CSSE2310/7231 — 2.1

SVN,
Ass1, output streams,
`main()` and pointers

SVN

Detour: (very) simple subversion(SVN)

Version control systems:

- Store development history of a project
  - As a series of *commits*/revisions
- Allow retrieval of previously committed states.
- report differences between revisions
- multiple parallel lines of development (branches) / merging
  - beyond the scope of this course

Subversion is one VCS among many, but it is the only suported one in this course.

SVN pieces

- Repository
  - Where history is stored
  - Only manipulate via svn commands
  - Located via a URL
  - In this course you each have a repo at `https://source.eait.uq.edu.au/svn/csse2310-s????`
- Working copy / copies
  - Where you do your editing / compiling / testing
  - Record "good" states back to the repo with `commits`
  - Nothing you modify in your working copy affects the repo unless you commit.
  - You can delete your working copy and it will not affect the repo (will lose any uncommitted changes though)
  - There could be multiple working copies checked out.

SVN operations

- `svn checkout` *`URL working_dir`*
  - Make a working copy of the most recent version
  - eg: `svn checkout https://source.eait.uq.edu.au/svn/csse2310-s??????/\trunk/ass1 ass1`
- `svn add` *`filename`*
  - Tells svn to track changes to this file.
  - Doesn't take affect until you commit
- `svn mv` *`oldname newname`*
  - Rename or move a file
  - Need to commit to make the change in Repo
- `svn rm` *`fname`*
  - Remove file locally and remove it from future repo revisions
  - Can not remove it from past versions

SVN operations

- `svn status`
  - Show which files have pending changes
  - M — modified, A — untracked file to be added, D — file to be removed, ? — Don't know anything about this file
- `svn diff`
  - Show the lines which have changed
- `svn commit`
  - Pending changes in the repo
  - Will ask for a log message (edit and save)
  - or `svn commit -m 'Things I changed'`
  - Can commit only specified files with:
    `svn commit f1 f2 f3`

SVN operations

- `svn revert` *`filename`*
  - Undo any pending changes to *filename*.
  - Can't be reversed.
- `svn update`
  - Bring working copy up to date with the latest version in the repo
  - Generally only needed if you've been using multiple working directories.
  - Can lead to conflicts

Putting it back

To put a file back the way it was a few versions ago:

- `svn update -r14` *`filename`*
- Make a copy of that version of the file
  `cp filename backup`
- `svn update` *`filename`*
- copy the older version over
  `cp backup filename`
- `svn commit filename`

Demo

Assignment 1

Outptut streams

C programs have two (default) streams of output:

- Standard out (`stdout`) where normal output goes
- Standard err (`stderr`) where error messages go

On the command line 2> to separate out error messages.

Parameters of `main()`

Parameters of main

Main takes 2 parameters — together, they describe an array of strings.

- `int argc` : The number of strings in the array
- `char** argv` or `char* argv[]` : The array itself.
- `argv[0]` is the program being run.

See `arg1.c`

- %s — placeholder for a string
- C arrays are not range checked
- Generally you can't reliably ask how big an array is (hence argc).

Paramters of main

See `arg2.c`

Casting doesn't work that way

See `arg3.c`

Parameters of main

```
total += (int)argv[i];
```

Need to use a function to do this.

See `arg4.c`

Note: Does not detect problems with "13spider".
For that we need `strtol()` and *pointers*.

Cue ominous music

Pointers!

- . . . Have a reputation.
- . . . Allow indirection (which is *really* important in programming).
- . . . Are strongly related to what Python or Java call references[1].
- . . . Are "exciting" in C because of the operations C allows on pointers.

[1] *COUGH* `java.lang.NullPointerException` *COUGH*

Pointers

A pointer:

- is a value
- is the address where another value can be found. (eg your address is both a value and a location where you live)
- has a type
  - What sort of thing does the pointer "point to"
  - `int*` and `char*` are different types but both are pointers.

Pointers

`int* var;`
declares `var` to be a variable which stores a "pointer-to-int".

As a *short hand*, people will often refer to variables by the type they store [2].

But understand the difference.

[2] See René Magritte's "La Trahison des images"
(Ceci n'est pas une pipe)

Pointing at ?

See `point1.c`
Would be better to initialise pointers to 0.

Getting valid pointers

See `point2.c`

- `malloc()` — Find me some memory I can use and give me a pointer to it.
- `p` gives the value(address) of the pointer
- `*p` gives the thing pointed to
- `malloc()` doesn't ask what type of value you want to store (just how much space you need).
- Use `sizeof` to find out how much space something takes.
- The return value from malloc should (probably) be cast.
- Memory aquired via `malloc()` is called "Dynamic memory".

Clean up

See `point3.c`

Memory leak.
Malloc'd memory is not cleaned up until

- it is explicitly released using `free()`
- The program ends
- Memory can be free'd in another function, provided the pointer is known.

See `point4.c`

Dangling pointers

See `point5.c`

The following is dangerous:

```
p3 = p2;
free(p2);
// use p3;
```

Once any pointer to a chunk of memory is passed to `free()`, all pointers to that address are invalid. You can't tell this by looking at them!

Dynamic arrays

For a 10 element array of *int*

```
int* arr = (int*)malloc(sizeof(int) * 10);
...
// arr[0]...arr[9] will be valid.
...
free(arr);
```

The name of an array can be treated as a pointer to the first element.

Pointer arithmetic == "excitement"

```
int* arr=(int*)malloc(sizeof(int)*5);
                    // arr points to an array of 5 ints
arr[0] = 5;
                    // [ ] operator works on pointers too!
arr[4] = 7;
                    // "Last" element in the array
arr[400] = 5;
                    // Ummm
```

What about `arr[-1]`?

Negative indexes count from the end right?

Ha, ha, *nope!*

Pointer arithmetic

To find an indexed location in an array, calculate:
&(arr[index]) → start_of_array + index * size_of_element.
When you add an integer to a pointer, C moves forward in multiples of the size of the type pointed at: eg:

```
int* p=1024;
char* c=1024;

c + 1 == 1025    // sizeof(char)==1
p + 1 == 1028    // on 32bit machines
```

Pointers don't check how much memory they point at, so you can construct a pointer to any location in memory based on where it is relative to a known pointer.

Null pointer

There is a special pointer called the null pointer.
It never points to anything valid.
It can be written a in number of ways, but the standard says that **0** where you expect a pointer will be treated as the null pointer.
New pointers should be initialised to null unless you have a proper value for them.

```
int* v = 0;

if (v == 0) ...
```