled into assembly language, which then had to be sent to the assembler.) To give the output of the C compiler, i.e., the executable, a more reasonable name, use the "-o" option.



What's gnu? It's a project of the Free Software Foundation and stands for "gnu's not Unix." That it's not Unix was pretty important when the gnu work was started in the 80s. At the time, AT&T was the owner of the Unix trademark and was very touchy about it. Today the trademark is owned by The Open Group, who is less touchy about it.



The use of the –Wall flag will probably produce lots of warning messages about things you had no idea might possibly be considered objectionable. In most cases, it's safe to ignore them (unless they also show up when you don't use the flag).

Unless you're really concerned about getting the last ounce of performance from your program, it's a good idea always to use the –g flag.

Most of what we will be doing is according to the C90 specification. The C99 specification cleaned a few things up and added a few features. There's also a C11 (2011) specification that is not yet supported by gcc.

Declarations in C

```
int main() {
 int i;
 float f;
 char c;
 return 0;
}
```

- Types are promises
  - promises can be broken
- Types specify memory sizes
  - cannot be broken

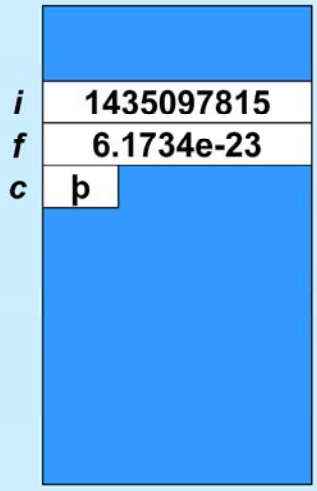
Declarations in C

```
int main() {
 int i;
 float f;
 char c;
 return 0;
}
```

- Declarations reserve memory space
  - where?
- Local variables are uninitialized
  - junk
  - whatever was there before

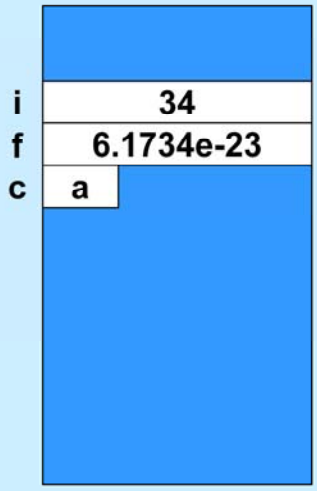
Declarations in C

```
int main() {
 int i;
 float f;
 char c;
 return 0;
}
```



Using Variables

```
int main() {
   int i;
   float f;
   char c;
   i = 34;
   c = 'a';
}
```



printf Again

```
int main() {
   int i;
   float f;
   char c;
   i = 34;
   c = 'a';
   printf("%d\n",i);
   printf("%d\t%c\n",i,c);
}
```

```
% ./a.out
34
34      a
```

printf Again

```
int main() {
   …
   printf("%d\t%c\n",i,c);
}
```

```
% ./a.out
34      a
```

Two parts

- formatting instructions
- arguments

printf Again

```
int main() {
   …
   printf("%d\t%c\n",i,c);
}
```

```
% ./a.out
34      a
```

Formatting instructions

- Special characters
  - \n : newline
  - \t : tab
  - \b : backspace
  - \" : double quote
  - \\ : backslash

printf Again

```
int main() {
   …
   printf("%d\t%c",i,c);
}
```

```
% ./a.out
34      a
```

Formating instructions

- Types of arguments
  - %d: integers
  - %f: floating-point numbers
  - %c: characters

printf Again

```
int main() {
   …
   printf("%6d%3c",i,c);
}
```

```
% ./a.out
   34  a
```

Formatting instructions

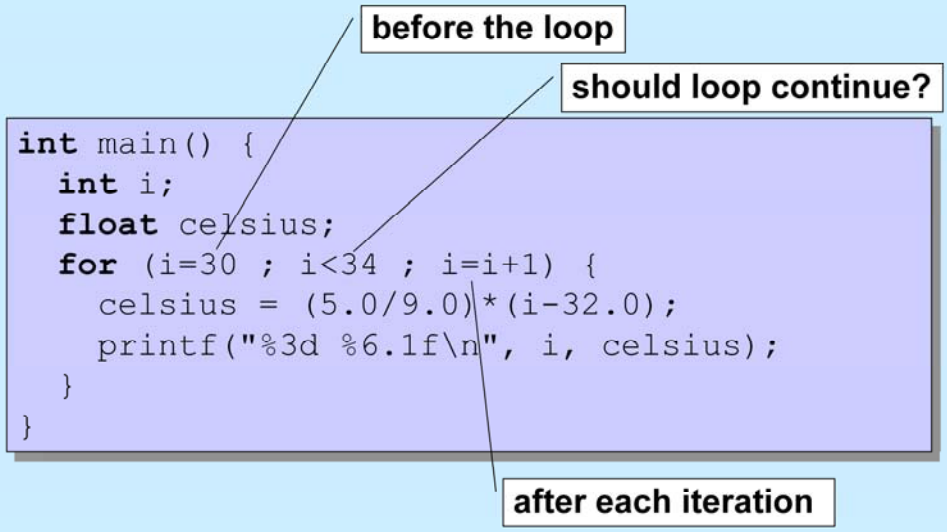
- %6d: decimal integers at least 6 characters wide
- %6f: floating point at least 6 characters wide
- %6.2f: floating point at least 6 wide, 2 after the decimal point

printf Again

```
int main() {
  int i;
  float celsius;
  for(i=30; i<34; i++) {
    celsius = (5.0/9.0)*(i-32.0);
    printf("%3d %6.1f\n", i, celsius);
  }
}
```

```
% ./a.out
 30   -1.1
 31   -0.6
 32    0.0
 33    0.6
```

For Loops



Some Primitive Data Types

int
- integer: 16 bits or 32 bits (implementation dependent)

long
- integer: either 32 bits or 64 bits, depending on the architecture

long long
- integer: 64 bits

char
- a single byte

float
- single-precision floating point

double
- double-precision floating point

The sizes of integers depends on the underlying architecture. In the earliest versions of C, the *int* type had a size equal to that of pointers on the machine. However, the current definitions of C apply this rule to the *long* type.

What is the size of my int?

```
int main() {
  int i;
  printf("%d\n", sizeof(i));
}
```

```
% ./a.out
4
```

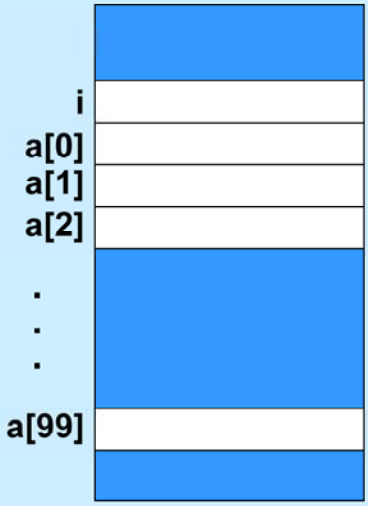
sizeof

- return the size of a variable in bytes
- very very very very very very important function in C

Note that the argument to *sizeof* need not be a variable, but could be the name of a type. For example, “sizeof(int)” is legal and returns 4 on most machines.

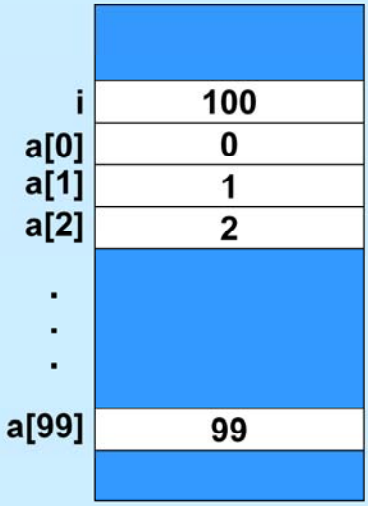
Arrays

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



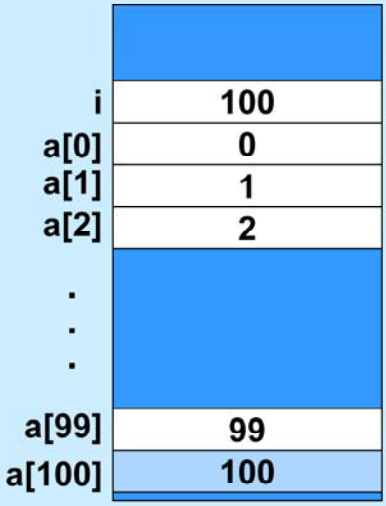
Arrays

```
int main() {
   int i;
   int a[100];
   for(i=0;i<100;i++)
     a[i] = i;
}
```



Array Bounds

```
int main() {
   int i;
   int a[100];
   for(i=0;i<=100;i++)
     a[i] = i;
}
```



Arrays in C

C Arrays = Storage + Indexing

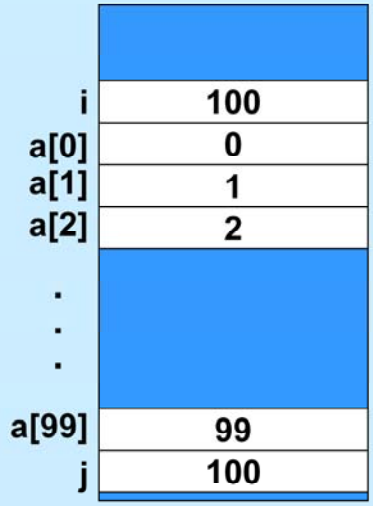
- no bounds checking
- no initialization

WELCOME TO THE JUNGLE

Welcome to the Jungle

```
int main() {
   int i;
   int a[100];
   int j;
   j = 8;
   for(i=0;i<=100;i++)
     a[i] = i;
   printf("%d\n", j);
}
```

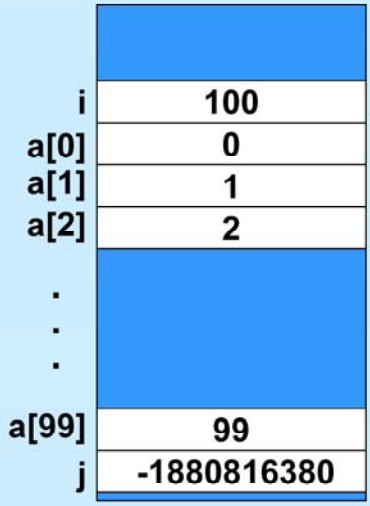
```
% ./a.out
100
```



Welcome to the Jungle

```
int main() {
   int i;
   int a[100];
   int j;
   for(i=0;i<100;i++)
     a[i] = i;
   printf("%d\n", j);
}
```

```
% ./a.out
-1880816380
```



Welcome to the Jungle

```
int main() {
  int i;
  int a[100];
  a[-3] = 25;
  printf("%d\n", a[-3]);
}
```

```
% ./a.out
25
```


Note that this code is not guaranteed to work!

Welcome to the Jungle

```
int main() {
  int i;
  int a[100];
  a[-3] = 25;
  a[11111111] = 6;
  printf("%d\n", a[-3]);
}
```

```
% ./a.out
Segmentation fault
```

What is a segmentation fault?

- **attempted access to an invalid memory location**

Sometimes the error message is “bus error.” Both terms (segmentation fault and bus error) come from the original C/Unix implementation on the PDP-11. A segmentation fault resulted from accessing memory that might exist, but for which the accessor has no permission. A bus error results from trying to use an address that makes no sense.

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

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;
}
```

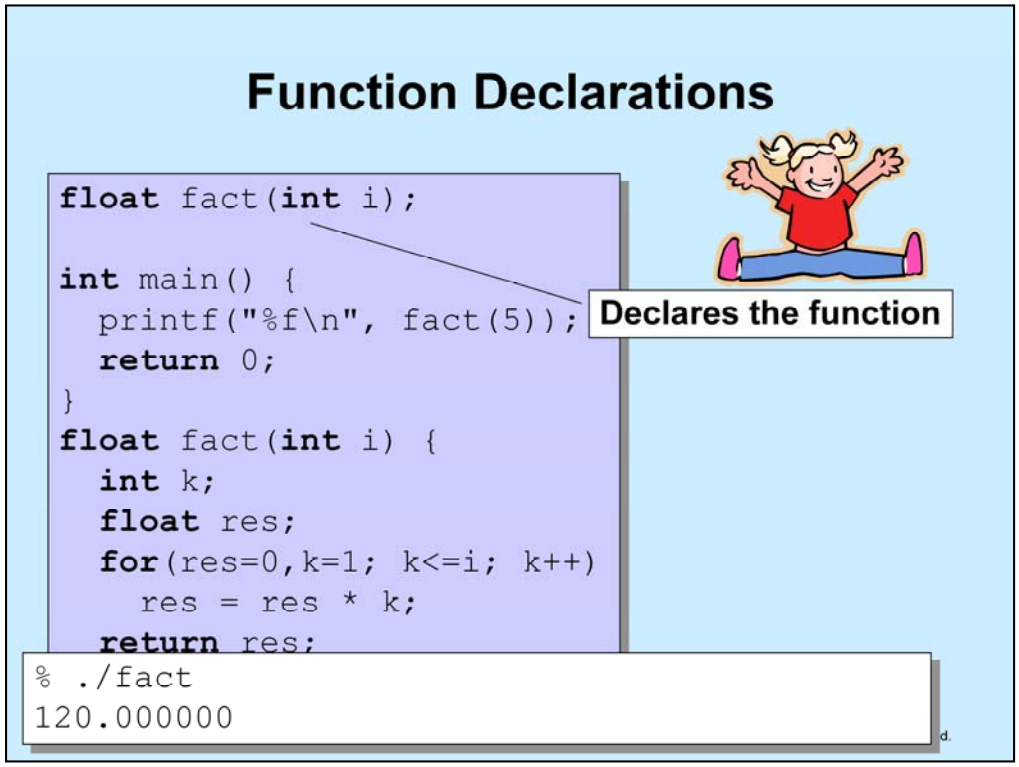

Note that not only has the definition of *main* been placed before the definition of *fact*, but that *fact* has been changed so that it now returns a *float* rather than an *int*.

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.)

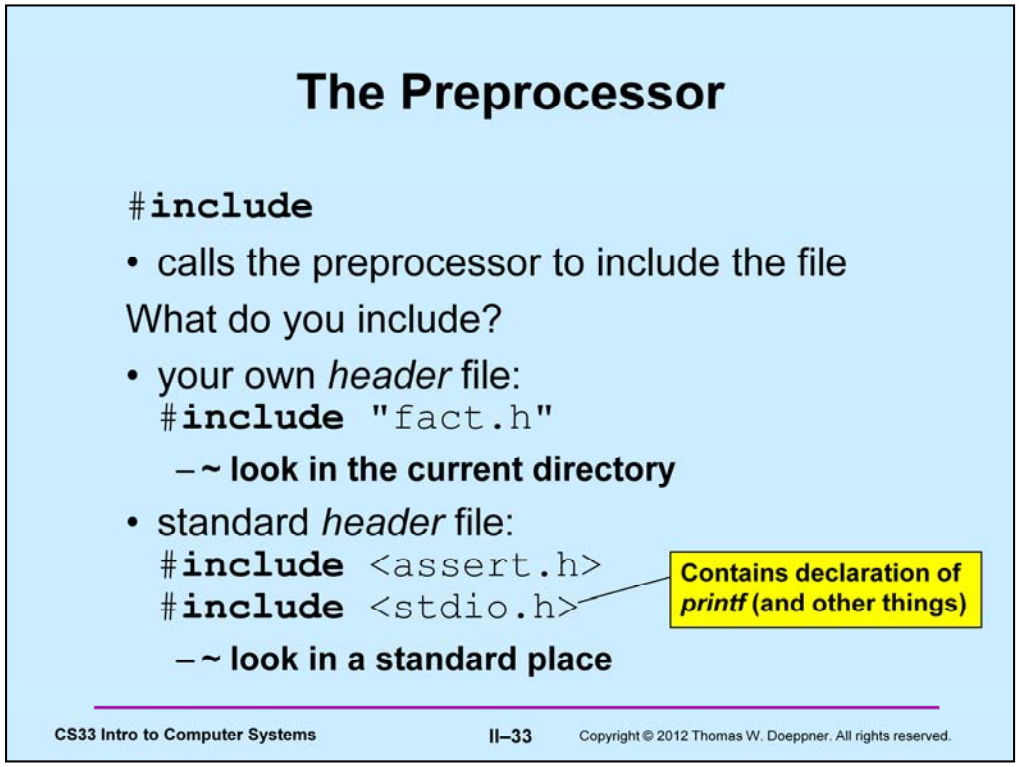
Function Declarations

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.

#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");
   }
}
```


assert

```
#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));
```

assert

- verify that the assertion holds
- abort if not

```
% ./fact
main.c:4: failed assertion 'i >= 0'
Abort
```

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 large it is**

Strings (or something like them)

Strings are arrays of characters terminated by '\0' (“null”)

- the '\0' is included at the end of string constants
  - `"Hello"`

Strings

```
int main() {
   printf("%s\n","Hello");
   return 0;
}
```

```
% ./a.out
Hello
```


Strings

```
void printString(char s[]) {
   int i;
   for(i=0; s[i]!='\0'; i++)
      printf("%c", s[i]);
}
int main() {
   printString("Hello");
   printf("\n");
   return 0;
}
```

Tells C that this function does not return a value

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 */
}
```

Swapping

Write a function to swap two ints

```
void swap(int i, int j) {
        
}
int main() {
   int a; a = 4;
   int b; 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; a = 4;
   int b; 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.

Memory addresses

In C

- you can take the address of any object
- just use the magical operator &

```
int main() {
   int a; 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

p is a pointer to an int

if you follow p, you find an int

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

p 3221224352

a:3221224352 4

```
% ./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; a = 4;
   int b; 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; 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; a = 4;
   p = &a;
   printf("%d\n", *p);
   *p = *p + 1;
   *p += 3;
   printf("%d\n", *p);
}
```

```
% ./a.out
4
8
```

Swapping

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

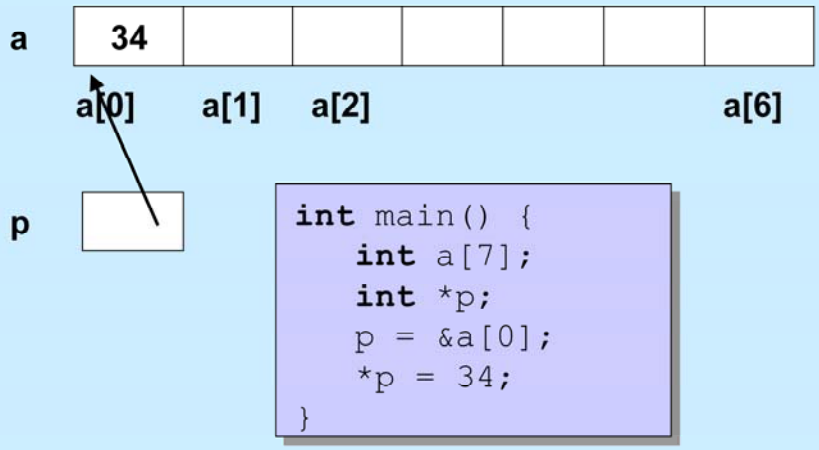

```
% ./a.out
a:8  b:4
```


Pointers and Arrays

- Strong relationship between them

```
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));
}
```


What is an array?

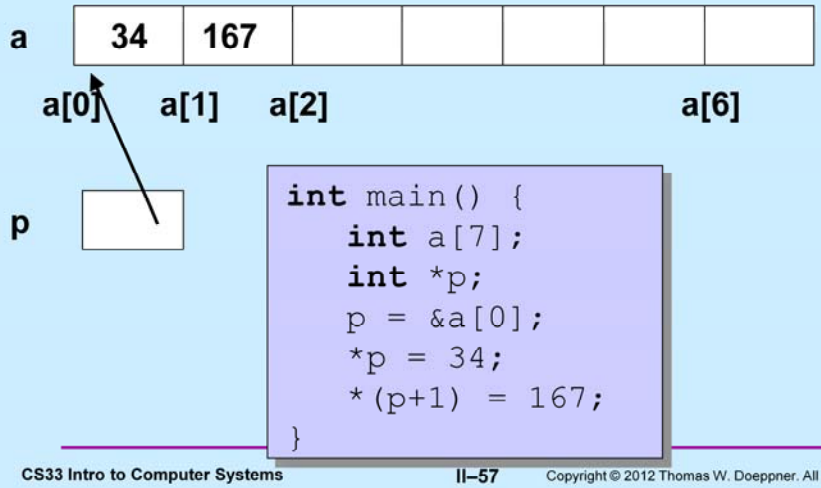



```
int main() {
    int a[7];
    int *p;
    p = &a[0];
    *p = 34;
}
```


Pointer Arithmetic

Pointers can be incremented/decremented

– what this does to the pointer depends on its type

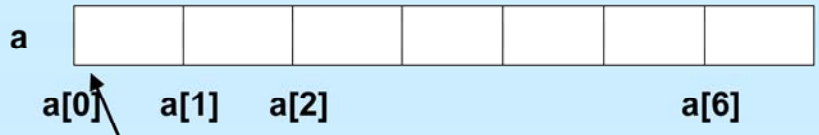



```
int main() {
    int a[7];
    int *p;
    p = &a[0];
    *p = 34;
    *(p+1) = 167;
}
```

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]

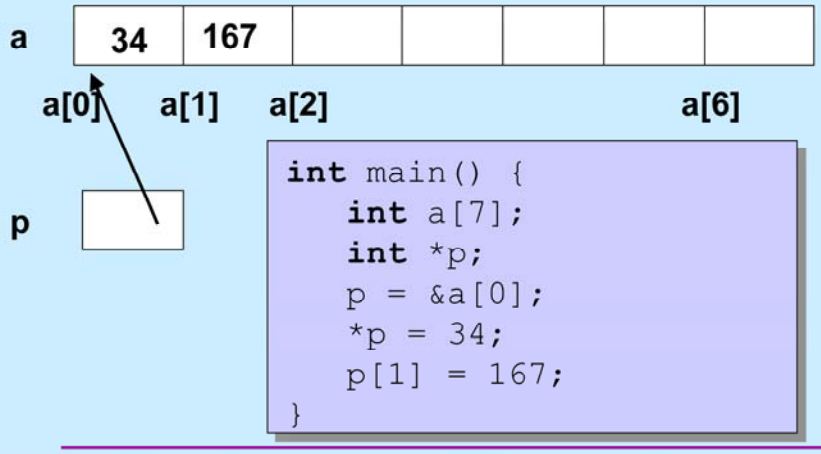

```
int main() {
   int a[7];
   int *p;
   p = &a[0];
}
```

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[0];
    *p = 34;
    p[1] = 167;
}
```

Pointer Arithmetic

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

can also be written as

```
p = a;
```

```
a[i];
```

really is

```
*(a+i)
```

Arrays and Parameters

```
void func(int arg[]) {
   /* arg points to the caller's array */
   int local[7];     /* seven ints */
   arg++;            /* legal */
   arg = local;      /* legal */
   local++;          /* illegal */
   local = arg;      /* illegal */
}
```

Pointer Arithmetic

```
int a[]
```

when declaring a parameter, it can also be written as

```
int *a
```

which also explains why

- arrays do not know they are arrays
- they do not know their sizes

C Arrays = Storage + Indexing

Dereferencing C Pointers

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

```
% ./a.out
-1073742732  3221224360
```


Operator precedence is hard to remember!

Dereferencing C Pointers

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

```
% ./a.out
5 3221224356
```

Dereferencing C Pointers

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

```
% ./a.out
5 3221224356
```


2-D Arrays

- Suppose `T` is a datatype (such as `int`)
- `T n[6]`
  - declares `n` to be an array of (six) `T`
  - the type of `n` is `T[6]`
- Thus `T[6]` is effectively a datatype
- Thus we can have an array of `T[6]`
- `T m[7][6]`
  - `m` is an array of (seven) `T[6]`
  - `m[i]` is of type `T[6]`
  - `m[i][j]` is of type `T`

Note that even though we might think of "int [6]" as being a datatype, to declare "n" to be of that type, we must write "int n[6]". Similarly, to have an array of seven of this type, we must write "int m[7][6]". We could have an array of eight of these 2-D arrays; such an array would be declared "int p[8][7][6]".

2-D Arrays

```
#define NUM_ROWS 3
#define NUM_COLS 4
…
int main() {
   int row, col;
   int m[NUM_ROWS][NUM_COLS];
   for(row=0; row<NUM_ROWS; row++)
     for(col=0; col<NUM_COLS; col++)
        m[row][col] = row*NUM_COLS+col;
   printMatrix(NUM_ROWS, NUM_COLS, m);
   return 0;
}
```

```
% ./a.out
   0   1   2   3
   4   5   6   7
   8   9  10  11
```

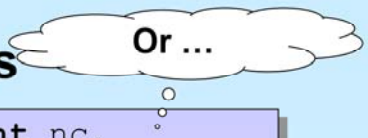
2-D Arrays


```
void printMatrix(int nr, int nc,
      int m[nr][nc]) {
   int row, col;
   for(row=0; row<nr; row++) {
     for(col=0; col<nc; col++)
        printf("%6d", m[row][col]);
     printf("\n");
   }
}
```


2-D Arrays


```
void printMatrix(int nr, int nc,
      int m[][nc]) {
   int row, col;
   for(row=0; row<nr; row++) {
     for(col=0; col<nc; col++)
        printf("%6d", m[row][col]);
     printf("\n");
}
```


2-D Arrays



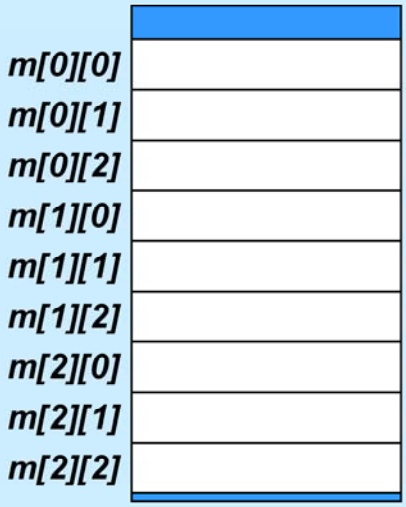

```
void printMatrix(int nr, int nc,
      int m[nr][nc]) {
   int i;
   for(i=0; i<nr; i++)
      printArray(nc, m[i]);
}
```

```
void printArray(int nc, int a[nc]) {
   int i;
   for(i=0; i<nr; i++)
      printf("%6d", a[i]);
   printf("\n");
}
```


Note that m is an array of arrays.

Memory Layout

```
#define NUM_ROWS 3
#define NUM_COLS 3
```



Lifetime of an object

Never, ever do this!

```
int *getArray() {
   int a[10]
   return a;
}
```

The array is allocated on the stack

- **it "disappears" after the function call**

Lifetime of an object

- All objects declared in functions
  - arguments
  - local variables

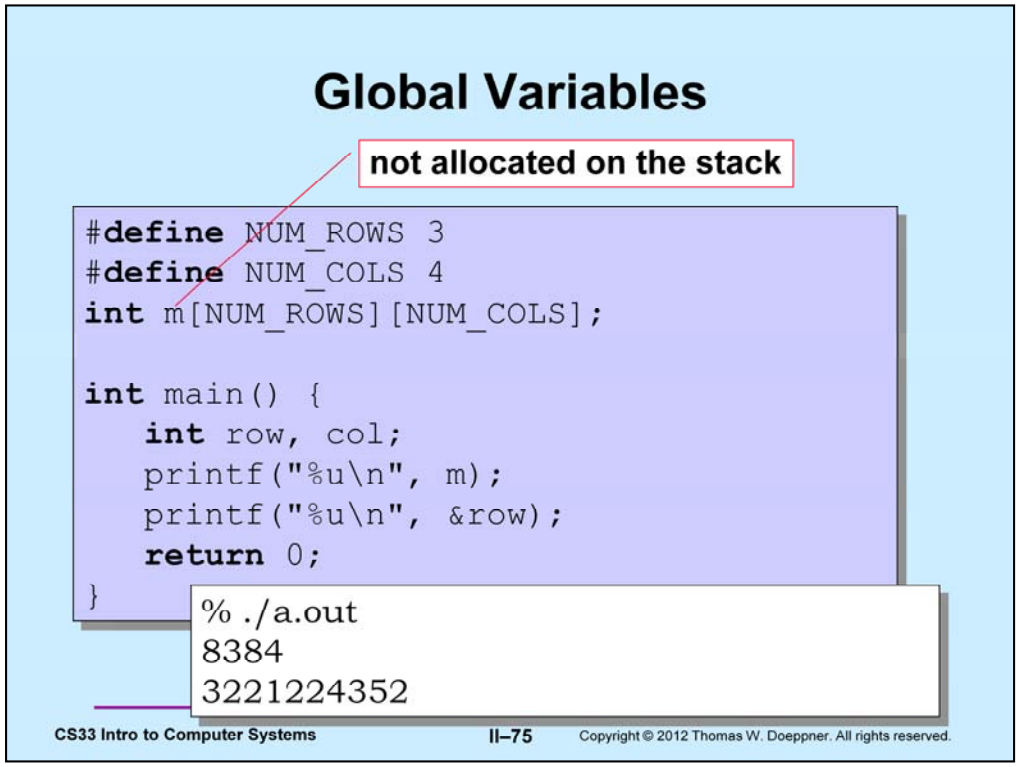
  are allocated on the stack†
- They are "deallocated" after the function returns;
- Never return/use an address to any of them after the function call
  - although C lets you do it

†not quite true — we discuss "static local" variables later

Global Variables


```
#define NUM_ROWS 3
#define NUM_COLS 4
int m[NUM_ROWS][NUM_COLS];

int main() {
   int row, col;
   for(row=0; row<NUM_ROWS; row++)
     for(col=0; col<NUM_COLS; col++)
        m[row][col] = row*NUM_COLS+col;
   return 0;
}
```

Note that the reference to “m” gives the address of the array in memory.

Global Variables are Initialized!

```
#define NUM_ROWS 3
#define NUM_COLS 4
int m[NUM_ROWS][NUM_COLS];

int main() {
   printf("%d\n", m[0][0]);
   return 0;
}
```

```
% ./a.out
0
```

scanf: Reading Data

```
int main() {
   int i, j;
   scanf("%d%d", &i, &j);
}
```

Two parts

- **formatting instructions**
- **arguments: must be addresses**
  - **why?**

