CSCI-UA.0201-003

Computer Systems Organization

Lecture 2-3-4: C Programming

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Many slides of this lecture are adapted from Lewis Girod, CENS Systems Lab
http://lecs.cs.ucla.edu/~girod/talks/c-tutorial.ppt
and Clark Barrett
In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of *The C Programming Language* by Kernighan & Ritchie caused a revolution in the computing world
Why C?

• Mainly because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:
  – Operating Systems
  – Language Compilers
  – Assemblers
  – Text Editors
  – Print Spoolers
  – Network Drivers
  – Modern Programs
  – Data Bases
  – Language Interpreters
  – Utilities
Your first goal: Learn C!

• Resources
  – These lectures
  – Additional online resources (some links on the website)

• Learning a Programming Language
  – The best way to learn is to write programs
  – Start using the virtual machine environment to play with C
  – Work your way through examples from lectures, KR, and/or additional online tutorials
  – Once you are comfortable writing simple programs in C, take a look at Lab 1
Writing and Running Programs

1. Write text of program (source code) using an editor such as emacs, save as file e.g. my_program.c

2. Run the compiler to convert program from source to an “executable” or “binary”:
   $ gcc -Wall -g -o my_program my_program.c

3-Compiler gives errors and warnings; edit source file, fix it, and re-compile

Run it and see if it works 😊
   $ ./my_program
   Hello World
   $
About C

• Hardware independent
• Programs portable to most computers
• Case-sensitive
• Four stages
  – Editing: Writing the source code by using some IDE or editor
  – Preprocessing or libraries: Already available routines
  – compiling: translates or converts source to object code for a specific platform source code \rightarrow object code
  – linking: resolves external references and produces the executable module
C Syntax and Hello World

```c
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv) {
    printf("Hello World\n");
    return 0;
}
```

The main() function is always where your program starts running.

Blocks of code are marked by `{ ... }`

Print out a message. ‘\n’ means “new line”.

What do the <> mean?

#include inserts another file. “.h” files are called “header” files. They contain stuff needed to interface to libraries and code in other “.c” files.

This is a comment. The compiler ignores this.

Can your program have more than one .c file?
Compilation occurs in two steps: “Preprocessing” and “Compiling”

Preprocess

In Preprocessing, source code is “expanded” into a larger form that is simpler for the compiler to understand. Any line that starts with ‘#’ is a line that is interpreted by the Preprocessor.

- Include files are “pasted in” (#include)
- Macros are “expanded” (#define)
- Comments are stripped out ( /* */ , // )
- Continued lines are joined ( \ )

The compiler then converts the resulting text into binary code the CPU can run directly.
OK, We’re Back.. What is a Function?

A **Function** is a series of instructions to run. You pass **Arguments** to a function and it returns a **Value**.

“main()” is a Function. It’s only special because it always gets called first when you run your program.

```
#include <stdio.h>

/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}
```

**Calling a Function**: “printf()” is just another function, like main(). It’s defined for you in a “library”, a collection of functions you can call from your program.

**Return type, or void**

**Function Arguments**

**Returning a value**
What is “Memory”?

Memory is like a big table of numbered slots where bytes can be stored.

The number of a slot is its **Address**. One byte **Value** can be stored in each slot.

Some “logical” data values span more than one slot, like the character string “Hello\n”

A **Type** names a logical meaning to a span of memory. Some simple types are:

| char | a single character (1 slot) |
| char [10] | an array of 10 characters |
| int | signed 4 byte integer |
| float | 4 byte floating point |
| int64_t | signed 8 byte integer |

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>6</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>7</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>8</td>
<td>‘o’ (111)</td>
</tr>
<tr>
<td>9</td>
<td>‘\n’ (10)</td>
</tr>
<tr>
<td>10</td>
<td>‘\0’ (0)</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
What is a Variable?

A **Variable** names a place in memory where you store a **Value** of a certain **Type**.

You first **Define** a variable by giving it a name and specifying the type, and optionally an initial value.

```
char x;
char y='e';
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
</tbody>
</table>

Initial value of x is undefined.

The compiler puts them somewhere in memory.

Type is single character (char)
Multi-byte Variables

Different types consume different amounts of memory. Most architectures store data on “word boundaries”, or even multiples of the size of a primitive data type (int, char)

char x;
char y='e';
int z = 0x01020304;

0x means the constant is written in hex

An int consumes 4 bytes

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>z</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Lexical Scoping

Every **Variable** is **Defined** within some scope. A Variable cannot be referenced from outside of that scope.

Lexical scopes are defined with curly braces `{ }`.

The scope of Function Arguments is the complete body of the function.

The scope of Variables defined inside a function starts at the definition and ends at the closing brace of the containing block.

The scope of Variables defined outside a function starts at the definition and ends at the end of the file. Called “**Global**” Vars.

(Returns nothing)

```c
void p(char x) {
    char y;
    char z;
}
char z;
```
Expressions and Evaluation

Expressions combine Values using Operators, according to precedence.

\[
1 + 2 \times 2 \rightarrow 1 + 4 \rightarrow 5 \\
(1 + 2) \times 2 \rightarrow 3 \times 2 \rightarrow 6
\]

Comparison operators are used to compare values. In C, 0 means “false”, and any other value means “true”.

\[
\text{int } x=4; \\
(x < 5) \rightarrow (4 < 5) \rightarrow \text{<true>} \\
(x < 4) \rightarrow (4 < 4) \rightarrow 0 \\
((x < 5) \text{ || } (x < 4)) \rightarrow (\text{<true>} \text{ || } (x < 4)) \rightarrow \text{<true>}
\]

Not evaluated because first clause was true
Precedence

• Highest to lowest
  • ()
  • *, /, %
  • +, -
Comparison and Mathematical Operators

== equal to
< less than
<= less than or equal
> greater than
>= greater than or equal
!= not equal
&& logical and
|| logical or
! logical not

+ plus
- minus
* mult
/ divide
% modulo

& bitwise and
| bitwise or
^ bitwise xor
~ bitwise not
<< shift left
>> shift right

The rules of precedence are clearly defined but often difficult to remember or non-intuitive. When in doubt, add parentheses to make it explicit.

Beware division:

• If second argument is integer, the result will be integer (rounded): 5 / 10 \(\rightarrow\) 0 whereas 5 / 10.0 \(\rightarrow\) 0.5
• Division by 0 will cause a FPE

Don’t confuse & and &&..
1 & 2 \(\rightarrow\) 0 whereas 1 && 2 \(\rightarrow\) <true>
Assignment Operators

\[
x = y \quad \text{assign } y \text{ to } x
\]
\[
x++ \quad \text{post-increment } x
\]
\[
++x \quad \text{pre-increment } x
\]
\[
x-- \quad \text{post-decrement } x
\]
\[
--x \quad \text{pre-decrement } x
\]
\[
x += y \quad \text{assign } (x+y) \text{ to } x
\]
\[
x -= y \quad \text{assign } (x-y) \text{ to } x
\]
\[
x *= y \quad \text{assign } (x*y) \text{ to } x
\]
\[
x /= y \quad \text{assign } (x/y) \text{ to } x
\]
\[
x %= y \quad \text{assign } (x\%y) \text{ to } x
\]

Note the difference between ++x and x++:

\[
\text{int } x=5; \\
\text{int } y; \\
y = ++x; \quad /* x == 6, y == 6 */
\]
\[
\text{int } x=5; \\
\text{int } y; \\
y = x++; \quad /* x == 6, y == 5 */
\]

Don’t confuse = and ==

\[
\text{int } x=5; \\
\text{if } (x==6) \quad /* \text{false} */ \\
\{
\quad /* ... */
\} \quad /* x \text{ is still } 5 */
\]
\[
\text{int } x=5; \\
\text{if } (x=6) \quad /* \text{always true} */ \\
\{
\quad /* x \text{ is now } 6 */
\} \quad /* ... */
\]
A More Complex Program: pow

```
#include <stdio.h>
#include <inttypes.h>

float pow(float x, uint32_t exp) {
    /* base case */
    if (exp == 0) {
        return 1.0;
    }
    /* "recursive" case */
    return x * pow(x, exp - 1);
}

int main(int argc, char **argv) {
    float p;
    p = pow(10.0, 5);
    printf("p = %.1f\n", p);
    return 0;
}
```

"if" statement

```
/* if evaluated expression is not 0 */
if (expression) {
    /* then execute this block */
} else {
    /* otherwise execute this block */
}
```

Tracing “pow()”:
- What does pow(5,0) do?
- What about pow(5,1)?
Recall lexical scoping. If a variable is valid “within the scope of a function”, what happens when you call that function recursively? Is there more than one “exp”?

Yes. Each function call allocates a “stack frame” where Variables within that function’s scope will reside.

```c
#include <stdio.h>
#include <inttypes.h>

float pow(float x, uint32_t exp) {
    /* base case */
    if (exp == 0) {
        return 1.0;
    }
    /* "recursive" case */
    return x * pow(x, exp - 1);
}

int main(int argc, char **argv) {
    float p;
    p = pow(5.0, 1);
    printf("p = %f\n", p);
    return 0;
}
```
The “for” loop

The “for” loop is just shorthand for this “while” loop structure.

```
float pow(float x, uint exp)
{
    float result=1.0;
    int i;
    i=0;
    while (i < exp) {
        result = result * x;
        i++;
    }
    return result;
}

int main(int argc, char **argv)
{
    float p;
    p = pow(10.0, 5);
    printf("p = %f\n", p);
    return 0;
}
```

```
float pow(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    return result;
}

int main(int argc, char **argv)
{
    float p;
    p = pow(10.0, 5);
    printf("p = %f\n", p);
    return 0;
}
```
When to Use What?

Different Loop-constructs
- while
- do-while
- for

Conditions
- if-else
- switch-case
Can a function modify its arguments?

What if we wanted to implement a function `pow_assign()` that modified its argument, so that these are equivalent:

```c
float p = 2.0;
/* p is 2.0 here */
p = pow(p, 5);
/* p is 32.0 here */
```

```c
float p = 2.0;
/* p is 2.0 here */
pow_assign(p, 5);
/* p is 32.0 here */
```

Would this work?

```c
void pow_assign(float x, uint exp) {
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}
```
Remember the stack!

```c
void pow_assign(float x, uint exp) {
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}
```

```c
{ float p=2.0;
  pow_assign(p, 5);
}
```

In C, all arguments are passed as values. But, what if the argument is the address of a variable?

<table>
<thead>
<tr>
<th>float x</th>
<th>32.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32_t exp</td>
<td>5</td>
</tr>
<tr>
<td>float result</td>
<td>32.0</td>
</tr>
<tr>
<td>float p</td>
<td>2.0</td>
</tr>
</tbody>
</table>
## Passing Addresses

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char x</td>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>char y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- address of x: 4
- memory at 4: 72
This is exactly how “pointers” work.

“address of” or reference operator: &
“memory_at” or dereference operator: *

address of x: &x
memory at y: *y

A “pointer type”: pointer to char

```
void f(char * p)
{
    *p = *p - 32;
}
```

```
char y = 101;    /* y is 101 */
f(&y);          /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

Pointers are used in C for many other purposes:
• Passing large objects without copying them
• Accessing dynamically allocated memory
• Referring to functions
A Valid pointer is one that points to memory that your program controls. Using invalid pointers will cause non-deterministic behavior, and will often cause Linux to kill your process (SEGV or Segmentation Fault).

There are two general causes for these errors:
• Program errors that set the pointer value to a strange number
• Use of a pointer that was at one time valid, but later became invalid

Will ptr be valid or invalid?

```c
char * get_pointer()
{
    char x=0;
    return &x;
}

{
    char * ptr = get_pointer();
    *ptr = 12;  /* valid? */
}
Answer: Invalid!

A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```c
char * get_pointer() {
    char x = 0;
    return &x;
}

char * ptr = get_pointer();
*ptr = 12; /* valid? */
other_function();
```

But now, `ptr` points to a location that’s no longer in use, and will be reused the next time a function is called!
More on Types

We’ve seen a few types at this point: char, int, float, char *

Types are important because:
• They allow your program to impose logical structure on memory
• They help the compiler tell when you’re making a mistake

In the next slides we will discuss:
• How to create logical layouts of different types (structs)
• How to use arrays
• How to parse C type names (there is a logic to it!)
• How to create new types using typedef
Structures

struct: a way to compose existing types into a structure

```c
#include <sys/time.h>

/* declare the struct */
struct my_struct {
    int counter;
    float average;
    struct timeval timestamp;
    uint in_use:1;
    uint8_t data[0];
};

/* define an instance of my_struct */
struct my_struct x = {
    in_use: 1,
    timestamp: {
        tv_sec: 200
    }
};
x.counter = 1;
x.average = sum / (float)(x.counter);

struct my_struct * ptr = &x;
ptr->counter = 2; /* equiv. */
(*ptr).counter = 3; /* equiv. */
```

- structs define a layout of typed fields
- structs can contain other structs
- fields can specify specific bit widths
- A newly-defined structure is initialized using this syntax. All unset fields are 0.
- Fields are accessed using `.` notation.
- A pointer to a struct. Fields are accessed using `->` notation, or (*ptr).counter
Arrays in C are composed of a particular type, laid out in memory in a repeating pattern. Array elements are accessed by stepping forward in memory from the base of the array by a multiple of the element size.

/* define an array of 10 chars */
char x[5] = {'t','e','s','t','\0'};

/* accessing element 0 */
x[0] = 'T';

/* pointer arithmetic to get elt 3 */
char elt3 = *(x+3); /* x[3] */

/* x[0] evaluates to the first element; */
/* * x evaluates to the address of the */
/* first element, or &(x[0]) */

/* 0-indexed for loop idiom */
#define COUNT 10
char y[COUNT];
int i;
for (i=0; i<COUNT; i++) {
    /* process y[i] */
    printf("%c\n", y[i]);
}
How to Parse and Define C Types

At this point we have seen a few basic types, arrays, pointer types, and structures. So far we’ve glossed over how types are named.

```c
int x;        /* int;                      */  typedef int T;
int *x;       /* pointer to int;           */  typedef int *T;
int x[10];    /* array of ints;            */  typedef int T[10];
int *x[10];   /* array of pointers to int; */  typedef int *T[10];
int (*x)[10]; /* pointer to array of ints; */  typedef int (*T)[10];
```

typedef defines a new type

C type names are parsed by starting at the type name and working outwards according to the rules of precedence:

- `int *x[10];` is an array of pointers to int
- `int (*x)[10];` is a pointer to an array of int

Arrays are the primary source of confusion. When in doubt, use extra parens to clarify the expression.
Dynamic Memory Allocation

So far all of our examples have allocated variables statically by defining them in our program. This allocates them in the stack.

But, what if we want to allocate variables based on user input or other dynamic inputs, at run-time? This requires dynamic allocation.

```c
int * alloc_ints(size_t requested_count) {
    int * big_array;
    big_array = (int *)calloc(requested_count, sizeof(int));
    if (big_array == NULL) {
        printf("can't allocate %d ints: %m\n", requested_count);
        return NULL;
    }
    /* now big_array[0] .. big_array[requested_count-1] are valid and zeroed. */
    return big_array;
}
```

`calloc()` allocates memory for N elements of size k

It's OK to return this pointer. It will remain valid until it is freed with `free()`.
Caveats with Dynamic Memory

Dynamic memory is useful. But it has several caveats:

Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()'d when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory is called a “memory leak”.

Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that has been freed. Whenever you free memory you must be certain that you will not try to use it again. It is safest to erase any pointers to it.

Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic
Macros can be a useful way to customize your interface to C and make your code easier to read and less redundant. However, when possible, use a static inline function instead.

Macros and static inline functions must be included in any file that uses them, usually via a header file. Common uses for macros:

```c
/* Macros are used to define constants */
#define FUDGE_FACTOR 45.6
#define MSEC_PER_SEC 1000
#define INPUT_FILENAME "my_input_file"

/* Macros are used to do constant arithmetic */
#define TIMER_VAL (2*MSEC_PER_SEC)

/* Macros are used to capture information from the compiler */
#define DBG(args...) do {
    fprintf(stderr, "%s:%s:%d: \\
        __FUNCTION__, __FILE__, __LINENO__); \\
    fprintf(stderr, args...); \\
} while (0)

/* ex. DBG("error: %d", errno); */
```

Float constants must have a decimal point, else they are type int

Put expressions in parens.

Multi-line macros need \\ args... grabs rest of args

Enclose multi-statement macros in do{}while(0)
Using “goto”

Some schools of thought frown upon goto, but goto has its place. A good philosophy is, always write code in the most expressive and clear way possible. If that involves using goto, then goto is not bad.

An example is jumping to an error case from inside complex logic. The alternative is deeply nested and confusing “if” statements, which are hard to read, maintain, and verify. Often additional logic and state variables must be added, just to avoid goto.

```c
goto try_again;
goto fail;
```
Unrolling a Failed Initialization using goto

```c
state_t *initialize()
{
    /* allocate state struct */
    state_t *s = g_new0(state_t, 1);
    if (s) {
        /* allocate sub-structure */
        s->sub = g_new0(sub_t, 1);
        if (s->sub) {
            /* open file */
            s->sub->fd =
                open("/dev/null", O_RDONLY);
            if (s->sub->fd >= 0) {
                /* success! */
            } else {
                free(s->sub);
                free(s);
                s = NULL;
            }
        } else {
            /* failed! */
            free(s)->sub);
            free(s);
            s = NULL;
        }
    } else {
        /* failed! */
        free(s);
        s = NULL;
    }
    return s;
}
```

```c
state_t *initialize()
{
    /* allocate state struct */
    state_t *s = g_new0(state_t, 1);
    if (s == NULL) goto free0;

    /* allocate sub-structure */
    s->sub = g_new0(sub_t, 1);
    if (s->sub == NULL) goto free1;

    /* open file */
    s->sub->fd =
        open("/dev/null", O_RDONLY);
    if (s->sub->fd >= 0) {
        /* success! */
    } else {
        free0:
        free(s->sub);
        free1:
        free(s);
        s = NULL;
    }
    return s;
}
```
Conclusions

• We took a quick look at the different features of C

• To get deeper look: check online tutorials. You will find some links at the course webpage

• To become an expert: write code ... write code ... write code