COMP10002 Workshop Week 4

Functions: The Lazy Programmers' Best Friend

1. Functions, discuss 6.02
2. Pointers, Pointers as Function Parameters

LAB:

- minimum requirements: 5.06 (amicable pair), 6.09 (coin change)
- other exercises (a lot!) in Chapters 5 and 6
- Q&A for the coming-soon MST

PROGRAM STRUCTURE

- A C program is a set of functions, with one and only one function main().
- Execution of a C program is the execution of its main().
- Good convention: main() be the first implemented function.
- Functions other than main() must be declared before use and implemented at some stage.

	my_program.c
#include & #define	#include <stdio.h>
	#include <stdlib.h>
	#define…
function declarations (prototypes)	int intPow(int n, int k) ;
	…
main() function –start	int main(int argc, char *argv[]) {
	int n= 2, m= 3;
	…
	printf(“n^m + m^n = %d\n”,
	intPow(n, m) + intPow(m, n));
	…
	return 0;
– end	}
implementation of declared function(s)	int intPow(int n, int k) {
	// supposing k>=0 and and no overflow
	int i, pow= 1;
	for (i=0; i<k; i++) {
	pow *= n;
	}
	return pow;
	}
	…

How Functions Work Independently & Together?

- All variables declared inside a function, including its parameters, are local to that function.
- A function can access (use and modify) its own local variables but cannot directly access the local variables of other functions.
- Functions share data by passing parameters and returning values.

my_program.c	

`#include <stdio.h>`
`#include <stdlib.h>`
`#define…`
`int intPow( int n, int k ) ;`
`...`	

`int main(int argc, char *argv[]) {`
`  int n= 2, m= 3;`
`  ...`
`  printf( "n^m + m^n = %d\n",`
`          intPow(n, m) + intPow(m, n));`
`  ...`
`  return 0;`
`}`
	main() can only access:
	argc
	argv
	n
	m

`int intPow( int n, int k ) {`
`  // supposing k>=0 and and no overflow`
`  int i, pow= 1;`
`  for (i=0; i<k; i++) {`
`    pow *= n;`
`  }`
`  return pow;`
`}`
`...`
	intPow can only access n, k, i, pow.
	
	Note that this n differs from that n declared in main()

Exercise 6.02: understanding local and global declarations and their scopes

***6.2**: For each of the 3 marked points, write down a list of all of the program-declared variables and functions that are in scope at that point, and for each identifier, its type. Don't forget* **main**, **argc**, **argv**. *Where there are more than one choice of a given name, be sure to indicate which one you are referring to.*

```
int bill(int jack, int jane);
double jane(double dick, int fred, double dave);

int trev;

int main(int argc, char *argv[]) {
	double beth;
	int pete, bill;		/* -- point #1 -- */
	return 0;
}

int bill (int jack, int jane) {
	int mary;
	double zack;	/* -- point #2 -- */
	return 0;
}

double jane(double dick, int fred, double dave) {
	double trev;		/* -- point #3 -- */
	return 0.0;
}
```

Why Functions in Programming?

Programs can get long and messy. Functions help us:

- **Organization**: Break down a large problem into smaller, manageable chunks.
- **Reusability**: Write a piece of code once and use it many times.
- **Readability**: Make your code easier for others (and yourself!) to understand.
- **Maintainability**: Easier to debug and update a small, specific function than a giant block of code.

Anatomy of a C Function

Structure: Each function has:

A **name**

Inputs (parameters)

A process (the body)

An output (return value)

```
return_type function_name(parameter_list) {
   // function body
   // statements to be executed
   return value; // if return_type is not void
 }
```

Key Components:

return_type: The type of data the function sends back after it's done (e.g., int, float, void).
function_name: A unique name that follows standard C naming conventions.
parameter_list: The list of variables the function takes as input. This can be empty.
function body: The code block that contains the instructions to perform the task.
return statement: Passes the result back to the calling function.

pass-by-value: parameters are always passed by values

When executing function call

add(a, b+2):

- the value of *expression* a+4 is computed and passed to m
- the value of *expressions* b is computed and passed to n

when the function finishes, the value of *expression* total is computed and sent back to the caller

```
1  int main(...) {
2    int a=2, b=4, sum= 0;
3
4    sum = add(a+4, b );
5    // sum= ?
     // n = ?
     // b = ?

    ...
   }

   int add(int m,    int n) { // m and n are local in add
     int total= m + n;
     n= 10;
     return total;
   }
```

6

4

6 is passed to m

4 is passed to n

8

- Via a parameter (such as n) a function receives a **copy** of the argument, so it cannot change the original variable (in this case: cannot change b).

- Via return, a function sends a value back to the caller.

Why Functions Matter for Algorithms?

Algorithms = step-by-step problem solving.
Functions help us to think abstractly!

Functions let us:

- Express each step as a function
- Build bigger algorithms from smaller ones
- Focus on logic, not messy details

Class Example:

- Discuss Ex5.06

How can function produce 2 (or more) outputs?

Consider the function call

sAndP(a,b, **sum, product**):

- Would it change sum to 6, product to 8?
- If not, how can sAndP send back two outputs, **s** and **p**, back to main()?

```
int main(...) {
  int a=2, b=4,      sum = 0, product = 0 ;

  sAndP(a, b, sum , product);



}


void sAndP( int m, int n, int  s , int  p ) {

  s = m + n ;

  p = m * n ;

}
```

How to produce 2 (or more) outputs? How about sending addresses?

Consider the function call

sAndP(a,b, **sum, product**):

- Would it change sum to 6, product to 8?
- If not, how can sAndP send back two outputs, **s** and **p**, back to main()?

```
int main(...) {
  int a=2, b=4,      sum = 0, product = 0 ;

  sAndP(a, b, address of sum , address of product);
     // need: sending address of sum and address of product
  //           to sAndP
  //
}


void sAndP( int m, int n, address s , address p ) {

  memory_at_address_s = m + n ;

  memory_at_address_p = m * n ;

}
```

Data type: `int *` ⇔ "pointer to int" ⇔ "address of int"

Declaration	Memory storages and its (supposed) address	
`int a= 18;`	1000 18 **a**	**a** is a **int** cell in the memory, with value of 18. Suppose that the cell is at address 1000.
`int *   pa;`	1000 1008 18 ? **a** **pa**	**pa** is an **int pointer**, it can hold the address of an **int** Here we supposed that **pa** itself is at address 1008.

Data type: `int *` ⇔ "pointer to int" ⇔ "address of int"

`int *pa;`	1000: **18** (`a`); 1008: **?** (`pa`)	
`pa= &a;`	1000: **18** (`a`) ← 1008: **1000** (`pa`)	`pa` now holds the address of `a`. We say that `pa` "points" to `a`. using the value of `pa` we can have *indirect access* to `a`

`(*pa)++;`	1000: **19** (`a`, `*pa`) ← 1008: **1000** (`pa`)	`pa` has value `1000`, `*pa` is `*(1000)` and has value `19`
		
		`(*pa)++;`
		is equivalent to:
		`a++;`
		So: `*pa` is an alias for `a`

Notes on pointers:

Symbol * has different meanings in:	
int* pA = &A; * specifies that pA is a pointer pA is an integer pointer (datatype int *)	*pA = 100; * is the dereference operator *pA is the memory cell pA points to

- Each datatype has its own pointer type. Example:

```
int I;      char C;      float F;      double D;
int *pI;    char *pC;    float *pF;    double *pD;
```

- Different pointer type are ... different, they cannot be cross-compared:

Example of Valid Use	Example of Invalid Use
if (*pI == *pC) ...	if (pI == pC) ...

How to produce 2 (or more) outputs? How about sending addresses?

```
int main(...) {
  int a=2, b=4,     sum = 0, product = 0 ;

  sAndP(a, b, &sum , &product);
    // need: sending address of sum and address of product
  //          to sAndP
  //
}



void sAndP( int m, int n, int * ps, int * pp ) {

  /* memory_at_address_ps */    *ps = m + n ;
  /* memory_at_address_pp */    *pp = m * n ;

}
```

Function parameters: passing values and passing addresses (references)

Consider the function call

sAndP(a,b, &**sum**,&**product**):

can it change the value of:

- a ?
- b ?
- **&sum** ?
- **&product** ?
- **sum** ?
- **product** ?

```
int main(...) {
  int a=2, b=4,      sum = 0, product = 0 ;

  sAndP(a, b, &sum ,          &product);

// a= ?
// sum= ?
// product= ?

 ...
}

void sAndP( int m, int n, int * ps , int *pp ) {

  *ps = m + n ;        // equivalent to sum= m + n;

  *pp = m * n ;
  m = 10;
}
```

2 is passed to m

&sum is passed to **ps**
so ***ps** is an alias for **sum**

pp= &product,
and so ***pp** and
product refer to
the same cell

What are passed to functions scanf and printf, why?

Compare:

```
scanf("%d", &n);
```

and

```
printf("%d", n);
```

What’s the difference in the values passed to functions? Why?

Quiz

Given function:

```
void foo(int a, int *b) {
    a= 1;
    *b = 2;
}
```

what are printed after the following fragment:

```
m= 5;
n= 10;
foo(m, &n);
printf("%d and %d\n", m, n);
```

A) 5 and 10	B) 1 and 2	C) 5 and 2	D) 1 and 10

Discuss: Top-Down Design with Exercise 6.9

6.9: *You'd better only read this description instead of the whole Exercises 6.09 and 3.06.*

The Task: Suppose that you're working at a shop's counter and need to return some money which is less than $10 to a customer, using only Australian coins.

Write a program that reads an integer amount of cents between 0 and 999 then prints out the coin changes, using coins from $2 to 5c.

As a programmer, you are required to have good function decomposition.

Note: valid coins are 200c, 100c, 50c, 20c, 10c, and 5c

Example of running the program:

```
./program
Enter amount in cents: 122
give 1 100-cent coins
give 1  20-cent coins
./program
Enter amount in cents: 533
give 2 200-cent coins
give 1 100-cent coins
give 1  20-cent coins
give 1  10-cent coins
give 1   5-cent coins
```

: Top-Down Design with Exercise 6.9

The Task: Suppose that you're working at a shop's counter and need to return some money which is less than $10 to a customer, using only Australian coins.
Write a program that reads an integer amount of cents between 0 and 999 then prints out the coin changes, using coins from $2 to 5c. As a programmer, you are required to have good function decomposition.
Note: valid coins are 200c, 100c, 50c, 20c, 10c, and 5c

Example of running the program:

```
./program
Enter amount in cents: 122
give 1 100-cent coins
give 1  20-cent coins
```

Class Exercises: writing functions [preferably in groups on papers]

5.06

Two numbers are an *amicable pair* if their factors (excluding themselves) add up to each other.
Write a function that takes two int arguments and returns true if they are an amicable pair.

5.03 & 5.13

notes: supposing no overflow

5.3: Write an int function `int_pow` that raises its first argument to the power of its second argument, where the first argument is an int, and the second argument is a positive int. For example, `int_pow(3, 4)` should return the value `81`.

5.13: Ditto, but write a recursive function.

5.05a

A number is *perfect* if it is equal to the sum of its factors (including one, but not including itself). (eg. 6, as 1+2+3=6)

Write a function that returns true if its argument n is a perfect number, and false otherwise.

Lab Time: Do as many as possible

Simplest:
- Ex5.01: Largest of two integers
- Ex5.02: Largest of four integers
- Ex5.x1: Median of three ints

Basic:
- **Ex5.03**: Integer power
- Ex5.04: Combinations
- **Ex5.05**: Perfect Numbers

- Ex5.06: Amicable pairs
- Ex5.06b: Searching for amicable pairs
- Ex5.07: isupper() and tolower()

Recursion:
- **Ex5.13**: Integer power with recursion
- Ex5.14: logstar, recursively
- Ex5.15: Mutually recursive

Others:
- Ex5.08: Near-equality for doubles
- Simpler near-equality for doubles
- Ex5.08b: cube_root near-equality
- Ex5.11: Sum a real-valued sequence

Then: all exercises in chapter 6 😃

Wrap-up

Good function decomposition is essential!

- Functions make your code **modular, reusable, and readable**.
- They help avoid **repetition** and make code easier to **debug** and **maintain**.
- Always think about which parts of your code could be encapsulated into a function.

In C, parameters are passed by value

- This means a function receives a **copy** of the argument, so it cannot change the original variable.
- To modify the original variable, you must pass its **memory address** using a **pointer**. The function then uses this address to directly access and change the variable's value.

Additional Slides

Recursive functions

Recursive function: function that calls itself

```
int factorial( int n ) {
  if (n) {
    return 1;
  }
  return n*factotial(n-1);
}
```

Recursive functions: How

- general case: reduce the task of size n to the same tasks of smaller sizes
- clearly describe the base cases where the solutions are trivial
- when writing code, start with *solving the base cases first*

Example1: factorial (n)

- general case: factorial(n) can be computed by using factorial(n-1)
- base case: when n==0 the solution is trivial
- writing code:

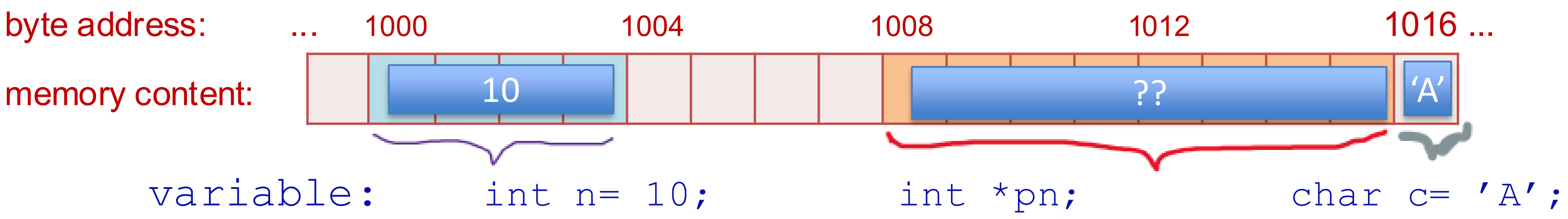
```
int factorial( int n ) {
  if (n==0) { // base case
    ???;
  }
  // general case    [note: else is not needed]
  ???
}
```

Example 2: **5.13**: int_pow or HanoiTower

- the task
- what's: relationship between the input size a smaller size?
- what's: the base case?

Pointers: variables and addresses

- Each variable `x` has a memory cell.
- The memory cell has an address `&x`, which is address of the variable.
- But variable has type, and so address is also bound to that type.
- The address of a cell is called the cell's reference, or a pointer to the cell.



Operator: & = reference

&n is 1000

Operator: * = dereference

`pn= &n;`

`*pn` now is another name for `n`