CSE 303
Lecture 9

C programming:
types, functions, and arrays

reading: *Programming in C* Ch. 4, 7-8

slides created by Marty Stepp
http://www.cs.washington.edu/303/
Lecture summary

- primitive data types: integers, real numbers, characters, Boolean
- functions
- arrays
- strings (briefly)
## Primitive numeric types

- **integer types:** char (1B), short (2B), int (4B), long (8B)
- **real numbers:** float (4B), double (8B)
- **modifiers:** short, long, signed, unsigned (non-negative)

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
<th>range of values</th>
<th>printf</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>0 to 255</td>
<td>%c</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>-32,768 to 32,767</td>
<td>%hi</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>2</td>
<td>0 to 65,535</td>
<td>%hu</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>-2,147,483,648 to 2,147,483,647</td>
<td>%d, %i</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>0 to 4,294,967,295</td>
<td>%u</td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>-9e18 to 9e18 - 1</td>
<td>%lli</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>approx. $10^{-45}$ to $10^{38}$</td>
<td>%f</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>approx. $10^{-324}$ to $10^{308}$</td>
<td>%lf</td>
</tr>
<tr>
<td>long double</td>
<td>12</td>
<td>Seattle temperature to Marty's IQ</td>
<td>%Lf</td>
</tr>
</tbody>
</table>
const variables

const type name = expression;

- declares a variable whose value cannot be changed

- Example:
  const double MAX_GPA = 4.0;
  
  ... 
  MAX_GPA = 4.5; // grade inflation! (error)

- The compiler will issue this warning:
  warning: assignment of read-only variable 'MAX_GPA'
Boolean type

```c
#include <stdbool.h>
...
bool b = false;
```

- C doesn't actually have a Boolean type (anything can be a test)
- including `stdbool.h` gives a pseudo-Boolean type `bool` (C99)
  - `false` is really a macro alias for 0
  - `true` is really a macro alias for 1

- what's wrong with the following statements?
  ```c
  if (x < y == true) {
      ...
  }
  bool b2 = x < 10;
  ```
Quintessential C bug

• What is wrong with this code?

```c
int x;
printf("Please type your age: ");
scanf("%d", &x);
if (x = 21) {
    printf("You came of drinking age this year!\n");
}
```
Defining a function

```java
returnType name(type name, ..., type name) {
    statements;
}

Example:

int sumTo(int max) {
    int sum = 0;
    int i;
    for (i = 1; i <= max; i++) {
        sum += i;
    }
    return sum;
}
```
Problem: function ordering

• You cannot call a function that has not been declared yet:

```c
int main(void) {
    int sum = sumTo(100);
    printf("The sum is %i\n", sum);
    return 0;
}
```

// sumTo is not declared until here
int sumTo(int max) {
    ...
}

• Solution: Reverse the order of function definition, or ...
Function declarations

```
returnType name(type name, ..., type name);
```

- declares (but does not define) a function, so it can be called

```c
int sumTo(int max);

int main(void) {
    int sum = sumTo(100);
    printf("The sum is %i\n", sum);
    return 0;
}

int sumTo(int max) {
    ...
}
```
More about declarations

```
returnType name(type, ..., type);
```

- don't need to list the parameter names; just types is sufficient

```
int sumTo(int);

int main(void) {
    int sum = sumTo(100);
    printf("The sum is %i\n", sum);
    return 0;
}

int sumTo(int max) {
    ...
}
```
Global vs. local variables

- global variables declared outside main can be seen by all code
- their use should be minimized (favor parameters/return instead)

```c
int x = 0;

void f(void) {
    x += 5;
}

int main(void) {
    x += 10;
    f();
    printf("x is %i\n", x);
    return 0;
}
```
Arrays

\textit{type name[size];}

Example:
\texttt{int scores[100];}

- the above statement allocates 100 ints' worth of memory
  - do not need to say new \texttt{int[100]} like in Java
  - initially each element of the array contains garbage data

- C arrays do not know their size
  - can call \texttt{sizeof(scores)}, but this is unreliable in many situations
  - only some recent versions of C allow an array's size to be a variable!:
    \begin{verbatim}
    int n = 20;
    int scores[n];   // works in C99 only
    \end{verbatim}
Array usage

type name[size] = {value, value, ..., value};

- allocates an array and fills it with pre-defined element values
- if fewer values are given than the size, the rest are filled with 0

name[index] = expression; // set an element

Example:
int primes[6] = {2, 3, 5, 6, 11, 13};
primes[3] = 7;

int allZeros[1000] = {0}; // 1000 zeros
Multi-dimensional arrays

\[
\text{type name}[\text{rows}][\text{columns}];
\]

- creates a two-dimensional array of given sizes, full of garbage data

\[
\text{type name}[\text{rows}][\text{columns}] = \{\{\text{values}\}, \ldots, \{\text{values}\}\};
\]

- allocates a 2D array and fills it with pre-defined element values

Example:
```
int grid[10][10];
int matrix[3][5] = {
    {10,  5, -3, 17, 82},
    { 9,  0,  0,  8, -7},
    {32, 20,  1,  0, 14}
};
```
Exercise

• Write a complete C program that outputs the first 16 Fibonacci numbers in reverse order, 8 numbers per line, 6 spaces per number.

987   610   377   233   144    89    55    34
   21     13     8      5      3      2      1      1
Arrays as parameters

• It is more difficult to use arrays as parameters/return than in Java.
  - arrays do not know their own size; they are just memory chunks

```c
int sumAll(int a[]);

int main(void) {
    int numbers[5] = {7, 4, 3, 15, 2};
    int sum = sumAll(numbers);
    return 0;
}

int sumAll(int a[]) {
    int i, sum = 0;
    for (i = 0; i < ... ???
```
Solution 1: declare size

- you can declare a function with the array's exact size
  - drawback: code is not flexible

```c
int sumAll(int a[5]);

int main(void) {
    int numbers[5] = {7, 4, 3, 15, 2};
    int sum = sumAll(numbers);
    return 0;
}

int sumAll(int a[5]) {
    int i, sum = 0;
    for (i = 0; i < 5; i++) {
        sum += i;
    }
    return sum;
}
```
Solution 2: pass size

- you can pass the array's size as a parameter

```c
int sumAll(int a[], int size);

int main(void) {
    int numbers[5] = {7, 4, 3, 15, 2};
    int sum = sumAll(numbers, 5);
    return 0;
}

int sumAll(int a[], int size) {
    int i, sum = 0;
    for (i = 0; i < size; i++) {
        sum += i;
    }
    return sum;
}
```
Returning an array

• arrays (as we have seen them) disappear at the end of the function
  ▪ this means they cannot be safely returned from a method

```c
int[] copy(int a[], int size);

int main(void) {
    int numbers[5] = {7, 4, 3, 15, 2};
    int numbers2[5] = copy(numbers, 5);  // no
    return 0;
}

int[] copy(int a[], int size) {
    int i;
    int a2[size];
    for (i = 0; i < size; i++) {
        a2[i] = a[i];
    }
    return a2;  // no
}
```
Solution: output parameter

• workaround: create the return array outside and pass it in
  ▪ "output parameter" works because arrays are passed by reference

```c
void copy(int a[], int a2[], int size);

int main(void) {
    int numbers[5] = {7, 4, 3, 15, 2};
    int numbers2[5];
    copy(numbers, numbers2, 5);
    return 0;
}

void copy(int a[], int a2[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        a2[i] = a[i];
    }
}
```
A bit about strings

• string literals are the same as in Java
  ```c
  printf("Hello, world!\n");
  ```
  but there is not actually a String type in C; they are just `char[]`

• strings cannot be made, concatenated, or examined as in Java:
  ```c
  String s = "hello";
  // no
  int answer = 42;
  printf("The answer is " + answer);
  // no
  int len = "hello".length();
  // no
  int printMessage(String s, int times) { ... // no
  ```

• Next week we will see how to create and manipulate strings.
Exercise

• Modify our previous program to prompt the user twice for a number and print that many Fibonacci numbers in reverse order, 8 numbers per line, 6 spaces per number.

How many Fibonacci numbers? 16
987  610  377  233  144  89  55  34
21  13   8   5   3   2   1   1

How many Fibonacci numbers? 10
55  34  21  13  8   5   3   2
1   1