Introduction to Programming

Week – *9*, Lecture – *2*

File Handling in C – Part 1

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

Text vs Binary files

You encounter two types of files on day-to-day basis – *text* files and *binary* files

Text vs Binary files

You encounter two types of files on day-to-day basis – *text* files and *binary* files

Text files are the files that you edit in ”simple text editors” like Notepad or vim

Text vs Binary files

You encounter two types of files on day-to-day basis – *text* files and *binary* files

Text files are the files that you edit in ”simple text editors” like Notepad or vim

- The term “simple” here implies that the files that you create in editors like Word, are usually not text files
- The convention, is that general-purpose text files are named with extension `.txt`

Text vs Binary files

You encounter two types of files on day-to-day basis – *text* files and *binary* files

Text files are the files that you edit in ”simple text editors” like Notepad or vim

- The term “simple” here implies that the files that you create in editors like Word, are usually not text files
- The convention, is that general-purpose text files are named with extension `.txt`

All other files are binary files

- Word files, Images, Music, Videos etc. all are binary files

Text vs Binary files

You encounter two types of files on day-to-day basis – *text* files and *binary* files

Text files are the files that you edit in ”simple text editors” like Notepad or vim

- The term “simple” here implies that the files that you create in editors like Word, are usually not text files
- The convention, is that general-purpose text files are named with extension `.txt`

All other files are binary files

- Word files, Images, Music, Videos etc. all are binary files

The major difference between the two is the representation of data

Text vs Binary files

You encounter two types of files on day-to-day basis – *text* files and *binary* files

Text files are the files that you edit in ”simple text editors” like Notepad or vim

- The term “simple” here implies that the files that you create in editors like Word, are usually not text files
- The convention, is that general-purpose text files are named with extension `.txt`

All other files are binary files

- Word files, Images, Music, Videos etc. all are binary files

The major difference between the two is the representation of data

Text files store data character-by-character by storing the ASCII code of the character

- On most systems, ASCII is the default “encoding” for text files; other possibilities include UTF-8 or UTF-16

Text vs Binary files

You encounter two types of files on day-to-day basis – *text* files and *binary* files

Text files are the files that you edit in ”simple text editors” like Notepad or vim

- The term “simple” here implies that the files that you create in editors like Word, are usually not text files
- The convention, is that general-purpose text files are named with extension `.txt`

All other files are binary files

- Word files, Images, Music, Videos etc. all are binary files

The major difference between the two is the representation of data

Text files store data character-by-character by storing the ASCII code of the character

- On most systems, ASCII is the default “encoding” for text files; other possibilities include UTF-8 or UTF-16

Binary files are a sequence of bytes, whose interpretation is application-specific

- Thus, the binary file created by one class of applications is usually incompatible with other applications

A Differentiating Example

If you store the number `0` in a text file, it occupies just one byte of space

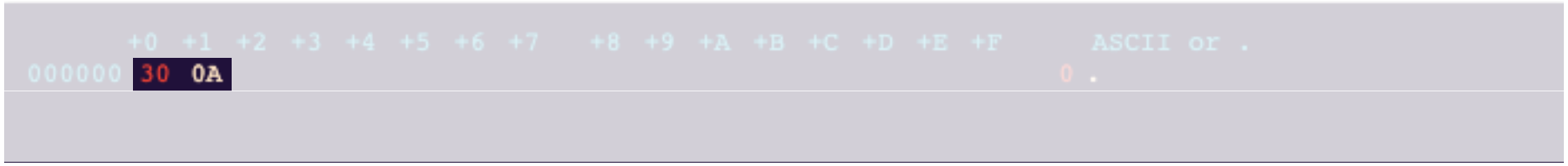
- If the encoding is ASCII, what is stored, is the binary representation of `48` – i.e. `0`'s ASCII code

A Differentiating Example

```
      +0  +1  +2  +3  +4  +5  +6  +7   +8  +9  +A  +B  +C  +D  +E  +F     ASCII or .
000000 30  0A                                                          0 .
```

This is the content of a text file, containing a single character, ‘0’ and a linefeed character

A Differentiating Example



30 in hexadecimal means 48, and 0A is 10 in hexadecimal (the ASCII code for a line feed)

A Differentiating Example

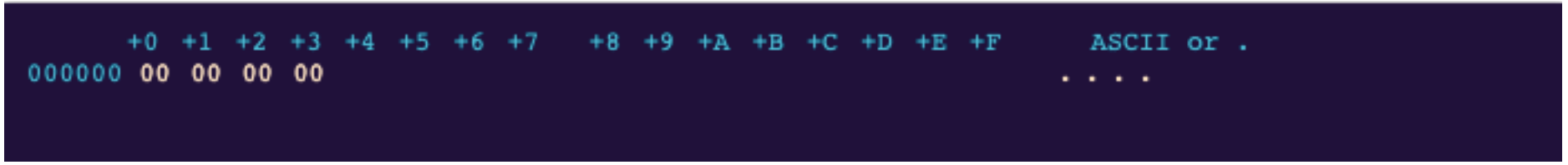
If you store the number `0` in a text file, it occupies just one byte of space

- If the encoding is ASCII, what is stored, is the binary representation of `48` – i.e. `0`'s ASCII code

If you store the number `0` in a binary file, say as an integer, it may occupy two or four bytes…

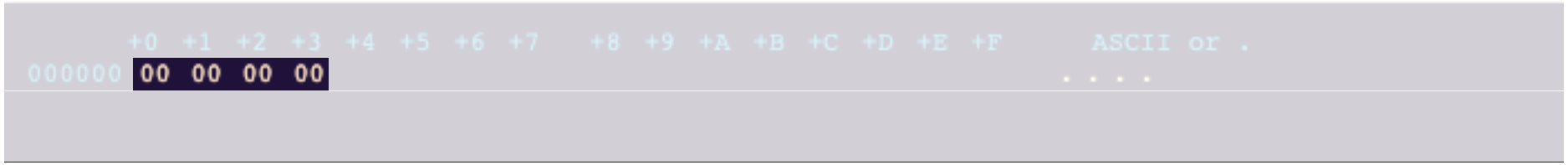
- … depending on the number of bytes that integers take on that machine
- What is stored is the binary representation of `0` in 16 or 32 bits

A Differentiating Example



This is the content of a binary file, containing a single integer 0

A Differentiating Example



0 is being represented in 4 bytes as well, by “all zeroes”

Accessing files through Programs

When you attempt to work with a file through a program, you should know the type of the file…

- … i.e., text or binary (as well as any related encoding or format)

Accessing files through Programs

When you attempt to work with a file through a program, you should know the type of the file…

- … i.e., text or binary (as well as any related encoding or format)

Next, you need to decide whether you want to read the file, write to it, or do both

Accessing files through Programs

When you attempt to work with a file through a program, you should know the type of the file…

- … i.e., text or binary (as well as any related encoding or format)

Next, you need to decide whether you want to read the file, write to it, or do both

In most programming languages, you get library support for four types of tasks:

Accessing files through Programs

When you attempt to work with a file through a program, you should know the type of the file…

- … i.e., text or binary (as well as any related encoding or format)

Next, you need to decide whether you want to read the file, write to it, or do both

In most programming languages, you get library support for four types of tasks:

- Reading from text files
- Writing to text files

Accessing files through Programs

When you attempt to work with a file through a program, you should know the type of the file…

- … i.e., text or binary (as well as any related encoding or format)

Next, you need to decide whether you want to read the file, write to it, or do both

In most programming languages, you get library support for four types of tasks:

- Reading from text files
- Writing to text files
- Reading from binary files
- Writing to binary files

Accessing files through Programs

When you attempt to work with a file through a program, you should know the type of the file…

- … i.e., text or binary (as well as any related encoding or format)

Next, you need to decide whether you want to read the file, write to it, or do both

In most programming languages, you get library support for four types of tasks:

- Reading from text files
- Writing to text files
- Reading from binary files
- Writing to binary files

Based on your needs, you must pick the correct set of library resources for your task

Interacting with files in C

In C, the starting point for interacting with a file, is creating an instance of a `FILE` pointer

Interacting with files in C

In C, the starting point for interacting with a file, is creating an instance of a `FILE` pointer

`FILE` is an alias for a structure that keeps important information about the file we wish to use

Interacting with files in C

In C, the starting point for interacting with a file, is creating an instance of a `FILE` pointer

`FILE` is an alias for a structure that keeps important information about the file we wish to use

You begin the operations on the file, by using the `fopen()` function

Interacting with files in C

In C, the starting point for interacting with a file, is creating an instance of a `FILE` pointer

`FILE` is an alias for a structure that keeps important information about the file we wish to use

You begin the operations on the file, by using the `fopen()` function

- You do so like this:
  `<file pointer> = fopen(<name of the file>, <access mode(s)>)`

Interacting with files in C

In C, the starting point for interacting with a file, is creating an instance of a `FILE` pointer

`FILE` is an alias for a structure that keeps important information about the file we wish to use

You begin the operations on the file, by using the `fopen()` function

- You do so like this:
  `<file pointer> = fopen(<name of the file>, <access mode(s)>)`

The access mode string provides information about the kind of operations we would perform

Interacting with files in C

In C, the starting point for interacting with a file, is creating an instance of a `FILE` pointer

`FILE` is an alias for a structure that keeps important information about the file we wish to use

You begin the operations on the file, by using the `fopen()` function

- You do so like this:
  `<file pointer> = fopen(<name of the file>, <access mode(s)>)`

The access mode string provides information about the kind of operations we would perform

- For instance, the access mode strings for reading and writing text files are `"r"` and `"w"` respectively…

Interacting with files in C

In C, the starting point for interacting with a file, is creating an instance of a `FILE` pointer

`FILE` is an alias for a structure that keeps important information about the file we wish to use

You begin the operations on the file, by using the `fopen()` function

- You do so like this:
  `<file pointer> = fopen(<name of the file>, <access mode(s)>)`

The access mode string provides information about the kind of operations we would perform

- For instance, the access mode strings for reading and writing text files are `"r"` and `"w"` respectively…
- … and that for reading and writing binary files are `"rb"` and `"wb"` respectively

Interacting with files in C

In C, the starting point for interacting with a file, is creating an instance of a `FILE` pointer

`FILE` is an alias for a structure that keeps important information about the file we wish to use

You begin the operations on the file, by using the `fopen()` function

- You do so like this:
  `<file pointer> = fopen(<name of the file>, <access mode(s)>)`

The access mode string provides information about the kind of operations we would perform

- For instance, the access mode strings for reading and writing text files are `"r"` and `"w"` respectively…
- … and that for reading and writing binary files are `"rb"` and `"wb"` respectively
- There are many other possible access modes, but we will not discuss them (check your homework though)

Interacting with files in C

In C, the starting point for interacting with a file, is creating an instance of a `FILE` pointer

`FILE` is an alias for a structure that keeps important information about the file we wish to use

You begin the operations on the file, by using the `fopen()` function

- You do so like this:
  `<file pointer> = fopen(<name of the file>, <access mode(s)>)`

The access mode string provides information about the kind of operations we would perform

- For instance, the access mode strings for reading and writing text files are `"r"` and `"w"` respectively…
- … and that for reading and writing binary files are `"rb"` and `"wb"` respectively
- There are many other possible access modes, but we will not discuss them (check your homework though)

The `FILE` pointer can then be passed to a set of file access functions in C to perform various tasks

Interacting with files in C

In C, the starting point for interacting with a file, is creating an instance of a `FILE` pointer

`FILE` is an alias for a structure that keeps important information about the file we wish to use

You begin the operations on the file, by using the `fopen()` function

- You do so like this:
  `<file pointer> = fopen(<name of the file>, <access mode(s)>)`

The access mode string provides information about the kind of operations we would perform

- For instance, the access mode strings for reading and writing text files are `"r"` and `"w"` respectively…
- … and that for reading and writing binary files are `"rb"` and `"wb"` respectively
- There are many other possible access modes, but we will not discuss them (check your homework though)

The `FILE` pointer can then be passed to a set of file access functions in C to perform various tasks

When done, the `FILE` pointer is passed to the `fclose()` function to free up any allocated resources

Writing *Characters* to text files

As discussed before, text files store data character by character

Writing *Characters* to text files

As discussed before, text files store data character by character

Two common functions that are often used for this purpose, are `fprintf()` and `fputs()`

Writing *Characters* to text files

As discussed before, text files store data character by character

Two common functions that are often used for this purpose, are `fprintf()` and `fputs()`

Both the functions expect a `FILE` pointer as input, with the file opened in writing mode

Writing *Characters* to text files

As discussed before, text files store data character by character

Two common functions that are often used for this purpose, are `fprintf()` and `fputs()`

Both the functions expect a `FILE` pointer as input, with the file opened in writing mode

If you write a string, they essentially write each character in that string to the file, in that order

Example – Writing a simple string

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

int main()
{
        char str[102];
        char file_name[22];
        FILE* fptr = NULL;

        printf("Enter a message (up to 100 characters): ");
        fgets(str, 100, stdin);

        printf("Enter the name of the file to store the message (max 20 chars): ");
        fgets(file_name, 20, stdin);
        file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

        fptr = fopen(file_name, "w"); //"w" - write

        fprintf(fptr, "%s", str);

        printf("Saved your message in %s. Use the MessageReader program to read it.\n", file_name);

        fclose(fptr);
}
```

Example – Writing a simple string

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

int main()
{
        char str[102];
        char file_name[22];
        FILE* fptr = NULL;

        printf("Enter a message (up to 100 characters): ");
        fgets(str, 100, stdin);

        printf("Enter the name of the file to store the message (max 20 chars): ");
        fgets(file_name, 20, stdin);
        file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

        fptr = fopen(file_name, "w"); //"w" - write

        fprintf(fptr, "%s", str);

        printf("Saved your message in %s. Use the MessageReader program to read it.\n", file_name);

        fclose(fptr);
}
```

The first step, as discussed, is to open the file in a writing mode, and obtain a `FILE` pointer to it

Example – Writing a simple string

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

int main()
{
        char str[102];
        char file_name[22];
        FILE* fptr = NULL;

        printf("Enter a message (up to 100 characters): ");
        fgets(str, 100, stdin);

        printf("Enter the name of the file to store the message (max 20 chars): ");
        fgets(file_name, 20, stdin);
        file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

        fptr = fopen(file_name, "w"); //"w" - write

        fprintf(fptr, "%s", str);

        printf("Saved your message in %s. Use the MessageReader program to read it.\n", file_name);

        fclose(fptr);
}
```

We can then use the `fprintf()` function to write a string to the beginning of the opened file

Example – Writing a simple string

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

int main()
{
        char str[102];
        char file_name[22];
        FILE* fptr = NULL;

        printf("Enter a message (up to 100 characters): ");
        fgets(str, 100, stdin);

        printf("Enter the name of the file to store the message (max 20 chars): ");
        fgets(file_name, 20, stdin);
        file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

        fptr = fopen(file_name, "w"); //"w" - write

        fprintf(fptr, "%s", str);

        printf("Saved your message in %s. Use the MessageReader program to read it.\n", file_name);

        fclose(fptr);
}
```

`fprintf()` is exactly like `printf()`, with one additional input – a `FILE` pointer

We can then use the `fprintf()` function to write a string to the beginning of the opened file

Example – Writing a simple string

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

int main()
{
        char str[102];
        char file_name[22];
        FILE* fptr = NULL;

        printf("Enter a message (up to 100 characters): ");
        fgets(str, 100, stdin);

        printf("Enter the name of the file to store the message (max 20 chars): ");
        fgets(file_name, 20, stdin);
        file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

        fptr = fopen(file_name, "w"); //"w" - write

        fprintf(fptr, "%s", str);

        printf("Saved your message in %s. Use the MessageReader program to read it.\n", file_name);

        fclose(fptr);
}
```

Finally, we close the resources associated with the file by calling the `fclose()` function

Example – Writing a simple string

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ gcc -o MessageSaver MessageSaver.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ ./MessageSaver
Enter a message (up to 100 characters): This is a sample message
Enter the name of the file to store the message (max 20 chars): message.txt
Saved your message in message.txt. Use the MessageReader program to read it.
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$
```

Example – Writing a simple string

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ gcc -o MessageSaver MessageSaver.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ ./MessageSaver
Enter a message (up to 100 characters): This is a sample message
Enter the name of the file to store the message (max 20 chars): message.txt
Saved your message in message.txt. Use the MessageReader program to read it.
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$
```

The message gets saved as *readable text* in the chosen file

Reading *Characters* from text files

To read the written text, the starting point again is to obtain the `FILE` pointer with proper access

Reading *Characters* from text files

To read the written text, the starting point again is to obtain the `FILE` pointer with proper access

Two common functions which can read text from a file are `fscanf()` and `fgets()`

Reading *Characters* from text files

To read the written text, the starting point again is to obtain the `FILE` pointer with proper access

Two common functions which can read text from a file are `fscanf()` and `fgets()`

The obtained `FILE` pointer is required in either case

Example – Reading a simple string

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

int main()
{
        FILE* fptr = NULL;
        char str[102];
        char file_name[22];

        printf("Enter the name of the file containing the message (max 20 chars): ");
        fgets(file_name, 20, stdin);

        file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

        fptr = fopen(file_name, "r"); //"r" - read

        fgets(str, 100, fptr);

        printf("The message is:\n%s", str);

        fclose(fptr);
}
```

Example – Reading a simple string

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

int main()
{
        FILE* fptr = NULL;
        char str[102];
        char file_name[22];

        printf("Enter the name of the file containing the message (max 20 chars): ");
        fgets(file_name, 20, stdin);

        file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

        fptr = fopen(file_name, "r"); //"r" - read

        fgets(str, 100, fptr);

        printf("The message is:\n%s", str);

        fclose(fptr);
}
```

The start is fairly similar – getting the `FILE` pointer in the correct access mode

Example – Reading a simple string

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

int main()
{
        FILE* fptr = NULL;
        char str[102];
        char file_name[22];

        printf("Enter the name of the file containing the message (max 20 chars): ");
        fgets(file_name, 20, stdin);

        file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

        fptr = fopen(file_name, "r"); //"r" - read

        fgets(str, 100, fptr);

        printf("The message is:\n%s", str);

        fclose(fptr);
}
```

Then, we can read text from the file using the `fgets()` function (you may have already used it by now)

Example – Reading a simple string

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

int main()
{
        FILE* fptr = NULL;
        char str[102];
        char file_name[22];

        printf("Enter the name of the file containing the message (max 20 chars): ");
        fgets(file_name, 20, stdin);

        file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

        fptr = fopen(file_name, "r"); //"r" - read

        fgets(str, 100, fptr);

        printf("The message is:\n%s", str);

        fclose(fptr);
}
```

This will read “up to” 100 characters from the text file, into the `str` `char` array

Then, we can read text from the file using the `fgets()` function (you may have already used it by now)

Example – Reading a simple string

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

int main()
{
        FILE* fptr = NULL;
        char str[102];
        char file_name[22];

        printf("Enter the name of the file containing the message (max 20 chars): ");
        fgets(file_name, 20, stdin);

        file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

        fptr = fopen(file_name, "r"); //"r" - read

        fgets(str, 100, fptr);

        printf("The message is:\n%s", str);

        fclose(fptr);
}
```

This will read “up to” 100 characters from the text file, into the `str` `char` array

The read may terminate before 100 characters, if a `\n` character is found

Then, we can read text from the file using the `fgets()` function (you may have already used it by now)

Example – Reading a simple string

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ gcc -o MessageReader MessageReader.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ ./MessageReader
Enter the name of the file containing the message (max 20 chars): message.txt
The message is:
This is a sample message
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$
```

Example – Reading a simple string

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ gcc -o MessageReader MessageReader.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ ./MessageReader
Enter the name of the file containing the message (max 20 chars): message.txt
The message is:
This is a sample message
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$
```

This is the same string that we wrote to the file

A word about binary files

Binary files are fairly different from text files

A word about binary files

Binary files are fairly different from text files

You need to come up with a *format*, before you read or write them

A word about binary files

Binary files are fairly different from text files

You need to come up with a *format*, before you read or write them

The format involves precise understanding of the role of each byte, that gets written to the file

- For example, you may choose a format which writes an `int`, followed by a `double`, and then a `char`

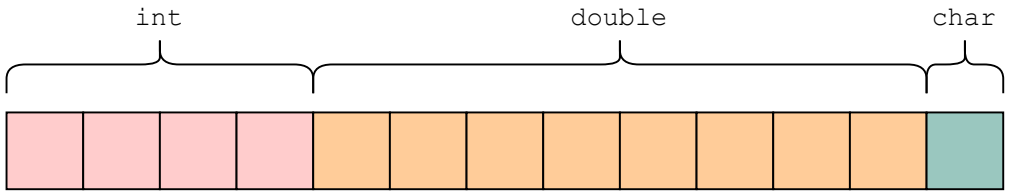
A word about binary files

Binary files are fairly different from text files

You need to come up with a *format*, before you read or write them

The format involves precise understanding of the role of each byte, that gets written to the file

- For example, you may choose a format which writes an `int`, followed by a `double`, and then a `char`
- This will write a total of 13 bytes to the file, in that precise order, as shown



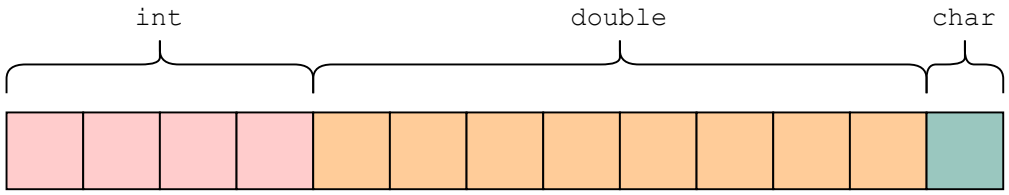
A word about binary files

Binary files are fairly different from text files

You need to come up with a *format*, before you read or write them

The format involves precise understanding of the role of each byte, that gets written to the file

- For example, you may choose a format which writes an `int`, followed by a `double`, and then a `char`
- This will write a total of 13 bytes to the file, in that precise order, as shown



- The file itself will just look like a set of 13 bytes (which can be interpreted in multiple ways)

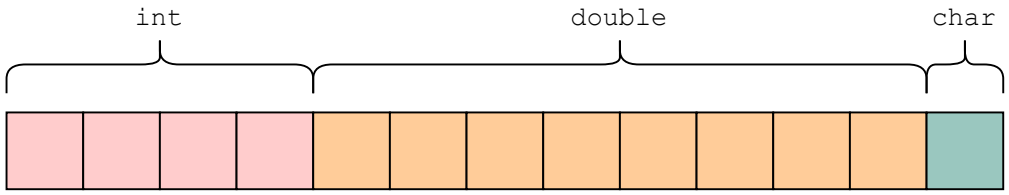
A word about binary files

Binary files are fairly different from text files

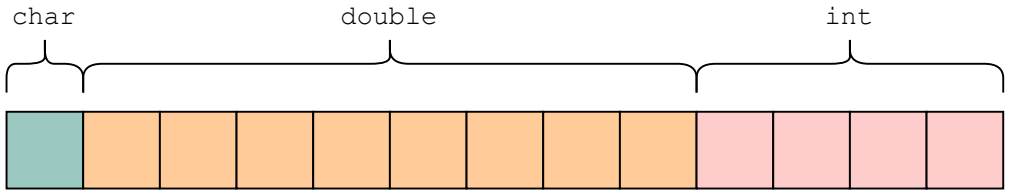
You need to come up with a *format*, before you read or write them

The format involves precise understanding of the role of each byte, that gets written to the file

- For example, you may choose a format which writes an `int`, followed by a `double`, and then a `char`
- This will write a total of 13 bytes to the file, in that precise order, as shown



- The file itself will just look like a set of 13 bytes (which can be interpreted in multiple ways)
- While reading, we must be aware of the format used for creating the file, since it can also be interpreted as



Writing *Bytes* to binary files

By default, if you open a file with `fopen()`, it is opened in text mode

Writing *Bytes* to binary files

By default, if you open a file with `fopen()`, it is opened in text mode

You need to add the suffix `b` in order to open it in binary mode

Writing *Bytes* to binary files

By default, if you open a file with `fopen()`, it is opened in text mode

You need to add the suffix `b` in order to open it in binary mode

After coming up with a specific format for your program or application…

- … you can write data to a file using the `fwrite()` function

Writing *Bytes* to binary files

By default, if you open a file with `fopen()`, it is opened in text mode

You need to add the suffix `b` in order to open it in binary mode

After coming up with a specific format for your program or application…

- … you can write data to a file using the `fwrite()` function

You'll need to provide the following to this function

- The `FILE` pointer, as always

Writing *Bytes* to binary files

By default, if you open a file with `fopen()`, it is opened in text mode

You need to add the suffix `b` in order to open it in binary mode

After coming up with a specific format for your program or application…

- … you can write data to a file using the `fwrite()` function

You'll need to provide the following to this function

- The `FILE` pointer, as always
- The size of the data type that you wish to write (e.g. `4` for `int` or `8` for `double`)

Writing *Bytes* to binary files

By default, if you open a file with `fopen()`, it is opened in text mode

You need to add the suffix `b` in order to open it in binary mode

After coming up with a specific format for your program or application…

- … you can write data to a file using the `fwrite()` function

You'll need to provide the following to this function

- The `FILE` pointer, as always
- The size of the data type that you wish to write (e.g. `4` for `int` or `8` for `double`)
- The number of elements to write – this makes sense when you are writing an array; otherwise it is `1`

Writing *Bytes* to binary files

By default, if you open a file with `fopen()`, it is opened in text mode

You need to add the suffix `b` in order to open it in binary mode

After coming up with a specific format for your program or application…

- … you can write data to a file using the `fwrite()` function

You'll need to provide the following to this function

- The `FILE` pointer, as always
- The size of the data type that you wish to write (e.g. `4` for `int` or `8` for `double`)
- The number of elements to write – this makes sense when you are writing an array; otherwise it is `1`
- A pointer to the data to be written – or the name of the array, if more than one elements are to be written

Example – Writing a list of integers

```c
int number_of_elements, i;
int* elements = NULL;
FILE* fptr = NULL;
char file_name[22];

printf("Enter the number of integers in your list: ");
scanf("%d", &number_of_elements);

elements = (int*) malloc(sizeof(int) * number_of_elements);
printf("Enter the elements of the list:\n");
for(i = 0; i < number_of_elements; i++)
        scanf("%d", &elements[i]);

clean_stdin();

printf("Enter the name of the file to store the list (max 20 chars): ");
fgets(file_name, 20, stdin);

file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

fptr = fopen(file_name, "wb"); //"w" - write, "b" - in binary mode (instead of text)

// Write the size of the list first
fwrite(&number_of_elements, sizeof(int), 1, fptr);

// Now write the list
fwrite(elements, sizeof(int), number_of_elements, fptr);

printf("Saved your list in %s. Use the ListReader program to read it.\n", file_name);

free(elements);
fclose(fptr);
```

Example – Writing a list of integers

```c
int number_of_elements, i;
int* elements = NULL;
FILE* fptr = NULL;
char file_name[22];

printf("Enter the number of integers in your list: ");
scanf("%d", &number_of_elements);

elements = (int*) malloc(sizeof(int) * number_of_elements);
printf("Enter the elements of the list:\n");
for(i = 0; i < number_of_elements; i++)
        scanf("%d", &elements[i]);

clean_stdin();

printf("Enter the name of the file to store the list (max 20 chars): ");
fgets(file_name, 20, stdin);

file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

fptr = fopen(file_name, "wb"); //"w" - write, "b" - in binary mode (instead of text)

// Write the size of the list first
fwrite(&number_of_elements, sizeof(int), 1, fptr);

// Now write the list
fwrite(elements, sizeof(int), number_of_elements, fptr);

printf("Saved your list in %s. Use the ListReader program to read it.\n", file_name);

free(elements);
fclose(fptr);
```

The starting point is similar, getting the `FILE` pointer…

Example – Writing a list of integers

```c
int number_of_elements, i;
int* elements = NULL;
FILE* fptr = NULL;
char file_name[22];

printf("Enter the number of integers in your list: ");
scanf("%d", &number_of_elements);

elements = (int*) malloc(sizeof(int) * number_of_elements);
printf("Enter the elements of the list:\n");
for(i = 0; i < number_of_elements; i++)
        scanf("%d", &elements[i]);

clean_stdin();

printf("Enter the name of the file to store the list (max 20 chars): ");
fgets(file_name, 20, stdin);

file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

fptr = fopen(file_name, "wb"); //"w" - write, "b" - in binary mode (instead of text)

// Write the size of the list first
fwrite(&number_of_elements, sizeof(int), 1, fptr);

// Now write the list
fwrite(elements, sizeof(int), number_of_elements, fptr);

printf("Saved your list in %s. Use the ListReader program to read it.\n", file_name);

free(elements);
fclose(fptr);
```

The starting point is similar, getting the `FILE` pointer…

… but with `fopen()`, the suffix `b` needs to be provided in the access mode string

Example – Writing a list of integers

```
int number_of_elements, i;
int* elements = NULL;
FILE* fptr = NULL;
char file_name[22];

printf("Enter the number of integers in your list: ");
scanf("%d", &number_of_elements);

elements = (int*) malloc(sizeof(int) * number_of_elements);
printf("Enter the elements of the list:\n");
for(i = 0; i < number_of_elements; i++)
        scanf("%d", &elements[i]);

clean_stdin();

printf("Enter the name of the file to store the list (max 20 chars): ");
fgets(file_name, 20, stdin);

file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

fptr = fopen(file_name, "wb"); //"w" - write, "b" - in binary mode (instead of text)

// Write the size of the list first
fwrite(&number_of_elements, sizeof(int), 1, fptr);

// Now write the list
fwrite(elements, sizeof(int), number_of_elements, fptr);

printf("Saved your list in %s. Use the ListReader program to read it.\n", file_name);

free(elements);
fclose(fptr);
```

Next, we first write the number of integers in the list

Example – Writing a list of integers

```
int number_of_elements, i;
int* elements = NULL;
FILE* fptr = NULL;
char file_name[22];

printf("Enter the number of integers in your list: ");
scanf("%d", &number_of_elements);

elements = (int*) malloc(sizeof(int) * number_of_elements);
printf("Enter the elements of the list:\n");
for(i = 0; i < number_of_elements; i++)
        scanf("%d", &elements[i]);

clean_stdin();

printf("Enter the name of the file to store the list (max 20 chars): ");
fgets(file_name, 20, stdin);

file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

fptr = fopen(file_name, "wb"); //"w" - write, "b" - in binary mode (instead of text)

// Write the size of the list first
fwrite(&number_of_elements, sizeof(int), 1, fptr);

// Now write the list
fwrite(elements, sizeof(int), number_of_elements, fptr);

printf("Saved your list in %s. Use the ListReader program to read it.\n", file_name);

free(elements);
fclose(fptr);
```

Next, we first write the number of integers in the list

Note that `number_of_elements` is an `int` variable, and we have given `1` as the argument for number of elements to write

Example – Writing a list of integers

```
int number_of_elements, i;
int* elements = NULL;
FILE* fptr = NULL;
char file_name[22];

printf("Enter the number of integers in your list: ");
scanf("%d", &number_of_elements);

elements = (int*) malloc(sizeof(int) * number_of_elements);
printf("Enter the elements of the list:\n");
for(i = 0; i < number_of_elements; i++)
        scanf("%d", &elements[i]);

clean_stdin();

printf("Enter the name of the file to store the list (max 20 chars): ");
fgets(file_name, 20, stdin);

file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

fptr = fopen(file_name, "wb"); //"w" - write, "b" - in binary mode (instead of text)

// Write the size of the list first
fwrite(&number_of_elements, sizeof(int), 1, fptr);

// Now write the list
fwrite(elements, sizeof(int), number_of_elements, fptr);

printf("Saved your list in %s. Use the ListReader program to read it.\n", file_name);

free(elements);
fclose(fptr);
```

Then, we write the list itself

Example – Writing a list of integers

```
int number_of_elements, i;
int* elements = NULL;
FILE* fptr = NULL;
char file_name[22];

printf("Enter the number of integers in your list: ");
scanf("%d", &number_of_elements);

elements = (int*) malloc(sizeof(int) * number_of_elements);
printf("Enter the elements of the list:\n");
for(i = 0; i < number_of_elements; i++)
        scanf("%d", &elements[i]);

clean_stdin();

printf("Enter the name of the file to store the list (max 20 chars): ");
fgets(file_name, 20, stdin);

file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

fptr = fopen(file_name, "wb"); //"w" - write, "b" - in binary mode (instead of text)

// Write the size of the list first
fwrite(&number_of_elements, sizeof(int), 1, fptr);

// Now write the list
fwrite(elements, sizeof(int), number_of_elements, fptr);

printf("Saved your list in %s. Use the ListReader program to read it.\n", file_name);

free(elements);
fclose(fptr);
```

Then, we write the list itself

Note that `elements` is an `int` array, and we have given `number_of_elements` as the argument for number of elements to write

Example – Writing a list of integers

```
int number_of_elements, i;
int* elements = NULL;
FILE* fptr = NULL;
char file_name[22];

printf("Enter the number of integers in your list: ");
scanf("%d", &number_of_elements);

elements = (int*) malloc(sizeof(int) * number_of_elements);
printf("Enter the elements of the list:\n");
for(i = 0; i < number_of_elements; i++)
        scanf("%d", &elements[i]);

clean_stdin();

printf("Enter the name of the file to store the list (max 20 chars): ");
fgets(file_name, 20, stdin);

file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

fptr = fopen(file_name, "wb"); //"w" - write, "b" - in binary mode (instead of text)

// Write the size of the list first
fwrite(&number_of_elements, sizeof(int), 1, fptr);

// Now write the list
fwrite(elements, sizeof(int), number_of_elements, fptr);

printf("Saved your list in %s. Use the ListReader program to read it.\n", file_name);

free(elements);
fclose(fptr);
```

Then, we write the list itself

Note that `elements` is an `int` array, and we have given `number_of_elements` as the argument for number of elements to write

In essence, this is our *format* for the binary file – number of elements in the list, followed by the list itself

Example – Writing a list of integers

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ gcc -o ListSaver ListSaver.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ ./ListSaver
Enter the number of integers in your list: 5
Enter the elements of the list:
7
11
3
29
23
Enter the name of the file to store the list (max 20 chars): list.bin
Saved your list in list.bin. Use the ListReader program to read it.
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$
```

Example – Writing a list of integers

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ gcc -o ListSaver ListSaver.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ ./ListSaver
Enter the number of integers in your list: 5
Enter the elements of the list:
7
11
3
29
23
Enter the name of the file to store the list (max 20 chars): list.bin
Saved your list in list.bin. Use the ListReader program to read it.
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$
```

This extension `.bin` or `.dat` are commonly used to signify binary data

Reading *Bytes* from binary files

Again, the `b` suffix is required with `fopen()`, to open a file in binary mode

Reading *Bytes* from binary files

Again, the `b` suffix is required with `fopen()`, to open a file in binary mode

The rest of the process is fairly similar; instead of `fwrite()` you can use `fread()` function

Reading *Bytes* from binary files

Again, the `b` suffix is required with `fopen()`, to open a file in binary mode

The rest of the process is fairly similar; instead of `fwrite()` you can use `fread()` function

The parameters to `fread()` are also similar to `fwrite()`

- The `FILE` pointer

Reading *Bytes* from binary files

Again, the `b` suffix is required with `fopen()`, to open a file in binary mode

The rest of the process is fairly similar; instead of `fwrite()` you can use `fread()` function

The parameters to `fread()` are also similar to `fwrite()`

- The `FILE` pointer
- The size of the data type that you wish to read (e.g. `4` for `int` or `8` for `double`)

Reading *Bytes* from binary files

Again, the `b` suffix is required with `fopen()`, to open a file in binary mode

The rest of the process is fairly similar; instead of `fwrite()` you can use `fread()` function

The parameters to `fread()` are also similar to `fwrite()`

- The `FILE` pointer
- The size of the data type that you wish to read (e.g. `4` for `int` or `8` for `double`)
- The number of elements to read – this makes sense when you are reading an array; otherwise it is `1`

Reading *Bytes* from binary files

Again, the `b` suffix is required with `fopen()`, to open a file in binary mode

The rest of the process is fairly similar; instead of `fwrite()` you can use `fread()` function

The parameters to `fread()` are also similar to `fwrite()`

- The `FILE` pointer
- The size of the data type that you wish to read (e.g. `4` for `int` or `8` for `double`)
- The number of elements to read – this makes sense when you are reading an array; otherwise it is `1`
- A pointer to a variable or array, in which the data will be read

Reading *Bytes* from binary files

Again, the `b` suffix is required with `fopen()`, to open a file in binary mode

The rest of the process is fairly similar; instead of `fwrite()` you can use `fread()` function

The parameters to `fread()` are also similar to `fwrite()`

- The `FILE` pointer
- The size of the data type that you wish to read (e.g. `4` for `int` or `8` for `double`)
- The number of elements to read – this makes sense when you are reading an array; otherwise it is `1`
- A pointer to a variable or array, in which the data will be read
  - You remember the use of call-by-reference to take outputs from a function? This is another example of that…

Example – Reading a list of integers

```c
int number_of_elements, i;
int temp;
FILE* fptr = NULL;
char file_name[22];

printf("Enter the name of the file containing the list (max 20 chars): ");
fgets(file_name, 20, stdin);

file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

fptr = fopen(file_name, "rb"); //"r" - read, "b" - in binary mode (instead of text)

// Read the number of elements in the list
fread(&number_of_elements, sizeof(int), 1, fptr);

printf("This is what I read:\n");
// Now read the list
for(i = 0; i < number_of_elements; i++)
{
        fread(&temp, sizeof(int), 1, fptr);
        printf("%d\n", temp);
}

fclose(fptr);
```

Example – Reading a list of integers

```
int number_of_elements, i;
int temp;
FILE* fptr = NULL;
char file_name[22];

printf("Enter the name of the file containing the list (max 20 chars): ");
fgets(file_name, 20, stdin);

file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

fptr = fopen(file_name, "rb"); //"r" - read, "b" - in binary mode (instead of text)

// Read the number of elements in the list
fread(&number_of_elements, sizeof(int), 1, fptr);

printf("This is what I read:\n");
// Now read the list
for(i = 0; i < number_of_elements; i++)
{
        fread(&temp, sizeof(int), 1, fptr);
        printf("%d\n", temp);
}

fclose(fptr);
```

The suffix `b` is important here

Example – Reading a list of integers

```
int number_of_elements, i;
int temp;
FILE* fptr = NULL;
char file_name[22];

printf("Enter the name of the file containing the list (max 20 chars): ");
fgets(file_name, 20, stdin);

file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

fptr = fopen(file_name, "rb"); //"r" - read, "b" - in binary mode (instead of text)

// Read the number of elements in the list
fread(&number_of_elements, sizeof(int), 1, fptr);

printf("This is what I read:\n");
// Now read the list
for(i = 0; i < number_of_elements; i++)
{
        fread(&temp, sizeof(int), 1, fptr);
        printf("%d\n", temp);
}

fclose(fptr);
```

As per the expected format, we read an integer from the file first, and expect that this is the size of the list

Example – Reading a list of integers

```
int number_of_elements, i;
int temp;
FILE* fptr = NULL;
char file_name[22];

printf("Enter the name of the file containing the list (max 20 chars): ");
fgets(file_name, 20, stdin);

file_name[strlen(file_name)-1] = '\0'; // To remove the extra \n character

fptr = fopen(file_name, "rb"); //"r" - read, "b" - in binary mode (instead of text)

// Read the number of elements in the list
fread(&number_of_elements, sizeof(int), 1, fptr);

printf("This is what I read:\n");
// Now read the list
for(i = 0; i < number_of_elements; i++)
{
        fread(&temp, sizeof(int), 1, fptr);
        printf("%d\n", temp);
}

fclose(fptr);
```

Then, we read *the expected* number of integers from the file, and show it as the stored list

Example – Reading a list of integers

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ gcc -o ListReader ListReader.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ ./ListReader
Enter the name of the file containing the list (max 20 chars): list.bin
This is what I read:
7
11
3
29
23
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$
```

Example – Reading a list of integers

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ gcc -o ListReader ListReader.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$ ./ListReader
Enter the name of the file containing the list (max 20 chars): list.bin
This is what I read:
7
11
3
29
23
saurabh@saurabh-VirtualBox:~/C/examples/Week 9$
```

This is the same list that we wrote

Homework !!

Appending, is a type of writing, with some major differences

- Find out the differences

There are several options at your disposal, when you open a file with `fopen()`

- Read the man page for `fopen()` to know about all of the possibilities (use the command `man fopen`)
- A summary with some sample usage is also provided here: https://www.geeksforgeeks.org/c-fopen-function-with-examples/