C - Functions, Pointers, Arrays

Jinyang Li

Some are based on slides of Tiger Wang
Today’s lecture

- Functions
- Pointers
- Array and its relationship to pointer
Functions
C program consists of functions (aka subroutines, procedures)

Why breaking code into functions?
- Readability
- Reusability
Advice from Linus

- Functions should be short and sweet, and do just one thing.
- The maximum length of a function is inversely proportional to the complexity of that function.  
  - For complex tasks, break it up into pieces and use helper functions with descriptive names.
- How to measure complexity?
  - Indentation level
  - # of local variables in a function should not exceed 5-10

Google: Linux kernel coding style
Local Variables

Scope (confining the usage of the variable)
- within the function
- local variables of the same name in different functions are unrelated

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

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

- Function arguments are basically local variables
- Local variables’ storage policy:
  - allocated upon function invocation
  - de-allocated upon function return

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

int main()
{
    int r;
    add(1, 2);  // r = add(1, 2);
    printf("r=%d\n", r);
    return 0;
}
```
Global Variables

Scope
- Can be accessed by all functions

Storage
- Allocated upon program startup, 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;
}
```

- Local variable `r` shadows global variable `r`
Function invocation

C (and Java) passes arguments by value

```c
void swap(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

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

Result  x: ?,  y: ?
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);
}
```

```c
void swap(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

Result  x: 1,  y: 2
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);
}

void swap(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

Result  x: 1,  y: 2
Pointers

Pointer is a memory address
char a = 1;
**Pointer**

```
char a = 1;
int b = 2;
```
char a = 1;
int b = 2;
char *x;

char * is a (pointer) type. char * is the same as char*

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

& gives address of variable

cchar *x; x = &a can be shorten as:
char *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;
int *y = &b;

How to make y a pointer that points to 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);

printf("a=%d\n", a);
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
Pointer

Always initialize pointers!

Dereferencing NULL pointer definitely results in “Segmentation fault”

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

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

*x = 3;
```

```
(gdb) r
Starting program: /oldhome/jinyang/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000000000004005ef in main () at foo.c:16
16       *x = 3;
(gdb) p x
$1 = 0x0
(gdb) 
```
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;
equivalent to
c

char** x x;
x x = &x;

what happens if I write
c

char*  xx;
x x = &x;

value of xx?
printf(“xx=%p”, xx); xx=0x15
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 dereference operator.

```c
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
```c
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;
    b = tmp;
}
```
Pass pointers to function

```c
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);
}
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>
<tbody>
<tr>
<td></td>
<td>0x10c ← p+2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x10b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x10a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x109</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x108 ← p+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a: 0x104 ← p</td>
<td></td>
</tr>
</tbody>
</table>
### Pointer arithmetic

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

<table>
<thead>
<tr>
<th>p+i</th>
<th>Point to the ith object of type int after object with address p</th>
<th>0x104 + i*4</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-i</td>
<td>Point to the ith object with int before object with address p</td>
<td>0x104 – i*4</td>
</tr>
</tbody>
</table>
**Pointer arithmetic**

\[
\text{short } a = 0; \\
\text{short } *p = &a; // assume the address of variable } a \text{ is } 0x104
\]

<table>
<thead>
<tr>
<th>p+i</th>
<th>Point to the (i)th object with type short after object with address (p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-i</td>
<td>Point to the (i)th object with type short before object with address (p)</td>
</tr>
</tbody>
</table>

???
Pointer arithmetic

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 i-th 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 i-th 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></th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+i</td>
<td>Point to the i'th object with type char * after object with address p</td>
<td>???</td>
</tr>
<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

```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 ith object with type char * after object with address p</th>
<th>0x104 + i*8</th>
</tr>
</thead>
<tbody>
<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>
What we’ve learnt and today’s plan

• Bitwise operations
• Pointers
  – Pointers are addresses
  – With pointers arguments, a callee can modify local variables in the caller.
• Today’s lesson:
  – Array and its relationship with pointer
  – Pointer casting
Array: a collection of contiguous objects with the same type
int a[3];
Array

```c
int a[3] = {1, 2, 3};
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
</tr>
</tbody>
</table>

If machine is little Endian
int a[3] = {1, 2, 3};

int *p;
p = &a[0]; //equivalent to p = a;

printf("%p\n", p); //output? 0x100

printf("%d\n", *p); //output? 1
Pointer arithmetic

int a[3] = {1, 2, 3};

int *p;
p = &a[0];

p = p + 1; //equivalent to p++

printf("%p\n", p); //output? 0x104

printf("%d\n", *p); //output? 2

Rule of pointer arithmetic:
p+i has address of i-th object after p, i.e. p+i’s value is p’s value plus i*sizeof(int)
Pointer arithmetic

```
int a[3] = {1, 2, 3};
int *p;
p = &a[0];

printf("%p\n", p+2); //output? 0x108
printf("%d\n", *(p+2)); //output? 3
```

Rule of pointer arithmetic:
p+i has address of i-th object after p, i.e. p+i’s value is p’s value plus i*sizeof(int)
Pointer arithmetic

char a[3] = {1, 2, 3};
char *p;
p = &a[0];

printf("%p\n", p+2);  //output? 0x102
printf("%d\n", *(p+2)); //output? 3

Rule of pointer arithmetic:
p+i has address of i-th object after p, i.e. p+i’s value is p’s value plus i*sizeof(int)
Array and pointer

```c
int a[10];
int *p;

p = &a[0]; // a is alias for &a[0];

for (int i = 0; i < 10; i++) {
    *(p+i) = 0; // p[i] is alias for *(p+i)
}
```
Array and pointer

```c
int a[10];
int *p;

p = a;    // a is alias for &a[0];

for (int i = 0; i < 10; i++) {
    p[i] = 0;    // p[i] is alias for *(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
Another Example

#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

// 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

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

Output? (when running on Intel laptop)
Pointer casting

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

Intel laptop is small endian

*`c`* is 0x78

What is `c+1`? `p+1`?
**Pointer casting**

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

![Diagram showing memory layout and pointer operations]

*(c+1) is 0x56

- p+1
- c+1
- p, c
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 && exp != 127);
}
```
**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></th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeof()</td>
<td></td>
</tr>
<tr>
<td>sizeof(int)</td>
<td></td>
</tr>
<tr>
<td>sizeof(long)</td>
<td></td>
</tr>
<tr>
<td>sizeof(float)</td>
<td></td>
</tr>
<tr>
<td>sizeof(double)</td>
<td></td>
</tr>
<tr>
<td>sizeof(int *)</td>
<td></td>
</tr>
</tbody>
</table>

64 bits machine
## function `sizeof`

<table>
<thead>
<tr>
<th>Function</th>
<th>Result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sizeof()</code></td>
<td></td>
</tr>
<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></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><code>10 * 4 = 40</code></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