Computer Programming

Command line

Robert Varga

Technical University of Cluj-Napoca
Computer Science Department

Course 12

Contents

1. Command line arguments
2. Command line system calls

Command line arguments

- C programs can receive input arguments from the command line
- these are input arguments to the main function
- it is an alternative way to provide input data
- most often employed in utility applications where we want to avoid user interaction and encourage automatization

Alternative form of the main function

```
int main(int argc, char** argv)
```

or equivalently

```
int main(int argc, char* argv[])
```

- argc - refers to argument count
  - the number of input arguments sent
  - includes the default first argument
- argv - refers to argument values
  - an array of strings
  - argv[0] = first argument is always the name of the executable
  - the array contains an additional element equal to NULL marking the end of it
  - argc[argv] = NULL

Conversions

- since arguments are sent as strings, they most likely need to be converted to the correct type, since they are sent as strings
- employ atoi/atof/atoll or sscanf for conversion
- arguments are separated by spaces
  - if we want to include the white-space characters in our arguments, they must be given between quotation marks in the command line, like "Program Files"
- the command line has special characters
  - all these must be introduced and handled differently
  - depends on the command line interpreter

Program 12.1 - Simple calculator

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

int main(int argc,
    char** argv)
{
    if (argc != 4){
      printf("Provide 3
          arguments");
      return -1;
    }

    double x, y;
    char op;
    x = atof(argv[1]);
    op = argv[2][0];
    y = atof(argv[3]);
    double rez;
    switch(op){
      case '+': rez = x+y; break;
      case '-': rez = x-y; break;
      case '*': rez = x*y; break;
      case '/': rez = x/y; break;
      case '^': rez = pow(x,y);
          break;
      default: rez = 0;
    }

    printf("%f", rez);
    return 0;
}
```

Stream redirection

- given an existing executable we can redirect
  - where the output is printed
  - from where the input is read
- can be done in the command line without changing the code
- redirecting output can be achieved using the $>$ symbol
  - example: main.exe $>$ outfile
  - will write to the file instead of writinf on the screen
  - equivalent to opening outfile with mode w
  - to append use the $>>$ symbols
  - example: main.exe $>>$ outfile
- redirecting input can be achieved using the $<$ symbol
  - example: main.exe $<$ infile
  - will read from the file instead of reading from the keyboard

System function

```
int system(const char* cmd);
```

- sends the string cmd to the command line
- transfer control to the interpreter and wait for the command to finish
- the return value can be checked to see if the command was executed successfully
- usually, 0 means success, but depends on the command
- interaction with the operating system
  - commands are different
  - security aspects = can run dangerous commands

DIR command

- allows us to view the files in a folder
- basic command syntax:

  `DIR folder options`
- will list the files in the given folder
- options change the way listing happens
- for full help on options run DIR /?

DIR command example

- Example command:

```
DIR "d:\code\c\codeblocks" /B > files.txt
```

- print all the files in the specified folder
- uses bare format: only file names with extensions
- instead of printing in the console, print to the given file

gcc command

- the gcc compiler is employed by your editor to compile the source code
- we can call it manually
- basic usage:

  ```
  gcc options source_file
  ```

- options start with - or –
- can be used to change the output executable file name, the standard, etc.
- for full help run gcc –help

gcc command example

- Example command:

```
gcc.exe -std=c99 -o main.exe main.c
```

- compiles and links the main.c source file
- use the c99 standard
- the name of the resulting executable should be main.exe
- gcc must be either in the current folder or on the system path
- returns 0 in case of success
- once main.exe was created in can be executed from our program using the following system call

```
system("main.exe");
```

Snipet - Running other programs

```
void eval(){
  char folder[] = "d:\\code\\c\\codeblocks\\tmp2\\";
  char cmd[256];
  sprintf(cmd, "dir %s\\*.c /b > files.txt",folder);
  system(cmd);

  char fname[256];
  char gcc_root[] =
  "c:\\Program Files (x86)\\CodeBlocks\\MinGW\\bin";
  FILE* f = fopen("files.txt","r");
  FILE* fout = fopen("results.csv", "w");
  mypair p;
  int idx = 1;
  while(1){
    if (fgets(fname, 99, f) < 1)
      break;
    fname[strlen(fname)-1] = 0;
    printf("%d:\n", idx++);
    printf("processing %s\n", fname);
    remove("output.txt");
    // ... eval
    printf("building ... ");
    sprintf(cmd, "\"%s\\gcc.exe\" -std=c99 -o a.exe %s\\%s",
            gcc_root, folder, fname);
    int ret = system(cmd);
    if (ret!=0){
      p.correct = -10;
      printf("build failed\n");
    }
    else{
      printf("ok\n");
      sprintf(cmd,"a.exe");
      int ret = system(cmd);
      if (ret!=0){
        p.correct = -9;
        printf("run failed\n");
      }

      p = compare("output.txt", "chess_out_gt.txt");
    }
    printf("correct: %d / %d\n\n",
        p.correct, p.total);
    fprintf(fout, "%s, %d, %d\n",
        fname, getNum(fname), p.correct);
  }
  fclose(f);
  fclose(fout);
}
```