"If it weren't for C, we'd be writing programs in BASI, PASAL, and OBOL."

http://www.gdargaud.net/Humor/C_Prog_Debug.html

Today

- Some C leftovers from Monday
- C memory model; stack allocation
  - computer memory and addressing
  - stack vs. heap
  - pointers
  - parameter passing
    - by value
    - by reference

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

```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;
}
```

```c
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

Memory hierarchy

- CPU registers: a few bytes
- L1/L2 cache (on CPU): 1-4 MB
- Physical RAM (memory): 1-2 GB
- Virtual RAM (on a hard disk): 2-8 GB
- Secondary/permanent storage (hard disks, removable drives, network): 500 GB

Virtual addressing

• each process has its own virtual address space of memory to use
  – each process doesn’t have to worry about memory used by others
  – OS maps from each process’s virtual addresses to physical addresses

Process memory layout

- stack (function calls)
- available memory
- heap (dynamically allocated data)
- global/static variables ("data segment")
- code instructions ("text segment")

- when a process runs, its instructions/globals load into memory
- address space is like a huge array of bytes
  – total: 2^32 bytes
  – each int = 4 bytes
- as functions are called, data goes on a stack
- dynamic data is created on a heap

Stack frames

• stack frame or activation record: memory for a function call
  – stores parameters, local variables, and return address to go back to

```c
int f(int p1, int p2) {
    int x, int a[3];
    ... return x + y;
}
```
Tracing function calls

```c
int main(void) {
  int n1 = f(3, -5);
  n1 = g(n1);
  int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x + y;
  }
  int g(int param) {
    return param * 2;
  }
}
```

The & operator

```c
#include <stdio.h>
int main(void) {
  int x, y;
  int a[2];
  // printf("x is at %d\n", &x);
  printf("x is at %p\n", &x);
  // x is at 0x0022ff8c
  printf("y is at %p\n", &y);
  // y is at 0x0022ff88
  printf("a[0] is at %p, a[0][0] is at %p\n", &a[0], &a[0][0]);
  // a[0] is at 0x0022ff80
  return 0;
}
```

Danger!

- array bounds are not enforced; can overwrite other variables

```c
#include <stdio.h>
int main(void) {
  int x = 10, y = 20;
  int a[2] = {30, 40};
  printf("x = %d, y = %d\n", x, y);
  // x = 10, y = 20
  a[2] = 999;
  // !!!
  a[3] = 111;
  // !!!
  printf("x = %d, y = %d\n", x, y);
  // x = 111, y = 999
  return 0;
}
```

Segfault

- segmentation fault ("segfault"): A program crash caused by an attempt to access an illegal area of memory

```c
#include <stdio.h>
void f() {
  f();
}
int main(void) {
  f();
  return 0;
}
```

The sizeof operator

```c
#include <stdio.h>
int main(void) {
  int x, int a[5];
  printf("int\=\%d, double\=\%d\n", sizeof(int), sizeof(double));
  printf("x uses %d bytes\n", sizeof(x));
  printf("a[0] uses %d bytes, sizeof(a[0]) uses %d bytes\n", sizeof(a[0]), sizeof(a[0]));
  return 0;
}
```

Output:

```
int=4, double=8
x uses 4 bytes
a[0] uses 20 bytes
```
sizeof continued
• arrays passed as parameters do not remember their size
#include <stdio.h>
void f(int a[]);
int main(void) {
    int a[5];
    printf("a uses %d bytes\n", sizeof(a));
    f(a);
    return 0;
}
void f(int a[]) {
    printf("a uses %2d bytes in f\n", sizeof(a));
}
Output:
a uses 20 bytes
a uses  4 bytes in f

Pointer: a memory address referring to another value

Dereferencing: access the memory referred to by a pointer

*L-values and R-values
• L-value: Suitable for being on left-side of an = assignment -- a valid memory address to store into
• R-value: Suitable for right-side of an = assignment

Pass-by-value: copy parameters' values
• Cannot change the original ("actual") parameter variable

L-values and R-values
• L-value: Suitable for being on left-side of an = assignment -- a valid memory address to store into
• R-value: Suitable for right-side of an = assignment
    int x = 42;
    int* p;
p = &x; // p stores address of x
    *p = 99; // go to the int p refers to; set to 99
    printf("x is %d\n", x);

Output: x is 99

* vs. &
• many students get * and & mixed up
  - & references (ampersand gets an address)
  - * dereferences (star follows a pointer)

    int x = 42;
    int* y = &x;
    printf("x is %d \n", x); // x is 42
    printf("y is %p \n", y); // y is 0x0022ff8c
    printf("*y is %d \n", *y); // *y is 42
    printf("&y is %p \n", &y); // &y is 0x0022ff88

    * What is *x ?

  L-values:
  x or *p (store into x), p (changes what p points to)
  * not 6x, 6p, *x, *(*p), 12
  • R-values: x or *p. 6x of p, 6p
  * not &6p, 642

  Pass-by-value: copy parameters' values
  • Cannot change the original ("actual") parameter variable

    int main(void) {
        int a = 42, b = -7;
        swap(a, b);
        printf("a = %d, b = %d\n", a, b);
        return 0;
    }

    void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
    }
Pass-by-reference: point to parameters

- Can change the actual parameter variable using the “formal”

```c
int main(void) {
    int a = 42, b = -7;
    swap(a, b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}

void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Questions?