CSCI 0330/1330

Introduction to Computer Systems

Welcome!

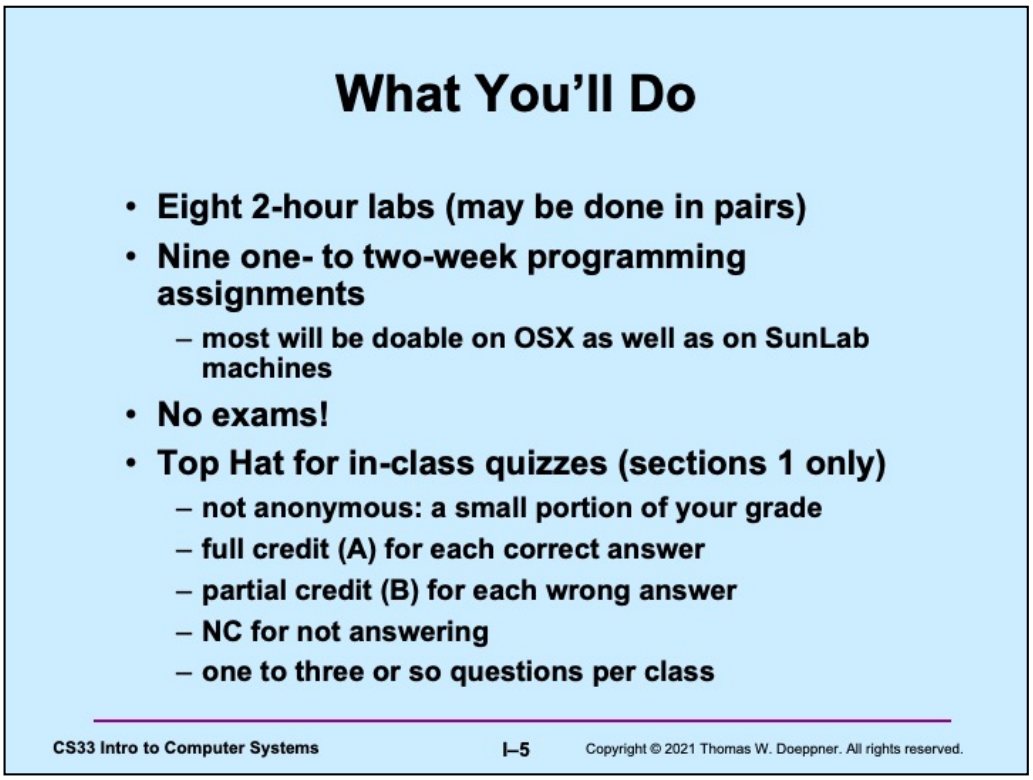
- Prof: Tom Doeppner
- HTAs: JC Loh, Ben Smith, Floria Tsui, Milanca Wang
- UTAs: Adrian Chang, Ashley Chang, Alan Gu, Divyam Dang, Sheridan Feucht, Yifan Ruan, Edward Xing, Emily Zhang, Harisen Luby, Jacob Axel, Zain Asghar, Ziwen Zhou, Jiaqi (Kaki) Su, Jenny Tan, Kenya Kimata, Kota Soda, Kelly Wang, Marc Mapeke, Nitya Thakkar, Peter Theodores, Peter Zubiago, Roman Hall, Richard Tang, Rosanna Zhao, Sahil Bansal, Fernando Cisneros, Aaron Rosario Jeyaraj, Jordan Walendom

What You'll Learn

- Programming in C
- Data representation
- Programming in x86 assembler language
- High-level computer architecture
- Optimizing programs
- Linking and libraries
- Basic OS functionality
- Memory management
- Network programming (Sockets)
- Multithreaded programming (POSIX threads)

Prerequisites: What You Need to Know

- Ability to program in an object-oriented or procedural language (e.g., Java) and knowledge of basic algorithms
  - CSCI 0160 or CSCI 0180



Please visit https://ithelp.brown.edu/kb/articles/top-hat-student-guide to find instructions for downloading the Top Hat software to your laptop or smartphone.

CSCI 1330

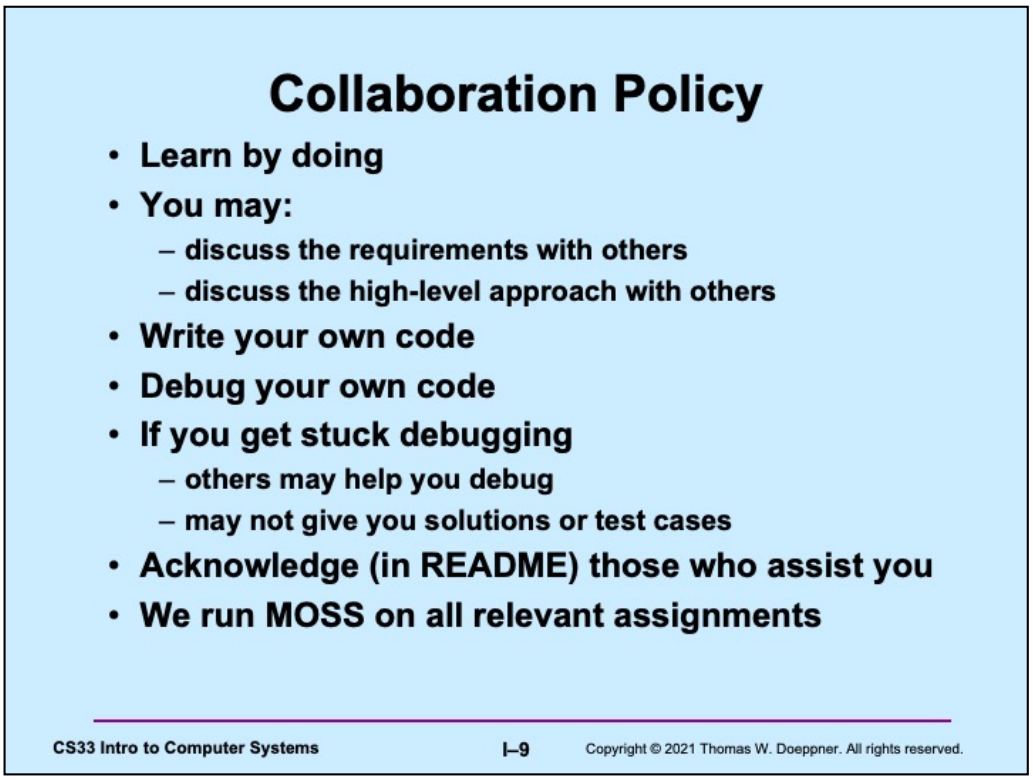
- Master's students only
- Weekly homeworks, just for you
  - 10% of your grade

Gear-Up Sessions

- Optional weekly sessions
  - handle questions about the week's assignment and course material
  - Soon after each assignment is released
    - » first session is 8pm Monday, 9/13
    - » via zoom (link TBD)

Take Aways

- A few questions on lecture material on the web site after each lecture
  - completely optional
  - not graded
- They help you digest the lecture material
  - you may discuss them with each other, with TAs, and with the instructor



Please see https://docs.google.com/document/d/1wEBDvxjsD4QjM-mTKbr8B86LFJZC8i-Hlg4LF2utIsI/edit#heading=h.mf0itv92115o for details.

Textbook

- ***Computer Systems: A Programmer's Perspective,*** **3rd Edition, Bryant and O'Hallaron, Prentice Hall 2015**

If Programming Languages Were Cars …

- Java would be an SUV
  - automatic transmission
  - stay-in-lane technology
  - adaptive cruise control
  - predictive braking
  - gets you where you want to go
    - » safe
    - » boring

- Pyret would be a Tesla
  - you drive it like an SUV
    - » (avoid autopilot)
    - » definitely cooler
    - » but limited range

If Programming Languages Were Cars …

- C would be a sports car
  - manual everything
  - dangerous
  - fun
  - you really need to know what you're doing!

For an interesting, though tough-going discussion of why C is relevant, see https://www.cs.kent.ac.uk/people/staff/srk21/research/papers/kell17some-preprint.pdf.

U-Turn Algorithm (Java and Pyret Version)

1. Switch on turn signal
2. Slow down to less than 3 mph
3. Check for oncoming traffic
4. Press the accelerator lightly while turning the steering wheel pretty far in the direction you want to turn
5. Lift your foot off the accelerator and coast through the turn; press accelerator lightly as needed
6. Enter your new lane and begin driving

U-Turn Algorithm (C Version)

1. Enter turn at 30 mph in second gear
2. Position left hand on steering wheel so you can quickly turn it one full circle
3. Ease off accelerator; fully depress clutch
4. Quickly turn steering wheel either left or right as far as possible
5. A split second after starting turn, pull hard on handbrake, locking rear wheels
6. As car (rapidly) rotates, restore steering wheel to straight-ahead position and shift to first gear
7. When car has completed 180° turn, release handbrake and clutch, fully depress accelerator

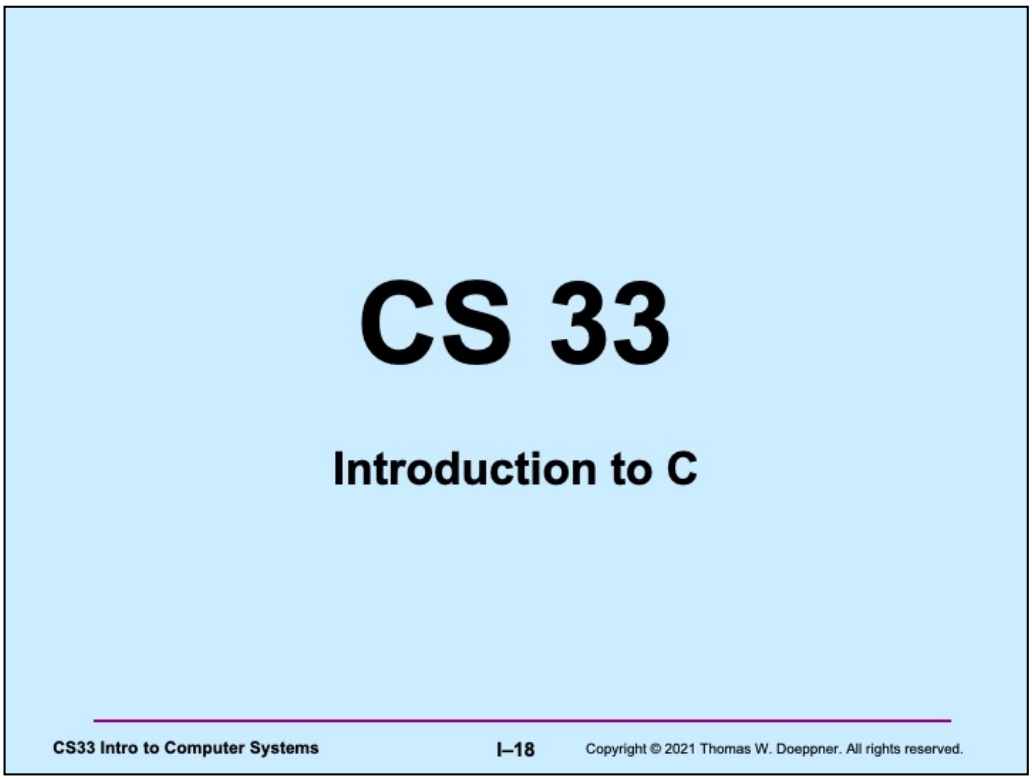
History of C

- Early 1960s: CPL (Combined Programming Language)
  - developed at Cambridge University and University of London
- 1966: BCPL (Basic CPL): simplified CPL
  - intended for systems programming
- 1969: B: simplified BCPL (stripped down so its compiler would run on minicomputer)
  - used to implement earliest Unix
- Early 1970s: C: expanded from B
  - motivation: they wanted to play "Space Travel" on minicomputer
  - used to implement all subsequent Unix OSes

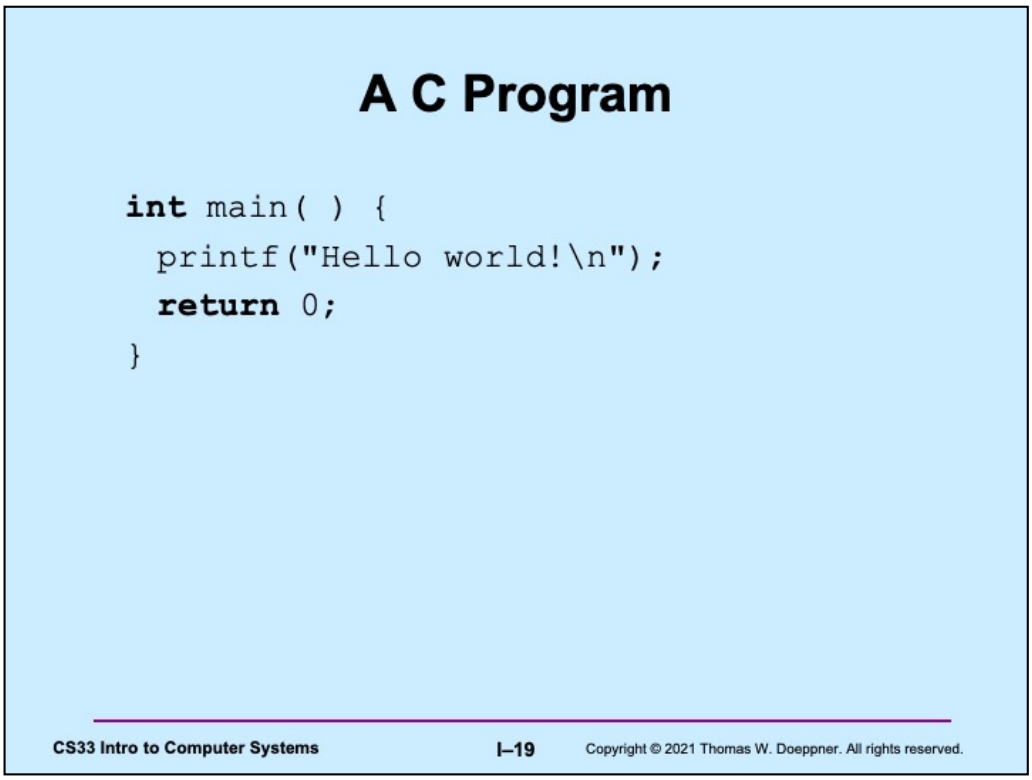
See http://en.wikipedia.org/wiki/C_programming_language.

More History of C

- 1978: Textbook by Brian Kernighan and Dennis Ritchie (K&R), 1st edition, published
  - de facto standard for the language
- 1989: ANSI C specification (ANSI C)
  - 1988: K&R, 2nd edition, published, based on draft of ANSI C
- 1990: ISO C specification (C90)
  - essentially ANSI C
- 1999: Revised ISO C specification (C99)
- 2011: Further revised ISO C specification (C11)
  - not widely used



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

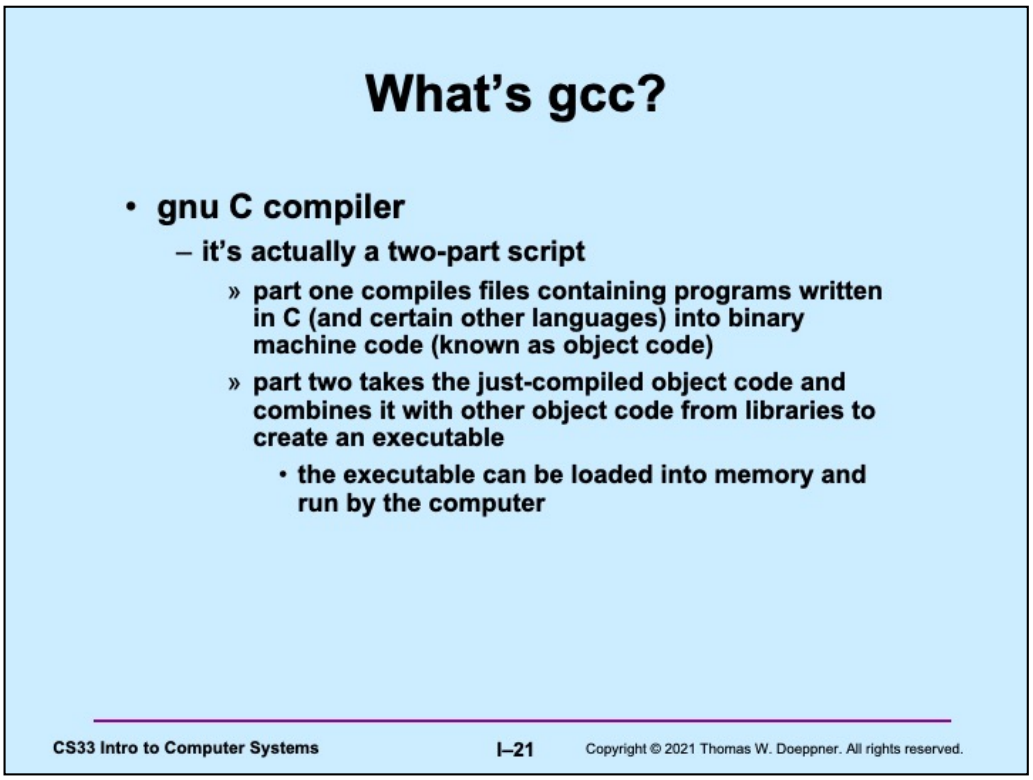


Following K&R, this is everyone's first C program. Note that C programs start in a function called *main*, which is a function returning an integer. This integer is interpreted as an error code, where 0 means no errors and anything else is some sort of indication of a problem. We'll see later how we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

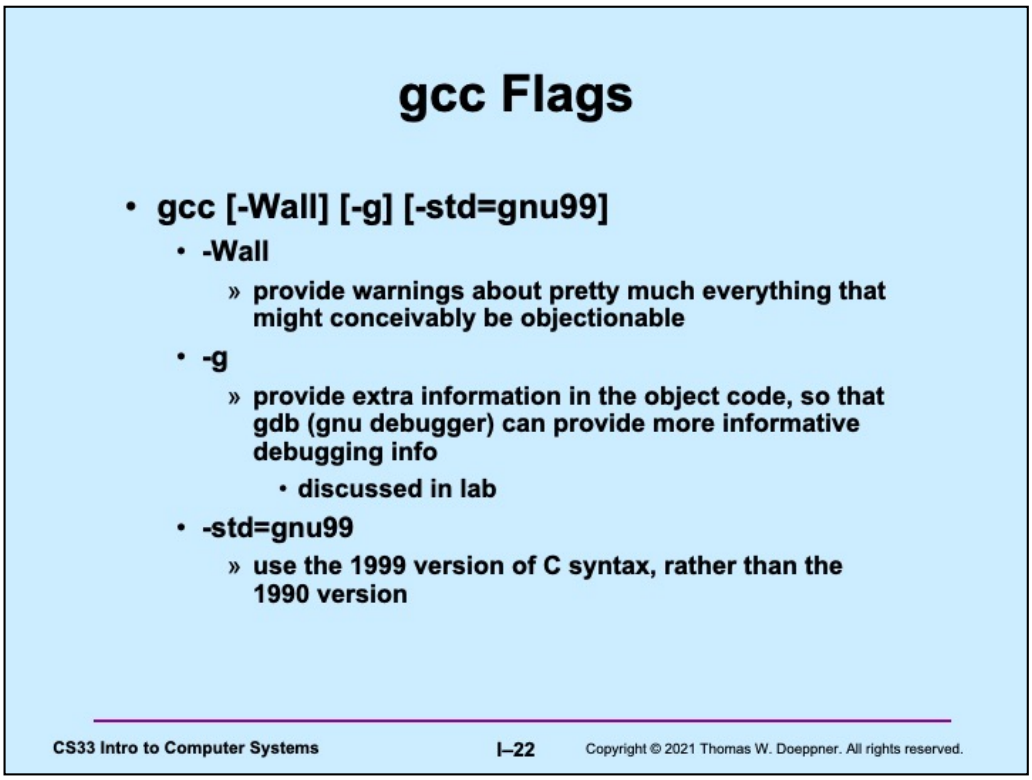
Compiling and Running It

```
$ ls
hello.c
$ gcc hello.c
$ ls
a.out       hello.c
$ ./a.out
Hello world!
$ gcc -o hello hello.c
$ ls
a.out       hello       hello.c
$ ./hello
Hello world!
$
```


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.

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 widely used.

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

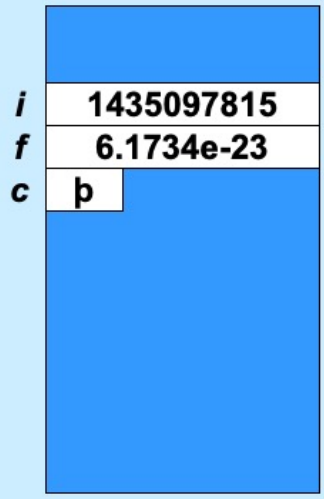
Declarations in C

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

- **Declarations reserve memory space**
  - **where?**
- **Local variables can be uninitialized**
  - **junk**
  - **whatever was there before**

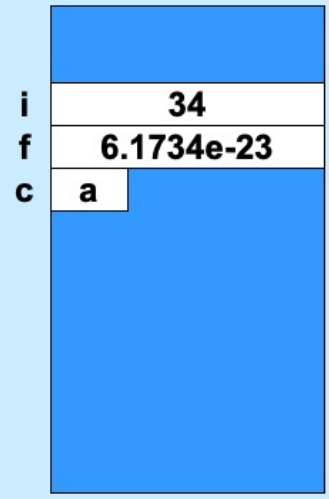
Declarations in C

```
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
```

Formatting instructions

- **Types of arguments**
  - **%d: integer**
  - **%f: floating-point number**
  - **%c: character**

printf Again

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

```
$ ./a.out
    34  a
```

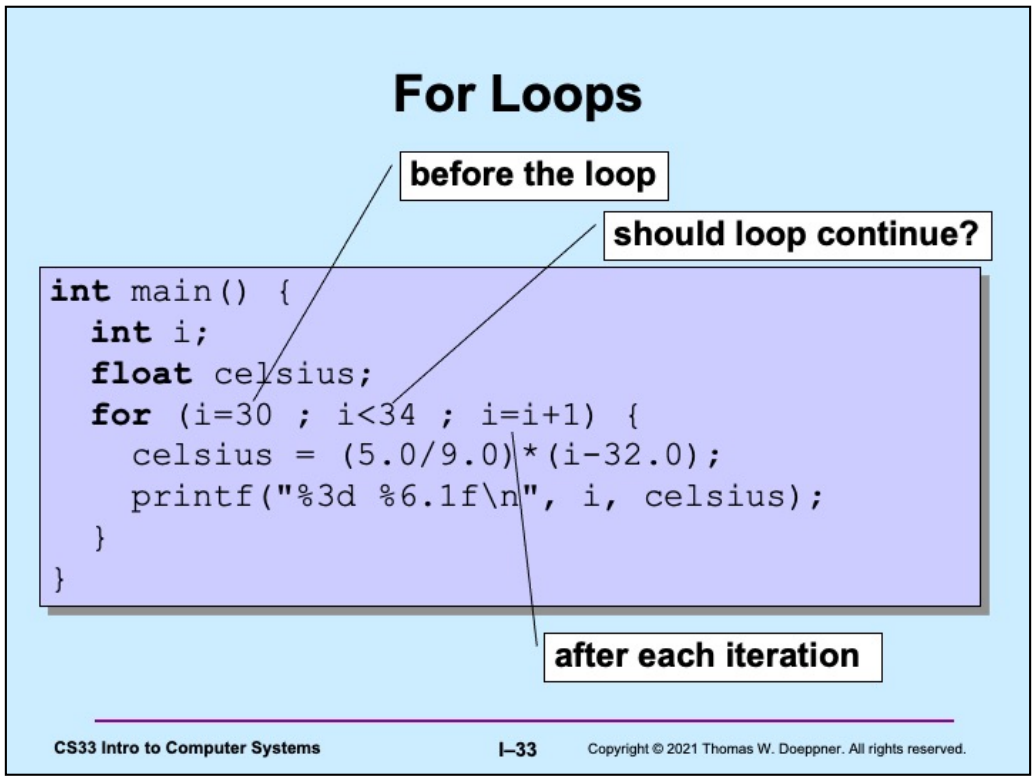
Formatting instructions

- **%6d: decimal integer 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**

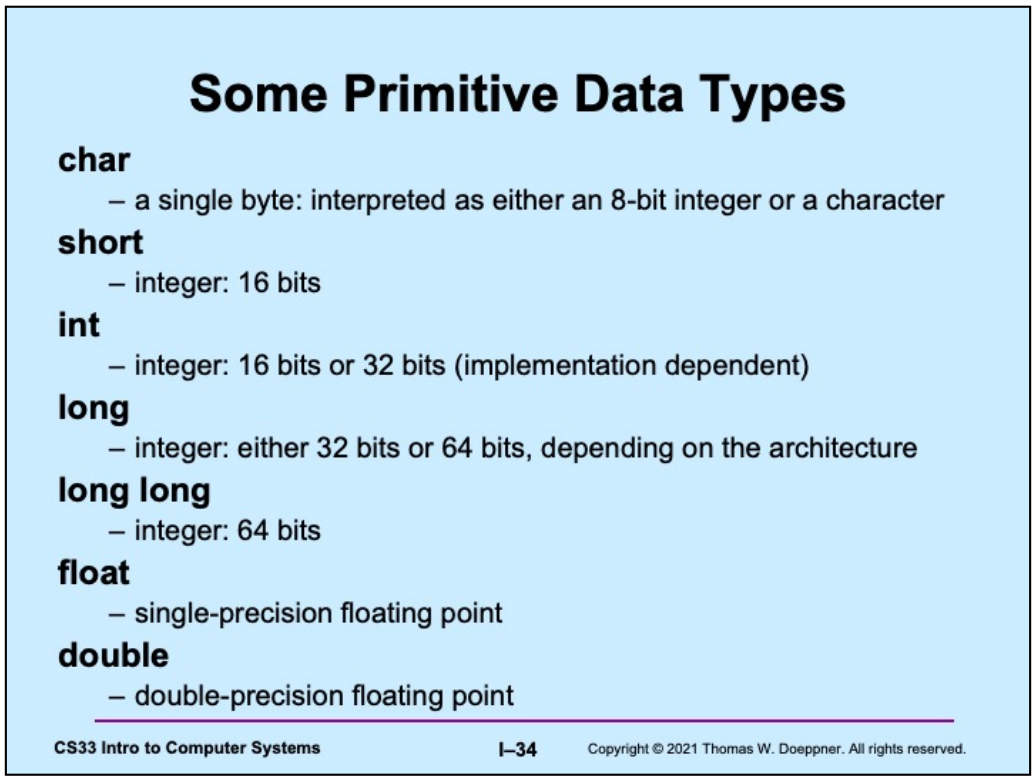
printf Again

```
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
```



Note that the "should loop continue" test is done at the beginning of each execution of the loop. Thus, if in the slide the test were "i<30", there would be no executions of the body of the loop and nothing would be printed.



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. The **int** type has a size of 32 bits on pretty much all of today's computers.

For the sunlab computers (and probably your own computer), a **long** is 64 bits.

What is the size of my int?

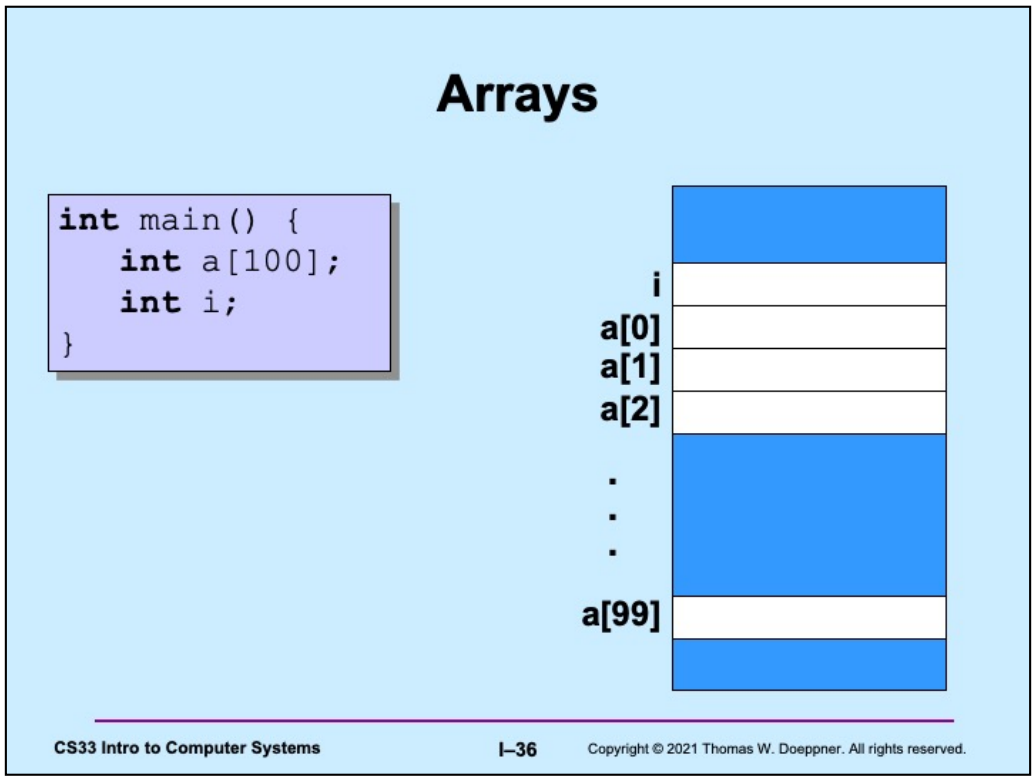
```
int main() {
  int i;
  printf("%d\n", sizeof(i));
}
```

```
$ ./a.out
4
```

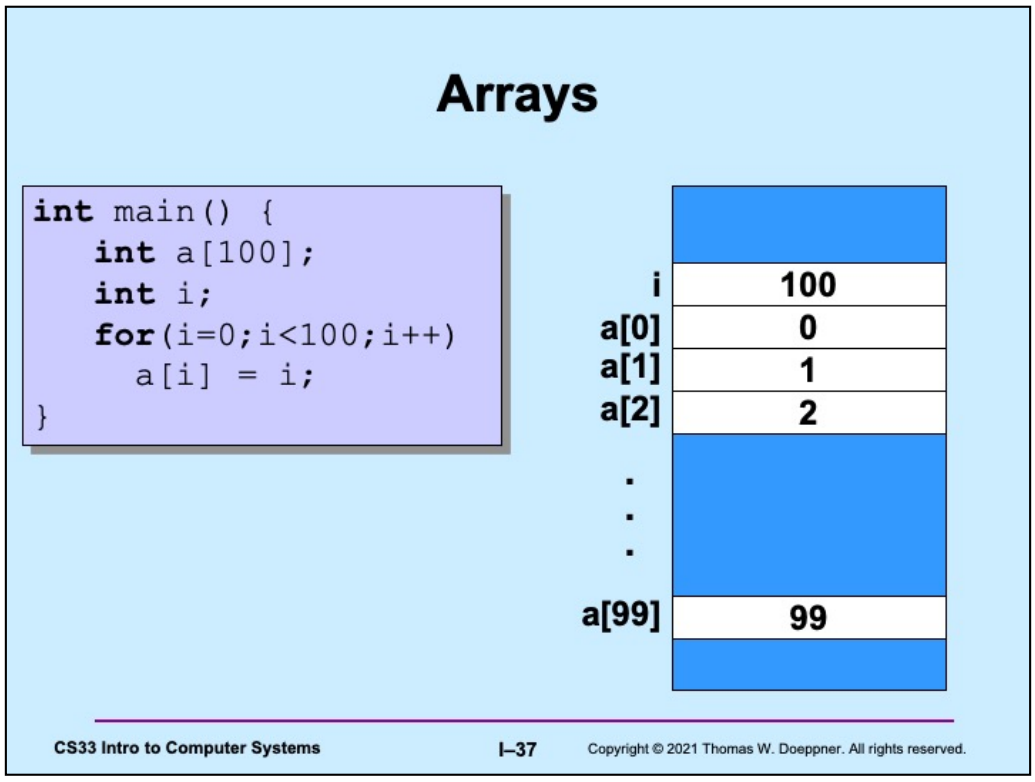
sizeof

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

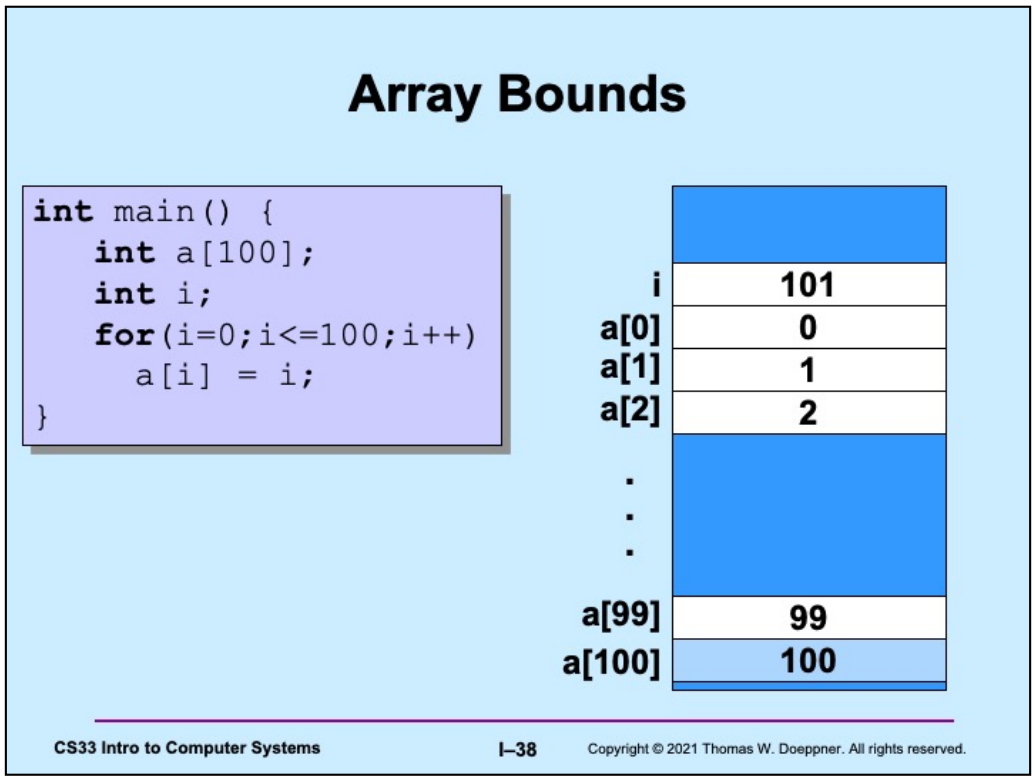
Note that the argument to **sizeof** need not be a variable but could be the name of a type. For example, “sizeof(int)” is legal and returns 4 on most machines.



The array *a* and the variable *i* really are arranged in memory as shown, assuming that "higher" memory addresses are at the bottom of the diagram and "lower" memory addresses are at the top. We draw it this way because this is how one normally draws pictures of memory. However, later in the course we will reverse this and arrange our memory diagrams so that higher addresses are at the top and lower addresses are at the bottom.



After executing the program, memory should contain what's shown in the diagram.



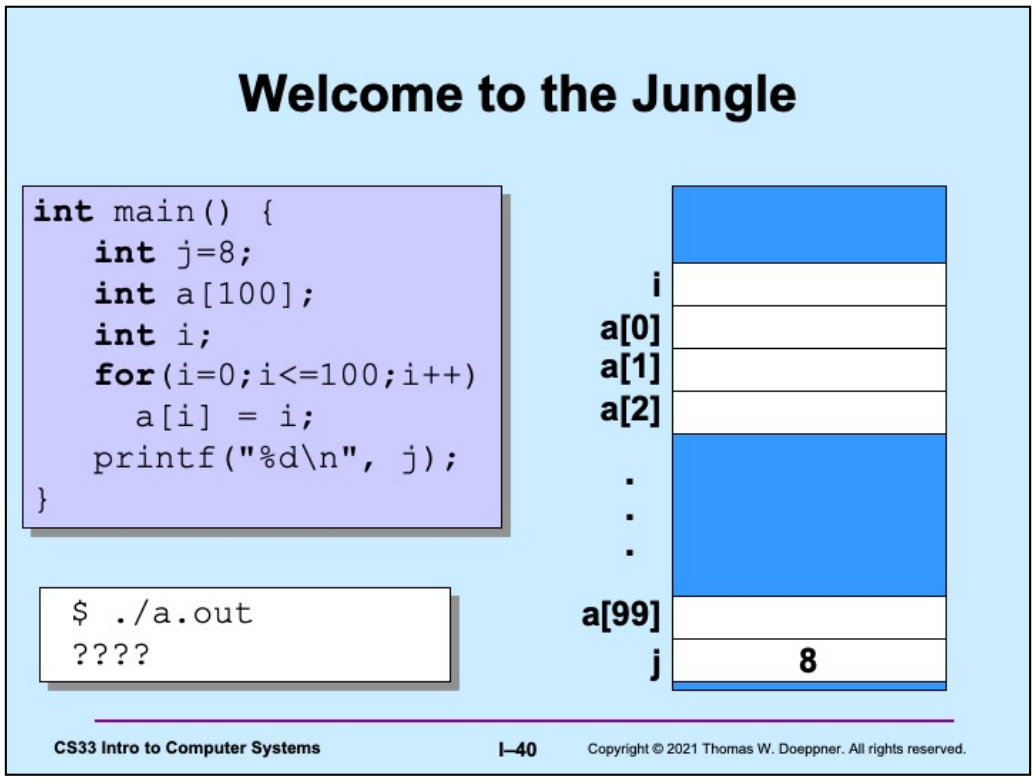
Here the for loop goes one element too far, storing 100 into a[100], despite the fact that we didn't declare the array to be that large.

Arrays in C

C Arrays = Storage + Indexing

- no bounds checking
- no initialization

WELCOME TO THE JUNGLE

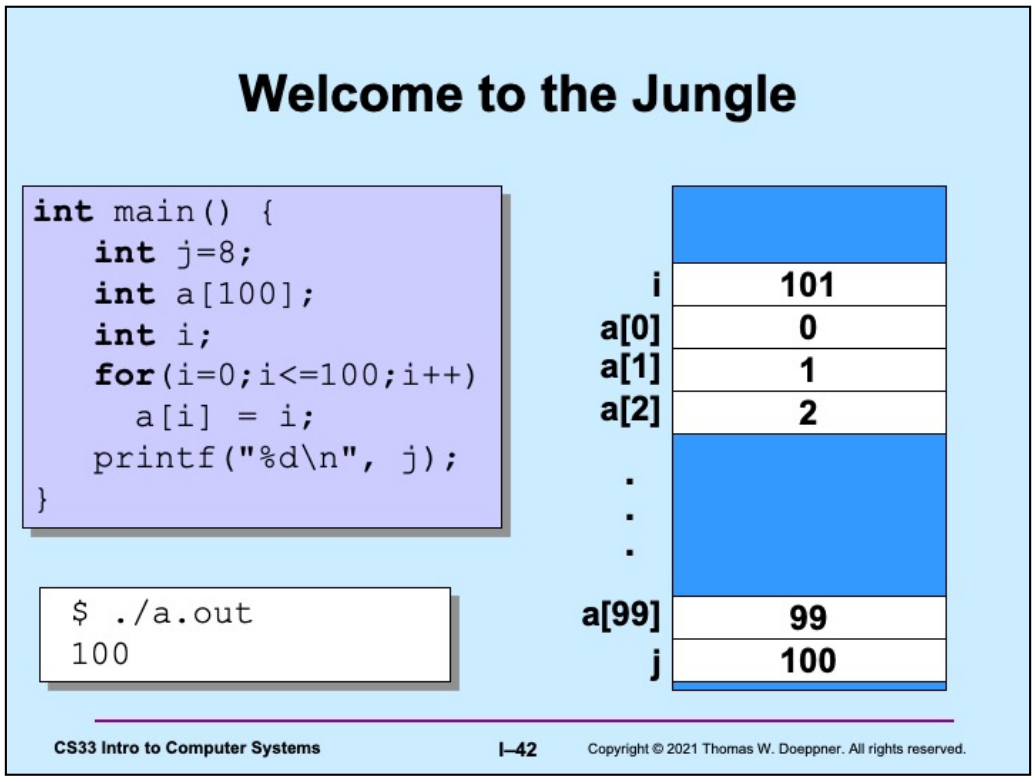


Note how j is both declared and initialized in the same statement.

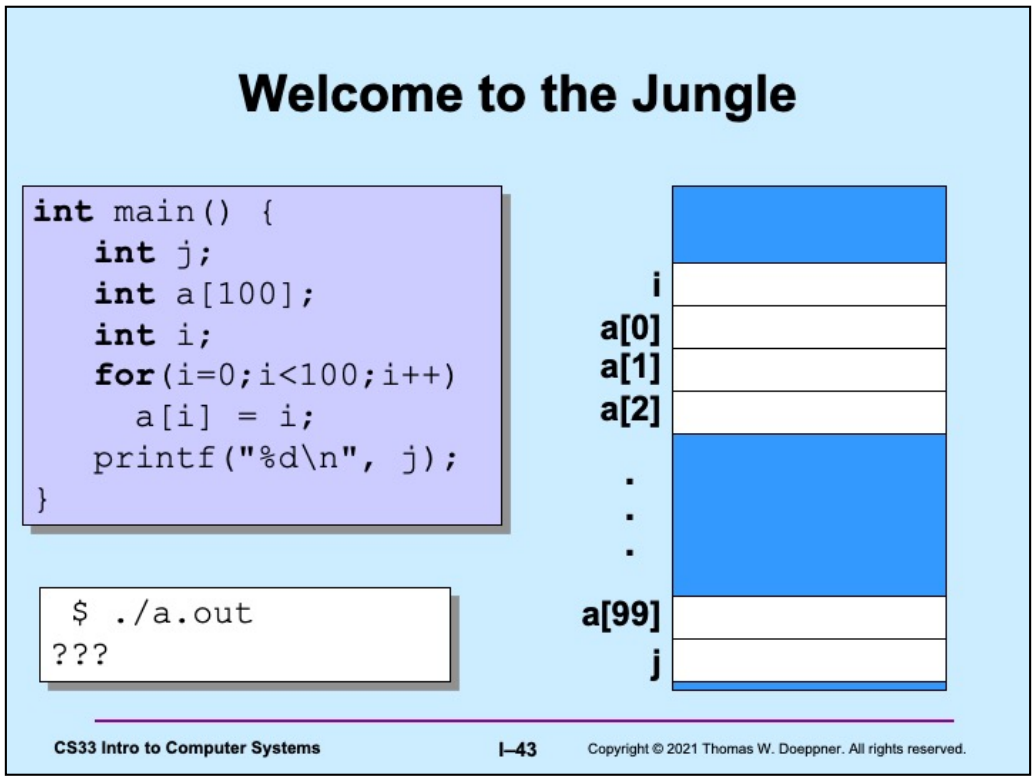
Quiz 1

- What is printed for the value of j when the program is run?
  a) 0
  b) 8
  c) 100
  d) indeterminate

This quiz doesn't count towards your grade!



Note that **j** occupies memory where **a[100]** would be if a were declared to be that large. Thus, j's location is overwritten when the program goes beyond the bounds for *a*.



Note that **j** no longer has an initial value. Note also that the for loop ends just after setting **a[99]** to 99.

Quiz 2

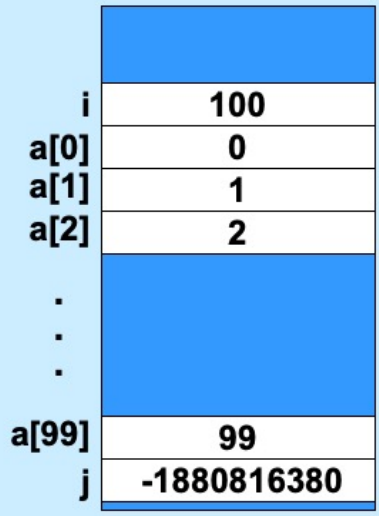
- What is printed for the value of j when the program is run?
  a) 0
  b) 8
  c) 100
  d) indeterminate

This quiz also doesn't count towards your grade.

Welcome to the Jungle

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

```
$ ./a.out
-1880816380
```



Welcome to the Jungle

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

```
$ ./a.out
25
```


This code is not guaranteed to work!

Welcome to the Jungle

```
int main() {
  int a[100];
  int i;
  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 computer. 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 main() {
  printf("%d\n", fact(5));
  return 0;
}

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

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;
}
```

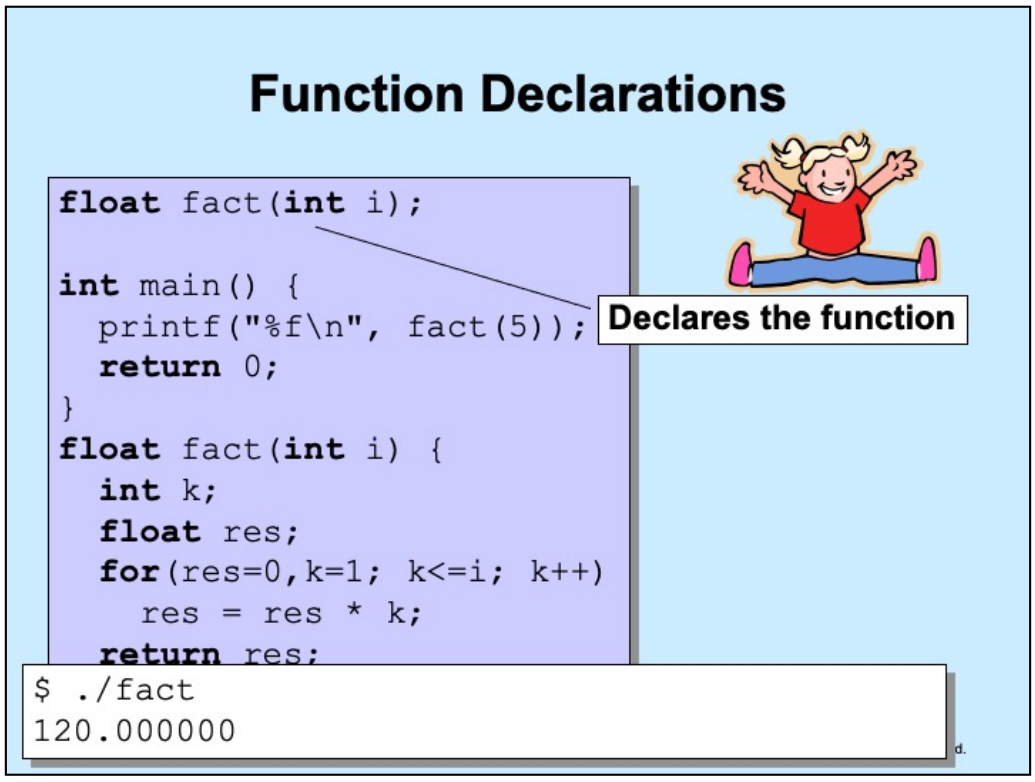

Not only has the definition of **main** been placed before the definition of **fact**, but also **fact** has been changed so that it now returns a **float** rather than an **int**.

Function Definitions

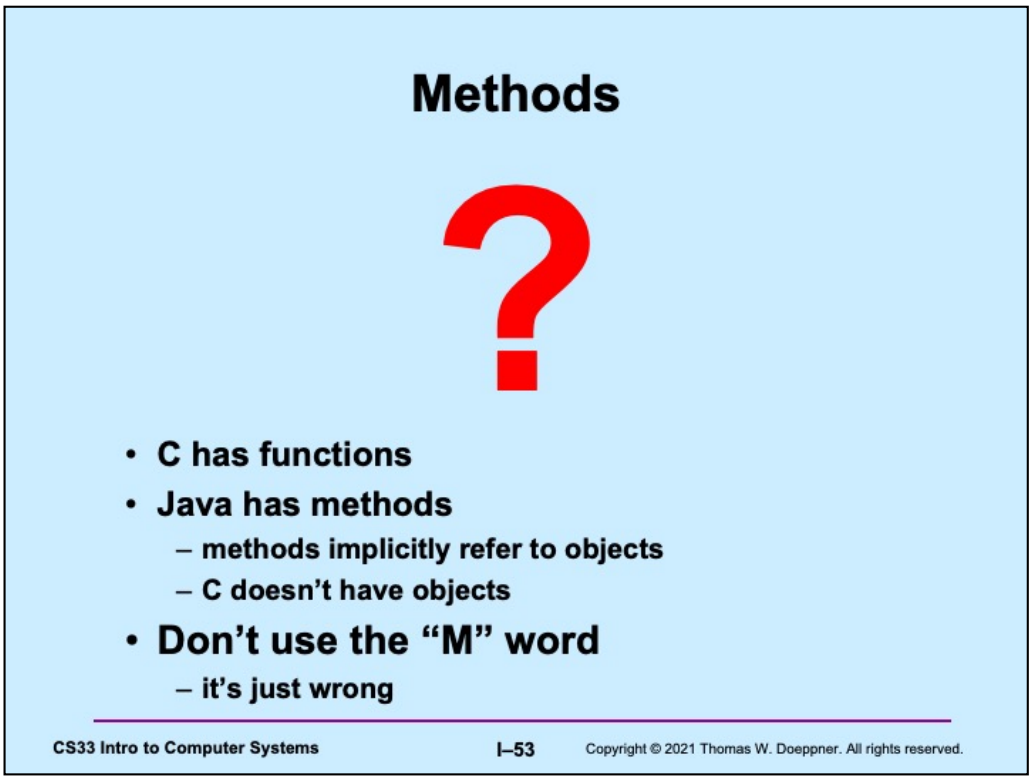
```
$ gcc -o fact fact.c
main.c:27: warning: type mismatch with previous implicit
declaration
main.c:23: warning: previous implicit declaration of
'fact'
main.c:27: warning: 'fact' was previously implicitly
declared to return 'int'
```

```
$  ./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.)



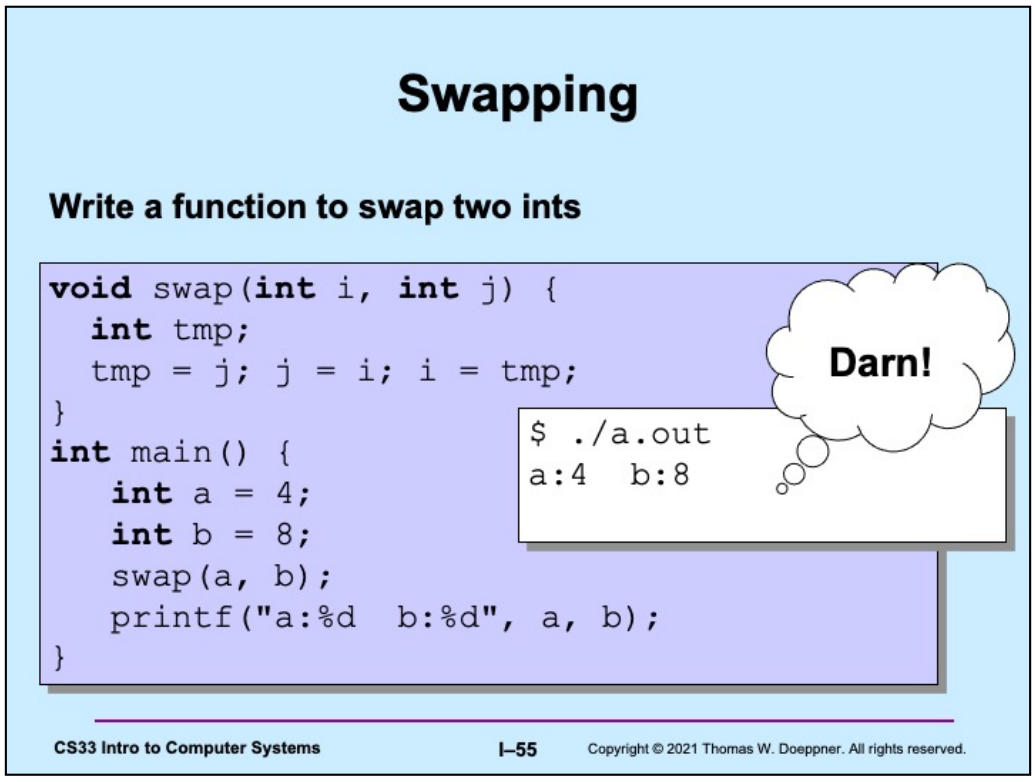

```
for (;;)
  printf("C does not have methods!\n");
```

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



This doesn't work because, when a function is called, copies are made of the arguments and it's these copies that are supplied to the function. Thus, if the function modifies its arguments, it's modifying only the copies. This is known as "pass by value".

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 the (ancient) Fortran programming language (by recognizing that **literals** such as "2" are special). Since C passes parameters by value, this has never been a problem in C.

Variables and Memory

What does

```
int x;
```

do?

- **It tells the compiler:**

  I want *x* to be the name of an area of memory that's big enough to hold an *int*.

What's memory?

We'll discuss "what's an int" in a couple weeks.

Industry Partners Program (IPP)

- Find and apply for jobs and internships in CS
- Learn about IPP member companies via tech talks
- Attend resumé reviews with industry professionals

- cs.brown.edu/about/partners

- To sign up for notifications about upcoming events:
  - http://bit.ly/brownipp

- Questions? Contact Lauren_Clarke@brown.edu