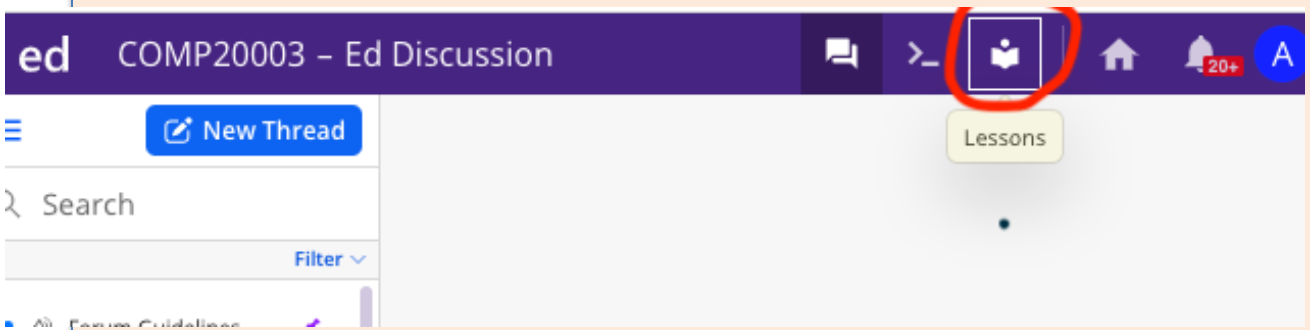
COMP20003 Workshop Week 1

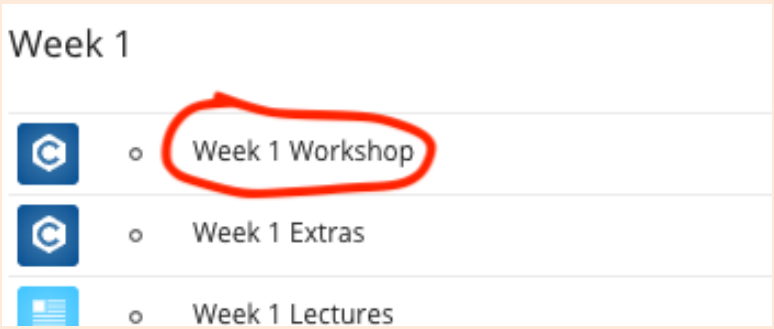
about Us and ... C

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





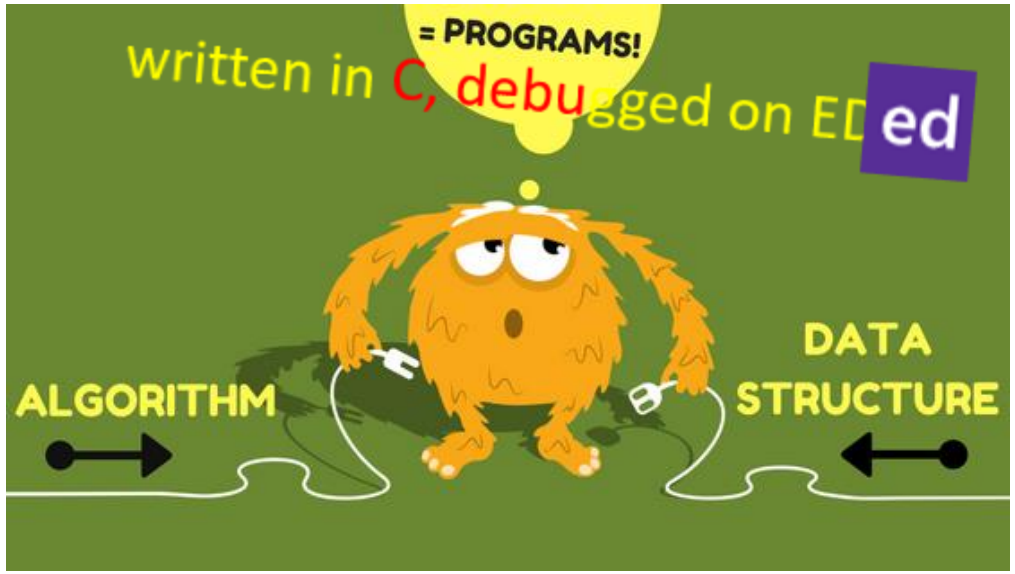
  ➢ click on W1.2 and fill in the survey

Plan

- Admin
- C review:
  - Memory Management
  - Automatic & Dynamic memory
  - Pointers, Strings, Arrays
- Survival Tools:
  - Command Line Interface: Essential Commands
  - Compiling, Executing, Debugging
  - Debugging with valgrind
- Having fun with Peer Programming

Our Workshops

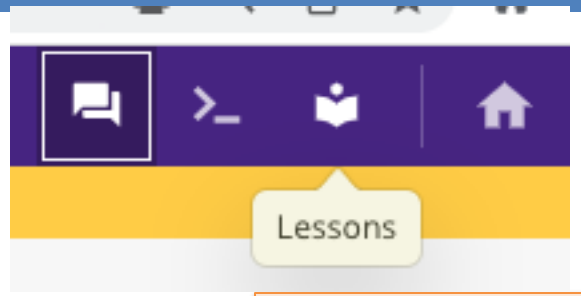
backup: the teaching team, ED, Internet, Google, ChatGPT, ...

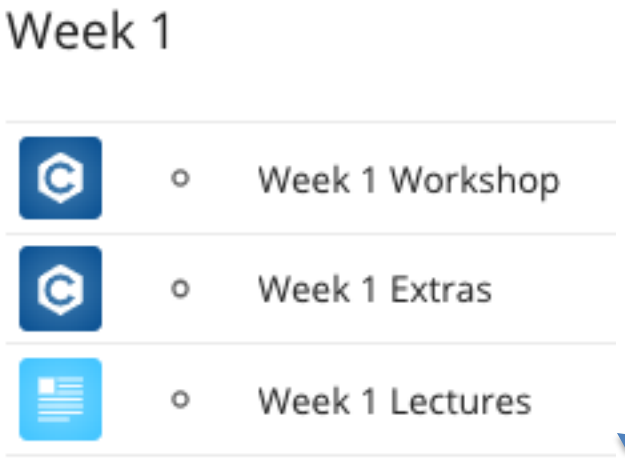


Tutor: a handy friend

= avo@unimelb.edu.au

ED forum and lessons for week x (here, x == 1)





Materials for the Workshop of week x.

Additional Works (not mandatory) for week x, including:

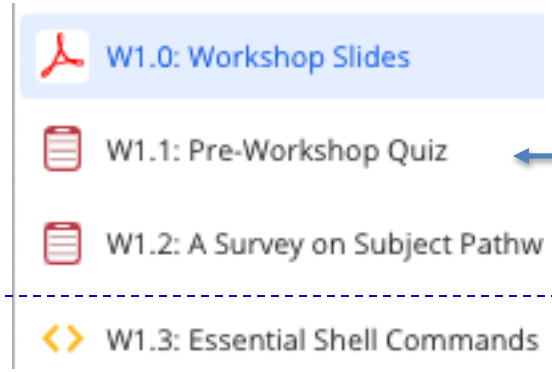
- Ex.1-Ex.9: exercises for further understanding
- Ex.10 onward: job-interviewed or challenging questions .

Lecture slides and videos.

Main Workshop Material: lesson Week x Workshop (x == 1 here)

Before the workshop:

- Review lectures
- Do Wx.0-Wx.2

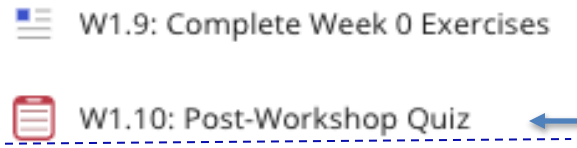


Wx.1: Peer Activity, for discussion

During the workshop:

- Discussions, be active
- Do Wx.3-Wx.9
- Peer Activity Wx.10

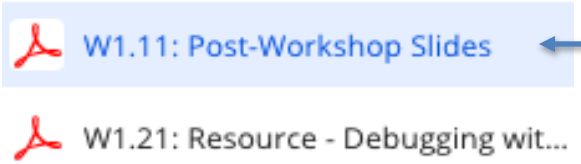
have fun during class



Same as Wx.1: need to fill in after the workshop.

After the workshop:

- Finish all items Wx.*
- Try "Week x Extras"
- Check with solution after 9AM Sunday



More detailed slides, for reviewing.

Bonus for weeks 1 to 12	Special for Week 1
Get the green tick for all items in "Workshop week x" by 9AM Sunday to be awarded 0.33 bonus mark.	• *Do Week 0 exercises to regain your C skills* • *Do "Week 0: Short Revival" to get 1 bonus mark*

Now: time for (social) networking

- Get to know your classmates
- Have a chat
- Also ensure that W**1**.1 and W**1**.2 completed (remember: W**x**.0 – W**x**.2 should be done prior to the workshop of week **x**)

Memory Management in C

Visualisation: Computers' Memory Lane

Two main categories of memory management in C

Automatic Memory:

- Used for local variables, including function parameters.
- Memory is allocated and deallocated automatically.
- Accessed directly by their variable names.

Dynamic Memory:

- Accessed via pointers to memory chunks.
- Allocated and deallocated explicitly by the programmer.

Automatic Memory: for All Local Variables

For a function (foo), all its declared variables (m, n and tmp) are *local*. Their memory:

- is automatically allocated when the function is called,
- is 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);
```

Q: Why a and b remain unchanged at line m4?

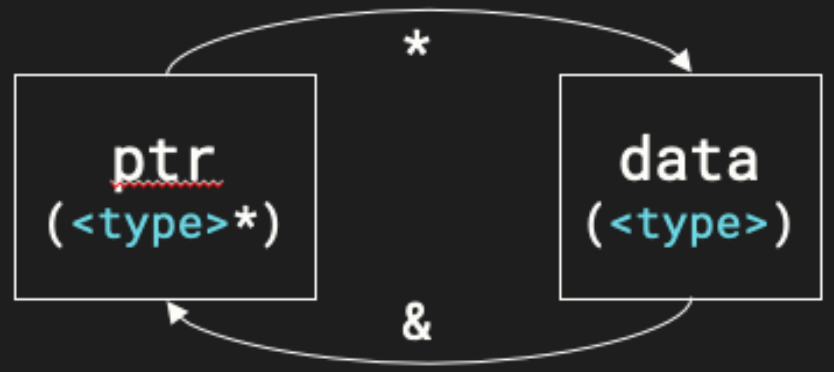
Stack at line f2	Stack at line f5	Stack at line m4
m n tmp 4 3 X a b sum 4 3 0	m n tmp 3 4 4 a b sum 4 3 0	a b sum 4 3 7

Pointers

A **pointer** is a variable that stores the **memory address** of another variable or a memory location.

Two basic **pointer operators**:

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

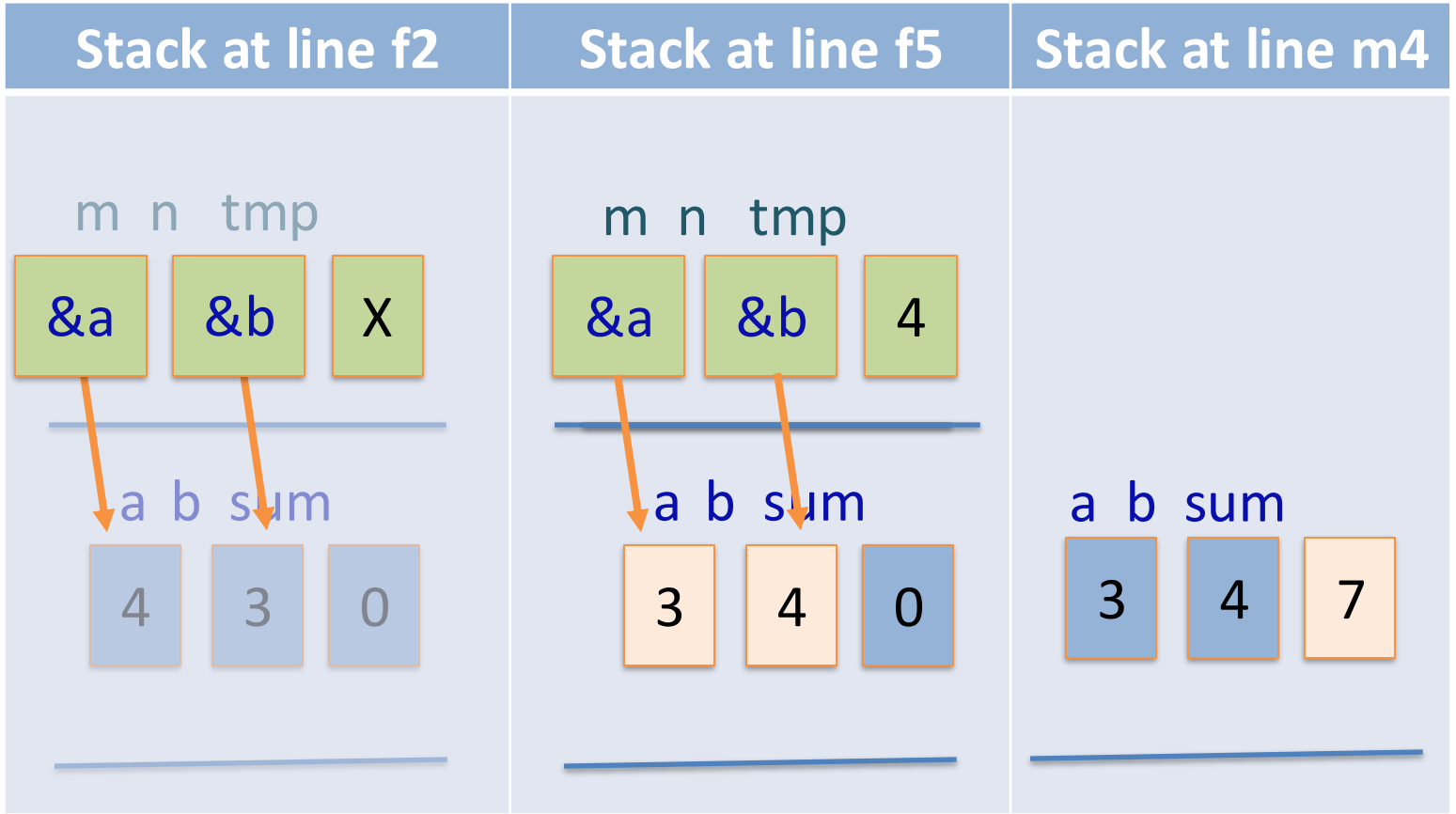


Pointers: examples of using as function parameters

int foo(int m, int n)

```
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);
```



Remember?

A function cannot directly access variables of other functions.

However, it can do so using pointers.

Pointers: Examples of Declaration

```
/* 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
```

Remember? An array name is a pointer, specifying the address of the array's first element

Size of basic data types, including pointers

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

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

Pointers: Examples with Memory Images

```
1  int main(void){
2     int *intPtr;
3     int number = 5;
4     int **intPtrPtr;
5     char *charPtr;
6     intPtrPtr = &intPtr;
7     *intPtrPtr = &number;
8     *intPtr = 10;
9     charPtr = (char *) intPtr;
10    charPtr[0] = 'a'; // a is 0x61 or 97
11    charPtr[1] = 'b'; // b is 0x62 or 98
12 }
```

Name	Address	Value	Memory (Bytes)							
intPtr	0x7fffffffffffe0b0									
number	0x7fffffffffffe0ac	5	05	00	00	00				
intPtrPtr	0x7fffffffffffe0b8									
charPtr	0x7fffffffffffe0c0									

Pointers: Examples with Memory Images

```
1  int main(void){
2    int *intPtr;
3    int number = 5;
4    int **intPtrPtr;
5    char *charPtr;
6    intPtrPtr = &intPtr;
7    *intPtrPtr = &number;
8    *intPtr = 10;
9    charPtr = (char *) intPtr;
10   charPtr[0] = 'a'; // a is 0x61 or 97
11   charPtr[1] = 'b'; // b is 0x62 or 98
12 }
```

Name	Address	Value	Memory (Bytes)							
intPtr	0x7fffffffffffe0b0									
number	0x7fffffffffffe0ac	5	05	00	00	00				
intPtrPtr	0x7fffffffffffe0b8	0x7fffffffffffe0b0	**b0**	**e0**	**ff**	**ff**	**ff**	**ff**	**ff**	**7f**
charPtr	0x7fffffffffffe0c0									

Pointers: Examples with Memory Images

```
1   int main(void){
2     int *intPtr;
3     int number = 5;
4     int **intPtrPtr;
5     char *charPtr;
6     intPtrPtr = &intPtr;
7     *intPtrPtr = &number;
8     *intPtr = 10;
9     charPtr = (char *) intPtr;
10    charPtr[0] = 'a'; // a is 0x61 or 97
11    charPtr[1] = 'b'; // b is 0x62 or 98
12  }
```

Name	Address	Value	Memory (Bytes)							
intPtr	0x7fffffffffffe0b0	0x7fffffffffffe0ac	**ac**	**e0**	**ff**	**ff**	**ff**	**ff**	**ff**	**7f**
number	0x7fffffffffffe0ac	5	05	00	00	00				
intPtrPtr	0x7fffffffffffe0b8	0x7fffffffffffe0b0	b0	e0	ff	ff	ff	ff	ff	7f
charPtr	0x7fffffffffffe0c0									

Pointers: Examples with Memory Images

```
1   int main(void){
2     int *intPtr;
3     int number = 5;
4     int **intPtrPtr;
5     char *charPtr;
6     intPtrPtr = &intPtr;
7     *intPtrPtr = &number;
8     *intPtr = 10;
9     charPtr = (char *) intPtr;
10    charPtr[0] = 'a'; // a is 0x61 or 97
11    charPtr[1] = 'b'; // b is 0x62 or 98
12  }
```

Name	Address	Value	Memory (Bytes)							
intPtr	0x7fffffffffffe0b0	0x7fffffffffffe0ac	ac	e0	ff	ff	ff	ff	ff	7f
number	0x7fffffffffffe0ac	10	**0a**	**00**	**00**	**00**				
intPtrPtr	0x7fffffffffffe0b8	0x7fffffffffffe0b0	b0	e0	ff	ff	ff	ff	ff	7f
charPtr	0x7fffffffffffe0c0									

Pointers: Examples with Memory Images

```
1  int main(void){
2    int *intPtr;
3    int number = 5;
4    int **intPtrPtr;
5    char *charPtr;
6    intPtrPtr = &intPtr;
7    *intPtrPtr = &number;
8    *intPtr = 10;
9    charPtr = (char *) intPtr;
10   charPtr[0] = 'a'; // a is 0x61 or 97
11   charPtr[1] = 'b'; // b is 0x62 or 98
12 }
```

Name	Address	Value	Memory (Bytes)							
intPtr	0x7fffffffffffe0b0	0x7fffffffffffe0ac	ac	e0	ff	ff	ff	ff	ff	7f
number	0x7fffffffffffe0ac	10	0a	00	00	00				
intPtrPtr	0x7fffffffffffe0b8	0x7fffffffffffe0b0	b0	e0	ff	ff	ff	ff	ff	7f
charPtr	0x7fffffffffffe0c0	0x7fffffffffffe0ac	**ac**	**e0**	**ff**	**ff**	**ff**	**ff**	**ff**	**7f**

Pointers: Examples with Memory Images

```
1   int main(void){
2     int *intPtr;
3     int number = 5;
4     int **intPtrPtr;
5     char *charPtr;
6     intPtrPtr = &intPtr;
7     *intPtrPtr = &number;
8     *intPtr = 10;
9     charPtr = (char *) intPtr;
10    charPtr[0] = 'a'; // a is 0x61 or 97
11    charPtr[1] = 'b'; // b is 0x62 or 98
12  }
```

Name	Address	Value	Memory (Bytes)							
intPtr	0x7fffffffffffe0b0	0x7fffffffffffe0ac	ac	e0	ff	ff	ff	ff	ff	7f
number	0x7fffffffffffe0ac	97	**61**	00	00	00				
intPtrPtr	0x7fffffffffffe0b8	0x7fffffffffffe0b0	b0	e0	ff	ff	ff	ff	ff	7f
charPtr	0x7fffffffffffe0c0	0x7fffffffffffe0ac	ac	e0	ff	ff	ff	ff	ff	7f

Pointers: Examples with Memory Images

```
1  int main(void){
2    int *intPtr;
3    int number = 5;
4    int **intPtrPtr;
5    char *charPtr;
6    intPtrPtr = &intPtr;
7    *intPtrPtr = &number;
8    *intPtr = 10;
9    charPtr = (char *) intPtr;
10   charPtr[0] = 'a'; // a is 0x61 or 97
11   charPtr[1] = 'b'; // b is 0x62 or 98
12 }
```

Name	Address	Value	Memory (Bytes)							
intPtr	0x7fffffffffffe0b0	0x7fffffffffffe0ac	ac	e0	ff	ff	ff	ff	ff	7f
number	0x7fffffffffffe0ac	25185	61	**62**	00	00				
intPtrPtr	0x7fffffffffffe0b8	0x7fffffffffffe0b0	b0	e0	ff	ff	ff	ff	ff	7f
charPtr	0x7fffffffffffe0c0	0x7fffffffffffe0ac	ac	e0	ff	ff	ff	ff	ff	7f

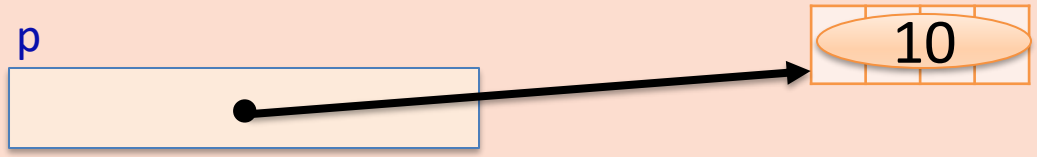
Dynamic Memory

Dynamic memory is allocated during the runtime on programmer's request

- the programmer request/allocate a memory chunk using malloc (or calloc)
- a call to malloc/calloc might fail, and the functions return NULL in this case
- when the memory chunk is no longer needed the programmer must de-allocate it using free.

✗	✗	✗	✓

int *p;
p= 10;
...
<end program>
	int *p;
	p= malloc(sizeof(int));
	p= 10;
	...
	<end program>
		int *p;
		p= malloc(sizeof(int));
		assert(p !=NULL);
		*p= 10;
		...
		<end program>
			int *p;
			p= malloc(sizeof(int));
			assert(p !=NULL);
			*p= 10;
			...
			free(p);
			<end program>

Memory before malloc	Memory after malloc, and just after *p= 10
p <undefined value>	p 

Pointers and dynamic memory: related tools in a nutshell

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);

Examples: Automatic Arrays and Dynamic Arrays

- an array occupies a consecutive chunk of memory
- the array's name is a pointer, pointing to the start of the array's memory chunk
- the array also has type, size (capacity) and length (ie. current number of elements)

	Automatic Arrays	Dynamic Arrays
Memory allocated	when function starts, automatically by compilers	at run time, by programmers, using malloc
Memory de-allocated	when function ends, automatically by compilers	at run time, by programmers, using free
Size (capacity)	a constant	a variable

Example	#define SIZE 1000
	int i, n= 0;
	
	int A[SIZE];
	
	for (n=0; n<SIZE; n++) A[n]= n;
		#define SIZE 1000
		int size= SIZE, i, n=0;
		int *A;
		A = malloc(size*sizeof(int));
		assert(A);
		for (n=0; n<size; n++) A[n]= n;

Differences	Now full, *Impossible* to add new data into A[] Won't work with large SIZE (say, 10^7).	Now *temporarily* full, still *possible* to add new data into A[]by reallocation of A and changing size Practically, size can be unlimited.

When having time, use ChatGPT/Google to explore why using dynamically allocated memory!

String Processing in an example

Task:
Write a function that returns the reverse of a string. The call

```
printf("%s\n", stringReverse("ABC"));
```

should print out:

```
CBA
```

```
char* stringReverse( char *s) {
  char *reverse;
  reverse= malloc( strlen(s) +1  );
          assert( reverse);
           for (int i=0; i<strlen(s); i++) {
     reverse[i]= s[strlen(s)-i-1];
   }
   revesrse[i]= '\0';

   // second method
   char *p= reverse;
   char *q= s + strlen(s) – 1;
   *p = *q;
   while (q >= s ) {
    *p= *q;
            p++;
    q--;
   }
   *p= '\0';



  return reverse;
}
```

Peer Activity: String Duplication

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 }
...
```

Peer Activity: String Duplication

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

a. Yes, it will.

b. No, it will not.

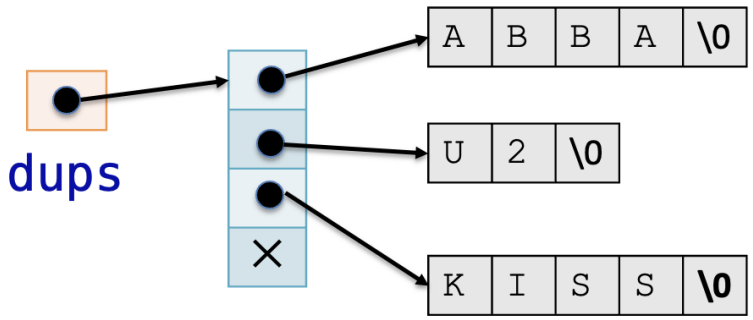
Supposing:

```
#define INITIAL 4
#define MAX_LEN 8
```

Inputs:

```
ABBA
U2
KISS
```

The Intention?




```
...
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';
      /* Store it into the array */
      dups[num_strings] = buf;
      num_strings++;
```

discuss with your peers

Peer Activity: String Duplication

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

b. No, it will not.

Why?

- line 11 is a pointer assignment
- every element of dups[ ] will point to buf

How do we fix it?

```
...
 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 }
...
```

Peer Activity: String Duplication

Will this code snippet work as intended?

Why or why not?

b. No, it will not.

Why?

- line 11 is a pointer assignment
- every element of dups[ ] will point to buf

How do we fix it?

Answer:

- `malloc()` + `strcpy()`
- `strdup()`

```
...
 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] = strdup(buf);
12     num_strings++;
13 }
...
```

C Program Development Tools on ED

Program Development Tools

- Creating/Editing program.c (and other text files) using the excellent built-in editor.
- Compiling with:

  gcc -Wall -g -o program program.c
- Using debugging tools such as valgrind:

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

That gives a report on current and/or potential issues of program.c, if any.

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...

Compiling: GCC

The **GNU Compiler Collection** does multiple things to C programs:

- **preprocessing:** processes `#includes`, preprocessor directives and macros
- **compiling:** 'converts' preprocessor output to assembler source code
- **assembling:** 'converts' assembler source code to object code
- **linking:** combines object code together into an executable

Compiling: GCC: Cheat Sheet

Synopsis:

```
gcc [gcc-options] infile... [more-gcc-options]
```

Essential options:

```
-c              # compile, assemble, but *not* link the source files
-g              # attach debugging information to the output
-lm             # search for the libm library (which includes math.h) while linking
                # has to supersede all infiles
-o file_name    # place the output in file_name
-O3             # optimise the compilation process as much as possible
-Wall           # turn on all optional warnings
```

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

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.

Notes: To obtain the manual page of any command/function, use the command line man. For examples:

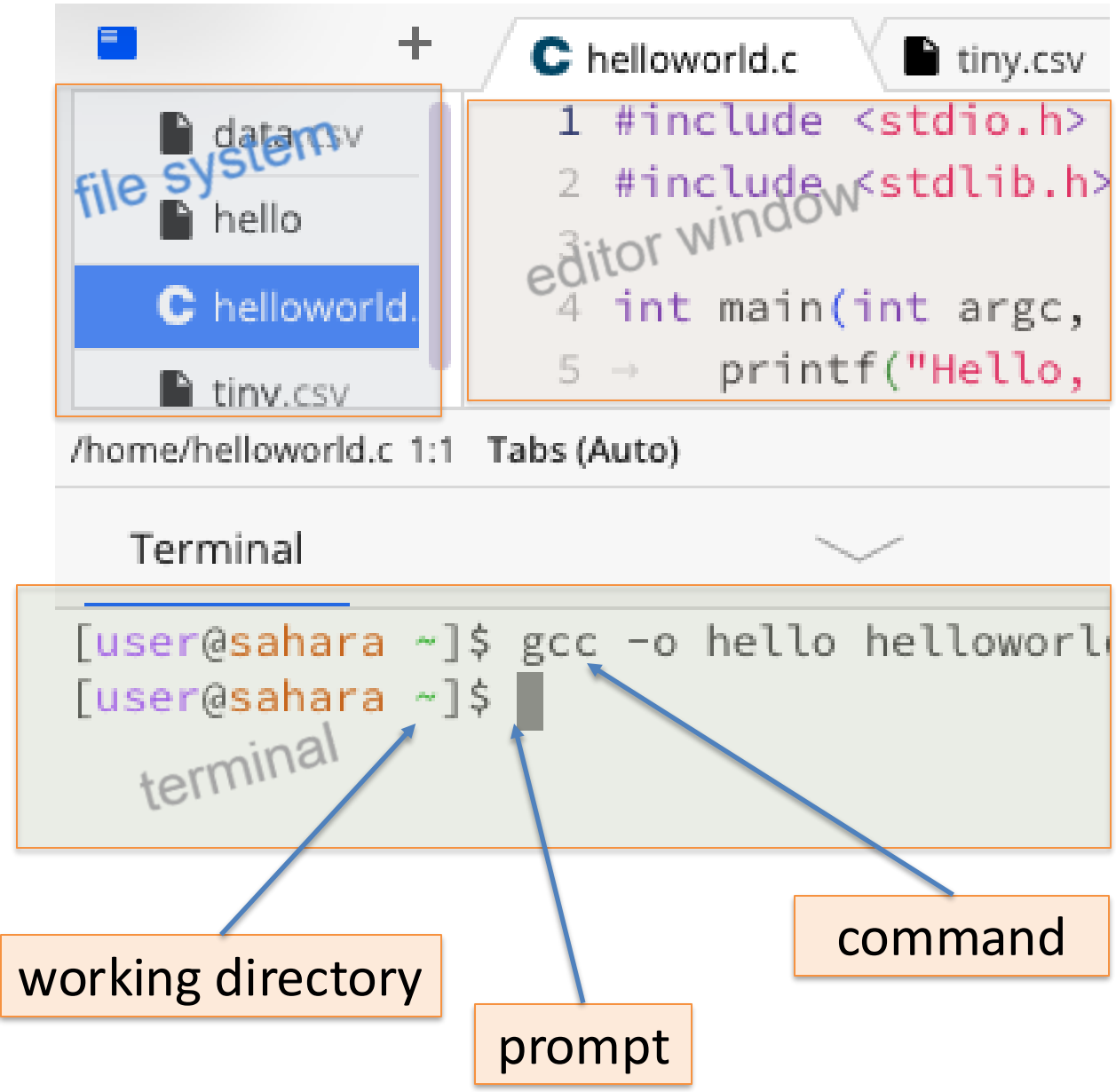
man gcc
man valgrind
man malloc

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>*`)

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) using text commands. The shell interprets and executes commands entered by us in the terminal.
- The Editor helps to create/edit program or text files.

Together with Tutor, do:

- W1.3: essential shell commands
- W1.4: using valgrind for debugging
- start W1.5: Peer Programming

Peer Programming

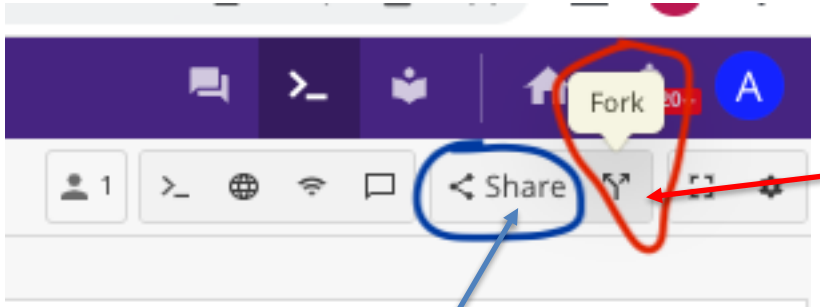
Collaboratively work on each week's exercises

1. Find this week's relevant workspace under the Workspaces tab
2. Fork and share said workspace with your peers
3. Work on solving the problems together
4. Summon me if you need help

LAB is fun!

- Do W1.5, W1.6, W1.7, W1.8 *with your teammate*, using a shared workspace cloned from:

  Ed → Week 2 Workshop → Workspaces → Public → Week 1 Lab



first, click here to clone your own workspace

then, click Share to share your own workspace with your teammates

- [Later:] Remember to individually copy back solutions from the shared workspace to the exercise spaces to get the green ticks
- To copy just a single .c file such as W1.7.c: use copy and paste
- To copy a whole directory such as W1.5:
  - right click on directory name W1.5 of the shared workspace
  - choose Download, it will zip the directory to W1.5.zip
  - go to Exercise W1.5, right click on any spot of file system, then choose Upload Here...
  - navigate to and open W1.5.zip, then click on Upload & Extract

Wrap Up

Done:

- got to know each other, workshop format and ED material, and how to use ED,
- learnt/review some stuffs on C and program development tools, and
- had fun with Peer Activity and Peer Programming.

To Do by 9AM this Sunday:

- finish W1.10 and the whole Week 1 Workshop to get 0.5 bonus mark,
- finish Week 0 Revival to get 1.0 bonus mark,
- do not-yet-done exercises in Week 0 Extras to regain C experience.

To Do before the next workshop:

- make sure that you can use Poll Everywhere
- try Week 1 Extras, especially its job interview questions,
- explore and review lectures of week 1,
- do items from W2.0 to W2.2 in Week 2 Workshop.

HELPS

- QA: after class, 45 minutes, on the ground level
- THIS WEEK ONLY: FYC, Friday 1PM-4PM to assist with C review

Supporting Slides

The Intension

Supposing:

```
#define INITIAL 4
#define MAX_LEN 8
```

Inputs:

```
ABBA
U2
KISS
```

The Intention?