COMP20003 Workshop Week 2

Dynamic Arrays + Program Development

1. File IO
2. More on Dynamic Arrays & Memory Management
3. Multi-file and Modular Programming

- 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 now
- Practice using a shared workspace today

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

We know how to use *standard I/O streamss*.

- standard I/O streams: automatically opened on program startup
- `scanf()` and `printf():` operates on standard I/O streams

But how do we do *file I/O*?

- How do we start and end working with a file?
- Any `scanf()` and `printf()`-like functions that operate on files?

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

Sample Task: File “numbers.txt” contains some integers like “1 23 52 5”, we want to produce a file “doubles.txt” that contains the doubled values.

Some Equivalent Solutions:

	Solution 1A	**Solution 1B**

Code	`int main() {`
	`  int x;`
	`  while (scanf(“%d”, &x)== 1) {`
	`    printf (“ %d”, x);`
	`  }`
	`  printf(“\n”);`
	`  return 0;`
	`}`
		`int main() {`
		`  int x;`
		`  while (fscanf(stdin, “%d”, &x)== 1) {`
		`    fprintf (stdout, “ %d”, x);`
		`  }`
		`  fprintf(stdout, “\n”);`
		`  return 0;`
		`}`

Exec	`./program <numbers.txt >doubled.txt`	`./program <numbers.txt >doubled.txt`
Note		`stdin` is the file handle that represents the standard input (the keyboard) `stdin` is the file handle that represents the standard output (the screen)

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

	Solution 1B	Solution 1C

Code	int main() {
	  int x;
	  while (fscanf(stdin, “%d”, &x)== 1) {
	    fprintf (stdout, “ %d”, x);
	  }
	  fprintf(stdout, “\n”);
	  return 0;
	}
		int main() {
		  FILE *in= stdin, *out= stdout;
		  int x;
		  while (fscanf(in, “%d”, &x)== 1) {
		    fprintf (out, “ %d”, x);
		  }
		  fprintf(out, “\n”);
		  return 0;
		}

Exec	./program <numbers.txt >doubled.txt	./program <numbers.txt >doubled.txt
Note	stdin is the *file handle* that represents the standard input (the keyboard) stdin is the *file handle* that represents the standard output (the screen)	“FILE *” is the data type for file handles. in and out are variable file handles. stdin and stdout are *constant* file handles.

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

	Solution 1B	Solution 2

Code	int main() {
	  FILE *in= stdin, *out= stdout;
	  int x;
	  while (fscanf(in, “%d”, &x)== 1) {
	    fprintf (out, “ %d”, x);
	  }
	  fprintf(out, “\n”);
	  return 0;
	}
		int main() {
		  FILE *in= stdin, *out= stdout;
		  int x;
		START in= fopen(“numbers.txt”, “r”);
		  out= fopen(“doubled.txt”, “w”);
		  assert (in && out);
		I/O operations while (fscanf(in, “%d”, &x)== 1) {
		    fprintf (out, “ %d”, x);
		  }
		  fprintf(out, “\n”);
		
		END fclose(in);
		  fclose(out);
		  return 0;
		}

Exec	./program <numbers.txt >doubled.txt	./program
Note		assert (in && out); is the same as: assert ((in!=NULL) && (out!=NULL)); ❤ we can open and work with many files in parallel ❤ we can use argv[] to get the filenames

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

stack:

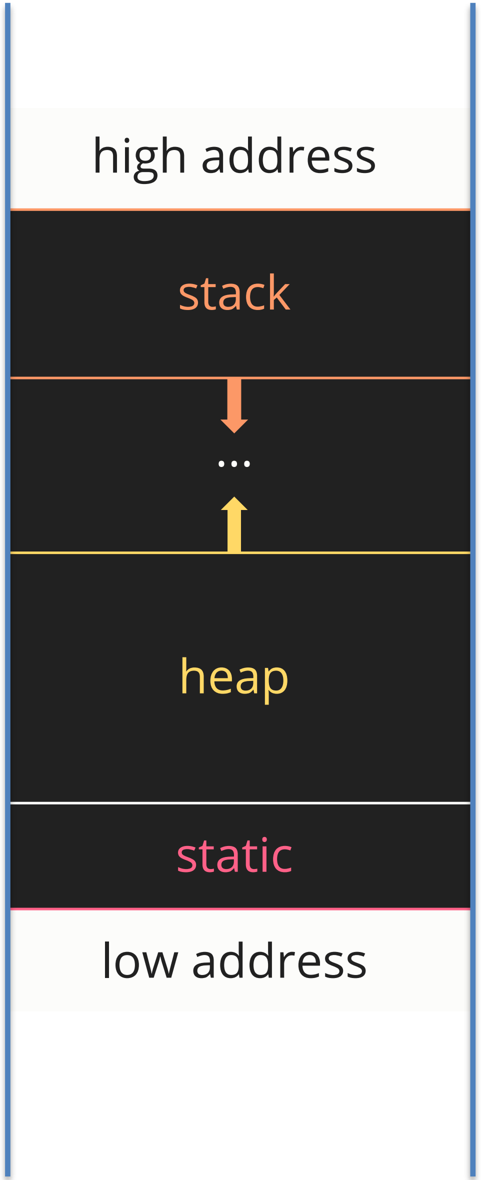
- where local variables live
- automatically allocated when a function starts
- automatically free-ed when the function ends
- has a limited size

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:

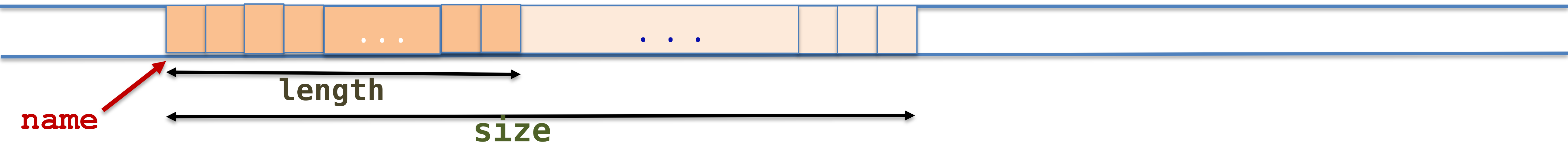
- for global and static variables



DYNAMIC ARRAYS: Using arrays for sequences of data

An array:

- is a consecutive chunk of memory
- has **name** (== pointer constant), **size** (aka. capacity), **length** (aka. **n**, aka. number of currently used elements)

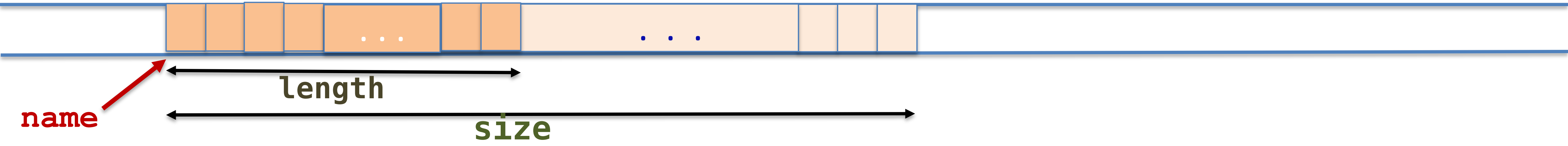


	Static Arrays	Dynamic Arrays
Memory allocated	when function starts, automatically by compilers	at run time, by programmers, using `malloc`
Memory freed	when function ends, automatically by compilers	at run time, by programmers, using `free`
Size (capacity)	a constant	a variable

Example	`#define SIZE 100`
	`int i, n= 0;`
	
	`int A[SIZE];`
	`for (n=0; n<SIZE; n++) A[n]=rand();`
		`int size=100, i, n=0;`
		`int *A;`
		`A= malloc(size * sizeof(*A) );`
		`assert(A);`
		`for (n=0; n<size; n++) A[n]= rand();`

Note	Now `A[]` is full. *Impossible* to add new data into `A[]`	Now `A[]` is *temporarily* full. Still *possible* to add new data into `A[]` by changing `size` and using `realloc`

DYNAMIC ARRAYS: Using arrays for sequences of data



<table>
<tr><th></th><th>Static Arrays</th><th>Dynamic Arrays</th></tr>
<tr><td>Example</td><td><pre>
#define SIZE 100
int i, n= 0, x;

int A[SIZE];

while (scanf("%d", &x)==1) {
  if (n==SIZE) {
    break;
  }

  A[n]= x;
  n++;
}
</pre></td><td><pre>
#define INIT_SIZE 4
int size=INIT_SIZE, i, n=0, x;
int *A;
A= malloc(size* sizeof(*A));
assert(A);

while (scanf("%d", &x)==1) {
  if (n==size) {
    size = size * 2;
    A= realloc(A, size*sizeof(*A));
    assert(A != NULL);
  }
  A[n]= x;
  n++;
}
</pre></td></tr>
</table>

Memory Pools: Caveats

C grants programmers the **great power** of governing over memory.
That 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

Peer Activity: Discuss, then fill in W2.10 (or W2.1 if haven't done)

Activity 1: Dynamic Arrays

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? Assume that there is a "`return 0;`" at the end of the function body.

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

```
/* 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*));
```

Activity 2: Memory Pools

Consider the following code snippet:

```
...
int x = 2;
int y = 0;
int z = 5;
*(&y + 1) = 1;
...
```

What is most likely to happen when the code snippet is run?

A. *Error: Invalid syntax*
B. `x == 1`
C. `y == 256`
D. `z == 1`
E. Error: Segmentation fault

NOTES:
From next week, all pre-workshop quiz should be done before the workshop starts

Lab’s Focus : 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

```
#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: Example

module "dynamic_array"

- interface: data type defs, and function prototypes
- implementation of all functions in the interfaces

module "list"

- interface: data type defs, and function prototypes
- implementation of all functions in the interfaces

application program

```
#include <stdio.h> ...
// include the interface of module list
// include the interface of module stack

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;
}
```

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

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
    gcc -Wall -c -o combination.o combination.c

factorial.o: factorial.c
    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)

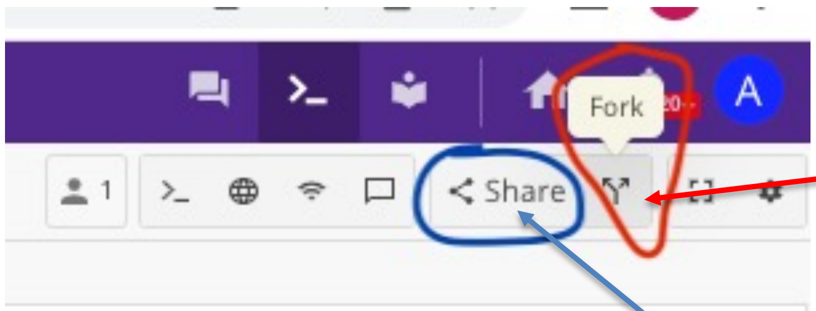
- target `all` depends on 3 targets `main.o`, `combination.o`, `factorial.o`
- first, executes 3 targets one-by-one
  - target `main.o` depends on `main.c`, which is ready as a file
  - executes command `gcc -Wall -c -o main.o main.c`
  - does similarly for the 2 remaining targets
- second, runs the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

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

LAB

- Do W2.3, W2.4 together with Anh, and help your classmates if you can
- Work with your teammate, using a shared workspace cloned from:

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



first, click here to clone your own workspace

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

- Do other exercises together with your teammate
- Have fun
- [Later:] Remember to individually copy back solutions from the shared workspace to the exercise spaces to get the green ticks

Wrap-up: Content of Week 1 Lectures & Week 2 Workhop

Lectures Week 1

Lecture 1:

- admin
- Movie Star Scheduling Problem

Lecture 2:

- intro to algorithms: concept, classification, correctness & efficiency
- Fibonacci numbers: algorithms and their efficiency

W2.0 PDF Slides

- Memory pools, PA on stack mem
- dynamic array, PA
- File IO
- Multi-file prog with make, file types: .c, .h, object, exec

#	ED exercise
2.1	pre quiz: memory pool; dynamic array
2.2	2-stages compilation, header files
2.3	make
*	reminding on shared workspace
2.4 2.4E	modularization: `qStud` module independence [reading only]
2.5	adding file IO
2.6	saving mem with dynamic arrays
E2.1	dynamic array of strings
E2.2	array-pointer relationship
E2.3	double pointers with 2D arrays
E2.4	tiny example for makefile
E2.5	makefile simple → complicated
E2.6	generate random data
E2.11 E2.12	challenges: emirp prime numbers

WORKSHOP

DISCUSSIONS:

- File IO
- review last week dynamic arrays
- dynamic arrays
- note on memory pools
- class discussions on Peer Activity

LAB:

- the importance of peer programming
- modular programming

NOTES:

- from next week: Wx.1 should be done before class