High Performance Programming

Programming in C – part 1

Anastasia Kruchinina

Uppsala University, Sweden

April 18, 2017

History of C

C is designed on a way to provide a full control of the computer.

C is the most widely used programming languages of all time.

C programming language

- Developed by Dennis Ritchie between 1969 and 1973 at Bell Labs
- Successor to B language
- The Unix system is written in C
- Different data types (e.g. byte, word, struct), efficient pointers

C standarts

Software developers writing in C are encouraged to conform to the standards, as doing so aids portability between compilers.

- C89 (ANSI C)
- C90 (ISO standard)
- C95
- C99
- C11

If you want to compile your programs according to the C standard C99, you should type `gcc -std=c99 -pedantic-errors`.

Structure of a program

```c
#include <stdio.h> // system header files
// #include "myheader.h" for your own header files

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

Functions

The syntax of a function definition:

```
return_type function_name( parameter list ) {
        /* body of the function */
}
```

Example of function declaration:

```
void f1();
int f2(int num1, int num2);  // call by value
void f3(char *s);   // call by reference
int* f4(int* arr);  // call by reference
```

Functions

Recursive functions:

```
void foo() {
  if(expression) return;
  foo(); /* function calls itself */
}
int main() {
  foo();
}
```

Q/A

If we run it, what will happen?

```
void foo() {
        foo(); /* function calls itself */
}
int main() {
  foo();
}
```

Answer: runtime error
During the execution the function foo() is called repeatedly and its return address is stored in the stack. After stack memory is full we will get stack overflow error.

main function arguments

```
#include <stdio.h>      /* printf */
#include <string.h>     /* strcpy */
#include <stdlib.h>     /* atoi */

int main(int argc, char const *argv[])
{
 if(argc != 4) {printf("Usage: %s string int double\n", argv[0]);
     return -1;} // Usage: a.out string int double
 char str[10];
 strcpy(str, argv[1]);
 int i = atoi (argv[2]);
 double f = atof (argv[3]);
 printf("%s %d %.1f\n", str, i, f);  // hello 5 6.5
 return 0;
}
```

We can run the program like this:
./a.out hello 5 6.5

Input/output

```
#include <stdio.h>
int main (){
char str [80]; int i, ihex;

scanf ("%s",str); // enter string
printf ("Entered string %s\n", str);

scanf ("%d",&i); // enter integer
printf ("Entered integer %d\n", i);

scanf ("%x",&ihex); // enter hexadecimal number
printf ("Entered hexadecimal %x (%d)\n", ihex, ihex);
```

Input/output

```
#include <stdio.h>
int main (){
 double f; char ch;

scanf ("%lf",&f); // enter double
printf ("Entered double %.3lf\n", f); // note format %.3lf

printf ("Address of f %p\n", &f);

scanf (" %c",&ch); // NOTE whitespace before %c, scanf
// does not skip any leading whitespace when reading char
printf ("Entered char %c\n", ch);

return 0;}
```

if..else

The syntax of a **if...else** statement is:

```
if(boolean_expression) {
 /* code will execute if the expression is true */
}
else {
 /* code will execute if the expression is false */
}
```

Relational operators: `!A`, `A&&B`, `A||B`

Note: do not confuse with bitwise operators `&`, `|`

Logical operators:
`A==B`, `A != B`, `A > B`, `A >= B`, `A < B`, `A <= B`

Note: `if(a=b)  is not the same as  if(a==b)`

if..else

C ternary operator:

```
condition ? expression1 : expression2
```

Meaning: if condition is true then execute expression1, otherwise execute expression2.

if..else

C ternary operator:

```
condition ? expression1 : expression2
```

Meaning: if condition is true then execute expression1, otherwise execute expression2.

Example:

```
int a = 36, b;
b = (a == 25) ? 1: -1;
printf( "b = %d, ", b );
b = (a == 36) ? 1: -1;
printf( "b = %d\n", b );
```

Output:

b = -1, b = 1

switch

The syntax of a **switch** statement is:

```
switch(expression)
{
        case constant-expression  :
// const-expression is a constant or a character
                /* code */
                break; // Optional

        case constant-expression  :
                /* code */
                break;  // Optional

        default : // Optional
                /* code */
}
```

Q/A

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int i = 1, j = 2;
switch(i)
{
case 1: printf("FIRST\n");
case j: printf("SECOND\n");
}
return 0;
}
```

Output: compiler error since "case j" is not allowed

for loop

The syntax of a for loop is:

```
for ( init; condition; increment ) {
  /* code */
}
```

for loop

The syntax of a `for` loop is:

```
for ( init; condition; increment ) {
  /* code */
}
```

Example of the for loop with `break` and `continue` statements:

```
int i;
for (i = 0; i < 1000; i++ ) {
  if(i < 5) continue;
  printf("%d ", i);
  if(i >= 10) break;
}
```

Output: 5 6 7 8 9 10

while loop

The syntax of a `while` loop is:

```
while(condition){
/* code */
}
```

The syntax of a `do...while` loop is:

```
do{
/* code */
} while(condition);
```

while loop

Example:

```
#include<stdio.h>

int main()
{
        int  i = 1;
        while(i++ < 10)
        {
          printf("%d ", i);
        }
        return 0;
}
```

Output: 2 3 4 5 6 7 8 9 10

Pointers

C allows to a programmer directly access and manipulate computer memory. It applies also to C++.
A lots of information and examples here:
http://www.c4learn.com/index/pointer-c-programming/
Pointer is a variable that can hold the address of other variable:

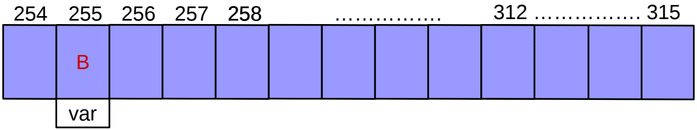
```
char var = 'B';
char *ptr = &var;
```

sizeof operator gives size in bytes of the object or type.
`sizeof(expression)` (or `sizeof expression`)
`sizeof(typename)`

Declare a variable

Declare an integer variable:

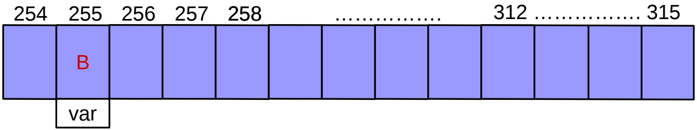
```
char var = 'B';
```



Declare a variable

Declare an integer variable and get its size:

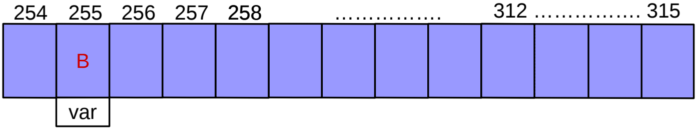
```
char var = ’B’;
printf("Sizeof(var): %lu\n", sizeof(var));  // 1 byte
printf("Sizeof(char): %lu\n", sizeof(char)); // 1
    byte
```



The address of a variable

The operator & - returns an address of a variable. Operating system decided on which memory location will be 'var'. Operator & just returns the address of this memory location (the address of the first byte in case the representation of the variable var needs more bytes).

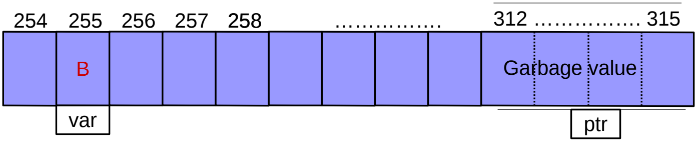
```
char var = 'B';
printf("The address is %p \n", &var);
// 0xff in hexadecimal, 255 in decimal
```



Pointers

Declare a pointer ptr:

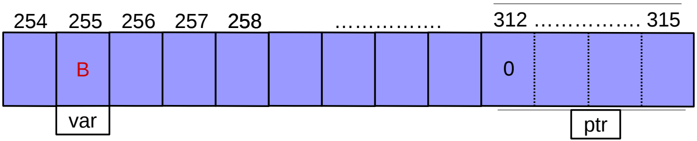
```
char var = 'B';  // allocates 1 byte of memory
char *ptr;    // or char* ptr;
/* ptr is a variable with type char*. In our example
    we use 32bit system and ptr allocates 4 bytes of
    memory. */
```



Pointers

Set ptr to NULL (macros for 0):

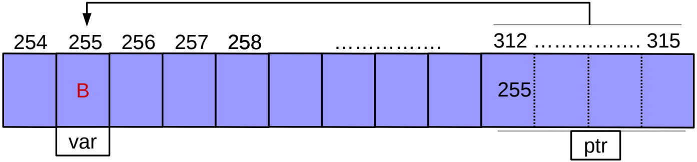
```
char var = 'B';
char *ptr = 0; // char *ptr = NULL // this telling to
    a developer that pointer is not used
```



Pointers

Assign an address of var to ptr:

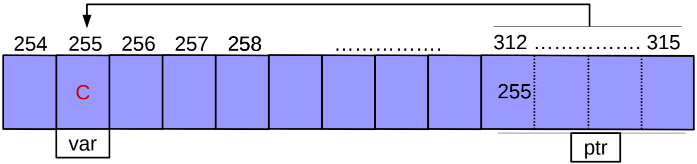
```
char var = 'B';
char *ptr = &var;   // 255
```



Dereferencing of pointer

Operator * access the content of the memory pointed by a pointer.

```
char var = 'B';
char *ptr = &var; // 255
*ptr = 'C';  // indirect way to
change a value of var
```



Examples:

```
char var = 'A';
char *ptr = &var; // var = 'A'
char var2 = *ptr; // var = 'A', var2 = 'A'
*ptr = 'B';       // var = 'B', var2 = 'A'
var = 'C';        // var = 'C', var2 = 'A'
ptr = &var2;
*ptr = 'D';       // var = 'C', var2 = 'D'
```

Q/A

What is the output of the following code:

```
int a = 2, b = 3;
int *p1 = &a;
int *p2 = &b;
*p1 = b;
*p2 = a;
printf("%d, %d", a, b);
```

Answer: 3, 3

Pointer arithmetics

See here for explanation and examples: http://www.c4learn.com/c-programming/c-pointer-arithmetic-operations/

p and r are pointers
Operator precedence:
p++ gives pointer to the next variable
p - r get distance between pointers
p + 5 moves pointer 5 positions forward
p + r not supported

Operator precedence:
*p++ increment pointer not value pointed by it
(*p)++ increment value pointed by p

Q/A

```
#include <stdio.h>

void f ( int *a , int b ) {
int *c;
c = &b;
(*a)++; b++; (*c)++;
}

int main () {
int a = 1, b = 10, c = 100;
f (&a, b );
printf( "a = %d , b = %d , c = %d\n" , a , b , c );
return 0 ;
}
```

Pointers

Write a function which sets pointer to a given value. What should be insteads of '?' ?

```
...
int *x;
tonull(? x, 3);
...
void set( ? x, int n ) {
? x = n;
}
```

(See answer on the next slide)

Pointers

Write a function which sets pointer to a given value.

```
...
int *x;
tonull(&x, 3);
...
void set( int **x, int n) {
*x = n;
}
```

Structures

Declare a structure with 2 members:

```
struct person
{
char  name[50];
int   age;
};
```

If we want to define an object: `struct person A;`
To access a member: `A.age = 19;`
Define a shortcut ”person_t” for ”struct person” using typedef:

```
typedef   struct person    person_t;
```

Pointers and structures

```
struct person
{
  char  name[50];
  int   age;
};
typedef   struct person    person_t;

person_t A;  // or struct person A;

A.age = 18; // set values of the structure members
strcpy(A.name, "Maria");
```

Pointers and structures

```
struct person
{
 char  name[50];
 int   age;
};
typedef   struct person    person_t;

person_t *A = (person_t *)malloc(sizeof(person_t));  //
    pointer to a structure

(*A).age = 18; // set values of the structure members
strcpy((*A).name, "Maria");
//or
A->age = 18; // set values of the structure members
strcpy(A->name, "Maria");
```

Pointers

A lots of information and examples here:
`http://www.c4learn.com/index/pointer-c-programming/`

Check this short summary about the pointers:
`http://nuclear.mutantstargoat.com/articles/pointers_explained.pdf`

Here you can read also about `void` pointers which can keep an address of the variable but cannot be dereferenced!

Arrays

`T[n]` is an array of n elements of type `T`: $0, \ldots, n-1$
Arrays values are stored in contiguous memory locations.

Initialization:

```
type arrayName [ size ]; // size is a constant !
```

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
//or
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Arrays

Arrays are closely related to pointers.

```
int a[] = {1, 2, 3, 4};
int *p = a;      // pointer to the first element
// equivalent:
int *p = &a[0]; // pointer to the first element
int *p = &a[3]; // pointer to the fourth element
```

Moreover:

```
int *p = a+2; // pointer to the third element
int v = *(a+2); // the third element
```

Arrays

Program writing in the standard output elements of the array arr:

```
#include <stdio.h>
int main()
{
int arr[5] = {11,22,33,44,55};
int i;
for(i=0;i<5;i++) {
 printf("%d ",arr[i]);
 // or
 printf("%d ",*(arr + i));
}
return 0;
}
```

String literal

String literal: `"hello"`, `"this is a string"`
Note: character: `'a'`, string `"a"`

Staring literal has a type `const char[size]`
The null character ('\0') is always appended to the string literal:

`"hello"` is a `const char[6]` holding the characters
'H', 'e', 'l', 'l', 'o', and '\0'.

On the compilation phase `"Hello," " world!"` yields the string `"Hello, world!"`

String literal and array of char

String literals can be assigned to char arrays:

```
char array1[] = "Hello" "world";
// same as
char array2[] = { ’H’, ’e’, ’l’, ’l’, ’o’, ’w’, ’o’,
    ’r’, ’l’, ’d’, ’\0’ };
```

Common functions:

- strlen – gives number of characters in the string, computed at the run time
- sizeof – gives size of the array, computed at the compile time

Strings

```
char arr[] = "Hello world";
int len = strlen(arr);
```

where

```
size_t strlen ( const char * str );
```

Implicit cast from array type `char[]` to the pointer to char `char*`.

Function gets the pointer to the first element of the array.
How does `strlen` knows the size of `str`?

Strings

```
char arr[] = "Hello world";
int len = strlen(arr);
```

where

```
size_t strlen ( const char * str );
```

Implicit cast from array type `char[]` to the pointer to char `char*`.

Function gets the pointer to the first element of the array.
How does `strlen` knows the size of `str`?
(String ends with a terminating null character '\0')

```
int len = strlen(arr); // answer is 11
```

Strings

What is the output of the following program?

```
#include<stdio.h>

int main()
{
  char s[20] = "Hello\0Hi";
  printf("%d %d", strlen(s), sizeof(s));
}
```

Answer: 5 20
strlen given the length of a string, so it is counting characters up to '0'.
sizeof reports the size of the array.

Strings

strcpy copies the string pointed by source into the array pointed by destination:

```
char * strcpy ( char * destination, const char *
    source );
```

Example:

```
char a[4];
//a = "hello";          This is an error!
strcpy(a, "hello");  // This is correct!
```

Q/A

```
#include <stdio.h>
int main()
{
 int *ptr , m = 100;
 ptr =  &m ;
 printf("%p",ptr);
 return 0;
}
```

Output: address of the variable m

Q/A

```
#include<stdio.h>

void foo(int *num1, int *num2) {
int temp;
temp = *num1;
*num1 = *num2;
*num2 = temp;
}
int main() {
int x=1, y=2;
foo(&x, &y);
printf("%d %d", x, y);
return 0;
}
```

Output: 2 1

Static vs dynamic memory

Static memory allocation: memory is allocated by the compiler.
Dynamic memory allocation: memory is allocated at the time of run time on *heap*.
You would use DMA if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.

Static allocation:

```
int arr[10];
```

Dynamic allocation

```
int *ptr;
ptr=(int *)malloc(sizeof(int)*10);
```

malloc, calloc, realloc from <stdlib.h>

`malloc` is used to allocate an amount of memory of **size bytes** during the execution of a program.

```
void* malloc (size_t size_of_block);
```

`calloc` allocates and zero-initialize array

```
void* calloc (size_t number_of_elements, size_t
   size_of_each_element);
```

`realloc` changes the size of the memory block pointed to by ptr

```
void* realloc (void* ptr_to_block, size_t
   size_of_block);
```

All functions return pointer to the allocated memory block.

free(p)

No garbage collector in C! You must free memory allocated with `malloc`, `calloc` or `realloc` using

```
void free (void* ptr);
```

A **memory leak** occurs when dynamically allocated memory has become unreachable.

Typical program:

```
int *ptr = (int *)malloc(sizeof(int));
if (ptr == 0)
{   printf("ERROR: Out of memory\n");
        return 1;}
/* code used ptr*/
free(ptr);
```

Q/A

Think before next lecture:

1. Can I increase the size of dynamically allocated array?

2. If you pass an array as an argument to a function, what is actually passed?

File I/O

A file represents a sequence of bytes, regardless of it being a text file or a binary file.

```
FILE *fopen( const char *filename, const char *mode )
```

`filename` is a string literal, the name of a file.
Most common file modes:

- 'r' : Opens an existing text file for reading purpose.
- 'w' : Opens a text file for writing. If it does not exist, then a new file is created.

Function returns NULL pointer if file cannot be opened.

Close file: `int fclose( FILE *fp );`

File I/O Example

```
#include <stdio.h>
int main() {
FILE *fp1;   // POINTER to an object FILE
FILE *fp2;
char buff[255];
fp1 = fopen("test.txt", "r");  // open file test.txt for reading
fp2 = fopen("out.txt", "w");   // open file out.txt for writing
fscanf(fp1, "%s", buff);   // read a string to buff (reads untill
     the space is encountered)
fprintf(fp2, "%s", buff);  // write data from buff to the file
    fp2
fclose(fp1);
fclose(fp2);
return 0; }
```

See also here: https://www.tutorialspoint.com/cprogramming/c_file_io.htm

Summary

What do you know after the lecture:

- syntax of basic constructions of C programming language
- what is a pointer and how to use it
- how to operate with strings and arrays
- how to allocate memory dynamically during runtime

Lab02 is available on the Studentportalen.

What is next?

It is important to understand the material of this lecture before the next lecture!

What will we do on the next lecture?

- how to use multidimensional arrays
- how to measure time in C
- what is time and space complexity
- data structures (binary tree, linked list and maybe some other)