"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

David Notkin ● Autumn 2009 ● CSE303 Lecture 10

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

```
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)

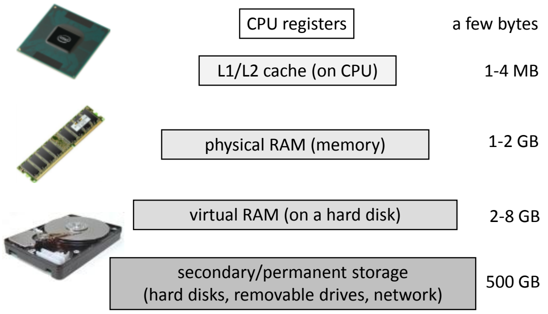
- String literals are the same as in Java
  - `printf("Hello, world!\n");`
  - but there is not actually a String type in C; they are just **`char[]`**
- Strings cannot be made, concatenated, or examined as in Java:

```
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
```

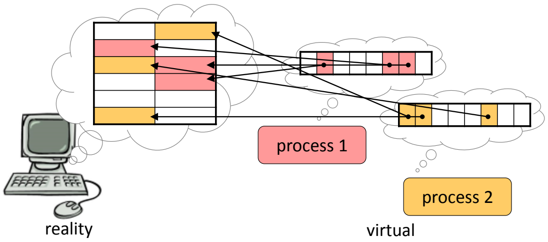
Memory hierarchy



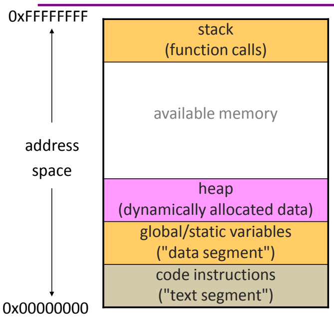
CPU registers	a few bytes
L1/L2 cache (on CPU)	1-4 MB
physical RAM (memory)	1-2 GB
virtual RAM (on a hard disk)	2-8 GB
secondary/permanent storage (hard disks, removable drives, network)	500 GB

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

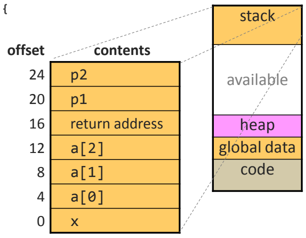


- when a process runs, its instructions/globals load into memory
- address space is like a huge array of bytes
  - total: 2^{32} bytes
  - each `int` = 4 bytes
- as functions are called, data goes on a **stack**
- dynamic data is created on a **heap**

Stack frames

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

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



offset	contents
24	p2
20	p1
16	return address
12	a[2]
8	a[1]
4	a[0]
0	x

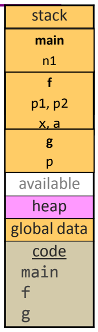
Tracing function calls

```
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

`&variable` produces **`variable`**'s memory address

```
#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\n", &a[0]);  // a[0] is at 0x0022ff80
    printf("a[1] is at %p\n", &a[1]);  // a[1] is at 0x0022ff84

    return 0;
}
```

  - **`%p`** placeholder in **`printf`** prints a memory address in hexadecimal

---

Danger!

- array bounds are not enforced; can overwrite other variables

```
#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

```
#include <stdio.h>

int main(void) {
    int a[2];
    a[999999] = 12345;  //out of bounds
    return 0;
}
```

Segfault

```
#include <stdio.h>

void f() {
    f();
}

int main(void) {
    f();
    return 0;
}
```

The `sizeof` operator

`sizeof(`***`type`***`)` or `(`***`variable`***`)` returns memory size in bytes

```
#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    uses %d bytes\n", sizeof(a));
    printf("a[0] uses %d bytes\n", sizeof(a[0]));
    return 0;
}
```

Output:

```
int=4, double=8
x    uses 4 bytes
a    uses 20 bytes
a[0] uses 4 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

```
type* name;                 // declare
type* name = address;       // declare/initialize


int x = 42;
int* p;
p = &x;         // p stores address of x

printf("x  is %d\n",    x);   // x  is 42
printf("&x is %p\n", &x);   // &x is 0x0022ff8c
printf("p  is %p\n",  p);   // p  is 0x0022ff8c



int* p1, p2;    // int* p1; int p2;
int* p1, *p2;   // int* p1; int* p2
```

Dereferencing: access the memory referred to by a pointer

```
*pointer                    // dereference
*pointer = value;           // dereference/assign


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("&x is %p\n", &x);       // &x is 0x0022ff8c
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 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 = &x;
```

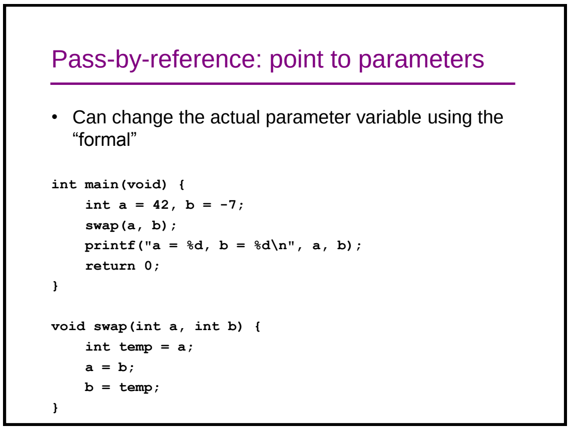
- L-values : `x` or `*p` (store into `x`), `p` (changes what `p` points to)
  - not `&x, &p, *x, *(*p), *12`
- R-values : `x` or `*p, &x` or `p, &p`
  - not `&(&p), &42`

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;
}
```

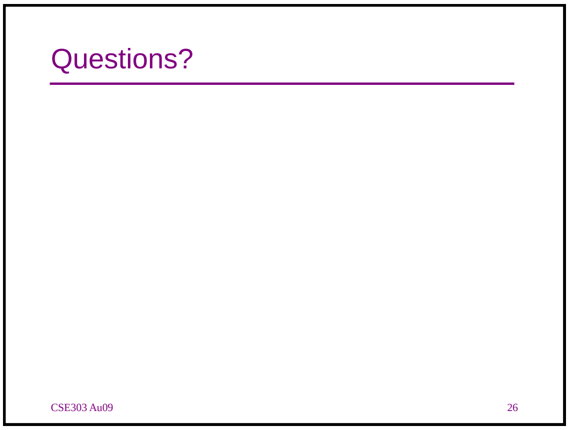


Pass-by-reference: point to parameters

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

```
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?