Introduction (part 2)

CS 241
January 23, 2012
Announcements

- Anonymous feedback
- Honors section
- Registration
Review: New concepts in C

- Pointers
- Memory allocation
- Arrays
- Strings

Theme: how memory really works
Pointers
Variables

Memory Address

Name

Value

Type of each variable (also determines size)

int x;
double y;
float z;
double* p;
int d;
The “&” Operator: Reads “Address of”

&y

&y

Name

Value

Value1

Value2

Value3

Value4

Value5

Copyright ©: University of Illinois CS 241 Staff
A pointer is a variable whose value is the address of another.
A pointer is a variable whose value is the address of another.

\[ \text{*p} = \text{“Variable p points to”} \]
Memory allocation
Memory allocation

- Two ways to dynamically allocate memory
  - Stack
    - Named variables in functions
    - Allocated for you when you call a function
    - Deallocated for you when function returns
  - Heap
    - Memory on demand
    - You are responsible for all allocation and deallocation
Sample layout for program image in main memory

Processes have three segments: text, data, stack

High address

Command-line arguments and environment variables

stack

heap

Uninitialized static data

Initialized static data

Program text

Allocations from malloc family

Activation record for function calls (return address, parameters, saved registers, automatic variables)

argc, argv, environment

Copyright ©: University of Illinois CS 241 Staff
Allocating and deallocating heap memory

- Dynamically **allocating** memory
  - Programmer explicitly requests space in memory
  - Space is allocated dynamically on the heap
  - E.g., using “malloc” in C, “new” in Java

- Dynamically **deallocating** memory
  - Must reclaim or recycle memory that is never used again
  - To avoid (eventually) running out of memory
  - Either manual or via automatic “garbage collection”
Heap memory allocation

- C++:
  - `new` and `delete` allocate memory for a whole object

- C:
  - `malloc` and `free` deal with unstructured blocks of bytes
    
    ```c
    void* malloc(size_t size);
    void free(void* ptr);
    ```
Example

```c
int* p;
p = (int*) malloc(sizeof(int));
*p = 5;
free(p);
```

How many bytes do you want?

**Cast** to the right type
Manual deallocation can lead to bugs

- **Dangling pointers**
  - Programmer frees a region of memory
  - … but still has a pointer to it
  - Dereferencing pointer reads or writes nonsense values

```c
int main(void) {
    int *p;
    p = malloc(sizeof(int));
    ...
    free(p);
    ...
    printf("%d\n",*p);
}
```

May print nonsense
Manual deallocation can lead to bugs

- **Memory leak**
  - Programmer neglects to free unused region of memory
  - So, the space can never be allocated again
  - Eventually may consume all of the available memory

```c
void f(void) {
    int *d;
    d = malloc(sizeof(int));
}

int main(void) {
    while (1) f();
}
```

Eventually, `malloc()` returns `NULL`
Manual deallocation can lead to bugs

- **Double free**
  - Programmer mistakenly frees a region more than once
  - Leading to corruption of the heap data structure
  - ... or premature destruction of a different object

```c
int main(void) {
    int *p, *q;
    p = malloc(sizeof(int));
    ...
    free(p);
    q = malloc(sizeof(int));
    free(p);
}
```

Might free space allocated by `q`!
I’m hungry. More bytes plz.

```c
int* p = (int*) malloc(10 * sizeof(int));
```

- Now I have space for 10 integers, laid out contiguously in memory. What would be a good name for that...?
Arrays
Arrays

- Contiguous block of memory
  - Fits one or more elements of some type

- Two ways to allocate
  - named variable
    ```c
    int x[10];
    ```
  - dynamic
    ```c
    int* x = (int*) malloc(10*sizeof(int));
    ```

Is there a difference?
Arrays

int p[5];

Name of array (is a pointer)

Shorthand:
*(p+1) is called p[1]
*(p+2) is called p[2]
etc..
Adding integers to pointers (pointer arithmetic)

- Compiler uses the type information
  - `long *p;`
  - `p [long][long][long]`

- What address is `p + 2`?
  - `...p + sizeof(long) * 2`
Example

```c
int y[4];
y[1]=6;
y[2]=2;
```

Array Name as Pointer

- What’s the difference between the examples?

- Example 1:

```c
int z[8];
int *q;
q=z;
```

- Example 2:

```c
int z[8];
int *q;
q=&z[0];
```
Questions

- What’s the difference between
  ```c
  int* q;
  int q[5];
  ```

- What’s wrong with
  ```c
  int ptr[2];
  ptr[1] = 1;
  ptr[2] = 2;
  ```
Questions

- What is the value of \( b[2] \) at the end?

```c
int b[3];
int* q;
b[0]=48; b[1]=113; b[2]=1;
q=b;
*(q+1)=2;
b[2]=*b;
```
Questions

- What is the value of `b[2]` at the end?

```c
int b[3];
int* q;
b[0]=48; b[1]=113; b[2]=1;
q=b;
*(q+1)=2;
b[2]=*b;
```
Questions

What is the value of b[2] at the end?

```c
int b[3];
int* q;

b[0]=48; b[1]=113; b[2]=1;

q=b;
*(q+1)=2;
b[2]=*b;
```
Questions

- What is the value of \( b[2] \) at the end?

```c
int b[3];
int* q;
b[0]=48; b[1]=113; b[2]=1;
q=b;
*(q+1)=2;
b[2]=*b;
```

<table>
<thead>
<tr>
<th></th>
<th>b[0]</th>
<th>b[1]</th>
<th>b[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>48</td>
<td>113</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>48</td>
<td>2</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>2</td>
<td>50</td>
</tr>
</tbody>
</table>
Strings
Strings
(Null-terminated Arrays of Char)

- String = array of `char` followed by a "Null" character `\0` to indicate end
  - Do not forget to leave room for the null character

- Example
  - `char s[5];`

Copyright ©: University of Illinois CS 241 Staff
String and character literals

- **Strings**
  - “this is a string”
  - “c”

- **Characters**
  - ‘c’
  - ‘X’

- **Example**
  - `printf("x = %c", 'x');`
Typecasting
Typecasting

- Syntax: type name in parentheses in front of another expression

```java
main() {
    float a;
    a = (float)5 / 3;
}
```

- Result is `a = 1.666666`
  - Integer 5 is converted to floating point value before division and the operation between float and integer results in float

- What would `a` be without the `float`?
Typecasting

- Take care about using typecast
- If used incorrectly, may result in loss of data
  - e.g., truncating a float when casting to an int
Typecasting pointers

- Does not change pointer value
- Does affect pointer arithmetic
- Avoids compiler warnings

```c
int* p = 500;
printf("%p %p\n", p+1, ((char*) p) + 1);
```
Typecasting pointers

- Does not change pointer value
- Does affect pointer arithmetic
- **Avoids compiler warnings**

```c
int* p = (int*) 500;
printf("%p %p\n", p+1, ((char*) p) + 1);
```
A puzzler
Can we make this work?!

```c
int x;

printf("%s is awesome!\n", &x);
```

241 is awesome!
Wednesday

- Lecture: OS structures
- Homework due 11 a.m. via SVN