JTSK-320111

Programming in C I

C-Lab I

Lecture 5 & 6

Xu He

Fall 2018

Slides modified from Dr. Kinga Lipskoch

This Week's Agenda

- Pointers and arrays
- Dynamic memory allocation
- Multi-dimensional arrays
- Recursive functions
- Dealing with larger projects
- Handling files
- Revision

Passing Arrays to Functions

- An array does not store its size
- This has to be provided as a parameter, or by making assumptions on the contents of the array (like for strings)
- When an array is passed to a function, a copy of the address of the first element is given
- Modifications to the elements are seen outside
- Modifications to the array are not seen outside
- Can you explain why?

Passing Arrays to Functions: Example

```c
#include <stdio.h>
#include <stdlib.h>
void strange_function(int v[], int dim) {
  int i;
  for (i = 0; i < dim; i++)
    v[i] = 287;
  // v = (int *) malloc(sizeof(int) * 1000);
}
int main() {
  int array[] = {1, 2, 9, 16};
  int *p = &array[0];
  strange_function(array, 4);
  printf("%d %x %x\n", array[0], p, array);
  return 0;
}
```

Dynamic Memory Allocation

- What if we do not know the dimension of the array while coding?
- Dynamic memory allocation allows you to solve this problem
    - And many others
    - But can also cause a lot of troubles if you misuse it

Pointers and Arrays

There is a strong relation between pointers and arrays

- Indeed an array is nothing but a pointer to the first element in the sequence
- We are looking at this in detail

Specifying the Dimension on the Fly

To specify the dimension on the fly you can use the `malloc()` function defined in the header file `stdlib.h`

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *dyn_array, how_many, i;
  printf("How many elements? ");
  scanf("%d", &how_many);
  dyn_array =
    (int*) malloc(sizeof(int) * how_many);
  if (dyn_array == NULL)
    exit(1);
  for (i = 0 ; i < how_many; i++) {
    printf("\nInput number %d:", i);
    scanf("%d", &dyn_array[i]);
  } return 0;
}
```

The `malloc()` Function (1)

- `void * malloc(unsigned int);`
- `malloc` reserves a chunk of memory
- The parameter specifies how many bytes are requested
- `malloc` returns a pointer to the first byte of such a sequence
- The returned pointer must be forced (cast) to the required type

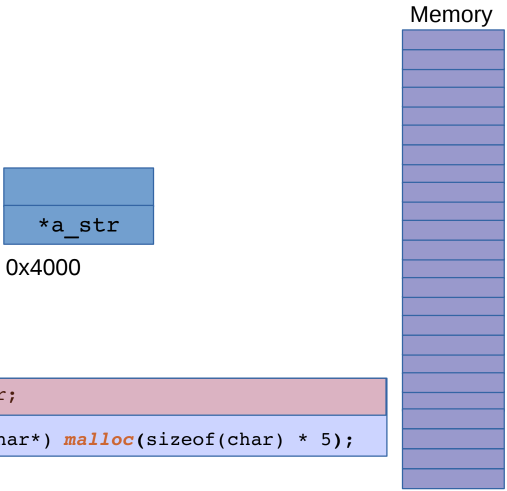
The malloc() Function (2)

```
pointer   = (cast) malloc(number of bytes);


char* a_str;
a_str = (char*) malloc(sizeof(char) * how_many);
```

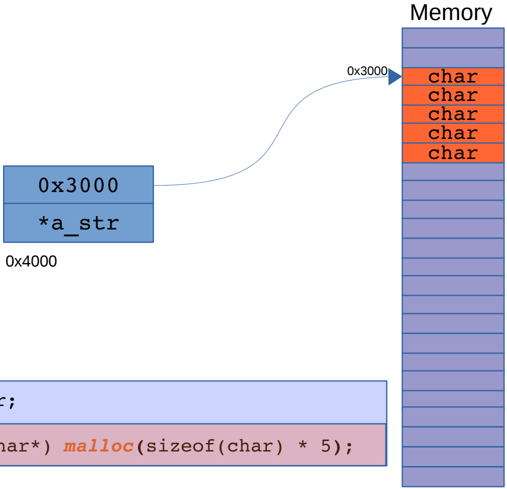
- malloc returns a void * pointer (i.e., a generic pointer) and this is assigned to a non void * pointer
- If you omit the casting you will get a warning concerning a possible incorrect assignment

Dynamically Allocating Space for an Array of `char`



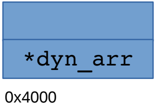

```
char *a_str;
a_str = (char*) malloc(sizeof(char) * 5);
```

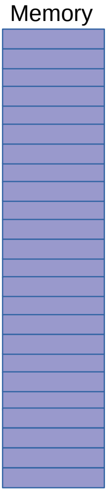
Dynamically Allocating Space for an Array of `char`




```
char *a_str;
a_str = (char*) malloc(sizeof(char) * 5);
```

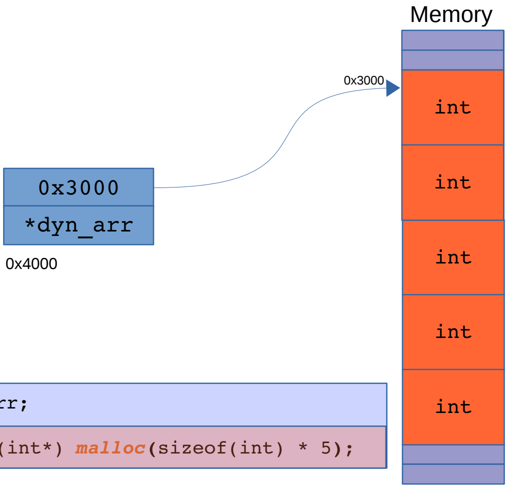
Dynamically Allocating Space for an Array of `int`






```
int *dyn_arr;
dyn_arr = (int*) malloc(sizeof(int) * 5);
```

Dynamically Allocating Space for an Array of `int`




```
int *dyn_arr;
dyn_arr = (int*) malloc(sizeof(int) * 5);
```

malloc() and free()

- All the memory you reserve via `malloc`, must be released by using the `free` function
- If you keep reserving memory without freeing, you will run out of memory

```
float *ptr;
int number;
...
ptr = (float*) malloc(sizeof(float) * number);
...
free(ptr);
```

Rules for `malloc()` and `free()`

- The following points are up to you (the compiler does not perform any control)
  1. Always check if `malloc` returned a valid pointer (i.e., not `NULL`)
  2. Free allocated memory just once
  3. Free only dynamically allocated memory
- Not following these rules will cause endless troubles
- `sizeof()` is compile time operator, it does not work on allocated memory

Review: Pointers, Arrays, Values

```
#include <stdio.h>
#include <stdlib.h>
int length[2] = {7, 9};
int *ptr1, *ptr2; int n1, n2;
int main() {
  ptr1 = &length[0];
  // &length[0] is pointer to first elem
  ptr2 = length;
  // length is pointer to first elem therefore
  // same as above
  n1 = length[0];
  // length[0] is value
  n2 = *ptr2;
  // *ptr2 is value therefore same as above
  printf("ptr1: %p, ptr2: %p\n", ptr1, ptr2);
  printf("n1: %d, n2: %d\n", n1, n2);
  return 0;
}
```

Multi-dimensional Arrays

- It is possible to define multi-dimensional arrays
  - Mostly used are bidimensional arrays, i.e., tables or matrices
- As for arrays, to access an element it is necessary to provide an index for each dimension
  - Think of matrices in mathematics

Multi-dimensional Arrays in C

- It is necessary to specify the size of each dimension
  - Dimensions must be constants
  - In each dimension the first element is at position 0

```
int matrix[10][20];    /* 10 rows, 20 cols */
float cube[5][5][5];   /* 125 elements */
```

- Every index goes between brackets

```
matrix[0][0] = 5;
```

Multi-dimensional Arrays in C: Example

```c
#include <stdio.h>
int main() {
  int table[50][50];
  int i, j, row, col;
  scanf("%d", &row);
  scanf("%d", &col);
  for (i = 0; i < row; i++)
    for (j = 0; j < col; j++)
      table[i][j] = i * j;
  for (i = 0; i < row; i++)
  {
    for (j = 0; j < col; j++)
      printf("%d ", table[i][j]);
    printf("\n");
  }
  return 0;
}
```

The `main` Function (1)

- Can return an `int` to the operating system
  - Program exit code (can be omitted)
  - print exit code in shell: `$> echo $?`
- Can accept two parameters:
  - An integer (usually called `argc`)
  - A vector of strings (usually called `argv`)
  - `argc` specifies how many strings contains `argv`

The main Function (2)

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  for (i = 1; i < argc; i++)
    printf("%d %s\n", i, argv[i]);
  return 0;
}
```

- Compile it and call the executable `paramscounter`
- Execute it as follows:

  `$> ./paramscounter first what this`
- It will print first, what and this, one word per line
- Note that argc is always greater or equal than one
- The first parameter is the program's name

The `const` Keyword

- The modifier `const` can be applied to variable declarations
- It states that the variable cannot be changed
  - i.e., it is not a variable but a constant
- When applied to arrays it means that the elements cannot be changed

const Examples

```
const double e = 2.71828182845905;
const char str[]= "Hello world";
e = 3;            /* error */
str[0] = 'h';     /* error */
```

- You can also use `#define` of the preprocessor
- But defines do not have type checking, while constants do

More const Examples

- `const char *text = "Hello";`
  - Does not mean that the variable `text` is constant
  - The data pointed to by `text` is a constant
  - While the data cannot be changed, the pointer can be changed
- `char *const name = "Test";`
  - `name` is a constant pointer
  - While the pointer is constant, the data the pointer points to may be changed
- `const char *const title = "Title";`
  - Neither the pointer nor the data may be changed

Recursive Functions (1)

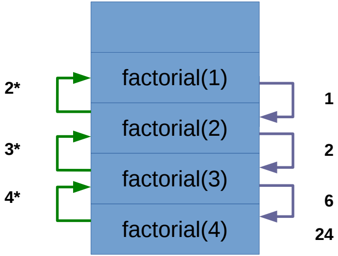
- Can a function call other functions?
  - Yes, indeed function calls appear only inside other functions (and everything starts with the execution of `main`)
- Can a function call itself?
  - Yes, but in this case special care should be taken
- A function which calls itself is called a **recursive function**
- Function A calls function A
- At a certain point function B calls A
  - A calls A then A calls A then A calls A ...
- When coding recursive functions attention should be paid to avoid endless recursive calls

Recursive Functions (2)

- Recursion theory can be studied for a longer time: here we will just scratch its surface from a basic coding standpoint
- Every recursive function must contain some code which allows it to terminate without entering the recursive step
  - Usually called **inductive base** or **base case**
- When recursion is executed, the new call should be driven "towards the inductive case"

Stack of Calls: Example

```
int factorial(int n) {
  if ((n == 0) || (n == 1))
    return 1;
  else
    return n * factorial(n - 1);
}
```



Tracing the Stack of Calls (1)

```
int factorial(int n) {
  int val;
  if ((n == 0) || (n == 1)) {
    printf("base\n");
    return 1;
  } else  {
    printf("called with par = %d\n", n);
    val = n * factorial(n - 1);
    printf("returning %d\n", val);
    return val;
  }
}
int main() {
  printf("%d\n", factorial(4));
  return 0;
}
```

Tracing the Stack of Calls (2)

From the `main`: call `factorial(4)`

factorial(1): n = 1, printf(“base”), return 1
factorial(2): n = 2, printf(2), val = 2 * factorial(1), printf(val), return val
factorial(3): n = 3, printf(3), val = 3 * factorial(2), printf(val), return val
factorial(4): n = 4, printf(4), val = 4 * factorial(3), printf(val), return val

~~factorial(1): n = 1,~~ printf(“base”), return 1
~~factorial(2): n = 2, printf(2), val = 2 * 1, printf(2)~~, return 2
~~factorial(3): n = 3, printf(3), val = 3 * 2, printf(6), return~~ 6
~~factorial(4): n = 4, printf(4), val = 4 * 6, printf(24), return~~ 24

One More Example: Fibonacci Numbers

$$F(N) = \begin{cases} 1, & N \leq 1 \\ F(N-1) + F(N-2), & N > 1 \end{cases}$$

```
int fibonacci(int n) {
  if ((n == 0) || (n == 1))
    return 1;
  else
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Dealing with Big Projects

- Functions are a first step to break big programs in small logical units
- A further step consists in breaking the source into many files
  - Smaller files are easy to handle
  - Objects sharing a context can be put together and easily reused
- C allows to put together separately compiled files to have one executable

Declarations and Definitions

- **Declaration**: introduces an object. After declaration the object can be used
  - Example: functions' prototypes
- **Definition**: specifies the structure of an object
  - Example: function definition
- Declarations can appear many times, definitions just once

Building from Multiple Sources

- C compilers can compile multiple sources files into one executable
- For every declaration there must be one definition in one of the compiled files
  - Indeed also libraries play a role
  - This control is performed by the linker
- `gcc -o name file1.c file2.c file3.c`

Libraries

- Libraries are collection of compiled definitions
- You include header files to get the declarations of objects in libraries
- At linking time libraries are searched for unresolved declarations
- Some libraries are included by `gcc` even if you do not specifically ask for them

Linking Math Functions: Example

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

int main() {
  double n;
  double  sn;

  scanf("%lf\n", &n); /* double needs %lf */
  sn = sqrt(n);
  /* conversion from double to float ok */
  printf("Square root of %f is %f\n", n, sn);
  return 0;
}

    gcc -lm -o compute compute.c
```

Compilers, Linkers and More

- Different compilers differ in many details
  - Libraries names, ways to link against them, types of linking
- Check your documentation
- But preprocessing, compilation and linking are common steps

File Handling in C

- Input and output can come from/go into files
- C treats files as streams of data
- A stream is a sequence of bytes (either incoming or outgoing)
- The language does not provide basic constructs for file handling, but rather the standard library does

Files, C and UNIX

- The file view of C is influenced by UNIX
- UNIX sees everything as a file
- You have already used two files/streams
  - `stdin` (standard input): associated with the keyboard
  - `stdout` (standard output): associated with the screen
  - These files are always tied to your program by the operating system

Working with Files

- The paradigm is the following:
  - open the file
  - read/write from/into file
  - close the file
- In C the information concerning a file are stored in a `FILE` structure (i.e., `struct`) defined in `stdio.h`

File Modes

Streams can be handled in two modes: (only important for MS Windows)

- Text streams: sequence of characters logically organized in lines. Lines are terminated by a newline ('`\n`')
  - Sometimes pre/post processed
  - Example: text files
- Binary streams: sequence of raw bytes
  - Examples: images, mp3, user defined file formats, etc.

Opening a File

- To open a file the `fopen` function is used
  ```
  FILE *fopen(const char * name, const char * mode)
  ```
- `name`: name of the file (OS level)
- `mode`: indicates the type of the file and the operations that will be performed
  ```
  FILE *fptr;
  fptr = fopen("myfile.txt", "r");
  ```

Mode Strings

String	Meaning
`"r"`	Open for reading, positions at the beginning
`"r+"`	Open for reading and writing, positions at the beginning
`"w"`	Open for writing, truncate if exists, positions at the beginning
`"w+"`	Open for reading and writing, truncate if exists, positions at the beginning
`"a"`	Open for appending, does not truncate if exists, positions at the end
`"a+"`	Open for appending and reading, does not truncate if exists, positions at the end

A `b` or a `t` can be added to indicate it is a binary/text file

Closing a File

- `int fclose(FILE *fp);`
- Forgetting to close a file might result in a loss of data
- After a file is closed it is not possible anymore to read/write

```
FILE *fptr;
fptr = fopen("myfile.txt", "r");
if (fptr == NULL) {
  printf("Some error occurred!\n");
  exit(1);
}
...
/* do some operations */
fclose(fptr);
...
```

Reading/Writing

Prototype	**Use**
`int getc(FILE *fp)`	Returns next `char` from `fp`
`int putc(int c, FILE *fp)`	Writes a `char` to `fp`
`int fscanf(FILE* fp, char * format, ...)`	Gets data from `fp` according to the format string
`int fprintf(FILE* fp, char * format, ...)`	Outputs data to `fp` according to the format string

Line Input and Line Output

```
char *fgets(char *line, int max, FILE *fp);
```

- Already seen with `stdin`
- Used for files as well

```
int fputs(char *line, FILE *fp);
```

- Outputs/writes a string to a file

Files: Example 1

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  char ch;
  FILE *fp;
  fp = fopen("file.txt", "r");
  if (fp == NULL) {
    printf("Cannot open file!\n");
    exit(1);
  }
  ch = getc(fp);
  while (ch != EOF) {
    putchar(ch);
    ch = getc(fp);
  }
  fclose(fp);
  return 0;
}
```

Files: Example 2

```c
# include <stdio.h>
# include <stdlib.h>
int main () {
  char ch;
  FILE * fp;
  fp = fopen("file.txt", "r") ;
  if (fp == NULL) {
    printf("Cannot open file!\n");
    exit(1);
  }
  while((ch=getc(fp))!=EOF) {
    putchar(ch);
  }
  fclose(fp);
  return 0;
}
```

Files: Example 3

```c
# include <stdio.h>
# include <stdlib.h>
int main () {
  char ch;
  FILE * fp;
  fp = fopen("file.txt", "r") ;
  if (fp == NULL) {
    printf("Cannot open file!\n");
    exit(1);
  }
  while(!feof(fp)) {
    ch=getc(fp);
    if (ch!=EOF)
      putchar(ch);
  }
  fclose(fp);
  return 0;
}
```

Revisiting: Casting

- It is possible to overcome standard conversions (casting)
- To force to a different data type, put the desired data type before the expression to be converted
  `(type name) expression`
- Casting is a unary operator with high precedence

Revisiting: Casting – Example

```
int a;
float f1 = 3.456, f2 = 1.22;
/* these operations imply demotions */
a = (int) f1 * f2;     /* a is now 3 */
a = (int) (f1 * f2);   /* a is now 4 */
```

- You have already used casting when using `malloc`. `malloc` returns a `void *` pointer (i.e., a generic pointer) and this is assigned to a non `void *` pointer
- If you omit the casting you might get a warning concerning a possible incorrect assignment

Revisiting: String Functions

- Defined in `string.h`
- `strlen` determines the length of a string
- `strcat` concatenates two strings
- `strcpy` copies one string into another
- `strcmp` compares two strings
- `strchr` searches a char in a string
- `strdup` duplicates a string
- See man pages (`man 3 string`) or section B3 in the Kernighan & Ritchie book

Revisiting: `void *`

- `void *` is a generic pointer holding a memory address
  - `malloc` returns a `void *`, thus the need for a cast
- Every pointer can be assigned to a `void *` pointer and vice versa, without explicit casts
  - This can create big problems

Revisiting: Misusing `void*`

```c
#include <stdio.h>
int main(void) {
  void * vp;     /* a generic pointer */
  int * ip;
  float f = 1.234;
  float * fp = &f;
  vp = fp;
  ip = vp;
  /*  float * assigned to int *
      via a generic pointer
      this will not work correctly ...
  */
  printf("%d\n", *ip);
  /* outputs some strange number */
  return 0;
}
```

Examples Revisited (1)

```
char a_string[] = "This is a string\0";
char *p;
int count = 0;
int main() {
  printf("%s\n", a_string);
  for (p = &a_string[0]; *p != '\0'; p++)
    count++;
  printf("The string has %d chars\n", count);
  p--;
  printf("Printing the reverse string:\n");
  while (count > 0) {
    printf("%c", *p);
    p--;
    count--;
  }
  return 0;
}
```

Examples Revisited (2)

To specify the dimension on the fly you can use the `malloc` function defined in the header file `stdlib.h`

```
...
int *dyn_array, how_many, i;
printf("How many elements? ");
scanf("%d", &how_many);
dyn_array = (int*) malloc(sizeof(int) *
  how_many);
if (dyn_array == NULL)
  exit(1);
for (i = 0 ; i < how_many; i++) {
  printf("\nInput number %d:", i);
  scanf("%d", &dyn_array[i]);
}
...
```

Examples Revisited (3) – Reading from the Keyboard

```c
#include <stdio.h>
int main() {
  int v;
  char str[30];
  char line[80];
  printf("Enter a string: ");
  fgets(line, sizeof(line), stdin);
  sscanf(line, "%s", str);  // not really needed
  // just read str directly

  printf("Enter a number: ");
  fgets(line, sizeof(line), stdin);
  sscanf(line, "%d", &v);
  printf("String: %s\n", str);
  printf("Number: %d\n", v);
  return 0;
}
```

Conversion Specification for `printf()`

Conversion	**Meaning**
`%c`	single character
`%d`	signed decimal integer
`%f`	double (also float)
`%e`	floating point (exponential notation)
`%s`	string (pointer needs to be passed)

Conversion Specification for `scanf()`

Conversion	**Meaning**
as above	
`%f`	float (decimal notation)
`%lf`	double (decimal notation)

Final Exam: Details

- Saturday, 13^{th} of October, 2018, Conference Hall, IRC, $10:00$ - $12:00$
- Programming exercises to be solved on paper
  - You have two hours to solve exercises
  - Similar to the programming assignments
  - Practice to write your programs on paper
- You do not need paper, it will be provided
- You may not use books or other documentation while taking the exam
- You may not use mobile phones, calculators or any other electronic devices
- Tutorial will be given by the TAs a few days before the exam