CSE 303
Lecture 10

C memory model;
stack allocation

reading: *Programming in C*  Ch. 11

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

• discuss ethics/society reading #3

• computer memory and addressing

• stack vs. heap

• pointers

• parameter passing
  ▪ by value
  ▪ by reference
Ethics/society reading #3

• Is DRM a hardware or software technology? What is one occasion in which you have run into DRM?

• Does DRM fundamentally conflict with fair use?

• Is DRM fair? If not, how can content creators ensure a suitable profit from their works without measures like DRM?
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

![Diagram showing virtual addressing](image)
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

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

<table>
<thead>
<tr>
<th>offset</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>a[0]</td>
</tr>
<tr>
<td>8</td>
<td>a[1]</td>
</tr>
<tr>
<td>12</td>
<td>a[2]</td>
</tr>
<tr>
<td>16</td>
<td>return address</td>
</tr>
<tr>
<td>20</td>
<td>p1</td>
</tr>
<tr>
<td>24</td>
<td>p2</td>
</tr>
</tbody>
</table>
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

&\texttt{variable} produces \texttt{variable}'s memory address

```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\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 \texttt{printf} prints a memory address in hexadecimal
OMG WTF BBQ

- 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;
}
```
• **segmentation fault** ("segfault"): A program crash caused by an attempt to access an illegal area of memory.

```c
#include <stdio.h>

void f() {
    f();  // infinite recursion
}

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

Output:  
Segmentation fault
```

```c
#include <stdio.h>

int main(void) {
    int a[2];
    a[999999] = 12345;  // oob
    return 0;
}

Output:  
Segmentation fault
```
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

```c
#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
```
Pointers

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

- **pointer**: A memory address that refers to another value.

```c
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
```

- **caution**: declaring multiple pointers on one line is tricky:

```c
int* p1, p2;  // incorrect => int* p1; int p2;
int* p1, * p2;  // correct
```
Dereferencing pointers

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

• dereference: To access the memory referred to by a pointer.

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

```c
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 the *left*-side of an `=` assignment.
  - in other words, a valid memory address that can be stored into

• **R-value**: A value suitable for the *right*-side of an `=` assignment.

```c
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` (42), `&x` or `p` (28fff8), `&p` (28fff8)
  - not `&(&p)`, `&42`
Pass-by-value

- **value semantics**: Parameters' values are copied.
  - impossible to affect change on the original parameter variable

```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;
    a = temp;
    b = temp;
}
```

Output:
```
a = 42, b = -7
```
Pass-by-reference

• **reference semantics**: Passed as references to / addresses of data.
  - can change the original parameter variable using the reference

```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;
}
```

Output:

```
a = -7, b = 42
```