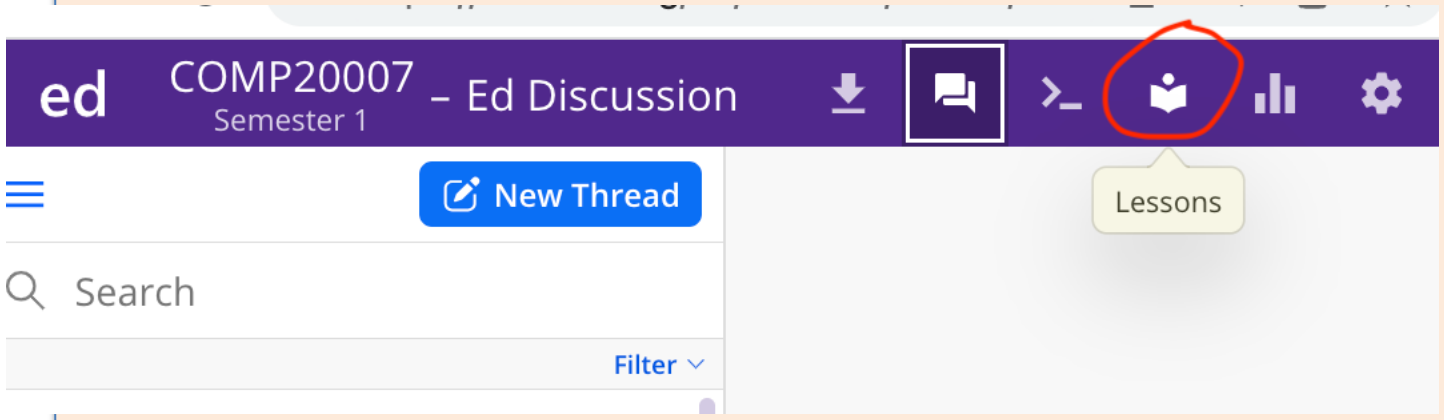
COMP20007 Workshop Week 1

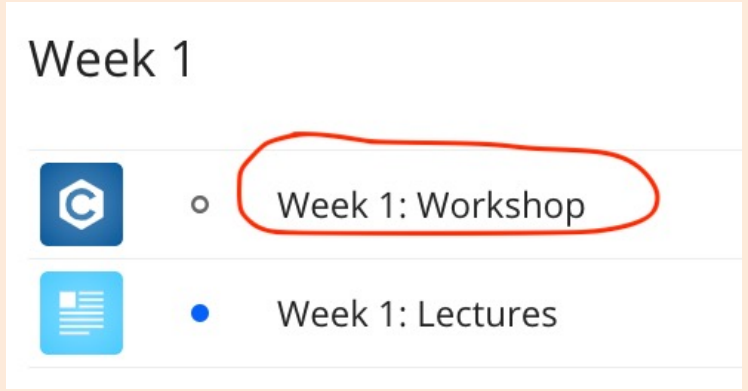
about us and … C

While waiting:

- Talk to classmates, make friends
- Open LMS and click on "Ed Discussion" to open ED, on ED:

  ➢ click on Lessons → Week 1 Workshop





➢ Inspect the first 2 slides: "Tutorial" and "Tutorial: Pre-Workshop"

A Bit on C: Variables, Pointers

Local Variables

For the function `main()`, all its local variables (`a, b` and `sum`):

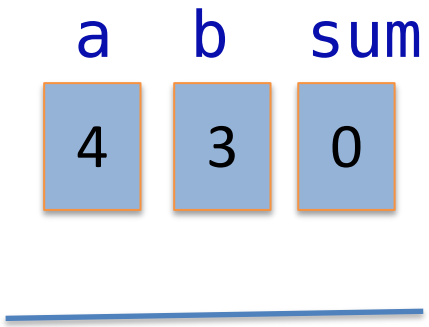
- are automatically allocated memory on the stack frame (SF) when the program starts,
- are automatically deallocated when the function exits.

main.c

```
f1  int foo(int m, int n) {
f2    int tmp;
f3    if (m > n) {
f4      tmp= m;
f5      m= n;
f6      n= tmp;
f7    }
      return m+n;
    }
```

```
m1  int main() {
m2    int a= 4, b= 3, sum=0;
m3    sum= foo(a, b);
m4    printf("%d %d %d\n",
                a, b, sum);
```

SF at line m2



Local Variables

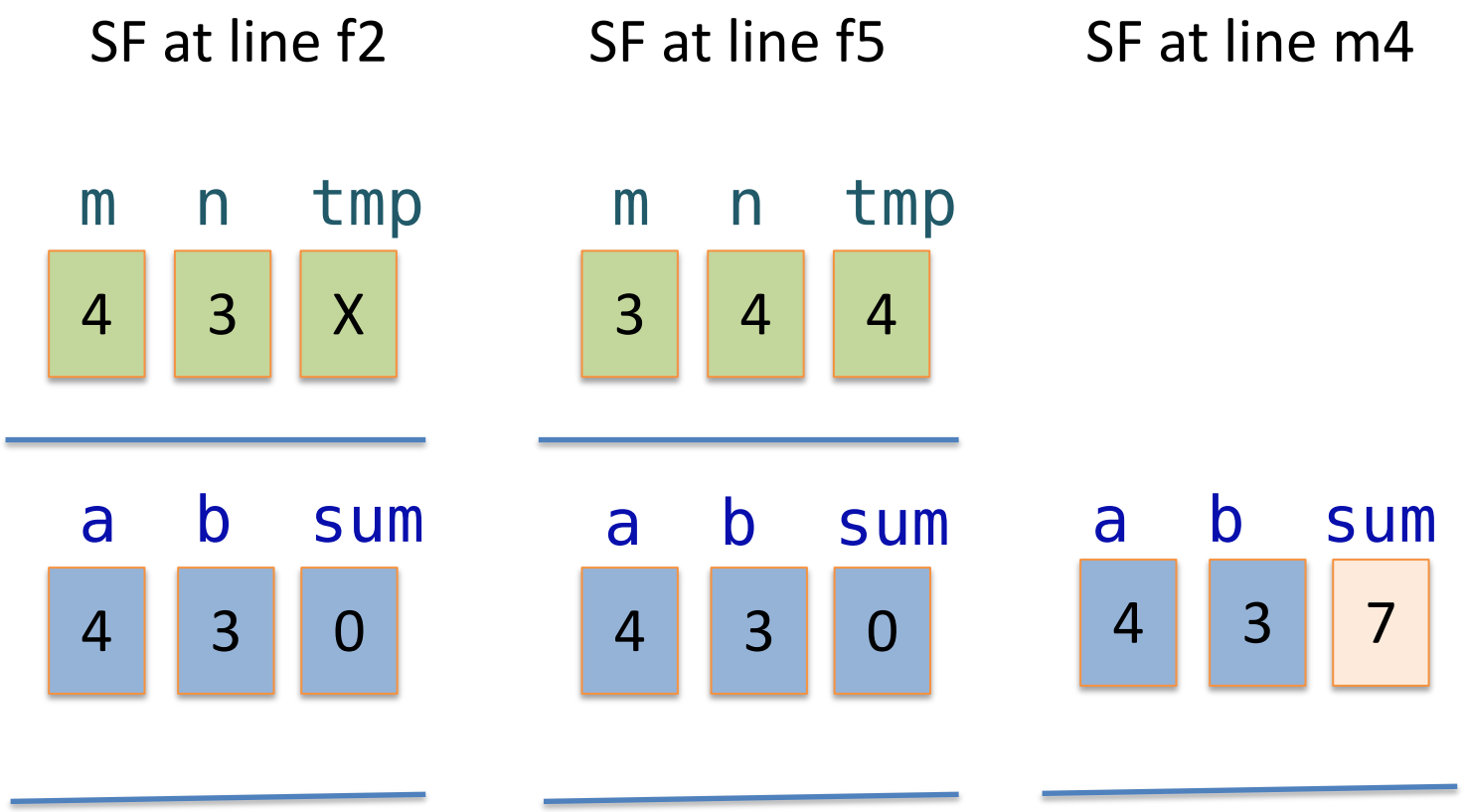
For a function (`foo`), all its local variables (`m` and `n`):

- are automatically allocated memory on the stack frame (SF) when the function is called,
- are automatically deallocated when the function exits.

main.c

```
f1  int foo(int m, int n) {
f2    int tmp;
f3    if (m > n) {
f4      tmp= m;
f5      m= n;
f6      n= tmp;
f7    }
      return m+n;
    }
```

```
m1  int main() {
m2    int a= 4, b= 3, sum=0;
m3    sum= foo(a, b);
m4    printf("%d %d %d\n",
                a, b, sum);
```



Pointers

If function `fool` also wants to change `a` and `b` it needs to receive the *address* of `a` and `b`. The address `&a` and `&b` are called *pointers*. The type of `&a` and `&b` is `int *`

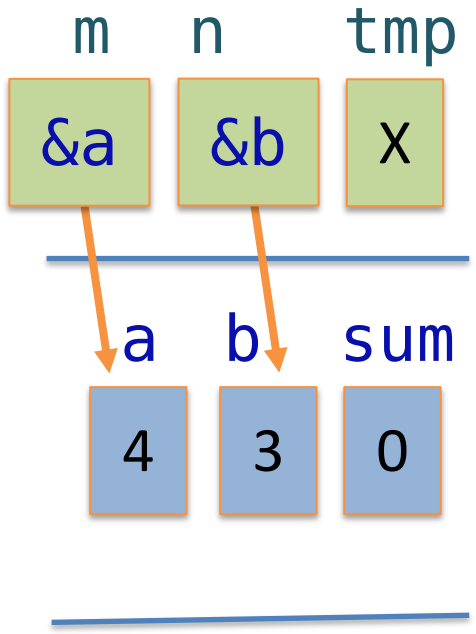
int foo(int m, int n)

```
f1  int fool(int *m, int *n) {
f2    int tmp;
f3    if (*m > *n) {
f4      tmp= *m;
f5      *m= *n;
f6      *n= tmp;
f7    }
      return *m + *n;
    }
```

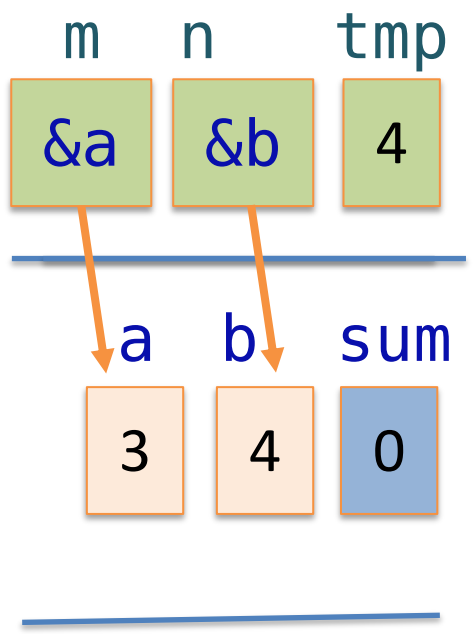
```
m1  int main() {
m2    int a= 4, b= 3, sum=0;
m3    sum= foo(&a, &b);
m4    printf("%d %d %d\n",
                a, b, sum);
```

Here `m` and `n` are *pointer variables*, `&a` and `&b` are *pointer constants*.

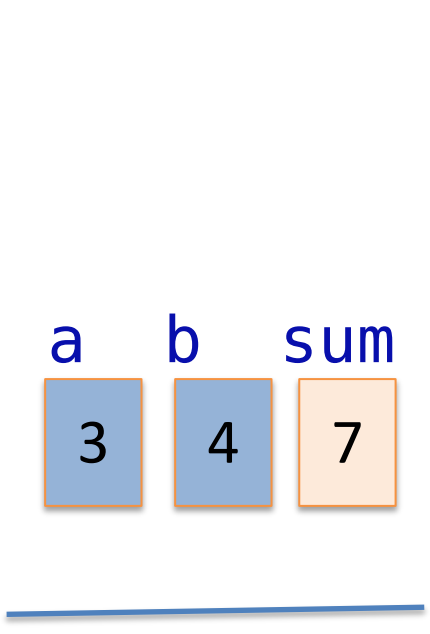
SF at line f2



SF line f5



SF at line m4



Visualisation: Memory Lane 




8 blocks	1 block	8 blocks	4 blocks	...
1 Foo St	13 Foo St	21 Foo St	42 Foo St	...

double	...	char	...	long	...	int	...
8 Bytes		1 Byte		8 Bytes		4 Bytes	...
0x075f5f		0x603e7a		0xbcde77		0xbd370a	...

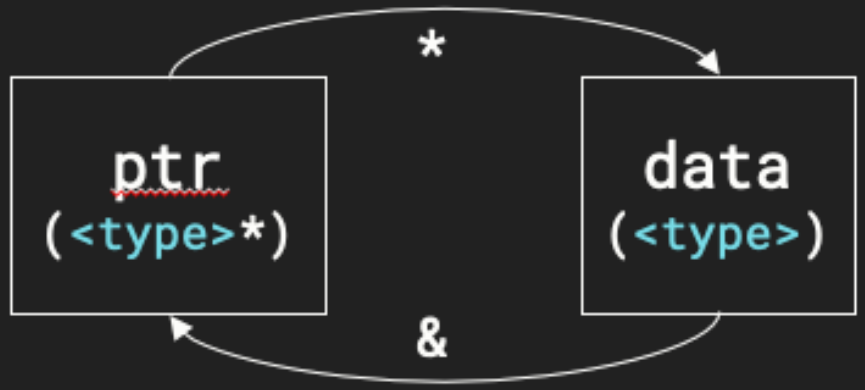
Pointers: Memory Allocation & Deallocation

```c
/* Return a pointer to memory space for an int */
void *ptr = malloc(sizeof(int));
/* Ensure that memory allocation succeeded */
assert(ptr);
/* Frees memory that was allocated with malloc() */
free(ptr);
```

Pointers in a nutshell

Two basic **pointer operators**:

- **dereference operator** (*): "value pointed by"
- **reference operator** (&): "address of"



Operation	How?
Pointer declaration for <type>	<type> *ptr;
Allocation, method 1	ptr= (<type>*) malloc(sizeof(<type>)); assert(ptr);
Allocation, method 2 (using automatic casting)	ptr= malloc(sizeof(*ptr)); assert(ptr);
Allocation, zero-initialised	ptr= calloc(1, sizeof(*ptr)); assert(ptr);
Deallocation	free(ptr);

Pointers: Examples

```
/* pointer variable declarations */
<type> *ptr;  // run-of-the-mill variable
int *example;    // pointer to an int
char *example2;  // pointer to a char
int **example3;  // pointer to a [pointer to an int]

/* Array access via pointers */
int arr[9];
arr[5] == *(arr + 5);  // these are functionally equivalent
```

Pointers: sizes

```
/* Expected values in bytes
 * (within most systems, anyways)
 */
sizeof(char) == 1;
sizeof(float) == 4;
sizeof(int) == 4;
sizeof(double) == 8;
sizeof(long) == 8;
```

```
/* All pointers have the same size
 * since addresses have equal sizes
 */
sizeof(int*) == 8;
sizeof(int**) == 8;
sizeof(char*) == 8;
```

Pointers: Example of Variables in Memory

```
> nl pointer.c
1  int main(void){
2      int *example;
3      int number = 5;
4      int **example2;
5      char *example3;
6      example2 = &example;
7      *example2 = &number;
8      *example = 10;
9      example3 = (char *) example;
10     example3[0] = 'a'; // a is 0x61 or
97
11     example3[1] = 'b'; // b is 0x62 or
98
12
```

Name	Address	Value	Memory (Bytes)							
example	0x7fffffffe0b0		?	?	?	?	?	?	?	?
number	0x7fffffffe0ac	5	5	0	0	0				
example2	0x7fffffffe0b8		?	?	?	?	?	?	?	?
example3	0x7fffffffe0c0		?	?	?	?	?	?	?	?

Discussion: storing a sequence of strings (supposing INITIAL large enough)

Will this code snippet work as intended? Why or why not?

a. Yes, it will.

b. No, it will not.

```
...
1  char buf[MAX_LEN];  // string buffer
2  char **dups =       // array of strings
3      (char**)malloc(INITIAL*sizeof(char*));
4  int num_strings = 0;
5  /* Read strings from stdin */
6  while (fgets(buf, MAX_LEN, stdin) != NULL) {
7      /* NULL-terminate the string */
8      if (buf[strlen(buf) - 1] == '\n')
9          buf[strlen(buf) - 1] = '\0';
10     /* Store it into the array */
11     dups[num_strings] = buf;
12     num_strings++;
13 }
...
```

Program Development on ED

C Program Development on ED

ED supplies

- files systems (aka. *workspaces*),
- a Text Editor for Creating/Editing programs and text files,
- a Shell (aka. *command-line interpreter*), which could be used for
  - compiling:

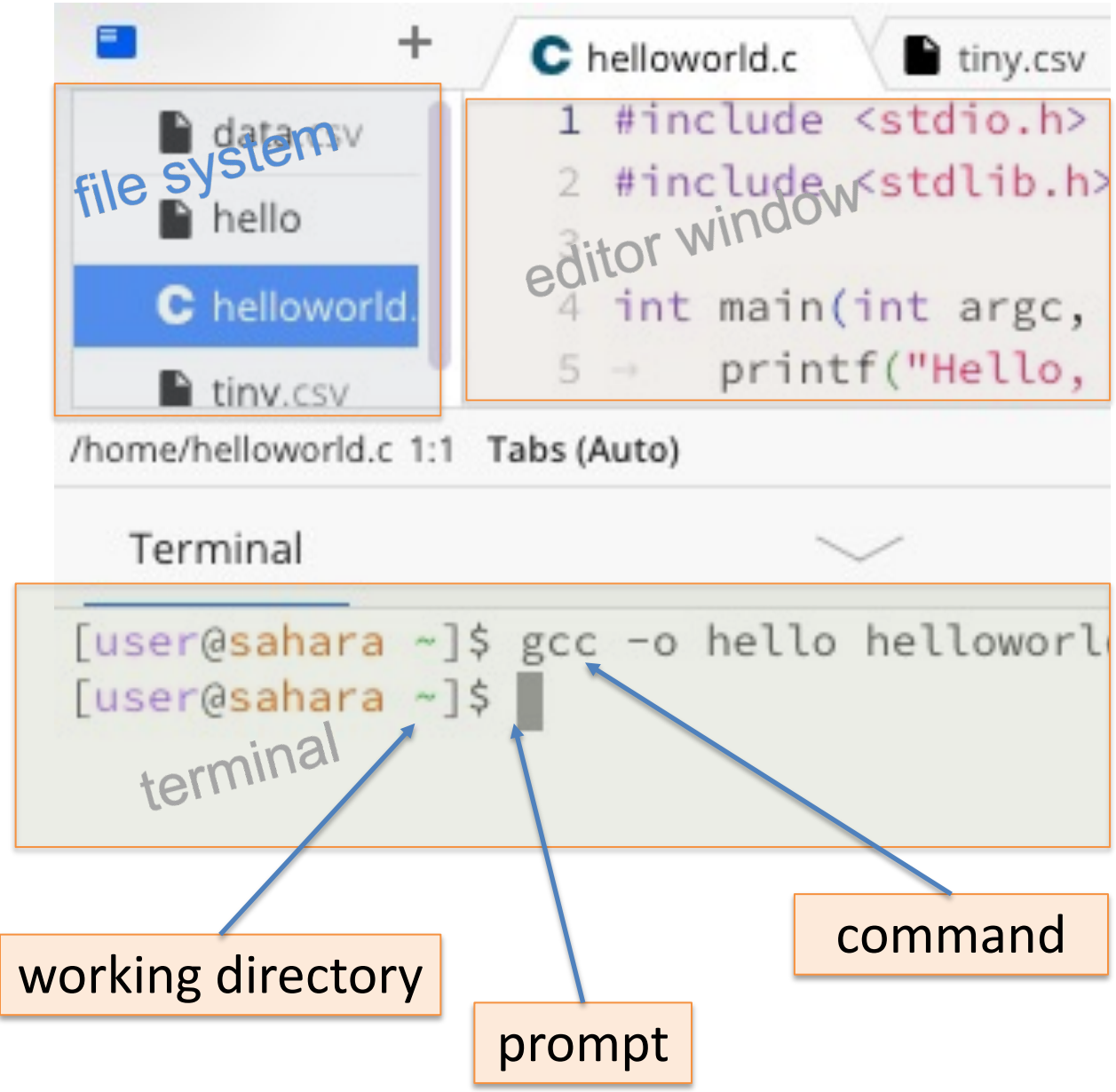
    ```
    gcc –Wall –g –o program program.c
    ```

  - debugging with some tools, for example:

    ```
    valgrind --leak-check=full --track-origins=yes ./program
    ```

  - and many other useful jobs such as seeing manual pages (`man`) of functions/tools, copying or displaying files...

Using ED Workspace for a Program Project



A *workspace* is used for a programming project:

- It has a *file system*, starting with a *home directory* (~). Recursively, a directory can host files and its own sub-directories.
- The terminal allows us to interact with a Unix-style shell (namely `bash`).

Together with Tutor, do:

- Problem 1: essential shell commands
- Problem 6
- Problem 2: using `valgrind` for debugging

Lab

Appendices

Testing: Simplest Failing Test Case

Test code with the **simplest test case** that:

- you know should work, but
- does not work :')

For example,

Debugging principles

- get no warnings/errors from `gcc -Wall`
- get a clean `valgrind` report
- when having problem, first debug with small and simple input
- having problems with larger inputs: focus on the first troublesome input part
- remember: all debug tools have limitations...

Memory Profiling: Valgrind

Valgrind is a suite of dynamic analysis tools.

Memcheck is the default, memory-analysing tool and allows us to:

- see a program's cumulative memory utilisation
- decode arcane segmentation faults
- find **memory leaks:** manually-allocated memory that was never freed

Memory Profiling: Valgrind: Cheat Sheet

Synopsis:

```
valgrind [valgrind-options] your-program [your-program-options]
```

Essential options:

```
--leak-check=full     # shows details of each individual memory leaks
--track-origins=yes   # tracks the origins of uninitialised values
```

Note: This is an excerpt of Valgrind's man page.

Memory Profiling: Valgrind: Common Errors

Error message	Possible cause(s)
`Conditional jump or move depends on uninitialised value(s)`	Uninitialised values were used in the guard of `for`/`if`/`switch`/`while` statements
`Invalid free() / delete / delete[]`	Non-`*alloc()`'d memory was `free()`'d Some `*alloc()`'d memory was `free()`'d multiple times
`Invalid read/write of size X`	Accessing/modifying restricted memory
`Use of uninitialised value of size X`	Accessing/modifying uninitialised values

Common sizes: 1 (`char`), 2 (`short`), 4 (`float`, `int`), 8 (`double`, `long`, `<type>*`)

Daily Tool: Command-Line Interface

```
cat file_name    # prints file_name's contents to stdout
cd dir_path      # changes the current directory to dir_path
clear            # wipes the terminal clean
ls dir_path      # lists the contents of dir_path
mkdir dir_name   # creates a new directory dir_name
rm file_name     # removes file_name
rm -r dir_name   # removes dir_name and everything inside it
```

Daily Tool: command `man` for manual pages

Man(ual) pages are a form of software documentation.

They usually **document**:

- formal standards and conventions (e.g. TCP/IP)
- libraries (e.g. `stdio.h`)
- system calls (e.g. `fork()`)

They can be **accessed** via the `man` command.

Examples:

- How to use `malloc`?
- How to use `fgets` or `getline`?