COMP20003 Workshop Week 2
Dynamic Arrays + Modular Programming

1. File IO
2. More on Dynamic Arrays & Memory Management
3. Multi-file and Modular Programming
4. LAB & Team Building for Assignment 1

Please:
- Open Ed
- You should do W2.0 to W2.2 before the workshop
- Skim W2.0 now if you forgot

1. File Operations

File I/O: Read-From and Write-To text files

We can read from a text file just like reading from the keyboard (aka. *standard input stream*).
We can write to a text file just like writing to the screen (aka. *standard output stream*).

standard I/O streams stdin *and* stdout.

- always ready (that is, always *opened*)
- operated with scanf() and printf()
- can be redirected using > and < as in:

    ./program < input.txt > output.txt

But how do we do *file I/O* without using redirection?

Sample Task: *File* "numbers.txt" *contains some integers like* "1 23 5", *we want to produce a file* "copy.txt" *that contains the same integers in the same order.*

How?

numbers.txt
1 23 5

?

copy.txt
1 23 5

File I/O: Read-From and Write-To text files

	Solution 1A (using redirection)	Solution 1B (still using redirection)

Code	int main() {
	  int x;
	
	  while (scanf("%d", &x)== 1) {
	    printf (" %d", x);
	  }
	  printf("\n");
	
	  return 0;
	}
		int main(int argc, char *argv[]) {
		  int x;
		  FILE *in= stdin, *out= stdout;
		
		  while (fscanf(in, "%d", &x)== 1) {
		    fprintf (out, " %d", x);
		  }
		  fprintf(out, "\n");
		
		  return 0;
		}

Exec	./program <numbers.txt >copy.txt	./program <numbers.txt >copy.txt
Note	with redirection we can only read from a single file and write to another single file	

START

I/O operations

END

File I/O: Read-From and Write-To text files

	Solution 1A (using redirection)	Solution 2

Code	int main() {
	  int x;
	
	  while (scanf(“%d”, &x)== 1) {
	    printf (“ %d”, x);
	  }
	  printf(“\n”);
	
	  return 0;
	}
		int main(int argc, char *argv[]) {
		  int x;
		  FILE *in= stdin, *out= stdout;
		  assert (argc > 2);
		  in= fopen(argv[1], “r”);
		  out= fopen(argv[2], “w”);
		  assert (in && out);
		  while (fscanf(in, “%d”, &x)== 1) {
		    fprintf (out, “ %d”, x);
		  }
		  fprintf(out, “\n”);
		
		  fclose(in);
		  fclose(out);
		  return 0;
		}

Exec	./program <numbers.txt >copy.txt	./program numbers.txt copy.txt
Note	with redirection we can only read from a single file and write to another single file	assert (in && out); is the same as: assert ((in!=NULL) && (out!=NULL)); ❤ we can open and work with many files in parallel

START

I/O operations

END

2. More on Memory and Dynamic Arrays

Memory Pools: a C program uses three memory pools during run-time

stack:

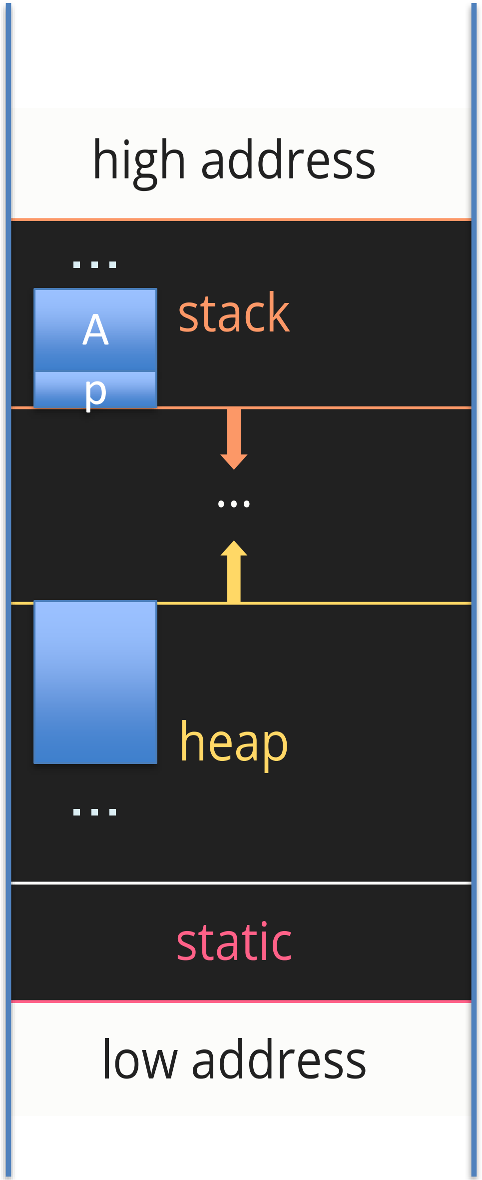
- where local variables live
- automatically allocated when a function starts
- automatically free-ed when the function ends
- has a limited size

heap:

- where dynamically-allocated memory lives
- allocated by programmers via *alloc() calls
- free-ed by programmers via free() calls
- virtually has unlimited size

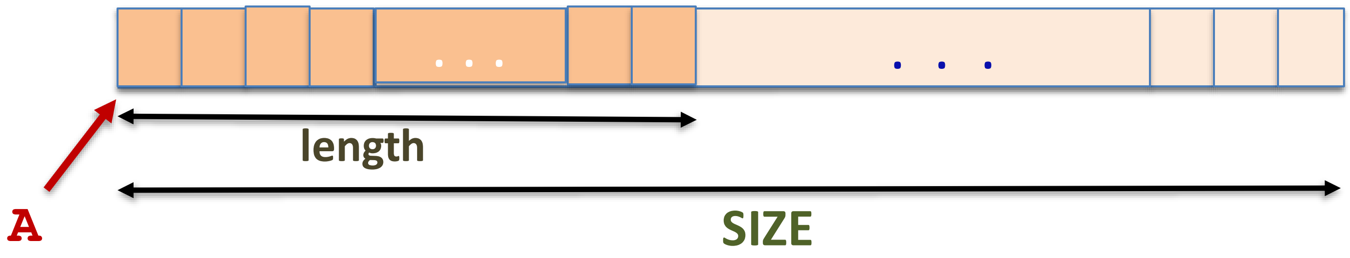
static data segment:

- for global and static variables




```
int foo() {
  double A[2];
  double *p;
  // the storage for A and p
are in the stack

  p = malloc(32);
  // the chunk of 32 bytes
(that p points to) is in the
heap)
```

Automatic (aka. Static) Arrays



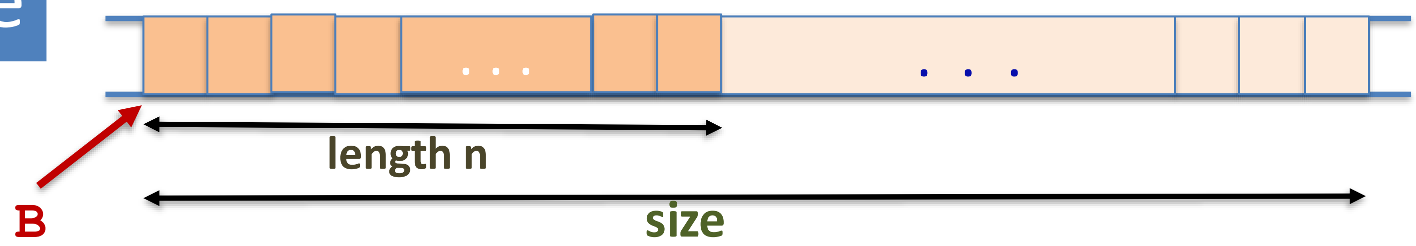
A static array is a consecutive chunk of memory, and has:

- **name** (== constant pointer),
- **SIZE** (aka. pre-defined capacity),
- **length** (or **n**, aka. number of currently used elements)

	A Static Array
Memory allocated	when function starts, automatically by compilers
Memory freed	when function ends, automatically by compilers
Size (capacity)	a constant

Example: *Read a sequence of integers and store in an array.*	`#define SIZE 100`
	`int A[SIZE], n= 0, x;`
	
	`while ( scanf("%d", &x)==1) {`
	`  if (n==SIZE)`
	`    break; // A[] is full`
	`           // Impossible to add new data`
	`  A[n]= x;`
	`  n++;`
	`}`

Dynamic Arrays: Standard Recipe



Dynamic Arrays	explanations
`#define INIT_SIZE 4`	initial size of the array (4 in this cases)
`int size=INIT_SIZE, n=0, x;`	
`int *B;`	B is a undefined pointer
`B= malloc(size * sizeof(*B) );`	B now points to a block (and becomes an array) of 4 elements
`assert(B != NULL);`	
`while ( scanf(“%d”, &x)==1) {`	
`  if (n==size) {`	if array B is full
`    size= szie * 2;`	
`    b= realloc(B, size*sizeof(*B));`	then resize it using realloc
`    assert(B)`	
`    // re-allocate memory for B`	
`    // B= realloc`	
`  }`	
`  B[n]= x;`	
`  n++;`	
`}`	free the memory used by B when B is no more needed

Tool for Memory Re-Allocation: realloc()

Prototype: `void *realloc(void *ptr, size_t size)`

Example :

```
... // B has been malloc-ed before with size elements
size = size * 2;
B= realloc(B, size * sizeof(*B));
assert (B != NULL);
```

Purpose: Resizes some previously-allocated memory block.

Arguments:

- `ptr`: pointer to a memory block to be resized
- `size`: new size of memory block, in bytes

Return values:

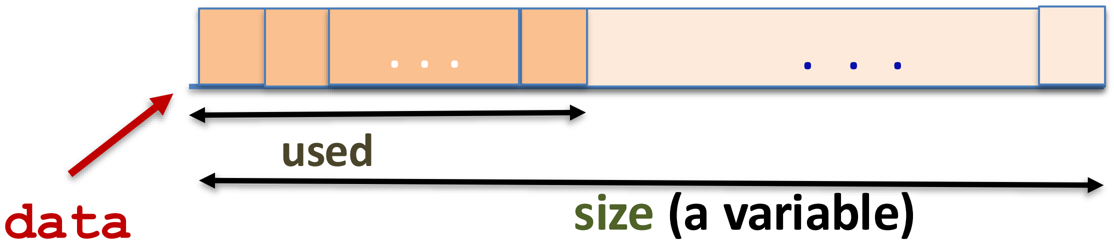
- pointer to newly-allocated memory if reallocation was successful, or
- `NULL` otherwise

Notes:

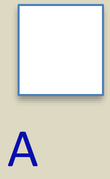
- the original content is preserved in the newly-allocated block

Example of Conventional Usage

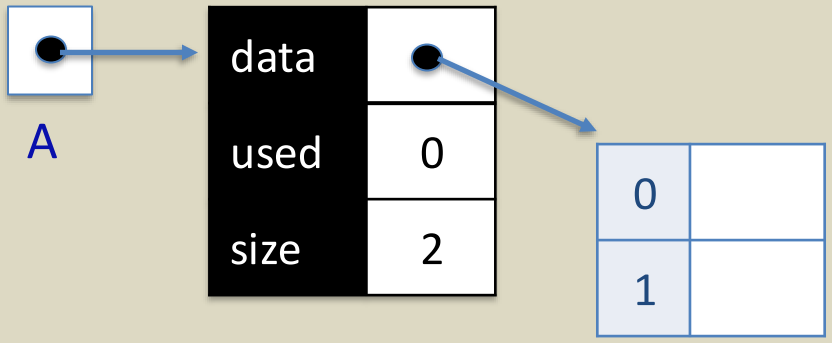
```
#define INIT_SIZE 2
typedef struct array {
  void **data;
  int size, used;
} array_t;

array_t *create_array() {
  array_t *arr= malloc(sizeof(*arr));
  assert(arr);
  arr->used= 0;
  arr->size= INIT_SIZE;
  arr->data= malloc(arr->size*sizeof(void *));
  assert(arr->data);
  return arr;
}

void ensure_array_size(array_t *A) {
  // realloc for A->data if needed
  // to ensure that A->size > A->used
  ...
}
...
```



array_t *A;



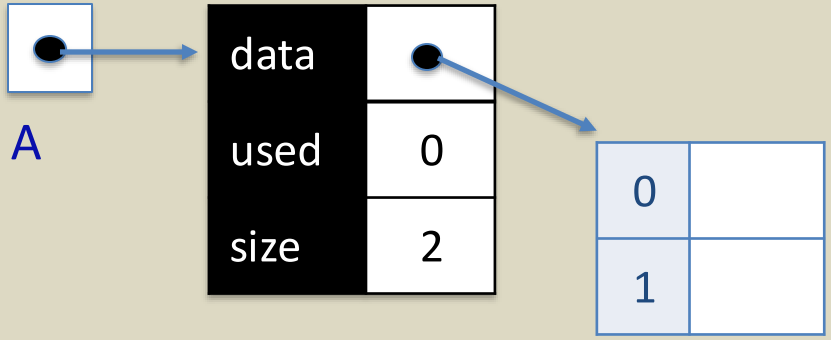
A= create_array();



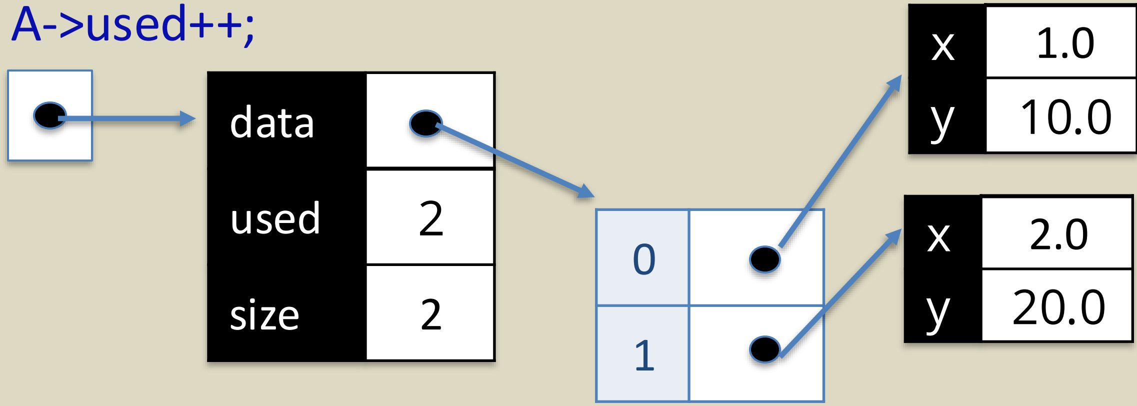
Example of Conventional Usage

```c
#define INIT_SIZE 2
typedef struct array {
  void **data;
  int size, used;
} array_t;

array_t *create_array() {
  ...
}

void ensure_array_size(array_t *A) {
  // realloc for A->data if needed
  // to ensure that A->size > A->used
  ...
}
...
typedef struct {
  double x,y;
} vector_t;

vector_t *create_vector(double x, double y) {
  vector_t *v= malloc(sizeof(*v));
  assert (v);
  *v= (vector_t) {x, y};
  return v;
}
```

```c
A= create_array();
```



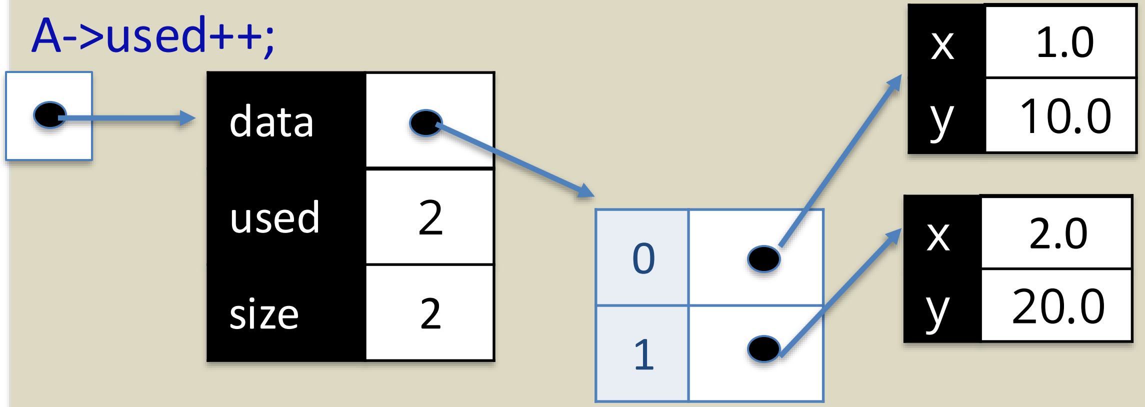

```c
A->data[0]= create_vector(1,10);
A->used++;
A->data[1]= create_vector(2,20);
A->used++;
```



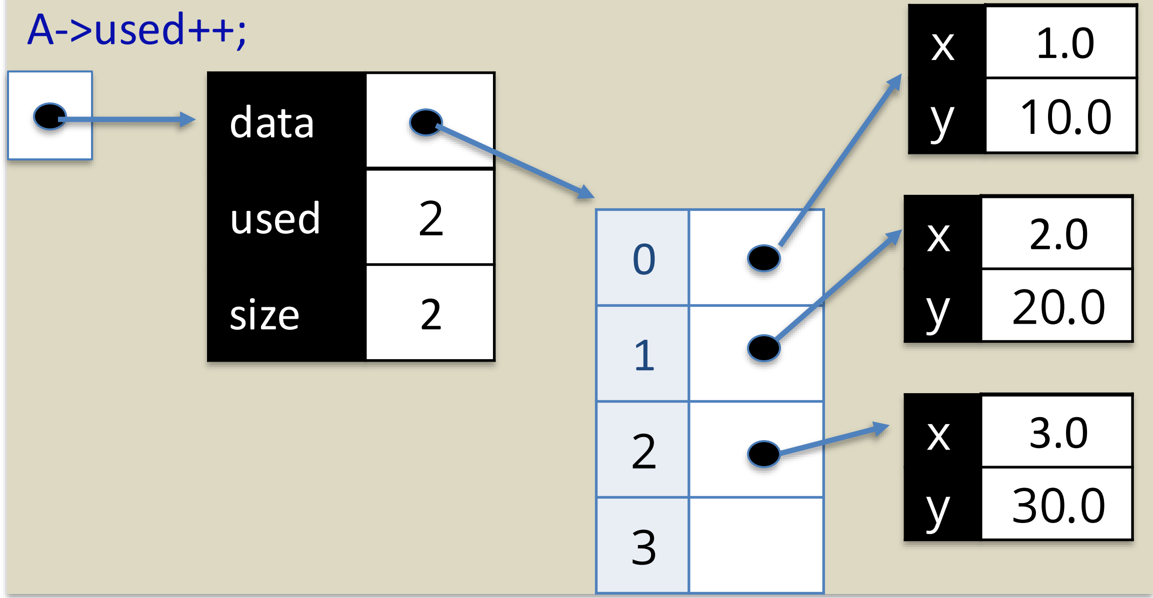

```c
A->data[2]= create_vector(2,20); ???
```

```c
#define INIT_SIZE 2
typedef struct array {
  void **data;
  int size, used;
} array_t;

array_t *create_array() {
  ...
}

void ensure_array_size(array_t *A) {
  // realloc for A->data if needed
  // to ensure that A->size > A->used
  ...
}
...
typedef struct {
  double x,y;
} vector_t;

vector_t *create_vector(double x, double y) {
  vector_t *v= malloc(sizeof(*v));
  assert (v);
  *v= (vector_t) {x, y};
  return v;
}
```

```c
A= create_array();
A->data[0]= create_vector(1,10);
A->used++;
A->data[1]= create_vector(2,20);
A->used++;
```




```c
ensure_array_size(A);
A->data[2]= create_vector(3,30);
A->used++;
```



Peer Activity: Dynamic Array Expansion

What is the right ordering for these code snippets to implement a function `int ensure_array_size(struct array *arr)` that expands a `struct array`'s data space when it is full (if memory is available)? Assume the function returns `0` after all statements if it hasn't returned yet.

a. 3-2-5-1-4
b. 3-2-5-4-1
c. 2-5-1-4-3
d. 2-5-4-1-3

```c
/* Snippet 1 */
arr->data = res;

/* Snippet 2 */
arr->size *= 2;

/* Snippet 3 */
if (arr->used < arr->size) return 0;

/* Snippet 4 */
if (res == NULL) {
    arr->size /= 2;
    return 1;
}

/* Snippet 5 */
void *res =
    realloc(arr->data, arr->size*sizeof(void*));
```

Peer Activity: Dynamic Array Expansion

What is the right ordering for these code snippets to implement a function `int ensure_array_size(struct array *arr)` **that expands a** `struct array`**'s data space when it is full (if memory is available)? Assume the function returns** `0` **after all statements if it hasn't returned yet.**

b. 3-2-5-4-1

Why?

- check if we actually need more space before anything else
- need to make sure `realloc()` succeeded
  - undo `arr->size` change on failure
  - otherwise update `arr->data`

```
int ensure_array_size(struct array *arr) {
    /* Array still has some space */
    if (arr->used < arr->size) return 0;

    /* Array needs more space; double its size */
    arr->size *= 2;
    void *res =
        realloc(arr->data, arr->size*sizeof(void*));
    if (res == NULL) {
        /* realloc() failed; undo changes */
        arr->size /= 2;
        return 1;
    }
    arr->data = res;
    return 0;
}
```

C Memory: Caveats

C grants programmers the **great power** of governing over dynamic memory. We can

- use virtually unlimited-size data structures,
- get and return memory on our demand.

The great power comes with a **great responsibility**.
Overstepping memory boundaries is a very real possibility with C.
Its consequences range from:

- **best:** getting immediate error (e.g. Segmentation fault) and crashing
- **worse:** overwriting memory 'housekeeping' data and crashing some time later
- **worst:** silently overwriting other variables and continuing execution

Notes: valgrind is a great tool for discovering potential memory problems in C codes.

3. Program Development with Modular Programming

Modular Programming

Modular programming: breaking down a large program into smaller, independent modules or functions that can be developed and tested separately.

Each module is designed to perform a specific task or set of tasks. A module communicates with application programs or other modules through well-defined interfaces.

Modular Programming: why?

```
#include <stdio.h> …
```

declarations & function prototypes for working with dynamic arrays and linked list

```
int main(…) {
  …
  using dynamic arrays
  using linked lists
  …
}
```

implementation of functions,
including the ones for dynamic arrays and linked lists

Why Modular Programming?

- program could be too long, complicated and un-manageable!
- Modular Programming breaks long code into manageable and reusable modules and files.

File Types

- **header** (*.h):
  - contains:
    - function prototypes
    - `struct`/type declarations
  - usually `#include`'d in source files

- **source** (*.c):
  - contains:
    - #include header files
    - function definitions
    - `struct`/type definitions

gcc –c …

- **object** (*.o):
  - contains **object** code that is:
    - non-executable
    - platform-specific

gcc –o …

- **executable:**
  - created by linking several object files together

Modular Programming: Example

module “dynamic_array”

- interface (array.h): data type defs, and function prototypes
- source (array.c) : implementation of all functions in the interface

module “list”

- interface (llist.h): data type defs, and function prototypes
- source (llist.c) : implementation of all functions in the interface

application program

```
#include <stdio.h> ...
#include “llist.h” // include the interface of module list
#include “array.h” // include the interface of module dynamic array

int main(...) {
 ...
  //using dynamic arrays & linked list facilities
   ...
}
```

Benefits:

- each module can be developed and tested separately
- modules are reusable
- ...

Simple example W2.2: module factorial

program.c

```
#include <stdio.h>

#define MAX_N 14
int factorial(int);

int main() {
   ...
   m= factorial(k);
   ...
}

int factorial(int n)
{
   ...
   return soln;
}
```

```
gcc -o prog program.c
```

main.c

```
#include <stdio.h>

#include "factorial.h"

int main() {
   ...
   m= factorial(k);
   ...
}
```

factorial.h

interface = header file

```
#define MAX_N 14
int factorial(int);
```

factorial.c

```
#include "factorial.h"

int factorial(int n) {
   ...
   return soln;
}
```

```
gcc -c factorial.c -o factorial.o
gcc -c main.c -o main.o
gcc -o prog   main.o   factorial.o
```

executable file

object file

Another example: modules factorial and combination

main.c

```
#include <stdio.h>

#include "factorial.h"
#include "combination.h"


int main() {
    ...
    m= factorial(k);
    choices= nCk(n,k);
    ...
}
```

factorial.h

```
#define MAX_N 14
int factorial(int);
```

combination.h

```
#include "factorial.h"
int nCk(int n, int k);
...
```

factorial.c

```
#include "factorial.h"

int factorial(int n) {
    ...
    return soln;
}
```

combination.c

```
#include "combination.h"

int nCk(int n, int k) {
    return factorial(n) /
        ( factorial(k) *
          factorial(n-k) );
}
...
```

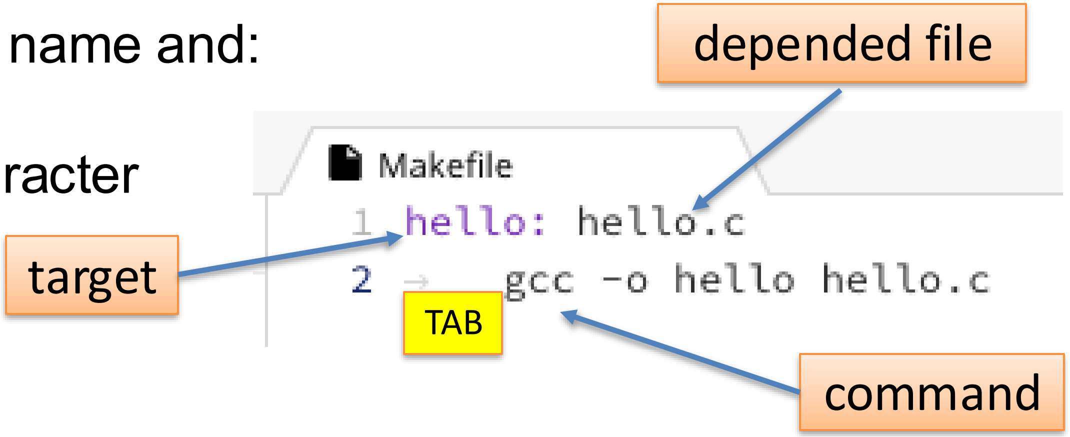
```
gcc -c factorial.c -o factorial.o [build factorial.o from factorial.c]
gcc -c main.c                     [build main.o from main.c, -o ... is by default ]
gcc -c combination.c
gcc -o newProg main.o factorial.o combination.o    [build executable newProg]
```

Multi-file auto-compilation with make and Makefile

Command make reads the user's file Makefile and *builds* the first *target* in that file.

File Makefile contains a sequence of targets. A target has name and:

- a list of *dependencies* (depended files or targets)
- shell's *commands*, preceding by a single TAB character



When building a target, the command make:

- builds all the depended targets
- runs the target's commands iif a depended file has been changed since the last build

We can use make for automatic compiling a C project by:

- build a file Makefile in the project's directory,
- organise Makefile so that the targets and their dependencies and commands create the needed sequence of compiling commands

Why make?

- simplify the repeated compiling process to just typing make,
- allowing to recompile only the changed files (great for multiple file projects)

Makefile

```
all: main.o combination.o factorial.o
	gcc -o combination main.o combination.o factorial.o

main.o: main.c
	gcc -Wall -c -o main.o main.c

combination.o: combination.c combination.h
	gcc -Wall -c -o combination.o combination.c

factorial.o: factorial.c factorioal.h
	gcc -Wall -c -o factorial.o factorial.c
```

target

dependencies

command

auto-compiling with `make`

$`make` reads file `Makefile` and executes the first target (`all`, in this case)

- target `all` depends on 3 targets `main.o`, `combination.o`, `factorial.o`
- first, executes 3 targets one-by-one
  - target `main.o` depends on `main.c`, which is ready as a file
  - executes command `gcc -Wall -c -o main.o main.c`
  - does similarly for the 2 remaining targets
- second, runs the accompanied command to build `combination`:

  `gcc -o combination main.o combination.o factorial.o`

Do Together with Tutor: Exercises W2.3.b and W2.4

- Skim W2.3.a
- Do 2.3.a together with your tutor (if you never created a Makefile)
- Then, for W2.3.b:
  - rename Makefile and rebuild a new Makefile from scratch,
  - noting that Makefile syntax is a bit picky with using TAB
- Do W2.4 with your tutor to quick understand the logic behind qStud - a project that will be expanded further in this and the next couple of workshops.

Modules in W2.4's qStud

driver.c

```
...
#include "data.h"
#include "array.h"

... main() ...
// Using stuffs declared in the interfaces
data.h and array.h
```

data.h

declare `student_t` and operations for read/write `student_t` data...

array.h

```
#include "data.h"
// declare operations on array of student_t
such as building an array, search, delete
```

data.c

```
#include ...
#include "data.h"
// implementation of operations
declared in data.h
```

array.c

```
#include ...
#include "array.h"
// implementation of operations declared
in array.h
```

Lab: Have Fun and Form Your Team for Assignment 1

- Finish W2.3, W2.4 if not yet done
- [Right Now:] Form your team (of 2 people) for the coming Assignment 1

- a Team Work
- continued in Assignment 2

Assignment 1
Released this week-end

- Be ready to start on Monday with your team
- Build your team **right now**
- Practice using a shared workspace today

- Do W2.5, W2.6 with your teammate, using a shared workspace cloned from:

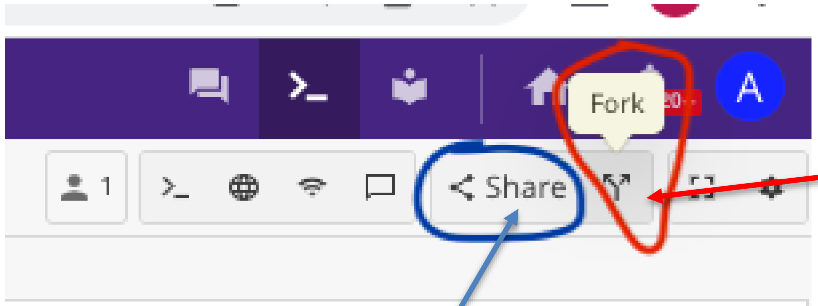
  Ed → Week 2 Workshop → Workspaces → Public → Week 2 Lab

- *It's important to build your team and learn to work together effectively.*

LAB is fun!

- Do W2.4, W2.5, W2.6 *with your teammate*, using a shared workspace cloned from:

  Ed → Week 2 Workshop → Workspaces → Public → Week 2 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 W2.5:
  - right click on directory name W2.5 of the shared workspace
  - choose Download, it will zip the directory to W2.5.zip
  - go to Exercise W2.5, right click on any spot of file system, then choose Upload Here...
  - navigate to and open W2.5.zip, then click on Upload & Extract

Appendices: Additional Slides for Reviewing