Computer Systems Design

Lesson 6

Basics of C programming

Alexander Antonov, Assoc. Prof., ITMO University

Hangzhou, 2025

Outline the lesson

- History
- Data types
- Operations
- Control flow
- Pointers
- Common limitations

History of C

C - general-purpose, procedural computer programming language

Originally developed in Bell Labs by Dennis Ritchie around ***1972*** for ***PDP-11*** computer programming

Standardized by **ANSI** since 1989 (ANSI C) and **ISO**

Commonly used nowadays for ***system*** programming (e.g. Linux) and ***embedded*** programming

Very close to hardware (sometimes designated as ***“cross-platform assembly”***)

Ken Thompson (sitting) and Dennis Ritchie working together at a PDP-11, 1972

“Hello World” in C

```c
// Write "Hello world!" to the console

#include <stdio.h>

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

Base data types

Can be assigned to variables (including function return values)

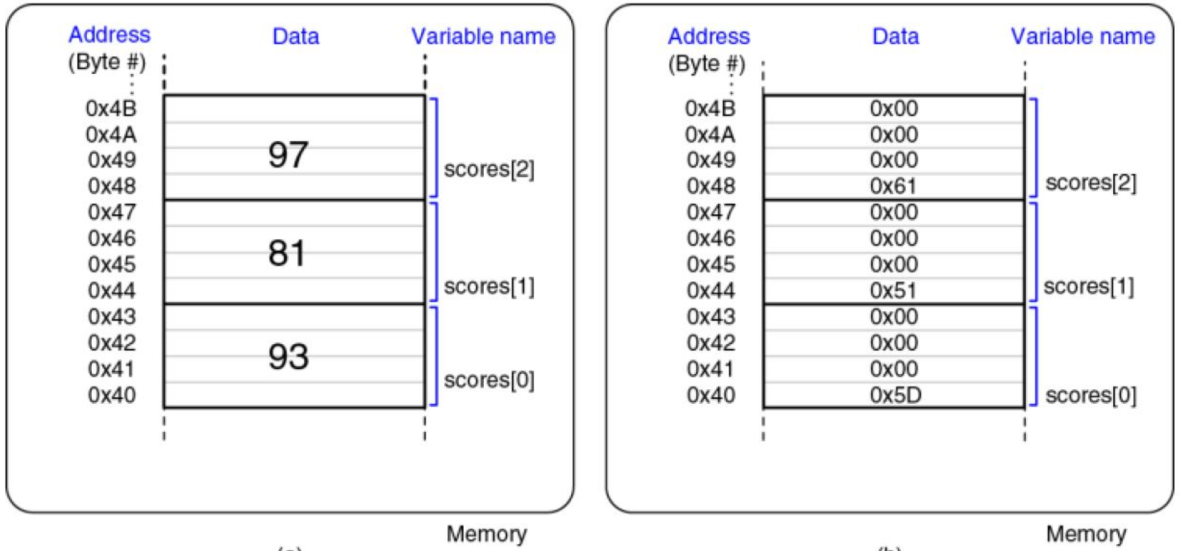
Data type	Memory (bytes)	Range	Format specifier
[signed] char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
[signed] short	2	-32,768 to 32,767	%hd
unsigned short	2	0 to 65,535	%hu
[signed] int	hardware-dependent	hardware-dependent	%d
unsigned int	hardware-dependent	hardware-dependent	%u
[signed] long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
[signed] long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
float	4	FLT_MIN to FLT_MAX	%f
double	8	DBL_MIN to DBL_MAX	%lf
long double	16	LDBL_MIN to LDBL_MAX	%Lf

Arrays

Array is a group of variables of the ***same type***, ***sequentially*** located in a ***contiguous*** area of memory.

```
long scores[3];

scores[0] = 93;
scores[1] = 81;
scores[2] = 97;
```



Watch out for out of range accesses
(no default protection from memory damages provided!)

Structures

Variable data types can be assembled in structures

```
struct contact {
    char name[30];
    int phone;
    float height; // in meters
};

struct contact c1;
strcpy(c1.name,"Ben Bitdiddle");
c1.phone = 7226993;
c1.height = 1.82;
```

iTMO

Base operations (1)

Category	Operation	Description	Example
Unary	`++`	increment	`++a; // a = a+1`
	`--`	decrement	`--x; // x = x-1`
	`++`	post-increment	`a++; // a = a+1`
	`--`	post-decrement	`x--; // x = x-1`
	`~`	bitwise not	`z = ~a;`
	`!`	logical not	`!x`
	`-`	unary negation	`y = -a;`
	`&`	getting address	`x = &y;`
	`(type)`	type casting	`x = (int)c;  // cast c to an int and assign it to x`
	`sizeof()`	getting type size	`long int y; x = sizeof(y);  // x = 4`
Additive	`+`	addition	`y = a + 2;`
	`-`	subtraction	`y = a - 2;`
Multiplicative	`*`	multiplication	`y = x *12;`
	`/`	division	`z = 9 / 3; // z = 3`
	`%`	remainder	`z = 5 % 2; // z = 1`

iTMO

Base operations (2)

Category	Operation	Description	Example
Shifts	<<	bitwise left	z = 5 << 2; // z = 0b00010100
	>>	bitwise right	x = 9 >> 3; // x = 0b00000001
Relational	==	equal	y == 2
	!=	not equal	x != 7
	<	less	y < 12
	>	more	val > max
	<=	less or equal	z <= 2
	>=	more or equal	y >= 10
Bitwise	&	and	y = a & 15;
	|	or	x && y
	^	exclusive or	y = 2 ^ 3;
Logical	&&	and	x && y
	||	or	x || y

Ternary	? :	conditional operator	y = x ? a : b;
			// if x is TRUE,
			// y=a, else y=b

Assignment	=	assignment	x = 22;
	<operation>=	assignment with operation	y += 3; // y = y + 3

ITMO

Control flow operations: if/else and switch

```c
// Assign amt depending on option

if (option == 1) {
    amt = 100;
} else if (option == 2) {
    amt = 50;
} else if (option == 3) {
    amt = 20;
} else if (option == 4) {
    amt = 10;
} else {
    printf("Error: unknown option.\n");
}
```

```c
// Assign amt depending on option

switch (option) {
    case 1: amt = 100;break;
    case 2: amt = 50; break;
    case 3: amt = 20; break;
    case 4: amt = 10; break;
    default: printf("Error: unknown option.\n");
}
```

Control flow operations: for loops

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of loop
}
```

```
// Print numbers from 1 to 10
#include <stdio.h>

int main() {
  int i;

  for (i = 1; i < 11; ++i)
  {
    printf("%d ", i);
  }
  return 0;
}
```

Control flow operations: while, do/while loops

```c
while (testExpression) {
  // the body of the loop
}
```

```c
// Print numbers from 1 to 5
#include <stdio.h>

int main() {
  int i = 1;

  while (i <= 5) {
    printf("%d\n", i);
    ++i;
  }

  return 0;
}
```

```c
do {
  // the body of the loop
}
while (testExpression);
```

```c
// adds input numbers to sum until
0 is entered

do {
  printf("Enter a number: ");
  scanf("%lf", &number);
  sum += number;
}
while(number != 0.0);
```

Functions

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

void bubble_sort(int vals[], int len)
{
    int i, j, temp;
    for (i=0; i<len; i++) {
        for (j=i+1; j<len; j++) {
            if (vals[i] > vals[j]) {
                temp = vals[i];
                vals[i] = vals[j];
                vals[j] = temp;
            }
        }
    }
}

#define ARR_SIZE 8
int main() {
    int data_array[ARR_SIZE] = {4, 227, 6, 12, 0, 45, 11, 123};
    bubble_sort(data_array, ARR_SIZE);
    for (int i=0; i < ARR_SIZE; i++) {
        printf("%d\n", data_array[i]);
    }
    return 0;
}
```

Result:

```
0
4
6
11
12
45
123
227
```

Global and local variables

Global and local variables differ is ***scope*** and ***life cycle*** (when and where they are allocated, and from where they are visible)

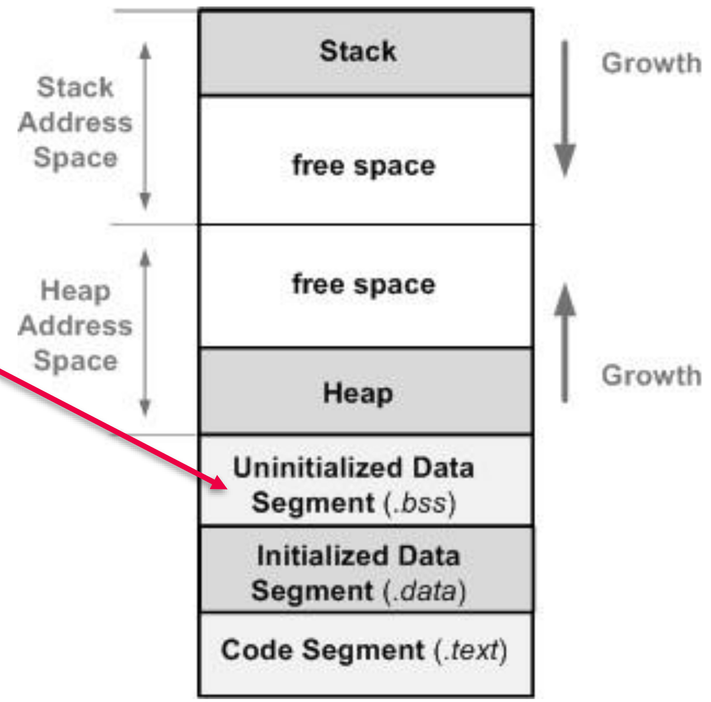
Global variable example:

```c
// Use a global variable to find and print the maximum of 3 numbers
int max; // global variable holding the maximum value

void findMax(int a, int b, int c) {
    max = a;
    if (b > max) {
        if (c > b) max = c;
        else max = b;
    } else if (c > max) max = c;
}

void printMax(void) {
    printf("The maximum number is: %d\n", max);
}

int main(void) {
    findMax(4, 3, 7);
    printMax();
}
```



Allocated in data segment, visible globally

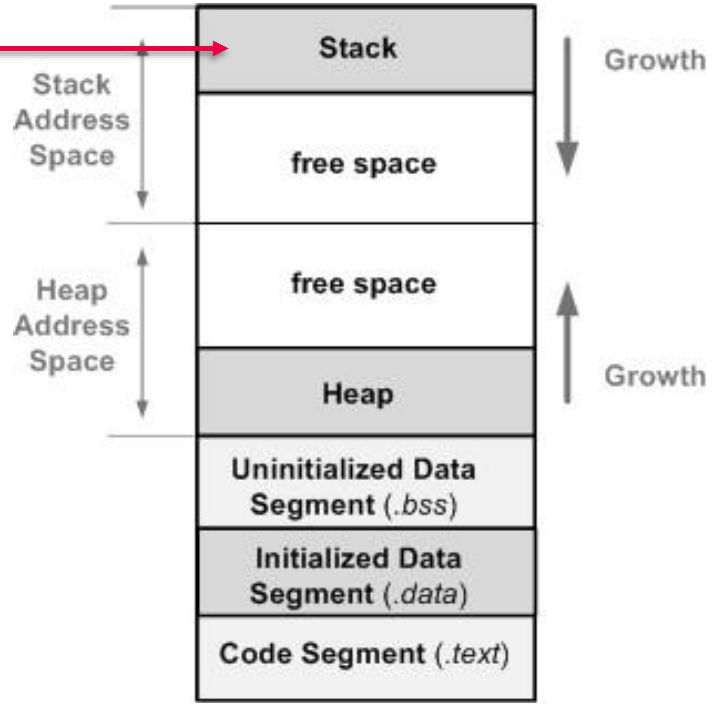
Global and local variables

Local variable example:

```c
// Use local variables to find and print the maximum of 3 numbers
int getMax(int a, int b, int c) {
    int result = a; // local variable holding the maximum value
    if (b > result) {
        if (c > b) result = c;
        else result = b;
    } else if (c > result) result = c;
    return result;
}

void printMax(int m) {
    printf("The maximum number is: %d\n", m);
}

int main(void) {
    int max;
    max = getMax(4, 3, 7);
    printMax(max);
}
```



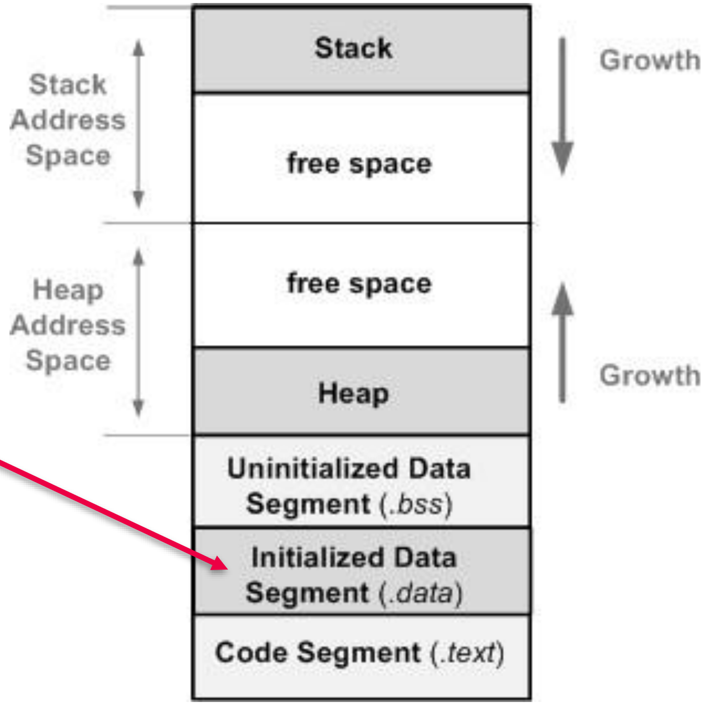
Allocated in stack, visible locally

Static variables

```c
#include <stdio.h>

int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d\n", fun());
  printf("%d\n", fun());
  printf("%d\n", fun());
  return 0;
}
```



Result:

1

2

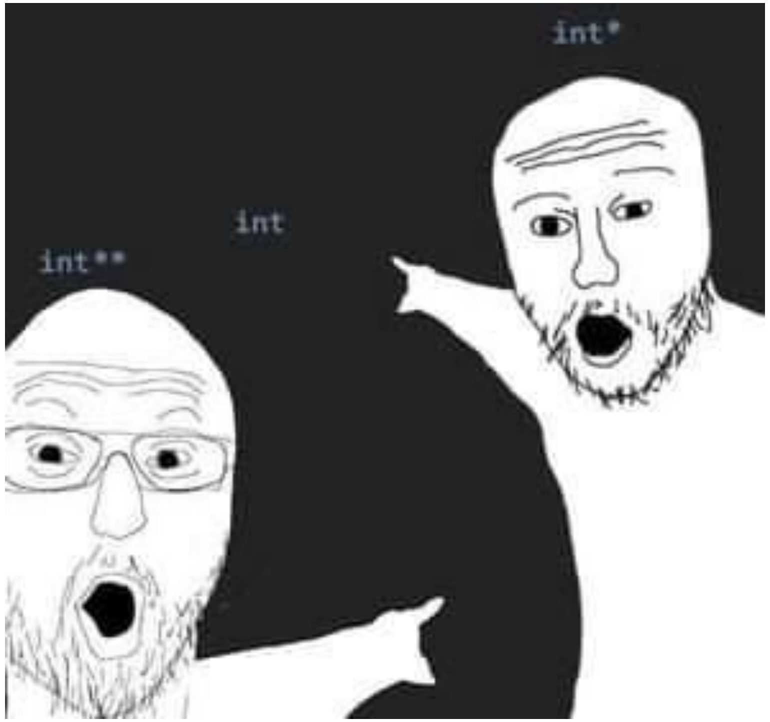
3

Preserving their value even after they are out of their scope!
Allocated in data segment, but visible locally

Pointers

Special variables ***containing addresses*** (***pointing***) to other variables

Address	Value	
0x00000000	...	
...	...	
0x210A345C	0x39FFAC34	
...	...	
0x48DF6784	0x210A345C	Pointer
...	...	
0xFFFFFFFC	...	



Pointers: example

```c
// Example pointer manipulations

int salary1, salary2; // 32-bit numbers
int *ptr; // a pointer specifying the address of an int variable

salary1 = 67500; // salary1 = $67,500 = 0x000107AC
ptr = &salary1; // ptr = 0x0070, the address of salary1

salary2 = *ptr + 1000;
/* dereference ptr to give the contents of address 70 = $67,500,
then add $1,000 and set salary2 to $68,500 */
```

Watch out for inconsistencies (dangling pointers, pointer arithmetic, etc.)

(no default protection from memory damage provided!)

Passing arguments by reference

By default arguments are passed in functions ***by value*** (value is ***copied***)

Passing ***by reference*** (by pointer) can ***save performance and memory***

no copying needed

Difference: modifications remain visible after return from function

```
#include <stdio.h>

struct contact {
    char name[30];
    int phone;
    float height; // in meters
    // much more data ...
};

void ProcessContact(contact * cnt) {
    cnt->phone = 8883344;
    // more processing...
}


int main() {
    struct contact c1;
    c1.phone = 7771122;
    printf("Before ProcessContact: %d\n", c1.phone);
    ProcessContact(&c1);
    printf("After  ProcessContact: %d\n", c1.phone);
    return 0;
}
```

Result:

```
Before ProcessContact: 7771122
After  ProcessContact: 8883344
```

Dynamic memory allocation

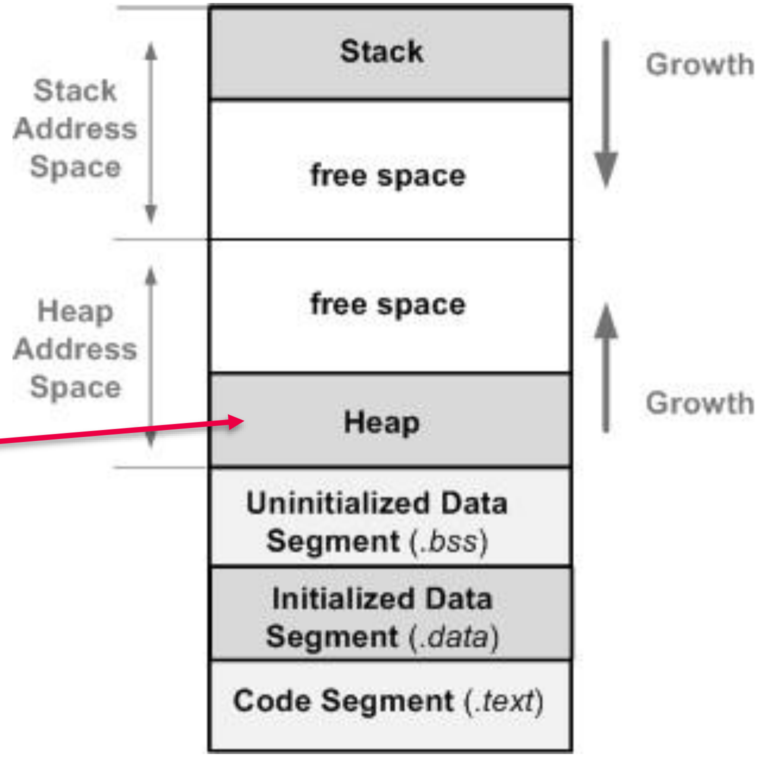
Memory for dynamic objects can be allocated in ***heap***

```c
// Dynamically allocate and de-allocate an array using malloc and free

#include <stdlib.h>

// getMean function definition

int main(void) {
    int len, i;
    int *nums;
    printf("How many numbers would you like to enter? ");
    scanf("%d", &len);
    nums = malloc(len*sizeof(int));
    if (nums == NULL) {
        printf("ERROR: out of memory.\n");
        return 1;
    else {
        for (i=0; i<len; i++) {
            printf("Enter number: ");
            scanf("%d", &nums[i]);
        }
        printf("The average is %f\n", getMean(nums, len));
        free(nums);
        ...
    }
}
```



Watch out for missed deallocations

(no default protection from memory leaks provided!)

Example: complex pointer processing

Function purpose: allocate sufficient memory for packet depending on packet type.

Solution: pass ***pointer to pointer*** (***“chain of pointers”***) to initialize pointer in function.

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

#define STATUS_OK   0
#define STATUS_FAIL 1

int InitMemForPacket(void ** packet_ptr, char packet_type) {
    if (packet_type == 0x0) {
        *packet_ptr = malloc(100);
    } else if (packet_type == 0x1) {
        *packet_ptr = malloc(200);
    } // other packet types ...

    if (*packet_ptr != NULL) return STATUS_OK;
    else return STATUS_FAIL;
}

int main() {
    void * new_packet_ptr;
    int InitStatus = InitMemForPacket(&new_packet_ptr, 0x1);
    printf("InitStatus: %d\n", InitStatus);
    return 0;
}
```

ITMO

Summary of variable allocation in segments

Declaration of data	Allocated in segment
Global variable (uninitialized)	`.bss`
Global variable (initialized)	`.data`
Local non-static variable	`Stack`
Local static variable	`.data`
Dynamic objects (managed by `malloc`/`free` functions)	`Heap`
Functions (code)	`.text`

Common standard libraries files

Header file	Description
stdio.h	I/O library. Contains functions for writing and reading data to/from a file or console (`printf`, `fprintf` and `scanf`, `fscanf`) and functions for opening and closing files(`fopen` and `fclose`).
stdlib.h	Standard Library. Contains functions for random number generation (`rand` and `srand`), dynamic memory allocation and deallocation (`malloc` and `free`), terminating the program (`exit`) and converting strings to numeric data types and vice versa (`atoi`, `atoll` and `atof`).
math.h	Mathematics Library. Contains standard mathematical functions such as `sin`, `cos`, `asin`, `acos`, `sqrt`, `log`, `log10`, `exp`, `floor` and `ceil`.
string.h	Library function for working with strings. Contains functions for comparing, copying, concatenating strings and calculating the length of a string.

Common limitations of C language

- No default protection from memory leaks and damages, many vulnerabilities for undefined behaviors (UBs)
- No automatic free of inaccessible memory (“garbage collection”), cannot be implemented
- No concept of parallelism (in the language itself)
  e.g. multicore programming is implemented via special libraries
- No concept of time (in the language itself)
  e.g. time management is implemented via interaction with OS and timers via special libraries
- Poor formalization, many ambiguities
  e.g. undefined order of arguments computation, confusions in parsing, etc.

Thank you for the lesson!

Alexander Antonov, Assoc. Prof., antonov@itmo.ru

Hangzhou, 2025