CSCI 0330/1330

Introduction to Computer Systems

Welcome!

- **Prof: Tom Doeppner**
- **HTAs: Naafi Ahmed, Nathan Benavides-Luu, Ed Bielawa, Vivian Lu**
- **UTAs: Siddharth Diwan, Jeremy Fleming, Michael Fu, Jamie Gabbay, Nathan Harbison, Jakobi Haskell, Nathan Nguyen, Patrick Peng, Seth Sabar, Anton Tarazi, Mikayla Walsh, Navaiya Williams, Matthias Yee, Camille Zhang**

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 or CSCI 0200

What You'll Do

- Nine 2-hour labs
- Eight one- to two-week programming assignments
  - one-on-one code review with a TA for each
- No written exams!
- Top Hat for in-class quizzes (sections 1 only)
  - not anonymous: a small portion of your grade
  - full credit (A) for each correct answer
  - partial credit (B) for each wrong answer
  - NC for not answering
  - one to three or so questions per class

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, 9/11
    - » 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

Collaboration Policy

- Goal is to learn from doing the assignments
- You may:
  - work with others in the design of your projects
  - help one another debug
- You may not:
  - use code from other sources (including AI tools)
- We run MOSS when relevant
- Details are here

Collaborative Hours

- TA hours are <u>collaborative</u>
  - TAs will work with you and connect you with other students with similar issues
    - » you may work out solutions with others
  - your code should be your own, but you may discuss it with others

Code Reviews

- After each project, you will meet with a TA for a code review – TA will ask you questions about your code
  - most (randomly selected) students will get just one question
  - others will get a lot of questions
  - 10-15 minutes per project
- Code reviews are easy and fun for those who did the assignment completely on their own
- They could be rather difficult for others

Textbook

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

Java™

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!

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

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

CS 33

Introduction to C

A C Program

```
int main( ) {
  printf("Hello world!\n");
  return 0;
}
```

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!
$
```

What’s gcc?

- gnu C compiler
  - it’s actually a two-part script
    - » part one compiles files containing programs written in C (and certain other languages) into binary machine code (known as object code)
    - » part two takes the just-compiled object code and combines it with other object code from libraries to create an executable
      - the executable can be loaded into memory and run by the computer

gcc Flags

- **gcc [-Wall] [-g] [-std=gnu99]**
  - **-Wall**
    - » provide warnings about pretty much everything that might conceivably be objectionable
  - **-g**
    - » provide extra information in the object code, so that gdb (gnu debugger) can provide more informative debugging info
      - discussed in lab
  - **-std=gnu99**
    - » use the 1999 version of C syntax, rather than the 1990 version

Declarations in C

```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

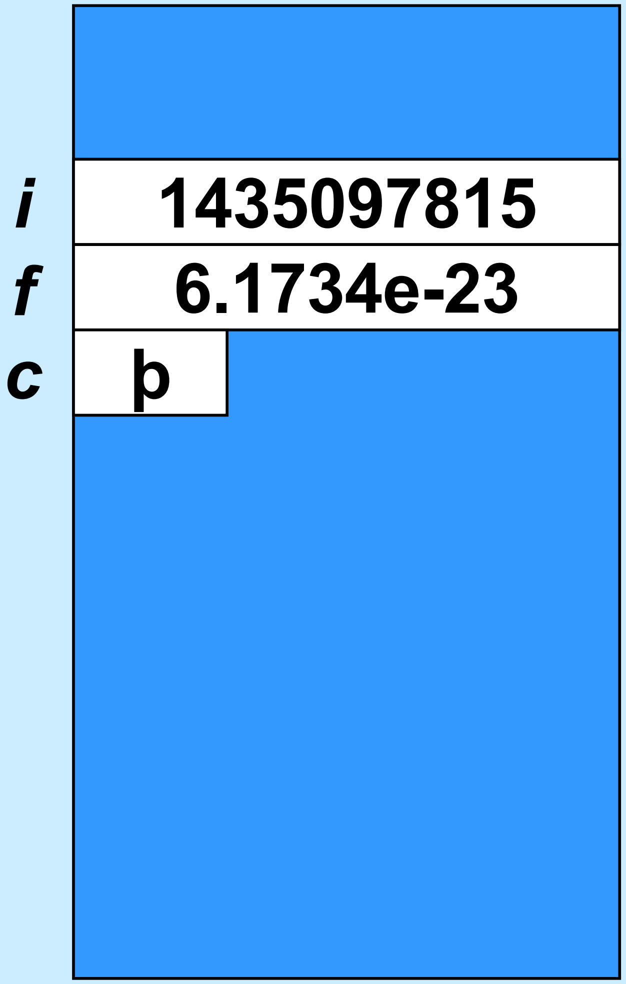
```
int main() {
 int i;
 float f;
 char c;
 return 0;
}
```

Declarations reserve memory space
- where?

Local variables can be uninitialized
- junk
- whatever was there before

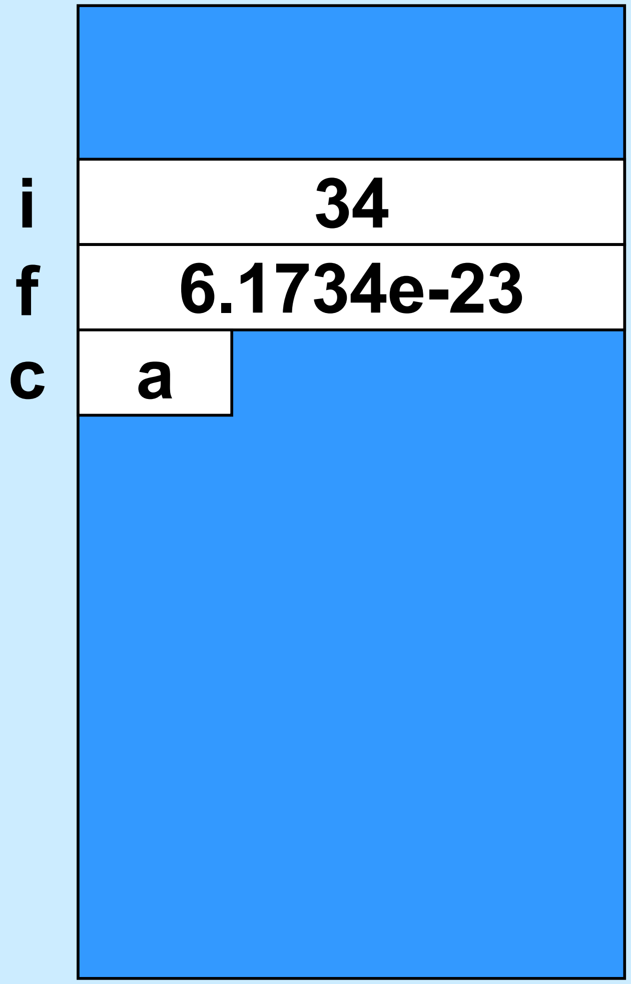
Declarations in C

```
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

```c
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
```

For Loops

before the loop

should loop continue?

```
int main() {
  int i;
  float celsius;
  for (i=30 ; i<34 ; i=i+1) {
    celsius = (5.0/9.0)*(i-32.0);
    printf("%3d %6.1f\n", i, celsius);
  }
}
```

after each iteration

Some Primitive Data Types

char

– a single byte: interpreted as either an 8-bit integer or a character

short

– integer: 16 bits

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

float

– single-precision floating point

double

– double-precision floating point

What is the size of my int?

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

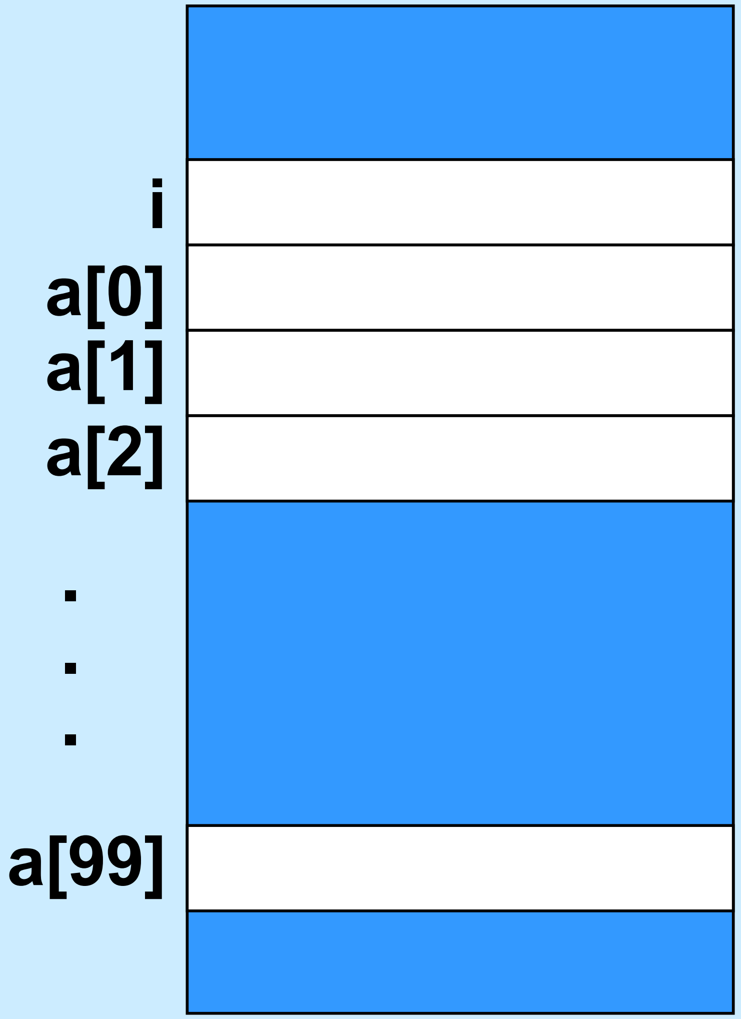
```
$ ./a.out
4
```

`sizeof`

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

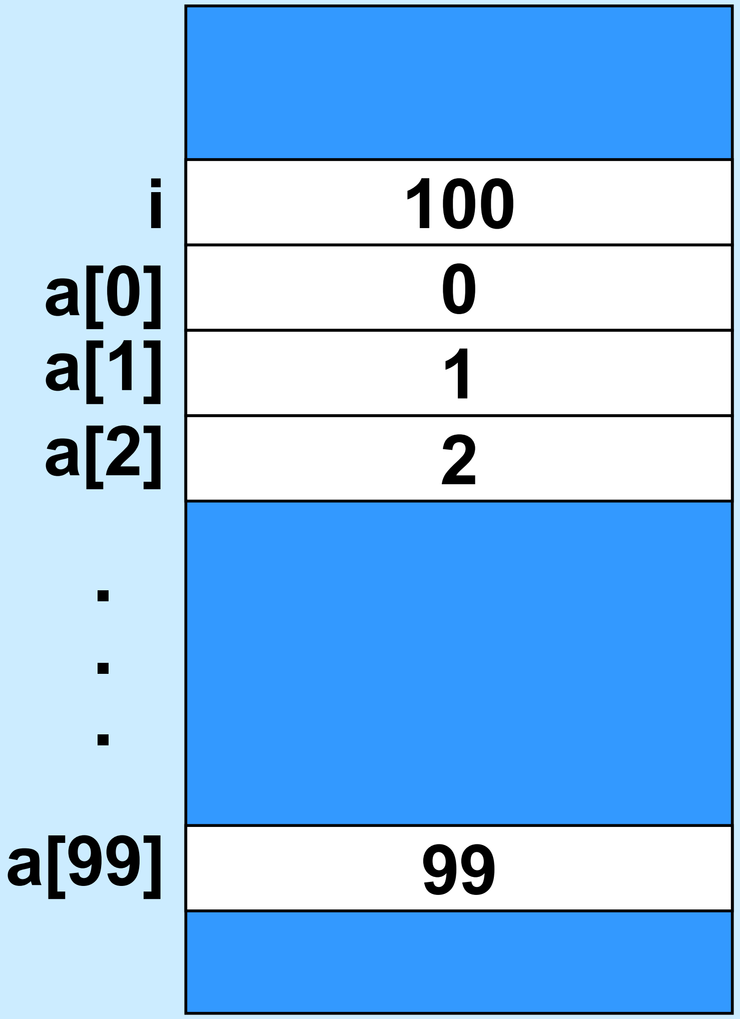
Arrays

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



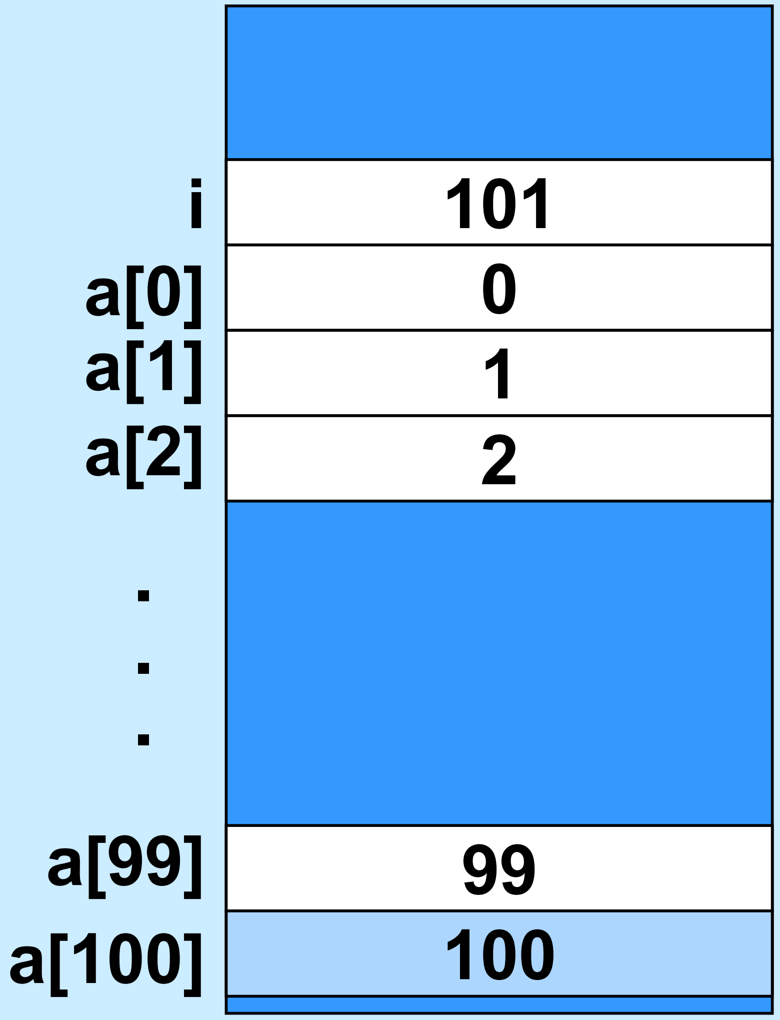
Arrays

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



Array Bounds

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



Arrays in C

C Arrays = Storage + Indexing

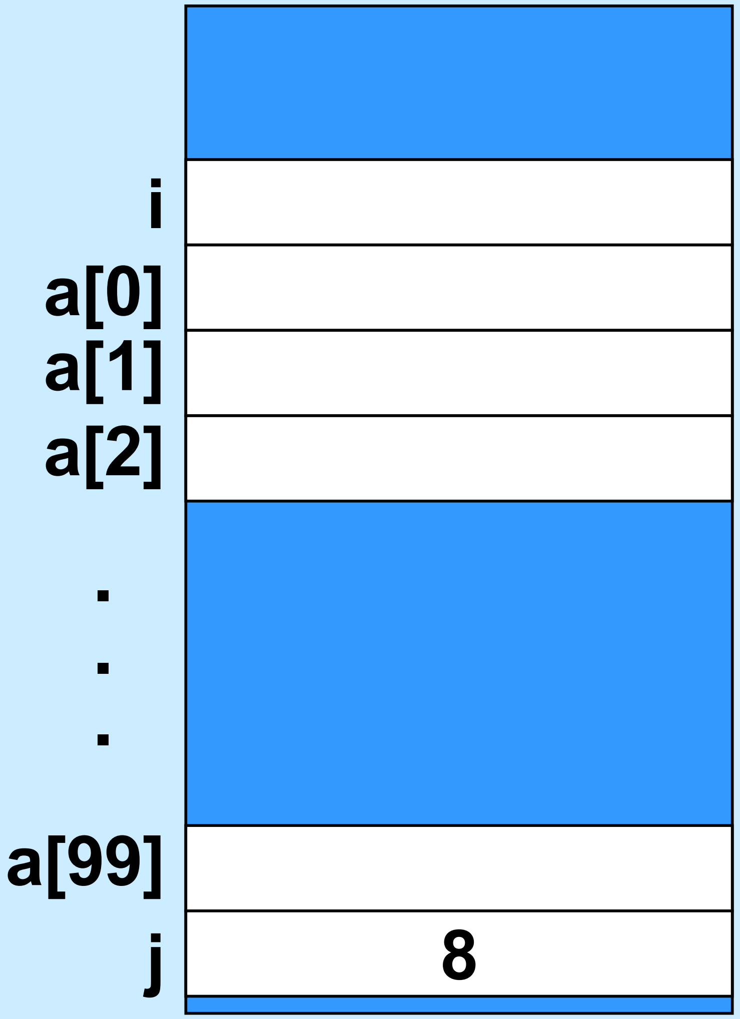
- no bounds checking
- no initialization

WELCOME TO THE JUNGLE

Welcome to the Jungle

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

```
 $ ./a.out
 ????
```



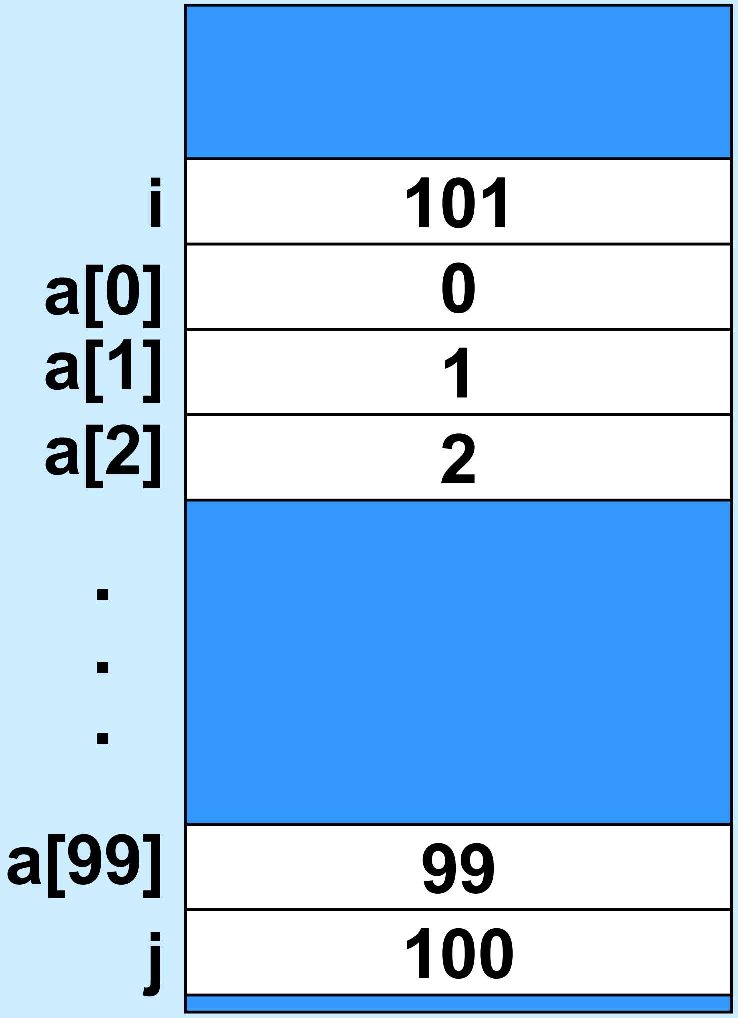
Quiz 1

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

Welcome to the Jungle

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

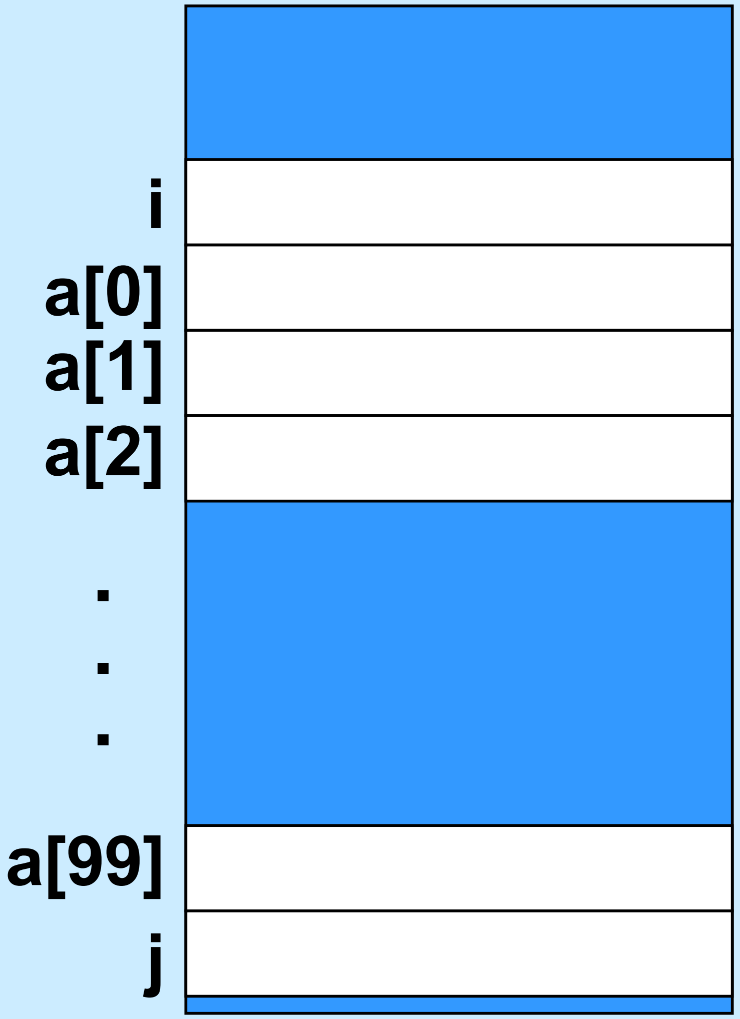
```
 $ ./a.out
 100
```



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
???
```



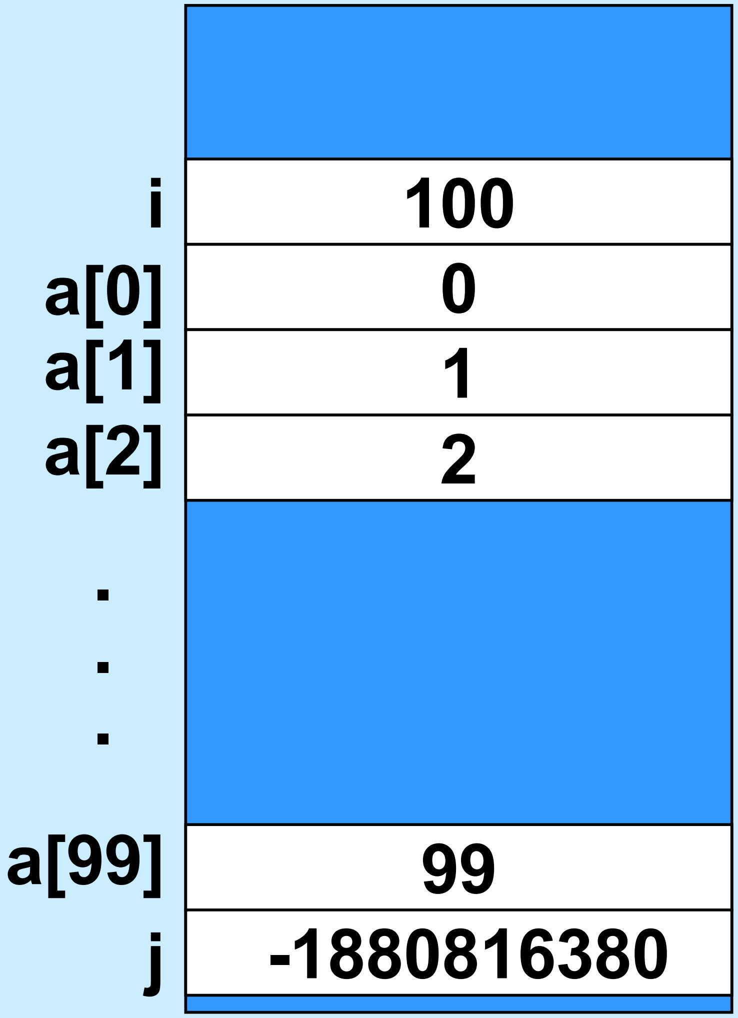
Quiz 2

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

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

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

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

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**

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**

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'
```

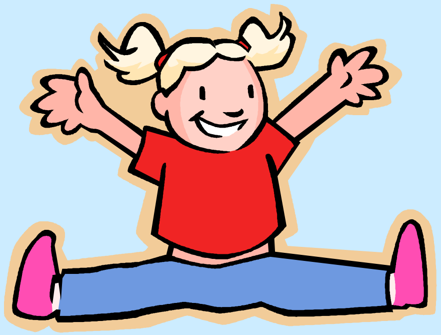
```
$  ./fact
1079902208
```

Function Declarations

```
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



```
$ ./fact
120.000000
```

d.

Methods

?

- C has functions
- Java has methods
  - methods implicitly refer to objects
  - C doesn't have objects
- Don't use the "M" word
  - it's just wrong

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);
}
```

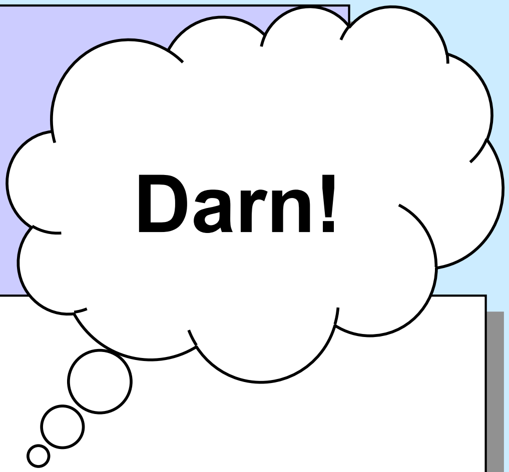
Parameters are passed by value

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
```

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?

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

- https://cs.brown.edu/about/partners

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

- Questions? Contact Lauren_Clarke@brown.edu