Assignment 2: blobby, Blobby

version: 1.7 last updated: 2020-11-21 17:30:00

Aims

- building a concrete understanding of file system objects
- practising C including byte-level operations
- understanding file operations, including I/O of binary data and robust error-handling

Background

A file archive is a single file which can contain the contents, names and other metadata of multiple files.
File archives make backups and transport of files more convenient, and make compression more efficient.

The some common archive formats are:

- tar for linux. (uses external compression tools)
- zip for windows. (with inbuilt compression)

Wikipedia's list of common file archive formats.

Your Task

Your task in this assignment is to write a C program **blobby.c**, a file archiver.

The file archives in this assignment are called ***blob***s.
Each ***blob*** contains one or more ***blobette***s.
Each ***blobette*** records one file system object.
Their format is described below.

blobby.c should be able to:

- list the contents of a *blob* (subset 0),
- list the permissions of files in a *blob* (subset 0),
- list the size (number of bytes) of files in a *blob* (subset 0),
- check the *blobette* magic number (subset 0),
- extract files from a *blob* (subset 1),
- check *blobette* integrity (hashes) (subset 1),
- set the file permissions of files extracted from a *blob* (subset 1),
- create a *blob* from a list of files (subset 2),
- list, extract, and create *blobs* that include directories (subset 3),
- list, extract, and create *blobs* that are compressed (subset 4).

Subset 0

Given the `-l` command line argument `blobby.c` should for each file in the specified *blob* print:

1. The file/directory permissions in octal
2. The file/directory size in bytes
3. The file/directory pathname

It should check *blob* magic number on each *blobette*, and emit an error if it is incorrect.

```
# List the details of each item in the blob called text_file.blob, which is in the examples directory
$ ./blobby -l examples/text_file.blob
100644    56 hello.txt
# List the details of each item in the blob called 4_files.blob, which is in the examples directory
$ ./blobby -l examples/4_files.blob
100644   256 256.bin
100644    56 hello.txt
100444   166 last_goodbye.txt
100464   148 these_days.txt
# List the details of each item in the blob called hello_world.blob, which is in the examples directory
$ ./blobby -l examples/hello_world.blob
100644    87 hello.c
100644    82 hello.cpp
100644    65 hello.d
100644    77 hello.go
100644    32 hello.hs
100644   117 hello.java
100644    30 hello.js
100755    47 hello.pl
100755    97 hello.py
100644    45 hello.rs
100644   107 hello.s
100755    41 hello.sh
100644    24 hello.sql
```

HINT:

Use *fopen* to open the blob file.

Use *fgetc* to read bytes.

Use C bitwise operations such as `<<` `&` and `|` to combine bytes into integers.

Think carefully about the functions you can construct to avoid repeated code.

Review `print_borts_file.c`, from our week 8 tutorial and `print_bytes.c` from our week 8 lab.

fseek can be used to skip over parts of the *blob* file, but you can also use a loop and *fgetc*

NOTE:

The order you list files is the order they appear in the *blob*.

blob files do not necessarily end with *.blob*.
This has been done with the provided example files purely as a convenience.

The starting code contains a suitable printf to match the required format.

The correct format string for this output `"%06lo %5lu %s\n"` is provided in the given `blobby.c` file.

Subset 1

Given the `-x` command line argument `blobby.c` should:

Extract the files in the specified *blob*.

It should set file permissions for extracted files to the permissions specified in the *blob*.

It should check *blob* integrity by checking each *blobette* hash, and emit an error if any are incorrect.

```
# Your program extracts the files into the current working directory.
# So we will run your program from a temporary directory (tmp) as to not clutter the base directory.
# Once in the tmp directory your program and examples/ will be in the parent directory.
# Hence the use of .. in pathnames.

# Make a directory called tmp.
$ mkdir -p tmp/
# Change into the tmp directory.
$ cd tmp/
# Forcibly remove all files inside the tmp directory.
$ rm -f * .*
# Use your program to extract the contents of text_file.blob.
$ ../blobby -x ../examples/text_file.blob
Extracting: hello.txt
# Show the contents of hello.txt in the terminal.
# You can manually open it in your text editor too, if you like.
$ cat hello.txt
Hello COMP1521
I hope you are enjoying this assignment.
# Forcibly remove all files inside the tmp directory.
$ rm -f * .*
# Use your program to extract the contents of hello_world.blob.
$ ../blobby -x ../examples/hello_world.blob
Extracting: hello.c
Extracting: hello.cpp
Extracting: hello.d
Extracting: hello.go
Extracting: hello.hs
Extracting: hello.java
Extracting: hello.js
Extracting: hello.pl
Extracting: hello.py
Extracting: hello.rs
Extracting: hello.s
Extracting: hello.sh
Extracting: hello.sql
# Show the first 25 lines from the extracted files to confirm the extraction was successful.
$ cat $(echo * | sort) | head -n 25
extern int puts(const char *s);

int main(void)
{
    puts("Hello, World!");
    return 0;
}
#include <iostream>

int main () {
  std::cout << "Hello, world!" << std::endl;
}
import std.stdio;

void main() {
    writeln("Hello, world!");
}
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
main = putStrLn "Hello, World!"
# Forcibly remove all files inside the tmp directory
$ rm -f * .*
# Use your program to extract the contents of meta.blob.
$ ../blobby -x ../examples/meta.blob
Extracting: 1_file.subdirectory.blob
Extracting: 1_file.subdirectory.compressed.blob
Extracting: 2_files.blob
Extracting: 2_files.compressed.blob
```

```
Extracting: 3_files.bad_hash.blob
Extracting: 3_files.bad_magic.blob
Extracting: 3_files.blob
Extracting: 3_files.compressed.blob
Extracting: 3_files.subdirectory.bad_hash.blob
Extracting: 3_files.subdirectory.bad_magic.blob
Extracting: 3_files.subdirectory.blob
Extracting: 3_files.subdirectory.compressed.blob
Extracting: 4_files.blob
Extracting: 4_files.compressed.blob
Extracting: all_the_modes.subdirectory.blob
Extracting: all_the_modes.subdirectory.compressed.blob
Extracting: binary_file.blob
Extracting: binary_file.compressed.blob
Extracting: hello_world.bad_hash.blob
Extracting: hello_world.bad_magic.blob
Extracting: hello_world.blob
Extracting: hello_world.compressed.blob
Extracting: lecture_code.subdirectory.blob
Extracting: lecture_code.subdirectory.compressed.blob
Extracting: text_file.bad_hash.blob
Extracting: text_file.bad_magic.blob
Extracting: text_file.blob
Extracting: text_file.compressed.blob
# Show the first 10 items in this directory alphabetically to check extraction was successful.
$ ls -1 $(echo * | sort) | head
1_file.subdirectory.blob
1_file.subdirectory.compressed.blob
2_files.blob
2_files.compressed.blob
3_files.bad_hash.blob
3_files.bad_magic.blob
3_files.blob
3_files.compressed.blob
3_files.subdirectory.bad_hash.blob
3_files.subdirectory.bad_magic.blob
# Go back into the directory with your code.
$ cd ../
# Remove the tmp directory and everything inside it.
$ rm -rf tmp/
```

> HINT:
>
> Use *fopen* to open each file you are extracting.
>
> Use *fputc* to write bytes to each file..
>
> In our lectures on files we covered copying bytes to a file in the cp_fgetc.c example and setting the permisisons of a file in the chmod.c example.
>
> Think carefully about the functions you can construct to avoid repeated code.
>
> For example, for every byte you read with `fgetc` you need to call `blobby_hash` to calculate a new hash value, so write a function that does both. Hint: have the function take a pointer to a hash value which it can update.

> NOTE:
>
> blobby should overwrite an files that already exist.
> blobby can leave already extracted/partially extracted files in the event of an error.

Subset 2

Given the `-c` command line argument `blobby.c` should:

Create a *blob* containing the specified files.

```
# These "echo" lines show you how to create these test files and what their contents are.

# Create a file called hello.txt with the contents "hello".
$ echo hello > hello.txt
# Create a file called hola.txt with the contents "hola".
$ echo hola > hola.txt
# Create a file called hi.txt with the contents "hi".
$ echo hi > hi.txt
# Set the permissions of these files to 644 (octal permission string (equivalent to rw-r--r--)).
# When you list the contents of the blob, the permissions should match this.
$ chmod 644 hello.txt hola.txt hi.txt
# Create a blob called selamat.blob with the files hello.txt, hola.txt, and hi.txt.
$ ./blobby -c selamat.blob hello.txt hola.txt hi.txt
Adding: hello.txt
Adding: hola.txt
Adding: hi.txt
# List the contents of selamat.blob.
$ ./blobby -l selamat.blob
100644     6 hello.txt
100644     5 hola.txt
100644     3 hi.txt
# Make a directory called tmp.
$ mkdir -p tmp/
# Change into the tmp directory.
$ cd tmp/
# Forcibly remove all files inside the tmp directory.
$ rm -f * .*
# Use your program to extract the contents of selamat.blob.
$ ../blobby -x ../selamat.blob
Extracting: hello.txt
Extracting: hola.txt
Extracting: hi.txt
# Check that the extracted file hello.txt is the same as the source file ../hello.txt.
$ diff -s ../hello.txt hello.txt
Files ../hello.txt and hello.txt are identical
# Check that the extracted file hola.txt is the same as the source file ../hola.txt.
$ diff -s ../hola.txt hola.txt
Files ../hola.txt and hola.txt are identical
# Check that the extracted file hi.txt is the same as the source file ../hi.txt.
$ diff -s ../hi.txt hi.txt
Files ../hi.txt and hi.txt are identical
# Go back into the directory with your code.
$ cd ../
# Remove the tmp directory and everything inside it.
$ rm -rf tmp/
```

> HINT:
>
> Use *fopen* and *fputc* to create the new blob.
>
> In our lectures on files we covered obtaining file metadata including its size and mode (permissions) in the stat.c example.

> NOTE:
>
> You must add/store files in the order they are given.

Subset 3

Given the **-c** command line argument `blobby.c` should:

Be able to add files in sub-directories, for examples:

```
# Create a blob called a.blob with the file "hello.txt" that is contained within 2 levels of directories.
$ ./blobby -c a.blob examples/2_files.d/hello.txt
Adding: examples
Adding: examples/2_files.d
Adding: examples/2_files.d/hello.txt
```

If a directory is specified when creating a *blob* `blobby.c` should add the entire directory tree to the *blob*.

```
# Create a blob called a.blob with *all* the contents within the directory "3_files.subdirectory.d"
# which is in the "examples" directory.
$ ./blobby -c a.blob examples/3_files.subdirectory.d
Adding: examples
Adding: examples/3_files.subdirectory.d
Adding: examples/3_files.subdirectory.d/goodbye
Adding: examples/3_files.subdirectory.d/goodbye/last_goodbye.txt
Adding: examples/3_files.subdirectory.d/hello
Adding: examples/3_files.subdirectory.d/hello/hello.txt
Adding: examples/3_files.subdirectory.d/these_days.txt
```

Given the **-l** command line argument and a *blob* containing directories, `blobby.c` should:

Be able to list files and directories, for example:

```
$ ./blobby -l examples/1_file.subdirectory.blob
040755     0 hello
100644    56 hello/hello.txt
```

Given the **-x** command line argument and a *blob* containing directories, `blobby.c` should:

Be able to extract files and directories, for example:

```
$ ./blobby -x examples/3_files.subdirectory.blob
Creating directory: goodbye
Extracting: goodbye/last_goodbye.txt
Creating directory: hello
Extracting: hello/hello.txt
Extracting: these_days.txt
```

> HINT:
>
> In our lectures on files we covered creating a directory in the mkdir.c example and listing a directories contents in the list_directory.c example.
>
> Traversing a directory tree is challenging and can be done in several ways.

> NOTE:
>
> The *blobby* reference implementation will add subdirectories in alphabetical order
> This behaviour doesn't need to be matched.
> Your implementation can add subdirectories in any order.
>
> If a file in a different directory is added to a *blob*, then the directories in the path need to be added to the *blob*.
>
> When extracting a *blob* with directories, the directory needs to be created if it does not already exist and the directories permissions need to be set to those specified in *blob*

Subset 4

Given the `-c` and the `-z` command line argument `blobby.c` should:

Compress all bytes of the *blob* using the external program *xz*.

The function *posix_spawn* or *posix_spawnp* must be used to run *xz*.

```
# Create a blob called h.blob with the file selamat.txt.
# Using the "xz" program for compression.
$ ./blobby -z -c h.blob selamat.txt
Adding: selamat.txt
# Show the file type of h.blob to confirm it was compressed using "xz".
$ file h.blob
h.blob: XZ compressed data
```

The `-z` option does not need to be specified when extracting (`-x`) or listing (`-l`) compressed *blob*s and is ignored if it is specified.

`blobby.c` should automatically recognize *blob*s created with `-z` when extracting (`-x`) or listing (`-l`)

```
$ ./blobby -l examples/text_file.compressed.blob
100644  56  hello.txt
$ ./blobby -l examples/4_files.compressed.blob
100644  256 256.bin
100644  56  hello.txt
100444  166 last_goodbye.txt
100464  148 these_days.txt
$ ./blobby -l examples/hello_world.compressed.blob
100644  87  hello.c
100644  82  hello.cpp
100644  65  hello.d
100644  77  hello.go
100644  32  hello.hs
100644  117 hello.java
100644  30  hello.js
100754  47  hello.pl
100754  97  hello.py
100644  45  hello.rs
100644  107 hello.s
100754  41  hello.sh
100644  24  hello.sql
```

HINT:

Subset 4 is highly challenging. The spawn_read_pipe example and the spawn_write_pipe.c example from our lectures on processes show most of the necessary operations but implementing this subset will require research and careful experimentation.

NOTE:

You are not permitted to use: *system*, *popen*, *fork*/*execve* or any other C functions to run *xz*.

You must run *xz* as an external program via *posix_spawn* or *posix_spawnp*.

Temporary files are not permitted.

Compressed *blob* files do not necessarily end with `.compressed.blob`.
This has been done with the provided example files purely as a convenience.

Reference implementation

A reference implementation is available as `1521 blobby` which can use to find the correct output for any input, like this:

```
$ 1521 blobby -l <blob file to list>
# [] means "optionally", ... means "repeated"
$ 1521 blobby [-z] -c <blob file to create> <file to add to the blob> [...]
$ 1521 blobby -x <blob file to extract>
```

HINT:

All the examples in the in the **Subsets** above are runnable using the reference implementation.

Simply replace `./blobby` with `1521 blobby`.

Provision of a reference implementation is a common, efficient and effective method to provide or define an operational specification, and it's something you will likely need to work with after you leave UNSW.

Where any aspect of this assignment is undefined in this specification you should match the reference implementation's behaviour.

Discovering and matching the reference implementation's behaviour is deliberately part of the assignment.

If you discover what you believe to be a bug in the reference implementation, report it in the class forum. If it is a bug, we may fix the bug, or indicate that you do not need to match the reference implementation's behaviour in this case.

Blob Format

*blob*s must follow exactly the format produced by the reference implementation.

A *blob* consists of 1 or more *blobettes*. Each *blobette* contains the information about one file or directory.

Field Length (bytes)	Field Name	Possible Values	Description

Field Length (bytes)	Field Name	Possible Values	Description
1	magic_number	0x42	byte 0 in every blobette must be 0x42 (ASCII 'B')
3	mode		file type and permissions as returned in `st_mode` field from *lstat*
2	pathname_length	1..65535	number of bytes in file/directory pathname
6	content_length	0..281474976710655	number of bytes in file 0 for directories
pathname_length	pathname		file/directory pathname
content_length	contents		bytes (contents) of file empty for directories
1	hash	0..255	`blobby_hash()` of all bytes of this blobette except this byte

blobette format

mode, *pathname_length* and *content_length* are always stored in big-endian format. It is recommended you construct their values from their individual bytes using bit-operations.

Side-note: an interesting property of the *blob* format is the concatenation of 2 or more *blobs* is a valid *blob*.

We can use the reference implementation to create simple *blobs* and inspect their contents.
For example, using the `print_bytes.c` program written as a lab exercise:

```
$ echo hola > hi.txt
$ chmod 640 hi.txt
$ 1521 blobby -c h.blob hi.txt
Adding: hi.txt
$ ./print_bytes h.blob
byte    0:  66 0x42 'B'
byte    1:   0 0x00
byte    2: 129 0x81
byte    3: 160 0xa0
byte    4:   0 0x00
byte    5:   6 0x06
byte    6:   0 0x00
byte    7:   0 0x00
byte    8:   0 0x00
byte    9:   0 0x00
byte   10:   0 0x00
byte   11:   5 0x05
byte   12: 104 0x68 'h'
byte   13: 105 0x69 'i'
byte   14:  46 0x2e '.'
byte   15: 116 0x74 't'
byte   16: 120 0x78 'x'
byte   17: 116 0x74 't'
byte   18: 104 0x68 'h'
byte   19: 111 0x6f 'o'
byte   20: 108 0x6c 'l'
byte   21:  97 0x61 'a'
byte   22:  10 0x0a
byte   23:  95 0x5f '_'
```

Note in the above example *pathname_length* is **6** *content_length* is **5** and the blobby hash is **0x5f**.

The Linux utility *xxd* is a good way to inspect *blobs*, for example, here is a *blob* containing 2 files (2 *blobettes*).

```
$ xxd examples/2_files.blob
00000000: 4200 81a4 0009 0000 0000 0038 6865 6c6c  B.........8hell
00000010: 6f2e 7478 7448 656c 6c6f 2043 4f4d 5031  o.txtHello COMP1
00000020: 3532 310a 4920 686f 7065 2079 6f75 2061  521.I hope you a
00000030: 7265 2065 6e6a 6f79 696e 6720 7468 6973  re enjoying this
00000040: 2061 7373 6967 6e6d 656e 742e 0a66 4200   assignment..fB.
00000050: 81a4 0010 0000 0000 00a6 6c61 7374 5f67  ..........last_g
00000060: 6f6f 6462 7965 2e74 7874 5468 6973 2069  oodbye.txtThis i
00000070: 7320 6f75 7220 6c61 7374 2067 6f6f 6462  s our last goodb
00000080: 7965 0a49 2068 6174 6520 746f 2066 6565  ye.I hate to fee
00000090: 6c20 7468 6520 6c6f 7665 2062 6574 7765  l the love betwe
000000a0: 656e 2075 7320 6469 650a 4275 7420 6974  en us die.But it
000000b0: 2773 206f 7665 720a 4a75 7374 2068 6561  's over.Just hea
000000c0: 7220 7468 6973 2061 6e64 2074 6865 6e20  r this and then
000000d0: 4927 6c6c 2067 6f0a 596f 7520 6761 7665  I'll go.You gave
000000e0: 206d 6520 6d6f 7265 2074 6f20 6c69 7665   me more to live
000000f0: 2066 6f72 0a4d 6f72 6520 7468 616e 2079   for.More than y
00000100: 6f75 276c 6c20 6576 6572 206b 6e6f 770a  ou'll ever know.
00000110: 64                                       d
```

In this example the first *blobette* starts at byte `0x00000000` and has the hash value `0x66 (ASCII 'f')`.
While the second *blobette* starts at byte `0x0000004E` and has the hash value `0x64 (ASCII 'd')`.

Also usable are the *od* and *hexdump* utilities.

Blobby Hash

Each *blobette* contains a hash value calculated from the other values of the *blobette*.

This allows us to detect if any bytes of the *blob* have been changed, for example by disk or network errors.

The hash value is calculated using the `blobby_hash()` function.
You are given code which already implements `blobby_hash()`.

If we create a tiny *blob* containing a single 1 byte file like this:

```
# write a single newline (ASCII '\n') to the file "a"
$ echo > a
$ 1521 blobby -c a.blob a
```

We can inspect the *blob* with *xxd* and see its hash is `0x6d (ASCII 'm')`.

```
$ xxd a.blob
00000000: 4200 81a4 0001 0000 0000 0001 610a 6d    B...........a.m
```

These are the calls to `blobby_hash()` which calculate this value:

```
blobby_hash(00, 42) = de
blobby_hash(de, 00) = 3f
blobby_hash(3f, 81) = 81
blobby_hash(81, a4) = fc
blobby_hash(fc, 00) = c8
blobby_hash(c8, 01) = 57
blobby_hash(57, 00) = a3
blobby_hash(a3, 00) = ab
blobby_hash(ab, 00) = 69
blobby_hash(69, 00) = 7f
blobby_hash(7f, 00) = 11
blobby_hash(11, 01) = fb
blobby_hash(fb, 61) = 3b
blobby_hash(3b, 0a) = 6d
```

The inital hash value starts as 0,
But then becomes the resulting hash value of the previos call to `blobby_hash()`

```
blobby_hash(<inital hash value>, <byte value>) = <resulting hash value>
```

Getting Started

Create a new directory for this assignment called `blobby`,
change to this directory, and fetch the provided examples:
by running these commands:

```
$ mkdir -m 700 blobby
$ cd blobby
$ 1521 fetch blobby
$ unzip examples.zip
```

Or, if you're not working on CSE, you can download the `examples.zip` and starting code

You have been given starting code for this assignment in `blobby.c` which already implements handling command line arguments.

```
$ dcc blobby.c -o blobby
$ ./blobby
Usage:
    ./blobby -l
    ./blobby -x
    ./blobby [-z] -c  pathnames [...]
```

The supplied code calls:

- the function `list_blob()` to list the contents of a *blob* (subset 0)
- the function `extract_blob()` to extract files from a *blob* (subset 1)
- the function `create_blob()` to create a *blob* (subset 2)

You need to add code to these functions.

You will need to add extra functions and `#defines`.

You may optionally create extra `.c` or `.h` files if you choose.

Error Handling

Error checking is an important part of this assignment and automarking will test error handling.

Error messages should be one line (only) and be written to `stderr` (not `stdout`).

`blobby.c` should `exit` with status 1 after an error.

`blobby.c` should check all file operations for errors.

As much as possible match the reference implementation's error messages exactly.

The reference implementation uses *perror* to report errors from file operations and other system calls.

It is not necessary to remove files and directories already created or partially created when an error occurs.

You may extract a file or directory from a *blobette* before determining if the *blobette* hash is correct.

You may *not* extract the file or directory from a *blobette* before determining if the *blobette* magic number is correct.
You can extract previos file or directory from a *blobette*.

Where multiple errors messages could be produced, for example, if two non-existent files are specified to be added to a *blob*, `blobby.c` may produce any one of the error messages.

Assumptions and Clarifications

Like all good programmers, you should make as few assumptions as possible.

If in doubt, match the output of the reference implementation.

`blobby.c` only has to handle ordinary files and directories.
`blobby.c` does not have to handle symbolic links, devices or other special files.
`blobby.c` will not be given directories containing symbolic links, devices or other special files.
`blobby.c` does not have to handle hard links.

If completing a `blobby` command would produce multiple errors, you may produce any of the errors and stop.
You do not have to prooduce the particular error that the reference implementation does.

If a blobette pathname contains a directory then a blobette for the directory will appear in the blob beforehand.
For example, if there is a blobette for the pathname **a/b/file.txt** then there will be preceding blobettes for the directories **a** and **a/b**,

You may also assume the blobbete for the directory, specifies the directory is writeable.

When adding an entire directory (subset 3) to a *blob* you may add the directory contents in any order to the *blob*, after the directory *blobette*.
Your do not have to match the order the reference implementation uses.

When a `blobby` command specifies adding files with common sub-directory.
You may add a *blobette* for the sub-directory multiple times. For example give this command:

```
$ ./blobby -c a.blob b/file1 b/file2
```

You may add two (duplicate) blobettes for `b`.

You can assume the pathname of a blob being created with **`-c`**, will not also be added to the blob, and will not be in a directory being added to the blob

You may call functions from any C library available by default on CSE Linux systems:
including e.g. `stdio.h`, `stdlib.h`, `string.h`, `math.h`, `assert.h`).

You may not use functions from other libraries; in other words, you cannot use dcc's `-l` flag.

You must submit C code only. You may not submit code in any other language.

You may not use *system*, *popen*, *fork*, *execve*, *posix_spawn*, or any other C functions to run external programs.

The only exception is that you are permitted to use *posix_spawn* or *posix_spawnp* to run *xz* (subset 4).

If you need clarification on what you can and cannot use or do for this assignment:
ask in the class forum.

You are not permitted to create or use temporary files.

You are required to submit intermediate versions of your assignment:
See below for details.

Your program must not require extra compile options.
It must compile with:

```
$ dcc *.c -o blobby
```

It will be run with `dcc` when marking.
Run-time errors from illegal C will cause your code to fail automarking.

If your program writes out debugging output, it will fail automarking tests:
make sure you disable debugging output before submission.

Assessment

Testing

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest blobby blobby.c [any other .c or .h files]
```

If you only want to run a subset of the tests avaible with autotest.
You can specify a filter:

```
$ 1521 autotest blobby subset? blobby.c [any other .c or .h files]
```

`1521 autotest` will not test everything.
Always do your own testing.

Automarking will be run by the lecturer after the submission deadline, using a superset of tests to those `autotest` runs for you.

> WARNING:
>
> The errors from `1521 autotest` will become less and less useful as you continue into later subsets.
>
> Our ability to show what error has occured becomes substantually harder than our ability to detect errors in subsets 3 and 4.
>
> Thus, `1521 autotest` may show a failed test and not give the best explanation as to why the test has failed in later subsets.

Submission

When you are finished working on the assignment, you must submit your work by running `give`:

```
$ give cs1521 ass2_blobby blobby.c [other .c or .h files]
```

You must run `give` before **Sunday November 22 21:00 2020** to obtain the marks for this assignment. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

You can run `give` multiple times.
Only your last submission will be marked.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

You *cannot* obtain marks by e-mailing your code to tutors or lecturers.

You can check your latest submission on CSE servers with:

```
$ COMP1521 classrun -check ass2_blobby
```

You can check the files you have submitted here.

Manual marking will be done by your tutor, who will mark for style and readability, as described in the **Assessment** section below. After your tutor has assessed your work, you can view your results here; The resulting mark will also be available via give's web interface.

Due Date

This assignment is due **Sunday November 22 21:00 2020**.

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 2%. For example, if an assignment worth 74% was submitted 10 hours late, the late submission would have no effect. If the same assignment was submitted 15 hours late, it would be awarded 70%, the maximum mark it can achieve at that time.

Assessment Scheme

This assignment will contribute ***15 marks*** to your final COMP1521 mark.

80% of the marks for assignment 2 will come from the performance of your code on a large series of tests.

20% of the marks for assignment 2 will come from hand marking.
These marks will be awarded on the basis of clarity, commenting, elegance and style.
In other words, you will be assessed on how easy it is for a human to read and understand your program.

An indicative assessment scheme follows.
The lecturer may vary the assessment scheme after inspecting the assignment submissions,
but it is likely to be broadly similar to the following:

HD (95+%)	beautiful documented code, perfectly readable code, all subsets (0-4) working for all blobs.
HD (85+%)	well documented code, very readable code, subsets 0-3 working for all blobs.
DN (75+%)	some documentation in code, readable code, subsets 0-2 working for all blobs.
CR (65%)	some documentation in code, readable code, subset 0-1 working for all blobs.
PS (60%)	subset 0 working for all blobs.
0%	knowingly providing your work to anyone; and it is subsequently submitted (by anyone).
0 FL for COMP1521	submitting any other person's work; this includes joint work.
academic misconduct	submitting another person's work without their consent; paying another person to do work for you.

Intermediate Versions of Work

You are required to submit intermediate versions of your assignment.

Every time you work on the assignment and make some progress you should copy your work to your CSE account and submit it using the `give` command below. It is fine if intermediate versions do not compile or otherwise fail submission tests. Only the final submitted version of your assignment will be marked.

All these intermediate versions of your work will be placed in a Git repository and made available to you via a web interface at `https://gitlab.cse.unsw.edu.au/z5555555/20T3-comp1521-ass2_blobby` (replacing *z5555555* with your own zID). This will allow you to retrieve earlier versions of your code if needed.

Attribution of Work

This is an individual assignment.

The work you submit must be entirely your own work, apart from any exceptions explicitly included in the assignment specification above. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted.

You are only permitted to request help with the assignment in the course forum, help sessions, or from the teaching staff (the lecturer(s) and tutors) of COMP1521.

Do not provide or show your assignment work to any other person (including by posting it on the forum), apart from the teaching staff of COMP1521. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, you may be penalized, even if that work was submitted without your knowledge or consent; this may apply even if your work is submitted by a third party unknown to you. You will not be penalized if your work is taken without your consent or knowledge.

Do not place your assignment work in online repositories such as github or any where else that is publically accessible. You may use a private repository.

Submissions that violate these conditions will be penalised. Penalties may include negative marks, automatic failure of the course, and possibly other academic discipline. We are also required to report acts of plagiarism or other student misconduct: if students involved hold scholarships, this may result in a loss of the scholarship. This may also result in the loss of a student visa.

Assignment submissions will be examined, both automatically and manually, for such submissions.

Change Log

Version 1.0 (2020-11-03 13:00:00)	• Initial release.
Version 1.1 (2020-11-06 11:00:00)	• Update subset 3 autotests to ignore order of files in the blob
Version 1.2 (2020-11-07 07:00:00)	• Clean-up the spec html, no changes to the assignment
Version 1.3 (2020-11-09 13:10:00)	• Hints improved - mainly links to relevant lecture examples
Version 1.4 (2020-11-10 15:30:00)	• Bug in reference implementation handling of blobs with invalid content length fixed
Version 1.4 (2020-11-12 13:30:00)	• Clarification added that blobette for any directory in a pathnames will appear first
Version 1.5 (2020-11-15 15:30:00)	• Clarification added, that a blob will not be added to itself.
Version 1.6 (2020-11-17 11:30:00)	• Clarification added, that directories in blobettes will specify they are writeable, if a file has to be added to the directory.
Version 1.7 (2020-11-21 17:30:00)	• Correctly show relative paths in the descriptions of autotests. The commands used by the autotests remain unchanged. • 1521 blobby will behave better in the (non-assessable) scenario that the destination blob file anme is attempted to be added into the blob.