DTS202TC Fundamental of Parallel Computing

Lecture 2: C Programming

Hongbin Liu

Xi'an Jiaotong-Liverpool University
西交利物浦大学

Administrations

- Your assessment group information should be submitted via LMO before 13 Nov, (otherwise will be randomly assigned to a group)
- Go to lab this Friday.
- Some clarifications on the Assessment 1.

Agenda

- Backgrounds
- Functions and Pointers
- Array
- C Strings
- Input/Output
- Structs
- Memory Allocation

History of C

- General-purpose computer programming language created in the 1970s at Bell Labs.
- C was originally developed to construct utilities running on Unix, and was applied to re-implementing the kernel of the Unix system.
- ANSI C, C99 and C11 etc.

Why C for this Module?

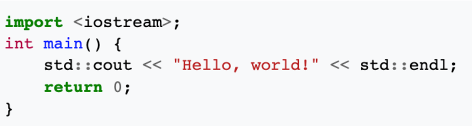
- Widely used in HPC/Parallel Computing
- Simple syntax
- Best performance

Hello World

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Spot the Differences

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```



```
import <iostream>;
int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Differences between C and C++

C	C++
C supports procedural programming paradigm	C++ supports both procedural and object oriented programming paradigms
C uses functions for input/output. For example **scanf** and **printf**.	C++ uses objects for input output. For example **cin** and **cout**.
C provides **malloc**() and **calloc**() functions for dynamic memory allocation, and **free**() for memory deallocation.	C++ provides **new** operator for memory allocation and **delete** operator for memory de-allocation.

Compiling

- *Compiler Tool*
  - – gcc (GNU Compiler)
  - – icc (Intel C compiler)
- Compiling/Building process: gcc hello.c –o hello
  - – *Command: gcc <options> <source_file.c>*
  - – Options:
    - *-Wall: Shows all warnings*
    - *-o output_file_name*
    - *-g: Include debugging information in the binary.*

Makefile

```
hello:
    gcc -g -Wall hello.c -o hello
pth:
    gcc -g -Wall hello_pth.c -o hello_pth
clean:
    rm -f hello hello_pth
```

Writing C Programs

- Many C developers use a text editor and a terminal to write their programs.
- For beginners, IDE would be a better choice, e.g. Clion
- There is a tutorial for setting up the IDE and C compiler (on Virtual Machine).

Testing Your Code

- **Very Important**: compile and test your code on the Ubuntu Virtual Machine before submitting your assessments
- Do not use Visual C++ for this module
- Be very careful with Microsoft Visual Studio

Agenda

- Backgrounds
- **Functions and Pointers**
- Array
- C Strings
- Input/Output
- Structs
- Memory Allocation

C Functions

- Functions are defined in C like this:

```
<return type> <function name>(<argument list>) {
    …
}
```

Passing by Value

- In C, everything is passed by value
- This menas that when you call a function, e.g.:

```
location(2, 4);
```

- Copies will be made of 2 and 4 and passed to the location function
- Changing these values inside the function doesn’t have an impact elsewhere

  They are internal to the function

Passing by Reference

- Sometimes we actually do want to change the value of a variable when it’s passed into a function:

```
int a = 3;
int b = 8;
printf(“%d, %d\n”, a, b);
Swap(a, b)
printf (“%d, %d\n”, a, b);
```

Prints:

3,8

8, 3

Passing by Reference

- We need to pass by reference
- In C, we accomlish this by passing in the meory address of the variable:
  - The address is passed by value
  - We can use the address to find the variable in memory and change it
- If you ever heard of pointers in C, this is what they’are used for.

New Syntax

- & the ‘address of’ operator. When a function takes a pointer as an argument, you need to give it an address, not the value of the variable
- int *x_p; defining a pointer. Note that this doesn’t create an integer, it creates a pointer to an integer.
- Finally when accessing a variable, *x_p is the dereference operator – it follows the address and looks up the actual value being pointed to

Demo

Agenda

- Backgrounds
- Functions and Pointers
- **Array**
- C Strings
- Input/Output
- Structs
- Memory Allocation

Array

- In C, arrays let us store a collection of values of the same type

- Creating an array:

```
int a_array[10];
double b_list[15] = {0};
```

- Accessing array

```
a_array[1] = 20;
b_list[2] = b_list[1] + 2;
```

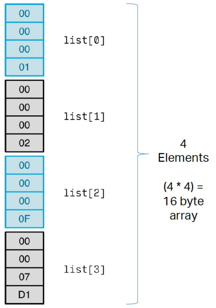
How about `a_array[10]`? Will it crash?

Memory Access

- What happens when you retrieve the value of list[5]?
  1. Find the location of list in the memory
  2. Move to the proper offset:
     5 * 4 = 20 byte
  3. Access the value

Visualizing Arrays in Memory

```
int list[] = {
      1,
      2,
      15,
      2001
    };
```



Behind the Scenes

- Arrays in C are actually pointers

  `int list[3];`

  `list` is the same as `&list[0];`

- There is another way to think of it

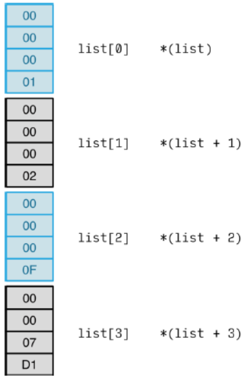
  `list[3]` is the same as `*(list + 3)`
  - Locate the start of the array
  - Move up 3 memory locations (4 bytes each*)
  - Dereference the pointer to get our value

Pointer Arithmetic

- Manipulating pointers in this way is called pointer arithmetic
- array[i] is the same as *(array + i);
- array[6] = 42 is the same as *(array + 6) = 42

Visualizing Arrays Again

```
int list[] = {
      1,
      2,
      15,
      2001
    };
```



Arrays as Function Arguments

- When we pass an array to a function, we are essentially passing the pointer to the function
- If we modify an array element inside of a function, will the change be replected in the calling function?
  - Why?
- In fact, when an array is pased to a function it decays to a pointer
  - The function just receives a pointer to the first element in the array. That’s it.

Array Decay

- When an array decays to a pointer, we lose some information
  - **Type** and **dimension**
- Let’s imagine someone just gives us a pointer
  - Do we know if it points to a single value?
  - Is it the start of an array?
- Functions are in the same situation: they don’t know where this pointer came from or where it’s been
  - **sizeof** doesn’t work as expected.

Avoiding Decay

- decay.c:6:39: warning: sizeof on array function parameter will return size of 'int *' instead of 'int[4]' [-Wsizeof-array-argument]

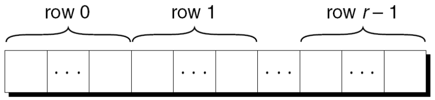
  ```
  printf("Size of array: %d", sizeof(array));
  ```

- To avoid this situation, we need to pass in the size of the array as well.
- You may have wondered why the sizes of arrays are always being passed around in C code
  - This is why.

Demo

Multidimensional Array

- `int a[NUM_ROWS][NUM_COLS];`
- Layout of an array with *r* rows:



- If `p` initially points to the element in row 0, column 0, we can visit every element in the array by incrementing `p` repeatedly.

Processing the Elements of a Multidimensional Array

- The obvious technique would be to use nested `for` loops:

  ```
  int row, col;
  …
  for (row = 0; row < NUM_ROWS; row++)
    for (col = 0; col < NUM_COLS; col++)
      a[row][col] = 0;
  ```

- If we view `a` as a one-dimensional array of integers, a single loop is sufficient:

  ```
  int *p;
  …
  for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
    *p = 0;
  ```

Processing the Rows of a Multidimensional Array

- A loop that clears row `i` of the array `a`:

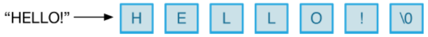
```
int a[NUM_ROWS][NUM_COLS], *p, i;
…
for (p = a[i]; p < a[i] + NUM_COLS; p++)
  *p = 0;
```

Agenda

- Backgrounds
- Functions and Pointers
- Array
- **C Strings**
- Input/Output
- Structs
- Memory Allocation

C Strings

- Let’s look at a C string:



“HELLO!” → H E L L O ! \0

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0’};
```

Or simply:

```
char greeting[] = "Hello";
```

What is the use of NUL?

- First, the presence of the NUL byte indicates a string rather than just a plain old array of characters
- As we know, we can’t always reliably determine how large an array is uless we keep track of its size
  - Array decay
- NUL allows the string manipulation functions to determine where the string ends

The C String Library

- #include <string.h>
- strcpy – copy one string to another
- strcat – concatenate two strings
- strcmp – test for string equality
- strlen – returns the length of the string (ignoring \0)
- strtok – tokenize the string (split it up)

Copying a String

- Let’s say you want to copy one string into another

```
char str1[] = “Hello World!”;
char *str2 = str1;
```

- This does not make a copy; str2 just points to str1.

Copying a String – Cont.

- We could loop through the array and copy each character into the other, but that is a lot of work
- Better solution: strcpy:

```
char str1[] = "Hello World!";
char str2[12];
strcpy(str2, str1);
printf("%s\n", str2);
```

But wait, this code has a big problem, array size.

Copying a String – Cont.

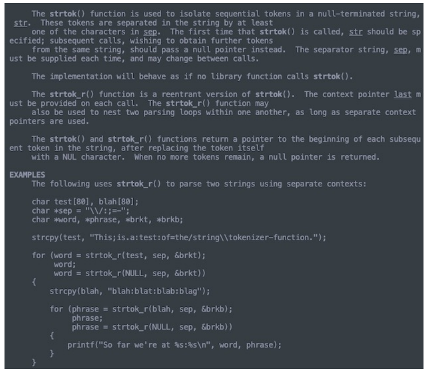
- Let’s fix our bug:

```
char str1[] = "Hello World!";
char str2[13];
strcpy(str2, str1);
printf("%s\n", str2);
```

- We could also create a much larger array to copy into
  - *– strcpy will go ahead and fill the rest with \0*

C Function Documentation

- Unix has a utility called man – short for manual
- There are several selections of man pages:
  - User Commands: e.g. rm, move etc
  - C Library Functions: e.g. strtok
  - others



```
     The strtok() function is used to isolate sequential tokens in a null-terminated string,
 str.  These tokens are separated in the string by at least
     one of the characters in sep.  The first time that strtok() is called, str should be sp
ecified; subsequent calls, wishing to obtain further tokens
     from the same string, should pass a null pointer instead.  The separator string, sep, m
ust be supplied each time, and may change between calls.

     The implementation will behave as if no library function calls strtok().

     The strtok_r() function is a reentrant version of strtok().  The context pointer last m
ust be provided on each call.  The strtok_r() function may
     also be used to nest two parsing loops within one another, as long as separate context
pointers are used.

     The strtok() and strtok_r() functions return a pointer to the beginning of each subsequ
ent token in the string, after replacing the token itself
     with a NUL character.  When no more tokens remain, a null pointer is returned.

EXAMPLES
     The following uses strtok_r() to parse two strings using separate contexts:

     char test[80], blah[80];
     char *sep = "\\/:;=-";
     char *word, *phrase, *brkt, *brkb;

     strcpy(test, "This;is.a:test:of=the/string\\tokenizer-function.");

     for (word = strtok_r(test, sep, &brkt);
          word;
          word = strtok_r(NULL, sep, &brkt))
     {
         strcpy(blah, "blah:blat:blab:blag");

         for (phrase = strtok_r(blah, sep, &brkb);
              phrase;
              phrase = strtok_r(NULL, sep, &brkb))
         {
             printf("So far we're at %s:%s\n", word, phrase);
         }
     }
```

Example of man strtok

Agenda

- Backgrounds
- Functions and Pointers
- Array
- C Strings
- **Input/Output**
- Structs
- Memory Allocation

Input/Output

- Most useful programs will provide some type of input or output.
- E.g. getting users’ keyboard input, printing message to screen, writing out to files.

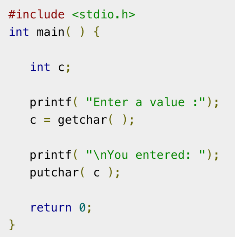
The Standard Files

- C programming treats all the devices as files

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	Stderr	Your screen

Basic input/output

- getchar() and putchar()
  - int getchar(void) function reads the next available character from the screen and returns it as an integer
  - int putchar(int c) function puts the passed character on the screen and returns the same character



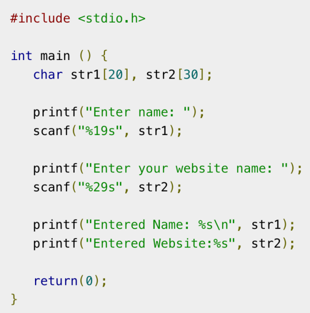
```
#include <stdio.h>
int main( ) {

   int c;

   printf( "Enter a value :");
   c = getchar( );

   printf( "\nYou entered: ");
   putchar( c );

   return 0;
}
```

Similarly,
char *gets(char *s)
int puts(const char *s)
Read write a line

https://www.tutorialspoint.com/cprogramming/c_input_output.htm

Basic input/output

- The C library function int **scanf(const char *format, ...)** reads formatted input from stdin.



```
#include <stdio.h>

int main () {
   char str1[20], str2[30];

   printf("Enter name: ");
   scanf("%19s", str1);

   printf("Enter your website name: ");
   scanf("%29s", str2);

   printf("Entered Name: %s\n", str1);
   printf("Entered Website:%s", str2);

   return(0);
}
```

Command Line Arguments

- Passing command line arguments is a common form of input:
  `./my_program train training_set_path`

- We see this often with Unix utilities:
  `ls –l /my/directory`

- In c, there is an alternative version of the main(void) function:
  `int main(int argc, char *argv[])`

Argument Attributes

- We receive two parameters:
  - **argc** the number of command line arguments
  - **argv** the arguments themselves

- Note,
  - argc will always be at least 1
  - argv will always start with the name of your program

Processing Arguments

- Command line arguments are C strings
  - They are terminated by \0
- So, we can do a string comparison:
  `strcmp(argv[1], “status”)`
- What if we want to accept an integer from the command line?

Converting Arguments

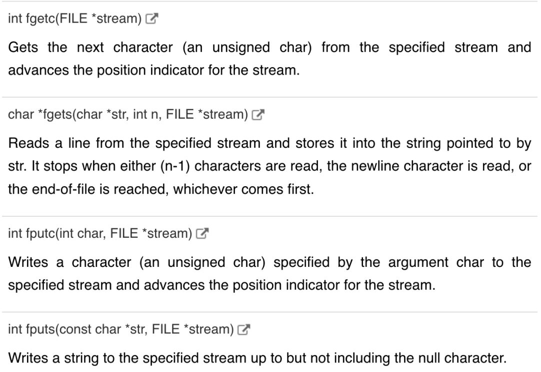
- In many cases, we want to accept an integer from the command line
- Converting a string to integer is accomplished with the `atoi()` function
  - Available in the C standard library `#include <stdlib.h>`
- Similarly, use atof() to convert a string to float, atol() to convert a string to long

File IO

```
/* This opens the file specified by the
   first command line argument: */
printf("Opening file: %s\n", argv[1]);
FILE *file = fopen(argv[1], “mode”);
```

- `r` – read
- `w` – write, create new file if does not exist
- `a` – append, create new file if does not exist
- `r+` - both reading and writing
- `w+` - both reading and writing, create new file if not exist
- `a+` - both reading and writing, reads from beginning, writing as appended

Reading and Writing to a File



int fgetc(FILE *stream)

Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.

char *fgets(char *str, int n, FILE *stream)

Reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

int fputc(int char, FILE *stream)

Writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream.

int fputs(const char *str, FILE *stream)

Writes a string to the specified stream up to but not including the null character.

https://www.tutorialspoint.com/c_standard_library/stdio_h.htm

Cleaning Up

- It is good practice to also close your files when you are done with them:
  - – fclose(file)
- Each file you open uses up a file descriptor
  - – The operating system limits on how many file descriptors can be open per program

File IO Demo

```
FILE *f = fopen("test.txt", "r");
int digit, count;
while ((count = fgetc(f)) != EOF) {
    ungetc(count, f); // what does this do?

    fscanf(f, "%d", &digit);
    printf("%d \n", digit);
}
fclose(f);
```

Agenda

- Backgrounds
- Functions and Pointers
- Array
- C Strings
- Input/Output
- **Structs**
- Memory Allocation

Structs

- C structs allow us to create groups of data
  - – Do not have to be all the same type like arrays
- These structures can contain multiple variables
- With structs, we can implement something similar to object-oriented programming
  - – However, rather than embedding data and methods, structs only contain data

Defining a Struct (1/2)

```
struct USER {
  int account_number;
  char *first_name;
  char *last_name;
};
```

Defining a Struture (2/2)

- Create a new type

```
typedef struct {
  int account_number;
  char *first_name;
  char *last_name;
} USER;
```

Creating a Struct

- USER user1;
- USER user1, *user2;

Direct Member Access

- User dot notation:

```
user1.account_number  = 111;
user1.first_name = "Matthew";
```

Indirect Member Access

```
void check_account(USER *user1) {
  user1->account_number = 100;
  printf("%s\n", user1->first_name);
}

/* Equivalent: */
(*user1).account_number = 100;
```

Agenda

- Backgrounds
- Functions and Pointers
- Array
- C Strings
- Input/Output
- Structs
- **Dynamic Memory Allocation**

Dynamic Memory Allocation

- You may have wondered why we often set up our arrays with a fixed size
- For example, char line[10]
- This simplifies programming in C
- What if you need bigger size?

Dynamic Memory Allocation

- void * malloc (size_t size)
  - Allocate contiguous blocks of memory

```
#include <stdlib.h>

int *array = malloc(sizeof(int));
```

Dynamic Memory Allocation

- void *calloc(size_t num, size_t element_size)
  - this also allocate memory, the difference is calloc initialises the memory to zero before returning the pointer
- void *realloc(void *ptr, size_t new size)
  - resize the previously allocated block of memory

Freeing Memory: free()

- Dynamically allocated memory must be freed when it is no longer needed.
  - otherwise, you are creating memory leak

```
#include <stdlib.h>

int *array = malloc(sizeof(int));

if (array == NULL) {
   printf("Can't allocate memory");
}
//do some other staff with array.
free(array); // free the memory
```

Wrap up

- We have not covered and could not cover everything in C
- If you need to use something else, search keyword using man command
- There are many other standard libraries in C, such as <time.h>, <stddef.h>, <math.h> and many more
- Recommended C textbook
  - C Programming: A Modern Approach, Second Edition, K. N. King

Wrap up (cont.)

- Start programming (trying)
  - Start with basics
  - Keep improving
- Learn from others (textbook examples, open-source code, documentations)
- Start learning a new language is easy, master it is hard
  - Practice, practice, practice

Next Week

- Shared memory programming using Pthreads by Dr. Maruf