Systemnära programmering

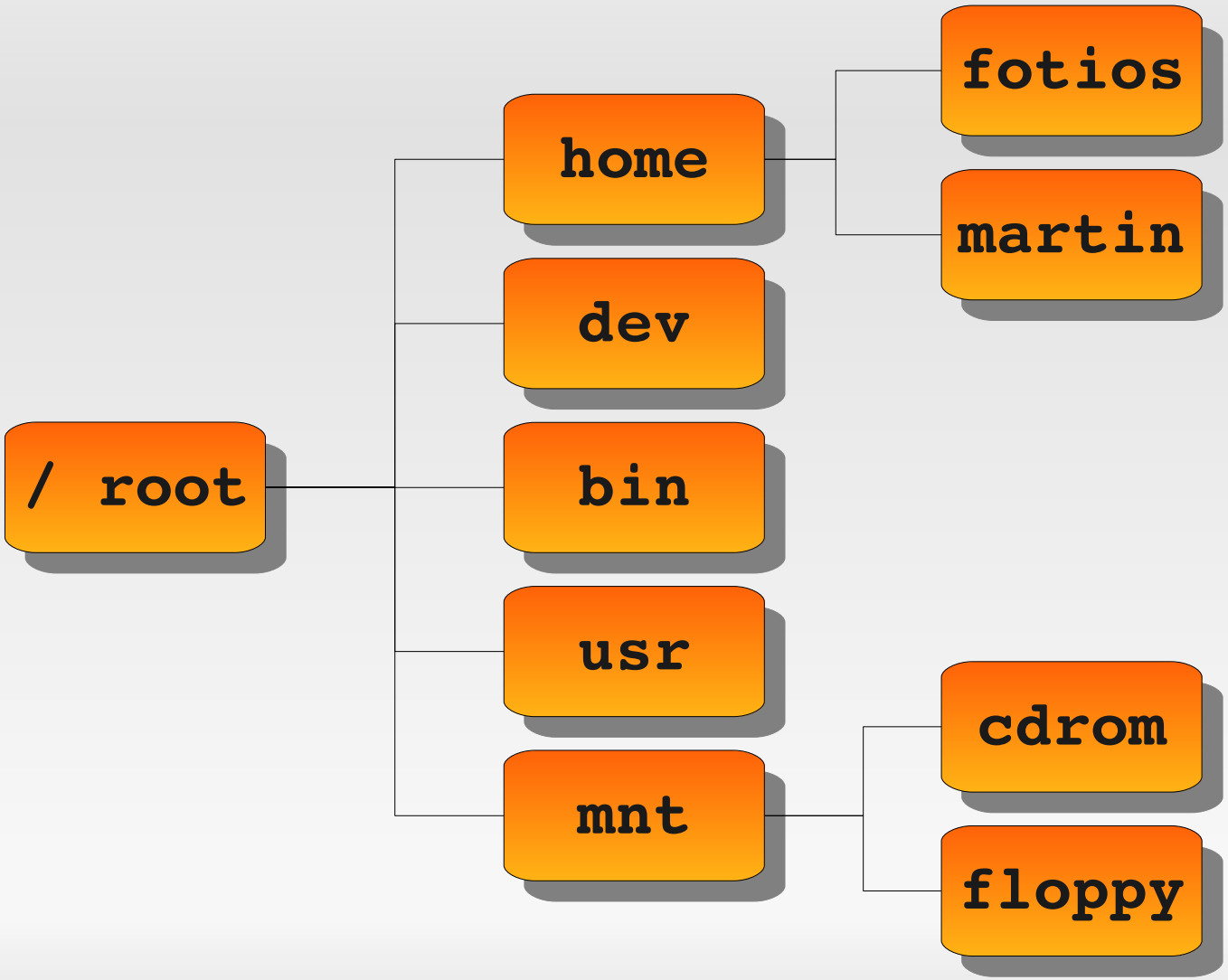
UMEÅ UNIVERSITY

Linux and C

Fotios Kasolis

THE SHELL

Linux directory structure



Enviroment control

- `cd d`: change to directory `d`
- `mkdir d`: create new directory `d`
- `mv f1 [f2...] d`: move file `f#` to directory `d`
- `mv d1 d2`: rename directory `d1` as `d2`
- `logout`: end terminal session
- `passwd`: change password

Enviroment status

- `ls -l`: list files in detail
- `date`: print date and time
- `who`: list logged in users
- `whoami`: display current user
- `finger user_name`: output user information
- `pwd`: print working directory
- `history`: display recent commands
- `!#`: submit recent command #

File manipulation

- `wc`: line, word and char count
- `cat`: list contents of file
- `more`: list file contents by screen
- `cmp f1 f2`: compare two files
- `diff f1 f2`: lists file differences
- `cp f1 f2`: copy file `f1` to file `f2`
- `sort`: alphabetically sort file contents of file `f`
- `mv f1 f2`: rename file `f1` to `f2`
- `rm f`: remove file `f` (`rm -r d`: remove directory `d`)
- `haid f`: output beginning of file `f`
- `tail f`: output end of file `f`

Example

```
$history>history_file1 # > directs the output to file
$sort history_file1>history_file2
$cmp history_file1 history_file2
$mkdir history_dir
$cp history_file1 /home/fotios/Desktop/history_dir
$cp history_file2 /home/fotios/Desktop/history_dir
$rm history_file1 history_file2
$cd history_dir
$ls -F
$cd ..
$rm -r history_dir
```

Write your first shell script

- Use an editor like `vi(m)`:

```
$vim my_first_script
```

- After writing shell script set execute permission for your script as follows:

```
$chmod +x my_first_script
```

- Execute your script:

```
$./my_first_script or
$bash my_first_script
```

Write your first shell script

- Here is an example:

```
$vim my_first_script
# This is a comment line
# My first script
clear
echo -e ”Welcome to\nLinux world”
$chmod +x my_first_script
./my_first_script
```

Vi(m) basics

- To insert new text: `esc+i`
- To save file: `esc+:+w`
- To save as: `esc+:+w "file_name"`
- To quit the editor: `esc+:+q`
- To save and quit the editor: `esc+:+wq`
- To quit the editor without saving: `esc+:+q!`

Script explanation

- `#` followed by any text is considered as comment.
- In the last syntax `./` means current directory.
- `clear` clears the screen.
- `echo "message"` prints message or value of variables on screen.
- `\n` means newline. To enable interpretation of the backslash characters we use `-e`.
- Give file extension such as `.sh`, which can be easily identified as shell script.

System variables

- `$BASH`: our shell name
- `$OSTYPE`: our OS type
- `$PATH`: our path settings
- `$SHELL`: our shell name
- `$USERNAME`: user name who is currently login

If you want to print your home directory location then you give command: `$echo $HOME`

UDV – User Defined Variables

- To define UDV use following syntax:

```
variable name=value
```

- To define variable called `my_var` having value `10`:

```
$my_var=10
$echo $my_var
```

- Shell could be used as a calculator:

```
$echo $[1+2+3] or $expr 1 + 2 + 3
```

The read statement

- Use `read` to get input (data from user) from keyboard and store (data) to variable.

```
$vim a_new_friend
#Script to read/print your name
echo "Type your first name please:"
read the_name
echo "Hello $the_name, Linux is your new
  friend!"
```

Other details

- To execute a command in an echo command use back quotes `command`:

```
$echo -e "My dirs-files are\n `ls`"
```

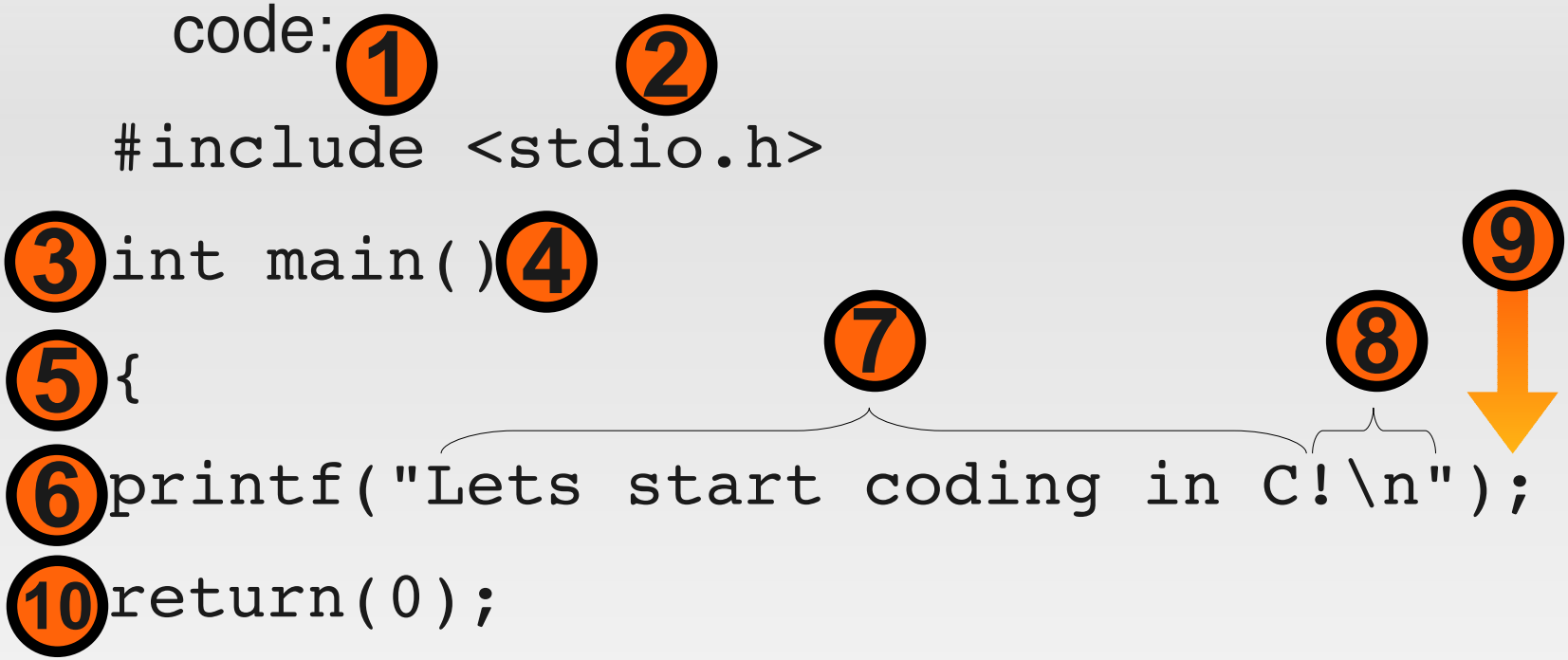
- By default in Linux if particular command/shell script is executed, it return two type of values which is used to see whether command or shell script executed is successful or not. If return value is zero (0), command is successful. If return value is nonzero, command is not successful or some sort of error executing command/shell script. This value is know as **Exit Status** and is stored in variable $?.

```
$ls

$echo $?

0
```

C WITH GCC

Your first program

- Use your text editor (like `vim`) to type the following source code:




```
#include <stdio.h>
int main()
{
printf("Lets start coding in C!\n");
return(0);
}
```

- Then save the file `coding.c` and type in the command line: `$gcc coding.c -o coding`
- Then `$./coding` and see what you get!

Your first program

- 1. `#include` is known as a preprocessor directive, which sounds impressive, and it may not be the correct term, but you' re not required to memorize it anyhow. What it does is tell the compiler to include text from another file, stuffing it right into your source code.
- 2. `<stdio.h>` is a filename hugged by angle brackets. The whole statement `#include <stdio.h>` tells the compiler to take text from the file `stdio.h` and stick it into your source code before the source code is compiled. The `stdio.h` contains information about the standard i/o functions required. The `.h` means header.
- 3. `int main` does two things. First, the `int` identifies the function `main` as an integer function, meaning that `main()` must return an integer value when it's done. Second, that line names the function `main`, which also identifies the first and primary function inside the program.

Your first program

- 4. Two empty parentheses follow the function name.
- 5. All functions in C have their contents encased by curly braces.
- 6. `printf()` is the name of a C language function. It's job is to display information on the screen.
- 7. In the parentheses of `printf()`, you find a string of characters to be printed on the screen.
- 8. The backslash character `\n` represents is the character produced by pressing the enter key, called a newline in C.
- 9. The semicolon tells the C compiler where one statement ends and another begins.
- 10. The second statement is the `return` command. This command sends the value `0` (zero) back to the operating system when the `main()` function is done.

The C language itself

- The C language is really rather brief. C has only 32 keywords:

```
auto, break, case, char, const, continue,
default, do, double, else, enum, extern,
float, for, goto, if, int, long, register,
return, short, signed, sizeof, static,
struct, switch, typedef, union, unsigned,
void, volatile, while.
```

- But these aren't all the words you use when writing programs in C. Other words or instructions are called **functions**. These include jewels like `printf()` and several dozen other common functions that assist the basic C language keywords in creating programs.

Other i/o functions

- `/* This is how a comment looks in the C language */`
- `scanf()` is a function like `printf()`. Its purpose is to read text from the keyboard.
- Compared to `scanf()`, the `gets()` function is nice and simple. Both do the same thing: They read characters from the keyboard and save them in a variable. `gets()` reads in only text, however `scanf()` can read in numeric values and strings and in a number of combinations (not safe→keybord overflow).
- `puts()` displays a string of text, but without all `printf()`'s formatting magic.

Types of variables

- `int:` store integers (`%i` or `%d`).
- `char:` is similar to `int` type, yet it is only big enough to hold one `ASCII` character (`%c`).
- `float:` is short for floating point – one machine word in size (`%f`).
- `double:` long floating point (%lf).

Good programming practice of initialization of characters

```
char letter='a'
```

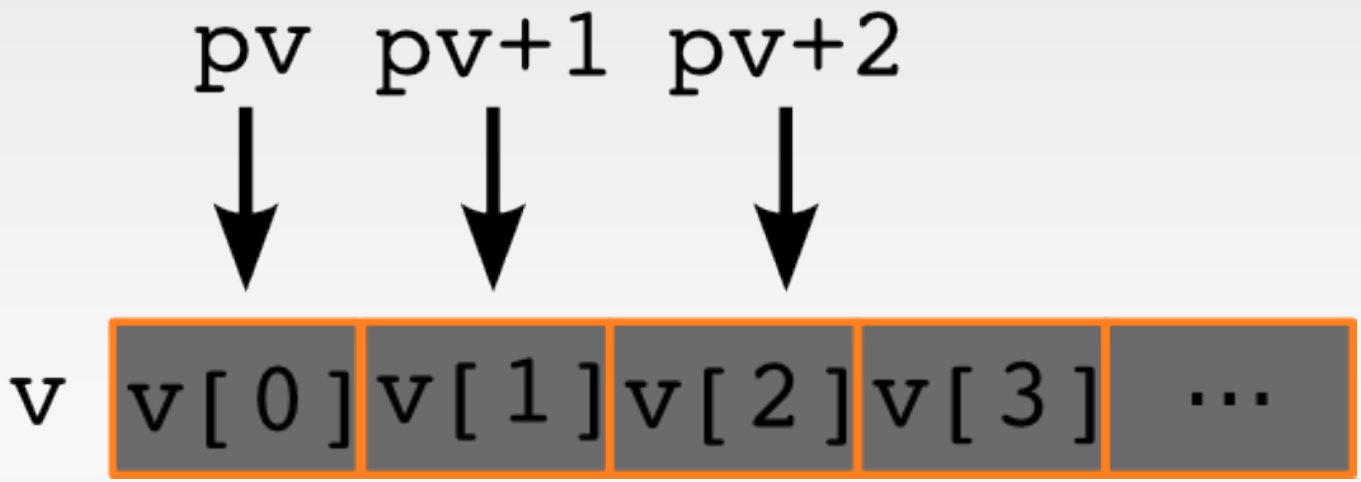
Pointers

- `pointer=&variable` means that the variable `pointer` should take its value `*pointer` from the adress `&variable`.

```
#include <stdio.h>

int main()

{

    float x, *fp;

    x = 6.5;

    printf("Value of x is %f, address of x %ld\n", x,
  &x);

    fp = &x;

      printf("Value in memory location fp is %f\n",
  *fp);

}
```

Arrays

- Arrays starts at position 0.
- The elements of the array occupy adjacent locations in memory.
- C treats the name of the array as if it were a pointer to the first element. Thus, if `v` is an array, `*v` is the same thing as `v[0]`, `*(v+1)` is the same thing as `v[1]`, and so on.



Array example

```
#include <stdio.h>

#define SIZE 3

int main()

{

float x[SIZE];

    float *fp;

    int i;

    for (i = 0; i < SIZE; i++)

       x[i] = 0.5*(float)i;

    for (i = 0; i < SIZE; i++)

       printf("%d\t%f\n", i, x[i]);

       fp = x;

    for (i = 0; i < SIZE; i++)

       printf("%d\t%f\n", i, *(fp+i));

    return 0;

}
```

Functions

```
#include <stdio.h>

int main(void)

{

    int x;

    scanf("%d",&x);

    printf("%d\n",square(x));

    return 0;

}

int square(int x)

{

    int square_of_x;

    square_of_x = x * x;

    return square_of_x;

}
```

```
return-type function-name
{
    ...local-declarations...

    ...statements...

    return return-value;
}
```

I/O to and from files

- The statements are: `FILE *fp, fp = fopen(name, mode), fscanf(fp, "format string", variable list), fprintf(fp, "format string", variable list), fclose(fp).`
- The logic here is that the code must
  - define a local `pointer` of type `FILE` (note that the uppercase is necessary here), which is defined in `<stdio.h>`.
  - `open` the file and associate it with the local pointer via `fopen`.
  - perform the i/o operations using `fscanf` and `fprintf`.
  - disconnect the file from the task with `fclose`.
- The `mode` argument in the `fopen` specifies the purpose/positioning in opening the file: `r` for reading, `w` for writing, and `a` for appending to the file.

I/O to and from files example

```
#include < stdio.h>
int main()
{
   FILE *fp;
   int i;
   fp = fopen(”foo.dat”, ”w”);  /*open foo.dat - writing*/
   fprintf(fp, ”\nSample Code\n\n”); /* write some info */
   for (i = 1; i <= 10 ; i++)
      fprintf(fp, "i = %d\n", i);
   fclose(fp);                          /* close the file  */
}
```

That was all!