ESc 101: Fundamentals of Computing

Lecture 28

Mar 18, 2010

Outline

Generating Many Big Numbers

- C compiler comes with many predefined functions.
- These functions are collected in a library referred as standard library.
- One of the functions is `rand()`.
- This function generates a random number between 0 and `RAND_MAX`.
- To generate k numbers, we call this function k times.

Generating Many Big Numbers

- C compiler comes with many predefined functions.
- These functions are collected in a library referred as standard library.
- One of the functions is `rand()`.
- This function generates a random number between 0 and `RAND_MAX`.
- To generate k numbers, we call this function k times.

Generating Many Big Numbers

- C compiler comes with many predefined functions.
- These functions are collected in a library referred as standard library.
- One of the functions is `rand()`.
- This function generates a random number between 0 and `RAND_MAX`.
- To generate k numbers, we call this function k times.

Generating Many Big Numbers

- C compiler comes with many predefined functions.
- These functions are collected in a library referred as standard library.
- One of the functions is `rand()`.
- This function generates a random number between 0 and `RAND_MAX`.
- To generate k numbers, we call this function k times.

Storing the Numbers in a File

- When there is a large amount of data to be read or written, it is easier to do this through a file.
- C provides a very simple way of working with files.
- To access a file, it first needs to be opened:
  `fopen(<filename>, <mode>)`
  opens the file `<filename>` for `<mode>` type of operations.
- `<filename>` is a string, representing the name of the file.
- `<mode>` is also a string, representing what we wish to do with the file.

Storing the Numbers in a File

- When there is a large amount of data to be read or written, it is easier to do this through a file.
- C provides a very simple way of working with files.
- To access a file, it first needs to be opened:
  `fopen(<filename>, <mode>)`
  opens the file `<filename>` for `<mode>` type of operations.
- `<filename>` is a string, representing the name of the file.
- `<mode>` is also a string, representing what we wish to do with the file.

Storing the Numbers in a File

- When there is a large amount of data to be read or written, it is easier to do this through a file.
- C provides a very simple way of working with files.
- To access a file, it first needs to be opened:
  `fopen(<filename>, <mode>)`
  opens the file `<filename>` for `<mode>` type of operations.
- `<filename>` is a string, representing the name of the file.
- `<mode>` is also a string, representing what we wish to do with the file.

Storing the Numbers in a File

- When there is a large amount of data to be read or written, it is easier to do this through a file.
- C provides a very simple way of working with files.
- To access a file, it first needs to be opened:
  `fopen(<filename>, <mode>)`
  opens the file `<filename>` for `<mode>` type of operations.
- `<filename>` is a string, representing the name of the file.
- `<mode>` is also a string, representing what we wish to do with the file.

Storing the Numbers in a File

- When there is a large amount of data to be read or written, it is easier to do this through a file.
- C provides a very simple way of working with files.
- To access a file, it first needs to be opened:
  `fopen(<filename>, <mode>)`
  opens the file `<filename>` for `<mode>` type of operations.
- `<filename>` is a string, representing the name of the file.
- `<mode>` is also a string, representing what we wish to do with the file.

Types of `<mode>`

`<mode>` can be:

- `"r"`: for reading from a file. If `<filename>` does not exist, results in error.
- `"w"`: for writing to a file. If `<filename>` does not exist, it is created. If it exists, its contents are deleted.
- `"a"`: for appending to a file. If `<filename>` does not exist, it is created. If it exists, its contents are retained.

There are more types of `<mode>` but we will not consider them in this course.

Types of `<mode>`

`<mode>` can be:

- `"r"`: for reading from a file. If `<filename>` does not exist, results in error.
- `"w"`: for writing to a file. If `<filename>` does not exist, it is created. If it exists, its contents are deleted.
- `"a"`: for appending to a file. If `<filename>` does not exist, it is created. If it exists, its contents are retained.

There are more types of `<mode>` but we will not consider them in this course.

Types of <mode>

<mode> can be:

- "r": for reading from a file. If <filename> does not exist, results in error.
- "w": for writing to a file. If <filename> does not exist, it is created. If it exists, its contents are deleted.
- "a": for appending to a file. If <filename> does not exist, it is created. If it exists, its contents are retained.

There are more types of <mode> but we will not consider them in this course.

Types of `<mode>`

`<mode>` can be:

- `"r"`: for reading from a file. If `<filename>` does not exist, results in error.
- `"w"`: for writing to a file. If `<filename>` does not exist, it is created. If it exists, its contents are deleted.
- `"a"`: for appending to a file. If `<filename>` does not exist, it is created. If it exists, its contents are retained.

There are more types of `<mode>` but we will not consider them in this course.

Types of <mode>

<mode> can be:

- "r": for reading from a file. If <filename> does not exist, results in error.
- "w": for writing to a file. If <filename> does not exist, it is created. If it exists, its contents are deleted.
- "a": for appending to a file. If <filename> does not exist, it is created. If it exists, its contents are retained.

There are more types of <mode> but we will not consider them in this course.

Return Value of `fopen()`

- `fopen()` returns a pointer to the file.
- It is defined to be of type `FILE *`.
- If there is an error, then the return value is `NULL`.

Return Value of fopen()

- fopen() returns a pointer to the file.
- It is defined to be of type FILE *.
- If there is an error, then the return value is NULL.

Return Value of fopen()

- fopen() returns a pointer to the file.
- It is defined to be of type FILE *.
- If there is an error, then the return value is NULL.

Reading and Writing

- We can use `fprintf()` and `fscanf()` to read from and write to a file after opening it.
- The only change is that there is an additional argument: the file pointer.
- The syntax is:
  `fprintf(fp, <format string>, arg-1, arg-2, ...)`
- Of course, `fprintf()` can only be used for files opened in `"w"` or `"a"` modes.
- And `fscanf()` can only be used for files opened in `"r"` mode.

Reading and Writing

- We can use `fprintf()` and `fscanf()` to read from and write to a file after opening it.
- The only change is that there is an additional argument: the file pointer.
- The syntax is:
  `fprintf(fp, <format string>, arg-1, arg-2, ...)`
- Of course, `fprintf()` can only be used for files opened in `"w"` or `"a"` modes.
- And `fscanf()` can only be used for files opened in `"r"` mode.

Reading and Writing

- We can use `fprintf()` and `fscanf()` to read from and write to a file after opening it.
- The only change is that there is an additional argument: the file pointer.
- The syntax is:
  `fprintf(fp, <format string>, arg-1, arg-2, ...)`
- Of course, `fprintf()` can only be used for files opened in `"w"` or `"a"` modes.
- And `fscanf()` can only be used for files opened in `"r"` mode.

Reading and Writing

- We can use `fprintf()` and `fscanf()` to read from and write to a file after opening it.
- The only change is that there is an additional argument: the file pointer.
- The syntax is:
  `fprintf(fp, <format string>, arg-1, arg-2, ...)`
- Of course, `fprintf()` can only be used for files opened in `"w"` or `"a"` modes.
- And `fscanf()` can only be used for files opened in `"r"` mode.

Closing a File

- The function call
  `fclose(fp)`
  closes the file whose file pointer is `fp`.
- Every opened file should be closed in the program when its use is finished.

Closing a File

- The function call
  `fclose(fp)`
  closes the file whose file pointer is `fp`.
- Every opened file should be closed in the program when its use is finished.

Checking End of File

- The function
  `feof(fp)`
  is useful to check if, while reading, the end of file is reached.
- The file pointer `fp` points to a certain location of the file.
- When we read from or write to the file, the data is read from or written to respectively the location pointed by `fp`.
- `fp` is then advanced to the next location of the file.
- `feof(fp)` returns a non-zero value if `fp` points to the end of the file. Else it returns 0.

Checking End of File

- The function
  `feof(fp)`
  is useful to check if, while reading, the end of file is reached.
- The file pointer `fp` points to a certain location of the file.
- When we read from or write to the file, the data is read from or written to respectively the location pointed by `fp`.
- `fp` is then advanced to the next location of the file.
- `feof(fp)` returns a non-zero value if `fp` points to the end of the file. Else it returns 0.

Checking End of File

- The function
  `feof(fp)`
  is useful to check if, while reading, the end of file is reached.
- The file pointer `fp` points to a certain location of the file.
- When we read from or write to the file, the data is read from or written to respectively the location pointed by `fp`.
- `fp` is then advanced to the next location of the file.
- `feof(fp)` returns a non-zero value if `fp` points to the end of the file. Else it returns 0.

sprintf() AND sscanf()

- The functions sprintf() and sscanf() work with strings.
- The format for sprintf() is:
  `sprintf(<string>, <format string>, arg-1, arg-2, ...).`
- The output is written to the <string> in the form of a string.
- Similarly, sscanf() reads input from a string.

sprintf() AND sscanf()

- The functions `sprintf()` and `sscanf()` work with strings.
- The format for `sprintf()` is:
  `sprintf(<string>, <format string>, arg-1, arg-2, ...)`.
- The output is written to the `<string>` in the form of a string.
- Similarly, `sscanf()` reads input from a string.

sprintf() AND sscanf()

- The functions `sprintf()` and `sscanf()` work with strings.
- The format for `sprintf()` is:
  `sprintf(<string>, <format string>, arg-1, arg-2, ...)`.
- The output is written to the `<string>` in the form of a string.
- Similarly, `sscanf()` reads input from a string.

Outline

Passing `<filename>` as Argument

- We may wish to create multiple files containing sequences of numbers.
- For this, the program can accept `<filename>` as input.
- It can also be done, in Linux at least, using the input redirection: `seq > <filename>`.
- There is another alternative: by providing the `<filename>` as a command line argument to the program: `seq <filename>`.
- This simplifies typing, as well as provides freedom to specify multiple input and output files.

Passing `<filename>` as Argument

- We may wish to create multiple files containing sequences of numbers.
- For this, the program can accept `<filename>` as input.
- It can also be done, in Linux at least, using the input redirection: `seq > <filename>`.
- There is another alternative: by providing the `<filename>` as a command line argument to the program: `seq <filename>`.
- This simplifies typing, as well as provides freedom to specify multiple input and output files.

Passing `<filename>` as Argument

- We may wish to create multiple files containing sequences of numbers.
- For this, the program can accept `<filename>` as input.
- It can also be done, in Linux at least, using the input redirection: `seq > <filename>`.
- There is another alternative: by providing the `<filename>` as a command line argument to the program: `seq <filename>`.
- This simplifies typing, as well as provides freedom to specify multiple input and output files.

Passing `<filename>` as Argument

- We may wish to create multiple files containing sequences of numbers.
- For this, the program can accept `<filename>` as input.
- It can also be done, in Linux at least, using the input redirection: `seq > <filename>`.
- There is another alternative: by providing the `<filename>` as a command line argument to the program: `seq <filename>`.
- This simplifies typing, as well as provides freedom to specify multiple input and output files.

Passing `<filename>` as Argument

- We may wish to create multiple files containing sequences of numbers.
- For this, the program can accept `<filename>` as input.
- It can also be done, in Linux at least, using the input redirection: `seq > <filename>`.
- There is another alternative: by providing the `<filename>` as a command line argument to the program: `seq <filename>`.
- This simplifies typing, as well as provides freedom to specify multiple input and output files.

Using `main()` to Accept Arguments

- When command line arguments are expected in a program, its `main()` function is written with parameters.
- The first one is an integer variable, storing the number of white-space separated strings in the command:
    - This also counts the name of the program as one string.
- The second argument is an array of strings, storing all the strings in the command.

Using `main()` to Accept Arguments

- When command line arguments are expected in a program, its `main()` function is written with parameters.
- The first one is an integer variable, storing the number of white-space separated strings in the command:
  - This also counts the name of the program as one string.
- The second argument is an array of strings, storing all the strings in the command.

Using `main()` to Accept Arguments

- When command line arguments are expected in a program, its `main()` function is written with parameters.
- The first one is an integer variable, storing the number of white-space separated strings in the command:
  - This also counts the name of the program as one string.
- The second argument is an array of strings, storing all the strings in the command.

Using `main()` to Accept Arguments

- When command line arguments are expected in a program, its `main()` function is written with parameters.
- The first one is an integer variable, storing the number of white-space separated strings in the command:
  - This also counts the name of the program as one string.
- The second argument is an array of strings, storing all the strings in the command.

Example

```
int main(int argc, char *argv[])
{
    for (int i = 0; i < argc, i++)
        printf("%s\n", argv[i]);
}
```

- The above program is compiled and stored in file, say, `test-command-line`.
- On typing `test-command-line xyz 123 Ad4`, the output will be:

```
test-command-line
xyz
123
Ad4
```

Example

```
int main(int argc, char *argv[])
{
    for (int i = 0; i < argc, i++)
        printf("%s\n", argv[i]);
}
```

- The above program is compiled and stored in file, say, `test-command-line`.
- On typing `test-command-line xyz 123 Ad4`, the output will be:

```
test-command-line
xyz
123
Ad4
```

Example

```
int main(int argc, char *argv[])
{
    for (int i = 0; i < argc, i++)
        printf("%s\n", argv[i]);
}
```

- The above program is compiled and stored in file, say, `test-command-line`.
- On typing `test-command-line xyz 123 Ad4`, the output will be:

```
test-command-line
xyz
123
Ad4
```

Example

```
int main(int argc, char *argv[])
{
    for (int i = 0; i < argc, i++)
        printf("%s\n", argv[i]);
}
```

- The above program is compiled and stored in file, say, `test-command-line`.
- On typing `test-command-line xyz 123 Ad4`, the output will be:

```
test-command-line
xyz
123
Ad4
```