C - Functions, Pointers, Arrays

Jinyang Li

based on the slides of Tiger Wang
Functions
C program consists of functions (aka subroutines, procedures)

Why breaking code into functions?

– Readability
– Reusability
The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that. Functions should not be 100 lines long. Functions should hardly ever be 20 lines long.
Why small size?

• It fits easily on your screen without scrolling

• It should be the code size that you can hold in your head

• It should be meaningful enough to require a function in its own right
Local Variables

Scope
  – within which the variable can be used

```c
int add(int a, int b)
{
    int r = a + b;
    return r;
}
```

r’s scope is in function `add`
Local Variables / function arguments

Scope (within which the variable can be used)
- Within the function it is declared in
- local variables of the same name in different functions are unrelated

Storage:
- allocated upon function invocation
- deallocated upon function return

```c
int add(int a, int b) {
    int r = a + b;
    return r;
}
```

```c
int subtract(int a, int b) {
    int r = a - b;
    return r;
}
```
Global Variables

Scope
- Can be accessed by all functions

Storage
- Allocated upon program start, deallocated when entire program exits

```c
int r = 0;

int add(int a, int b) {
    r = a + b;
    return r;
}

int subtract(int a, int b) {
    int r = a - b;
    return r;
}
```

- modifies global variable r
- local variable r shadows global variable r
**Function invocation**

C (and Java) passes arguments by value

```c
int main()
{
    int x = 1;
    int y = 2;
    swap(x, y);

    printf("x: %d, y: %d", x, y);
}

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

Result  x: ?,  y: ?
Function invocation

C passes the arguments by value

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

int main() {
    int x = 1;
    int y = 2;
    swap(x, y);
    printf("x: %d, y: %d", x, y);
}
```

Result  x: 1,  y: 2

[Diagram showing the variables and their values before and after the swap function call]
Function invocation

C passes the arguments by value

```c
int main()
{
    int x = 1;
    int y = 2;
    swap(x, y);
    printf("x: %d, y: %d", x, y);
}
```

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

Result  x: 1,  y: 2
New Office Hour

**Computer Systems Organization**
CSCI-UA.0201(005), Spring 2018

**Lecture:**
MW 3:30 - 4:45pm, Location: Wav 366

**Recitation:**
R 12:30-1:45pm, Location: CIWW 312

**Resources:**
Piazza, cso-staff at cs.nyu.edu

**Lecturer:**
Jinyang Li

**Recitation Instructor:**
Lingfan Yu

**Graders:**
Jingyu Liu

**Office Hour:**
- Jinyang Li (Wed 1-2pm, 60 5th Ave Room 410)
- Lingfan Yu (Thu 2-3pm, 60 5th Ave Room 406)
- Jingyu Liu (Mon 1-2pm, and Tue 5-6pm, 60 5th Ave Room 406)
Announcements

• You must always read emails from Piazza
  – All announcements are made on Piazza first.
• Lab 1 is out, but 8 students have not signed up for lab yet.
  – sign up on github classroom
    https://classroom.github.com/a/rzOBdXtS
  – Follow lab instructions (see course webpage)
Today’s lecture

• Pointers
• Array and its relationship to pointer
• Pointer casting
• 2D array
Pointers

Pointer is a memory address
char a = 1;
char a = 1;
int b = 2;
char a = 1;
int b = 2;
char *x = &a;

& gives address of variable
equivalent to:
char *x;
x = &a;

equivalent to:
char* x;
x = &a;

what happens if I write
char x = &a;
or
int *x = &a;

type mismatch!
char a = 1;
int b = 2;
char *x = &a;

Size of pointer on a 64-bit machine?
8 bytes
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;

*x = 3;

Value of variable a after this statement?
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;

*x = 3;
// value of variable a?
//printf("a=%d\n", a);
Pointer

```
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;
*x = 3;
```

what if x is uninitialized?

Dereferencing an arbitrary address value may result in “Segmentation fault” or a random memory write
char a = 1;
int b = 2;
char *x = NULL;
int *y = &b;

*x = 3;

Always initialize pointers!

Dereferencing NULL pointer definitely results in “Segmentation fault”
char a = 1;
int b = 2;
char *x = NULL;
int *y = &b;

*x = 3;
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;

*x = 3;
*y = 127;
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;

*x = 3;
*y = 127;

char **xx = &x;

what happens if I write
char* xx;
xx = &x;

value of xx?
printf("xx=%p", xx);  xx=0x15

equivalent to
char **xx;
xx = &x;

equivalent to
char** x x;
xx = &x;
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;

*x = 3;
*y = 127;

char **xx = &x;
int **yy = &y;

value of yy?
printf("yy=%p", yy); yy=0x1b
Common confusions on *

* has two meanings!!
1. part of a pointer type name, e.g. char *, char **, int *
2. the deference operator.

```
char a = 1;
char *p = &a;
*p = 2;

char *b, *c;
char **d,**e;
char *f=p, *g=p;
char **m=&p, **n=&p;
```

C’s syntax for declaring multiple pointer variables on one line
```
char* b, c; does not work
```

C’s syntax for declaring and initializing multiple pointer variables on one line
Pass pointers to function

```c
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    a = tmp;
}
```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

    printf("x:%d, y:%d", x, y);
}

Size and value of a, b, tmp upon function entrance?
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

    printf("x:%d, y:%d", x, y);
}
```c
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main() {
    int x = 1;
    int y = 2;
    swap(&x, &y);
    printf("x:%d, y:%d", x, y);
}
```
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

    printf("x:%d, y:%d", x, y);
}
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);
    printf("x:%d, y:%d", x, y);
}
```c
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);
    printf("x:%d, y:%d", x, y);
}
```
### Pointer arithmetic

```c
int a = 0;
int *p = &a; // assume the address of variable a is 0x104
```

<table>
<thead>
<tr>
<th>p+1</th>
<th>Point to the next object with type int (4 bytes after current object of address p)</th>
<th>???</th>
</tr>
</thead>
</table>

```
0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00
```

```
0x10a
0x109
0x108
0x107
0x106
0x105
0x104
0x103
```

```
0x10b
0x10c
```

```
a: 0x104
p: 0x10c
p+1: 0x108
p+2: 0x10c
```
# Pointer arithmetic

```c
int a = 0;
int *p = &a;  // assume the address of variable a is 0x104
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Address Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+i</td>
<td>Point to the $i$th object of type int after object with address $p$</td>
<td>$0x104 + i \times 4$</td>
</tr>
<tr>
<td>p-i</td>
<td>Point to the $i$th object with int before object with address $p$</td>
<td>$0x104 - i \times 4$</td>
</tr>
</tbody>
</table>
# Pointer arithmetic

```c
short a = 0;
short *p = &a; // assume the address of variable a is 0x104
```

<table>
<thead>
<tr>
<th>p+i</th>
<th>Point to the ith object with type short after object with address p</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-i</td>
<td>Point to the ith object with type short before object with address p</td>
<td>???</td>
</tr>
</tbody>
</table>
Pointer arithmetic

```c
short a = 0;
short *p = &a; // assume the address of variable a is 0x104
```

<table>
<thead>
<tr>
<th>p+i</th>
<th>Point to the ith object with type short after object with address p</th>
<th>0x104 + i*2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-i</td>
<td>Point to the ith object with type short before object with address p</td>
<td>0x104 - i*2</td>
</tr>
</tbody>
</table>
**Pointer arithmetic**

```c
char *a = NULL;
char **p = &a; // assume the address of variable a is 0x104
```

<table>
<thead>
<tr>
<th>p+i</th>
<th>Point to the i-th object with type char * after object with address p</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-i</td>
<td>Point to the i-th object with type char * before object with address p</td>
<td>???</td>
</tr>
</tbody>
</table>
Pointer arithmetic

```
char *a = NULL;
char **p = &a; // assume the address of variable a is 0x104
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p+i</td>
<td>Point to the ith object with type char * after object with address p</td>
<td>0x104 + i*8</td>
</tr>
<tr>
<td>p-i</td>
<td>Point to the ith object with type char * before object with address p</td>
<td>0x104 – i*8</td>
</tr>
</tbody>
</table>
Arrays

Array is a collection of contiguous objects with the same type
Array

Strong relationship with pointer
  – array access can be done using pointers.

A block of n consecutive objects.
  – int a[10];

```
a: int [10]
```
Array

a:

```
<table>
<thead>
<tr>
<th>int</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
```

length of a[0]: 4 bytes $\rightarrow$ a[1] is 4 bytes next to a[0]
Array

length of a[0]: 4 bytes → a[1] is 4 bytes next to a[0]

int *p = &a[0] → p+1 points to a[1]
Array

length of $a[0]$: 4 bytes $\rightarrow$ $a[1]$ is 4 bytes next to $a[0]$

int *$p = \&a[0] \rightarrow p+1$ points to $a[1]$  
$\rightarrow p + i$ points to $a[i]$
Array

length of a[0]: 4 bytes \( \rightarrow \) a[1] is 4 bytes next to a[0]

int *p = &a[0] \( \rightarrow \) p+1 points to a[1]
\( \rightarrow \) p + i points to a[i]

int *p = a \quad \longleftrightarrow \quad \text{int} *p = &a[0]
Array

length of a[0]: 4 bytes → a[1] is 4 bytes next to a[0]

int *p = &a[0] → p+1 points to a[1]
   → p + i points to a[i]

int *p = a  ←  int *p = &a[0]
p++  ✓
a++  ✓  compilation error
p = &a  ✗
**Array**

- Length of `a[0]`: 4 bytes \(\rightarrow\) `a[1]` is 4 bytes next to `a[0]`

- `int *p = &a[0] \rightarrow p + 1` points to `a[1]`
- `\rightarrow p + i` points to `a[i]`

- `int *p = a` \(\leftrightarrow\) `int *p = &a[0]`
- `*(p+1)` \(\leftrightarrow\) `p[1]`
- `*(p + i)` \(\leftrightarrow\) `p[i]`
```c
#include <stdio.h>

int main() {
    int a[3] = {100, 200, 300};
    int *p = a;
    *p = 400;
    for (int i=0; i<3; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}
```

Output? 400 200 300

What if change to: *(p+1) = 400;
Output: 100 400 300

Equivalent to p[0] = 400;
Another Example

```c
#include <stdio.h>

int main() {
    int a[3] = {100, 200, 300};
    int *p = a;
    p++;
    *p = 400;
    for (int i=0; i<3; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

Output? 100 400 300
```
Pass array to function via pointer

// multiply every array element by 2
void multiply2(int *a) {
    for (int i = 0; i < ???; i++) {
        a[i] *= 2;
    }
}

int main() {
    int a[2] = {1, 2};
    multiply2(a);
    for (int i = 0; i < 2; i++) {
        printf("a[%d]=%d", i, a[i]);
    }
}
Pass array to function via pointer

```c
// multiply every array element by 2
void multiply2(int *a, int n) {
    for (int i = 0; i < n; i++) {
        a[i] *= 2; // (*a+i) *= 2;
    }
}

int main() {
    int a[2] = {1, 2};
    multiply2(a, 2);
    for (int i = 0; i < 2; i++) {
        printf("a[%d]=%d", i, a[i]);
    }
}
```
Pointer casting

int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
printf("%x\n", *c);

Output? (when running on Intel laptop)
Pointer casting

```c
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
```

Intel laptop is small endian
*\(c\) is 0x78

What is \(c+1\)? \(p+1\)?
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;

*(c+1) is 0x56
Pointer casting

```c
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
```

*(c+1) is 0x56

What about big endian?
Another example of pointer casting

```cpp
bool is_normalized_float(float f)
{
}
```
Another example of pointer casting

```c
bool is_normalized_float(float f)
{
    unsigned int i;
    i = *(unsigned int *)&f;

    unsigned exp = (i&0x7fffffff)>>23;
    return (exp != 0);
}
```
function `sizeof`

`sizeof(type)`
- Returns size in bytes of the object representation of type

`sizeof(expression)`
- Returns size in bytes of the type that would be returned by expression, if evaluated.
## function `sizeof`

<table>
<thead>
<tr>
<th><code>sizeof()</code></th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sizeof(int)</code></td>
<td></td>
</tr>
<tr>
<td><code>sizeof(long)</code></td>
<td></td>
</tr>
<tr>
<td><code>sizeof(float)</code></td>
<td></td>
</tr>
<tr>
<td><code>sizeof(double)</code></td>
<td></td>
</tr>
<tr>
<td><code>sizeof(int *)</code></td>
<td></td>
</tr>
</tbody>
</table>

`64 bits machine`
## function `sizeof`

<table>
<thead>
<tr>
<th><code>sizeof()</code></th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sizeof(int)</code></td>
<td>4</td>
</tr>
<tr>
<td><code>sizeof(long)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>sizeof(float)</code></td>
<td>4</td>
</tr>
<tr>
<td><code>sizeof(double)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>sizeof(int *)</code></td>
<td>8</td>
</tr>
</tbody>
</table>

64 bits machine
### function `sizeof`

<table>
<thead>
<tr>
<th>expr</th>
<th><code>sizeof()</code></th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int a = 0;</code></td>
<td><code>sizeof(a)</code></td>
<td></td>
</tr>
<tr>
<td><code>long b = 0;</code></td>
<td><code>sizeof(b)</code></td>
<td></td>
</tr>
<tr>
<td><code>int a = 0; long b = 0;</code></td>
<td><code>sizeof(a + b)</code></td>
<td></td>
</tr>
<tr>
<td><code>char c[10];</code></td>
<td><code>sizeof(c)</code></td>
<td></td>
</tr>
<tr>
<td><code>int arr[10];</code></td>
<td><code>sizeof(arr)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>sizeof(arr[0])</code></td>
<td></td>
</tr>
<tr>
<td><code>int *p = arr;</code></td>
<td><code>sizeof(p)</code></td>
<td></td>
</tr>
</tbody>
</table>

64 bits machine
# Function `sizeof`

<table>
<thead>
<tr>
<th>expr</th>
<th><code>sizeof()</code></th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int a = 0;</code></td>
<td><code>sizeof(a)</code></td>
<td>4</td>
</tr>
<tr>
<td><code>long b = 0;</code></td>
<td><code>sizeof(b)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>int a = 0; long b = 0;</code></td>
<td><code>sizeof(a + b)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>char c[10];</code></td>
<td><code>sizeof(c)</code></td>
<td>10</td>
</tr>
<tr>
<td><code>int arr[10];</code></td>
<td><code>sizeof(arr)</code></td>
<td>10 * 4 = 40</td>
</tr>
<tr>
<td></td>
<td><code>sizeof(arr[0])</code></td>
<td>4</td>
</tr>
<tr>
<td><code>int *p = arr;</code></td>
<td><code>sizeof(p)</code></td>
<td>8</td>
</tr>
</tbody>
</table>

64 bits machine