C:
a grade or mark, as in school or college, indicating the quality of a student's work as fair or average.

Dictionary.com, "c," in *The American Heritage® Abbreviations Dictionary, Third Edition*. Source location: Houghton Mifflin Company, 2005. http://dictionary.reference.com/browse/c. Available: http://dictionary.reference.com. Accessed: October 19, 2009.

David Notkin ● Autumn 2009 ● CSE303 Lecture 9

Today

- Some C leftovers from Friday
- Primitive data types: integers, real numbers, characters, Boolean
- Functions
- Arrays
- Strings (briefly)

Mostly the same as Java

- Variables
  - can be used without being initialized (!)
  - must be declared at the start of a function or block (changed in C99)
- **`for`** loops
  - variable cannot be declared in the loop header
- **`if/else`** statements, **`while`** and **`do/while`** loops
  - there is no **`boolean`** type (changed in C99)
  - any type of value can be used as a test
  - 0 means false, every other number means true
- Parameters / returns
  - C has certain features for values vs. references ("pointers")

Very different from Java

- Strings
  - very clunky to use in C; arrays of characters
  - are not objects; do not contain methods (external string functions)
- I/O to/from console and files
  - no Scanner; must use input functions such as scanf
  - console I/O different than file I/O
- Errors and exceptions
  - C has no try/catch and does not represent errors as objects
  - errors are usually returned as integer error codes from functions
  - crashes are mostly called "segmentation faults" and are not of much direct utility in figuring out what is wrong

Also very different

- Arrays
  - are just bare contiguous blocks of memory
  - have no methods and do not know their own length (!)
- Objects
  - C doesn't have them
  - closest similar feature: struct (a set of fields; no methods)
- Memory management
  - most memory that you consume, you must explicitly free afterward
- API and provided libraries
  - C doesn't have very many, compared to Java
  - you must write many things yourself (even data structures)

printf continued

- A placeholder can specify the parameter's width or precision:
  - **`%8d`** an integer, 8 characters wide, right-aligned
  - **`%-8d`** an integer, 8 characters wide, left-aligned
  - **`%.4f`** a real number, 4 digits after decimal
  - **`%6.2f`** a real number, 6 total characters wide, 2 after decimal
- Examples:

```
int age = 45;
double gpa = 1.2345678;
printf("%8d %7.3f\n", age, gpa);
printf("%8.2f %.1f %10.5f", gpa, gpa, gpa);
```

scanf

function	description
`scanf`	reads formatted input from console

- `scanf("format string", variables);`
- uses same syntax for formatted strings, placeholders as `printf`
- Must precede each variable with an `&` (address-of operator)

```
int x;
int y;
printf("Type your x and y values: ");
scanf("%d %d", &x, &y);
```

scanf continued

- `scanf` returns the number of values successfully read: can be examined to see whether the reading was successful
- if # of variables listed doesn't match # of format placeholders
  - too many variables: later ones ignored
  - too few variables: program crashes!

Practice exercise [if you want]

- Write a C program that makes change:
  - prompts the user for an amount of money
  - reports the number of pennies, nickels, dimes, quarters, and dollars
- Example

```
Amount of money? 17.93
Pennies :   2
Nickels :   1
Dimes   :   1
Quarters:   3
Dollars :  17
```

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)

type	bytes	range of values	printf
`char`	1	0 to 255	%c
`short int`	2	-32,768 to 32,767	%hi
`unsigned short int`	2	0 to 65,535	%hu
`int`	4	-2,147,483,648 to 2,147,483,647	%d, %i
`unsigned int`	4	0 to 4,294,967,295	%u
`long long int`	8	-9e18 to 9e18 - 1	%lli
`float`	4	approx. 10^{-45} to 10^{38}	%f
`double`	8	approx. 10^{-324} to 10^{308}	%lf
`long double`	12	A lot!	%Lf

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

```
#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

Anything wrong here

```
if (x < y == true) {
    ...
}
bool b2 = x < 10;
```


Quintessential C bug

```
int x;
printf("Please type your age: ");
scanf("%d", &x);
if (x = 18) {
    printf("You can now vote!\n");
}
```

Defining a function

```
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 (defined) yet

```
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 ...

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

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

int allZeros[1000] = {0};      // 1000 zeros
```

Multi-dimensional arrays

- `type name[rows][columns];`
  - creates a two-dimensional array of given sizes, full of garbage data
- `type name[rows][columns] = {{values}, ..., {values}};`
  - allocates a 2D array and fills it with pre-defined element values

```
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

- Arrays do not know their own size; they are just memory chunks – harder than in Java

```
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

- Declare a function with the array's exact size

```
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

- Pass the array's size as a parameter

```
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 (so far) disappear at the end of the function: this means they cannot be safely returned

```
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

```
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 (more soon)

- String literals are the same as in Java
  - `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:

```
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
```

Exercise

- Modify the 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
```

Questions?