CS240: From Java to C

Agenda

JAVA to C: Basic Syntax

Read/write to terminal

C types: Booleans, arrays (1D), strings, structs

Stack diagrams

Pointers

File I/O

Parsing with strtok

From Java to C

Syntax is very similar!

- if/else statements, loops (while/for), variables, functions
- {} for statement blocks
- ; for statements
- operators and operator precedence the same
  - +, -, *, /, +=, &&, ||, % etc!

Data types

- int (1, -5, 0)
- double, float (1.0, -4.7)
- char (‘a’, ‘$’, ‘\n’)

From Java to C

What is the output of this program?

```
int func(int a, int b) {
    a += 5;
    if (a > 10) return 0;
    return a - b;
}

int main() {
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    for (int i = 0; i < 3; i++) {
      printf("%d, %d\n", x, y);
    }

    return 0;
}
```

Table 2. C Numeric Types

Type name	Usual size	Values stored	How to declare
`char`	1 byte	integers	`char x;`
`short`	2 bytes	signed integers	`short x;`
`int`	4 bytes	signed integers	`int x;`
`long`	4 or 8 bytes	signed integers	`long x;`
`long long`	8 bytes	signed integers	`long long x;`
`float`	4 bytes	signed real numbers	`float x;`
`double`	8 bytes	signed real numbers	`double x;`

Java has these too…

Questions:

1. What is a byte?
2. Why might we want to use an `int` instead of a `long`, or `float` instead of `double`?
3. Why might we want to use an `long` instead of a `int`, or `double` instead of `float`?

printf function

- Same as System.out.printf():

Java: `System.out.printf("%d %s\t %f", 6, "hello", 3.4);`

C: `printf("%d %s\t %f\n", 6, "hello", 3.4);`

%d, %i	int
%f or %g	float or double
%c	char
%s	string
%lu	Long unsigned int
%x	Unsigned hexidecimal
\t \n	tab character, new line character

To add padding: %03d, %.2f

See: https://www.cplusplus.com/reference/cstdio/printf/

scanf

- For reading in values of different types
- Uses format string like printf
- The arguments are the memory locations into which the values will be stored (the address of program variables or base addr of arrays):

```
int x;
float y;
char s[100];

scanf(" %s%d%f", s, &x, &y);

// s is the base address of the string array
// &x is the address of the variable x in memory
// &y is the address of the variable y in memory
```


Slide by Tia Newhall, CS31, Swarthmore

scanf: Example

```
int main() {
  printf("Do you like jokes? (y/n): ");

  char response = 0;
  scanf("%c", &response);

  printf("You chose: %c\n", response);
  if (response == 'y') {
    printf("Yes, me too!\n");
  } else if (response == 'n') {
    printf("Me neither!\n");
  } else {
    printf("I don't get you.\n");
  }

  return 0;
}
```

What about Booleans?

No booleans in C

0 is **always false**

Every other value is **true**

Boolean: Examples

```
#include <stdio.h>

int isEven(int val) {
  if (val % 2 == 0) {
    return 1;
  }
  return 0;
}
```

```
int main() {

  int a = 0;
  int b = 7;

  if (isEven(a)) {
    printf("a is even\n");
  } else {
    printf("a is odd\n");
  }

  if (b) {
    printf("Any non-zero value is true!\n");
  }
  return 0;
}
```

Arrays

```
class Array {
  public static void main(String[] args) {
    int i, size = 0;
    int[] my_arr = new int[10];

    for (i = 0; i < 10; i++) {
        my_arr[i] = i;
        size++;
    }
   my_arr[3] = 100;

   System.out.printf("array of %d items:\n",
         size);

    for (i = 0; i < 10; i++) {
        System.out.printf("%d\n", my_arr[i]);
    }
  }
}
```

```
int main() {
    int i, size = 0;
    int my_arr[10];

    for (i = 0; i < 10; i++) {
        my_arr[i] = i;
        size++;
    }

    my_arr[3] = 100;
    printf("array of %d items:\n", size);

   for (i = 0; i < 10; i++) {
        printf("%d\n", my_arr[i]);
    }

    return 0;
}
```

Arrays

static allocation – the size of the array is set at compile time *and does not change at runtime*

dynamic allocation – the size of the array can be set (and changed) *while the program is running*

Arrays

- No bounds checking in C!

- Easy to write/read elements out of bounds of array
  - You may (or may) not see a problem depending on what memory you access…

Invalid Index: Example in C

```
#include <stdio.h>

int main() {
  int bad[4] = {1, 2, 3, 4};
  printf("bad[2] = %d\n", bad[2]);
  printf("bad[4] = %d\n", bad[4]);
  return 0;
}
```

```
WSL alinen@Xin:~/cs223/cs223-devel/chpt01$ ./a.out
bad[2] = 3
bad[4] = -1514561552
```

Invalid Index: Example in Java

```
class BadArray {
  public static void main(String[] args) {
    int[] bad = {1, 2, 3, 4};
    System.out.printf("bad[2] = %d\n", bad[2]);
    System.out.printf("bad[4] = %d\n", bad[4]);
  }
}
```

```
WSL alinen@Xin:~/cs223/cs223-devel/chpt01$ java BadArray
bad[2] = 3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4
out of bounds for length 4
        at BadArray.main(BadArray.java:5)
```

What about strings?

In C, a string is an **array of char**

const char* message = “This is the message”;

char greeting[5] = “ciao”;

Not every array of char is a string,
but every string is an array of char

IMPORTANT: strings in C MUST be large enough to hold # characters + 1

Last character is a null character that denotes the end of the string

Terminating character: `\0` (equivalent to zero)

C strings are null-terminated

Example: Draw the contents of memory of the variable greeting below

char greeting[5] = “ciao”;

C strings must be stored in arrays with sufficient size

Example: Which of the following declarations are safe?

```
char greeting[5] = “ciao”;

char greeting[5] = “Salutations”;

char greeting[5] = “hello”;

char greeting[5] = “Hi!”;

const char* greeting = “Salutations”;
```

C strings

```
include <stdio.h>
#include <string.h>

int main() {
  char greeting[5] = "ciao";
  const char* message = "This is the message";

  char buffer[10];
  buffer[0] = 'h';
  buffer[1] = 'i';
  buffer[2] = '\0';

  printf("%s\n", greeting);
  printf("%s\n", message);
  printf("%s\n", buffer);

  int len = strlen(message);
  printf("%d\n", len);

  strcpy(buffer, "test");
  printf("%s\n", buffer);
}
```

Pointers

A **pointer** stores an address in memory

Example:

```
int a; // integer
int* a; // a pointer to an integer
```

A **NULL pointer** represents an empty, or unset, address

Example:

```
int* a = NULL; // an empty pointer to an integer
if (a == NULL) printf("a is not safe to use.");
```

Passing arrays to functions

All parameters are passed by value in C , e.g. values are copied

BUT arrays are denoted using their base address, using a special variable type called a **pointer.** A **pointer** stores an address in memory.

When we **pass by pointer** the memory address of the variable is copied, not the variable's data.

```
void print(const char* str, int n) {
  for (int i = 0; i < n; i++) {
    printf("%c-", str[i]);
  }
}

int main() {
  int a = 5;
  const char* greeting = "hello";
  print(greeting, a);
  return 0;
}
```

What is the output of this program?

Structs

No classes in C!

If we want to bundle data together in C we use a **struct**

Classes hold data and define methods

"know stuff" and "do stuff"

In C, only **structs** hold data and only **functions** "do stuff"

Struct: Example

```
struct studentT {
   char name[64];
   int age;
   float gpa;
   int grad_yr;
};

 struct studentT student1, student2;

   strcpy(student1.name, "Kwame Salter");  // name field is a char array
    student1.age = 18 + 2;                // age field is an int
   student1.gpa = 3.5;                   // gpa field is a float
   student1.grad_yr = 2020;               // grad_yr field is an int

   /* Note: printf doesn't have a format placeholder for printing a
    * struct studentT (a type we defined).  Instead, we'll need to
    * individually pass each field to printf. */
   printf("name: %s age: %d gpa: %g, year: %d\n",
         student1.name, student1.age, student1.gpa, student1.grad_yr);
```

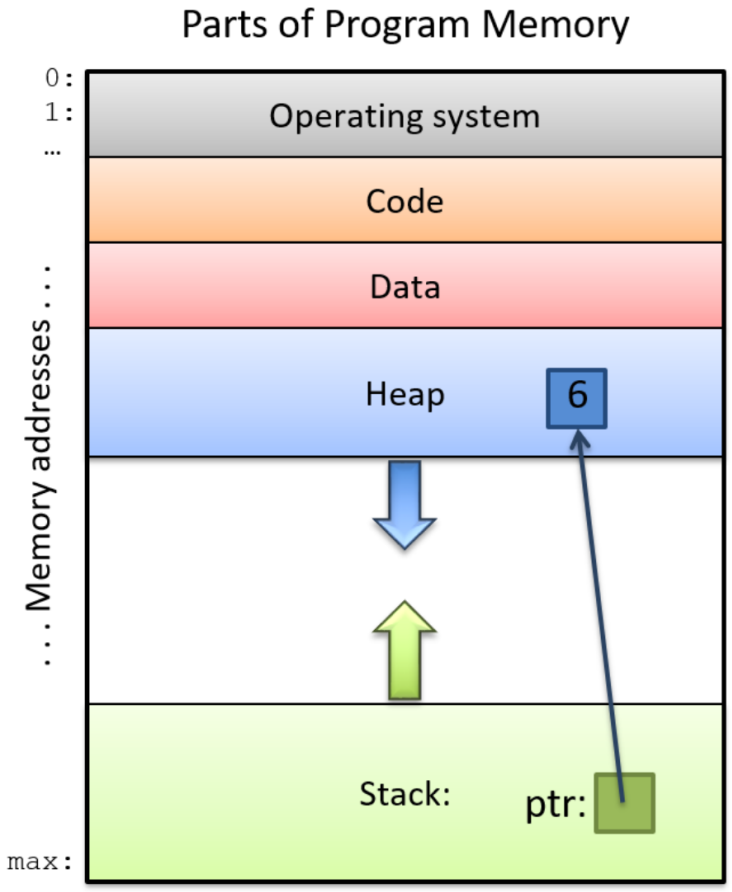
Function Stack

Execution stack – keeps track of active functions

Stack frame – created by each function

- contains parameters, local variables
- topmost frame is currently executing function
- frame is pushed to stack when function is called
- frame is popped off stack when it returns

Pass by value – values are *copied* into the frame's function parameters



Draw the stack diagram

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
    // draw the stack here
    return 0;
}
```

Draw the stack diagram

```
void print(const char* str,int n) {
  for (int i = 0; i < n; i++) {
    printf("%c-", str[i]);
  }
}
int main() {
  int a = 5;
  const char* greeting = "hello";
  print(greeting, a);
  return 0;
}
```

Draw the stack diagram

```
void print(char* str, int n) {
   for (int i = 0; i < n; i++) {
     greeting[i] = 'z';
  }
  // print stack here
}

int main() {
  char greeting[2];
  greeting[0] = 'a';
  greeting[1] = 'b';
  print(greeting, 2);
  return 0;
}
```

Example: Command line arguments

```
#include <stdio.h>

int main(int argc, char** argv)
{
  // Draw the stack here
  for (int i = 0; i < argc; i++)
  {
    printf("%d) %s\n", i, argv[i]);
  }
}
```

Draw the function stack for the following command:

$ ./a.out apple banana carrot

NOTE: We can also declare main like so: int main(int argc, char* argv[])