COMP2113 Programming Technologies / ENGG1340 Computer Programming II

Module 9. C programming (Part 2) – C basics

Objectives

At the end of this self-learning lab, you should be able to:

- Know how to use variables, flow of control and array in a C program.
- Know how to use function (pass by value / **pass by reference**) in a C program.
- Know some simple **C-string** functions for manipulating `char` arrays.

Section 1. Variables and flow of control

- **Variable -** Both C and C++ support the same data types like `int`, `double` and `char`.
- **Flow of control -** The syntax for control statements are the same. That is, `if-else` statements, `for`-loops and `while`-loops are defined identically.

Some important technical notes:

- **In old version of C (ANSI C89)** you are limited to declaring variables just after an opening brace (i.e., `{`).
- Let's browse to `~/module9/`

```
$ cd module9
```

- Consider the file `variable.c`

```
$ gedit variable.c & / vi variable.c
```

```
#include <stdio.h>
int main(){
    int a = 0;
    for ( int i = 0 ; i < 10 ; i++ )
        a += 1;
    printf ( "%d", a );
    return 0;
}
```

- Note that we define the variable `int i` in the `for` loop initialization part, which is a valid syntax in C++.
- Unfortunately, it is not a valid syntax if we use `gcc` to compile `variable.c`. It is because `gcc` by default is not using the c99 standard.

```
$ gcc variable.c -o variable
Error 'for' loop initial declaration used outside C99 mode
...
```

- **From C99 onwards (and C++)** the standard also allows variables to be declared inside `for` loops
- We need to add the flag `-std=c99` to ask the `gcc` compiler to use the c99 standard.

```
$ gcc -std=c99 variable.c -o variable
(No error messages, compile success)
$ ./variable
45
```

Section 2. Function (pass by value)

- **Function -** The syntax for functions (**pass by value, or called pass by copy**) is the same as the syntax in C++.
- Consider the file `function1.c`

```
$ gedit function1.c & / vi function1.c
```

```
#include <stdio.h>
double sum( double a, double b){
    return a + b;
}
int main(){
    double a = 12, b = 14.5;
    printf ( "The sum is %g \n", sum(a , b) );
    printf ( "The sum is %g \n", sum(5.5 , 4.5) );
    return 0;
}
```

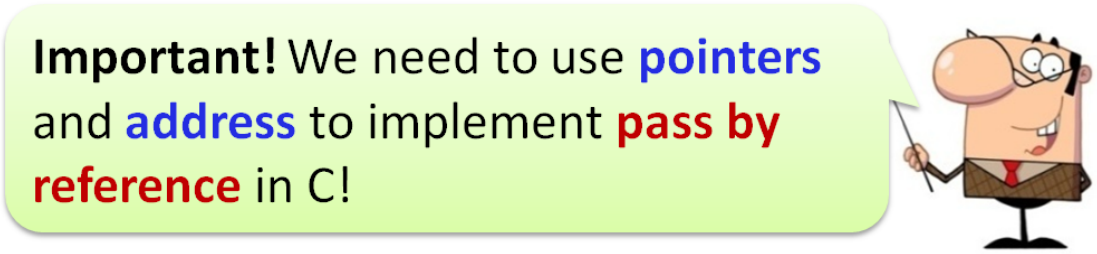
Code explanations:

1. In this example we define a function called `sum()` with two input parameters `double a` and `double b`.
   - Note that both input parameters are **pass by value, or we call it pass by copy**.
   - It is okay in this case because the `sum()` function does not update the variables `a` and `b` in the `main()` function, it just returns the sum, which is a `double` value .
   - Note that when `sum()` is called, the value of `a` and `b` are **copied** to input parameters `a` and `b` in `sum()`, i.e., `sum()` works on a copy of the values.
2. We use the `%g` conversion specifier in the `printf()` function to output the `double` value without trailing zeros on screen.

- Let's try to compile the program and run it.

```
$ gcc function1.c –o function1
$ ./function1
The sum is 26.5
The sum is 10
```

Section 3. Function (pass by reference)



- As a revision, let's look at the two approaches for pass by reference in C++:
- Consider the file `function2.cpp`. There are three `swap` functions.

```
$ gedit function2.cpp & / vi function2.cpp
```

- **Pass by value** `swap1(double a, double b)`

```
void swap1(double a, double b){
    double temp = a;
    a=b;
    b=temp;
}
```

  - Note that the `swap1()` function failed to swap the values of the variables `a` and `b` in the `main()` function because they are **pass by value** (or called pass by copy).
  - In `swap1()` it is just swapping a copy of the value of the variables `a` and `b` in the `main()` function, therefore `swap1()` cannot swap the variables that we pass in.

- **Pass by reference method 1** - `void swap2(double & a, double & b)`

```
void swap2(double & a, double & b){
    double temp = a;
    a=b;
    b=temp;
}
```

  - Note that this `swap2()` function can swap the values of the variables `a` and `b` in the `main()` function because they are **pass by reference**.

- However, this way of passing by reference is supported in C++ only! **C doesn't support this method to specify the pass by reference** ☹

- **Pass by reference method 2 -** `void swap3(double *a, double *b)`

```
void swap3(double *a, double *b){
    double temp = *a;
    *a=*b;
    *b=temp;
}
```

  - Note that this `swap3()` function can swap the values of the variables `a` and `b` in the `main()` function.
  - It is because in the `main()` function we **pass the address** of the variables `a` and `b` into the `swap3()` function.
  - Inside the `swap3()` function we **dereference the addresses** using the "`*`" operator so that `swap3()` is accessing and altering the values of the variables `a` and `b` in the `main()` function.
  - Note that we need to pass in the **address** of the variables to call `swap3()` function. Therefore we will have the "`&`" operator before the variables in the function call.

```
swap3(&a , &b);
```

- Let's try to compile the program and run it, this C++ code works.

```
$ g++ function2.cpp -o function2_cpp
$ ./function2_cpp
a = 5.5, b = 10.5
a = 5.5, b = 10.5
a = 10.5, b = 5.5
a = 5.5, b = 10.5
```

In C programming, only **swap3()** is a valid way to pass by reference, the technique used in **swap2()** is not supported.

- Let's consider `function2.c` as an illustration.

```
$ gedit function2.c & / vi function2.c
```

- Now we have `swap1()`, `swap2()` and `swap3()` functions in the C program `function2.c`. Try to compile the source code, can the compiler understand `swap2()`?

```
$ gcc function2.c –o function2
Error messages returned, C doesn't understand swap2()...
```

- Let's try to fix `function2.c` by commenting out the definition and function call of `swap2()`.

```
$ gcc function2.c –o function2
$ ./function2
a = 5.5, b = 10.5
a = 5.5, b = 10.5
a = 10.5, b = 5.5
```

Checkpoint 9.2a (Please submit your answer to Moodle.)

Let's write a function that computes the sine and cosine values of a user input degree value.

- Consider the file `sincos.c`

```
$ gedit sincos.c & / vi sincos.c
```

- `sincos.c` contains the following unfinished code

```
#include <stdio.h>
#define PI 3.14159265

// Task 2. Build the GetSinCos() function

int main(){

    double dSin;
    double dCos;
    int degree;

    // Task 1. Read in user input to variable degree

    // Task 3. Call the GetSinCos() function

    printf( "The sin is %g \n",dSin );
    printf( "The cos is %g \n",dCos );

    return 0;
}
```

Note: You need to pay attention to the use of the **address of** operator "**&**" when writing the C program.

- **Task 1. Read in user input to variable `degree`.**

```
scanf("%d", ??? );
```

- **Task 2. Build the `GetSinCos()` function.**
  - First we need to decide what is the input parameter of the function.

```
void GetSinCos( ??? , ??? , ??? ){
      // Compute sin and cos here...
}
```

- The first parameter - a simple `int` variable for the degree, let's call it `d` in the function. And we **pass this variable by value**.
- The second parameter - a `double` variable for storing the sine value of `d`, let's call it `dSin`, we need to **pass this variable by reference**.
- The third parameter - a `double` variable for storing the cosine value of `d`, let's call it `dCos`, we need to **pass this variable by reference**.

o We need to include the `<math.h>` library to use the buildin sine and cosine functions (`sin()` and `cos()`)

```
#include <math.h>    // for sin() and cos()
```

o With the `<math.h>` library, we can use the `sin()` and `cos()` function. However, both functions accept radian (But not degree) as their input. Given the degree in variable `d`, the following function call return the sine value of `d`.

```
??? = sin(d*PI/180);
```

- Note that `d*PI/180` changes the degree value in `d` into the corresponding radian value.
- You need to assign the computed sin value to `dSin`, note that `dSin` is pass by reference and it should be a **pointer to `double`** in the `GetSinCos()` function. Therefore we need to **dereference the pointer `dSin`** to access the double value.

o The following function call return the cosine value of `d`.

```
??? = cos(d*PI/180);
```

- You need to assign the computed cosin value to `dCos`, note that `dCos` is pass by reference and it should be a **pointer to `double`** in the `GetSinCos()` function.

- **Task 3. Call the `GetSinCos()` function.**
  - o The first parameter is pass by value, we need to provide `degree` as the first parameter.
  - o The 2nd and 3rd parameters are pass by reference, we need to provide the **addresses of `dSin` and `dCos`** to the function.
- Let's try to compile the program and run it. Note that we have to add the flag `-lm` to link the executable with the math library (even if we have included `math.h` header)

```
$ gcc sincos.c -o sincos -lm
$ ./sincos
30
The sin is 0.5
The cos is 0.866025
```

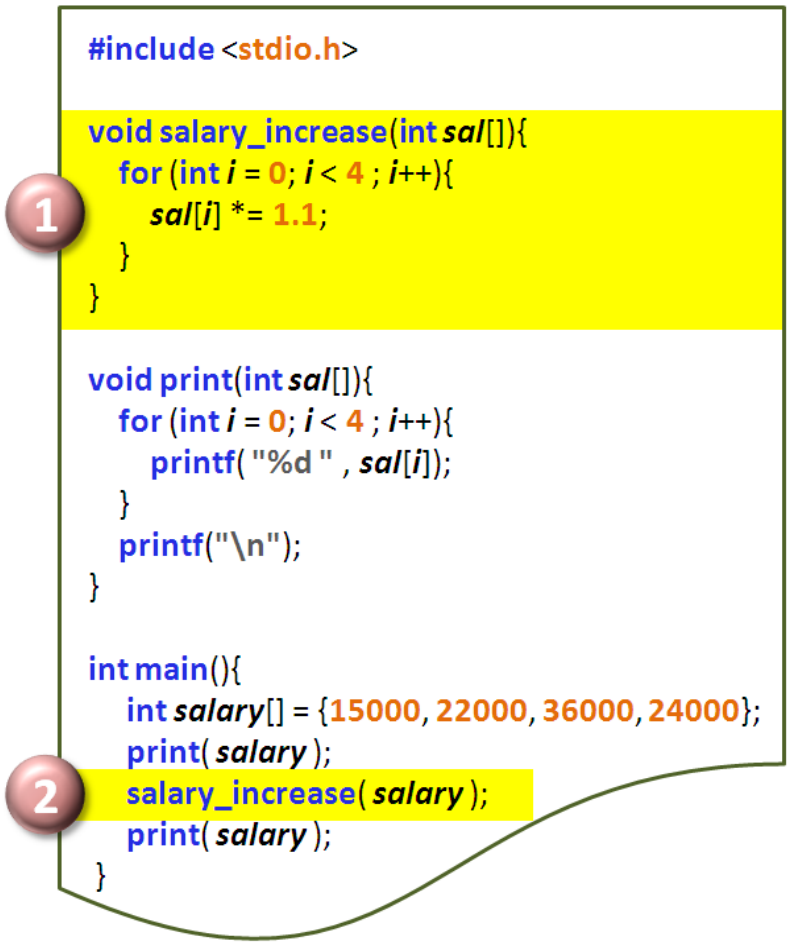
Please submit the **sincos.c** source file to Moodle.

Section 4. Array

- The syntax for `arrays` is the same as C++.
- Consider the file `array1.c`

```
$ gedit array1.c & / vi array1.c
```

Code explanations:



1. Define a function `salary_increase()` with an integer array as input parameter.
   - Same as C++, **passing an array into function behaves like pass by reference.** i.e., If you change the value of the array inside the function, the input array in the function caller also changes.
     - In this case, the `int` array `salary` in the `main()` function is updated after calling `salary_increase()`.
2. Calling a function and pass an array variable into the function is the same as C++. Note that we do not need to have the "address of" operator (i.e., `salary_increase(&salary)` is wrong.)

- Let's try to compile the program and run it.
- Note that as we define the variable `int i` in the `for` loop, we need to compile the C program using c99 standard.

```
$ gcc -std=c99 array1.c -o array1
$ ./array1
15000 22000 36000 24000
16500 24200 39600 26400
```

The relationship between array and pointer

Same as C++, when a **pointer** is referring to an **array**, **it is storing the address of the 1st slot of the array.** ***a*[*i*]** is in fact the short form of ***(a+i)**

- The following function is equivalent to the `salary_increase()` in `array1.c`.

```
void salary_increase(int *sal){
    for (int i = 0; i < 4 ; i++){
        (*sal) = (*sal) * 1.1;
        sal++;
    }
}
```

Code explanations:

- Input parameter `int *sal` is equivalent to `int sal[]`. (Same as C++)
    - It is because when we pass an array into a function, it is actually the address of the 1st slot of the array that is passed in.
    - This also explains why passing an array into a function behaves like pass by reference; it is because the array is NOT copied to the workspace of the function. Instead it is the pointer to the first slot of the array that is passed into the function.
- `(*sal)`
    - We use the "`*`" operator to dereference the pointer variable `sal`.
    - Note that `sal` is an `int` type pointer variable, which stores the address of a memory cell that stores an integer value.
    - `(*sal)` means that we are refereeing to **the memory cell** with the address equal to the address stored in `sal` (but not just the address value stored in `sal`).
- `sal++`
    - We can use the increment / decrement operator on pointer variable to go to the next / previous slot of the array.
- Let's try to replace the `salary_increase()` function in `array1.c` by the above function and recompile the code, you will notice that the two functions are equivalent to each other.

Section 5. C-String

- C does not provide the string class.
- Instead, a string in C is simply **an array of `char`**.
- A null character '`\0`' is used to indicate the end of the string.
- For example, if we define a `char` array as follows.

```
char line[15] = "Hello world";
```

- The content of `line` would be as follows.

char *line*[*i*]	H	e	l	l	o		w	o	r	l	d	\0	?	?	?
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- Note that the **null character** is inserted by the compiler automatically in position 11.
- The characters stored in position 12 to 14 are not part of the string and ignored during string operations like copying (`strcpy()`) and comparison (`strcmp()`).
- Assume we want to change the string to "`Hello world!`" we can do it as follows.

```
line[11] = '!';
line[12] = '\0';
```

- It is important to add the null character at position 12.
  - Otherwise, the compiler will consider all character to be part of the string up to the first null character is found.
  - Such a null character may occur far after the end of the array, leading to different types of error like **array access out of bound error**.

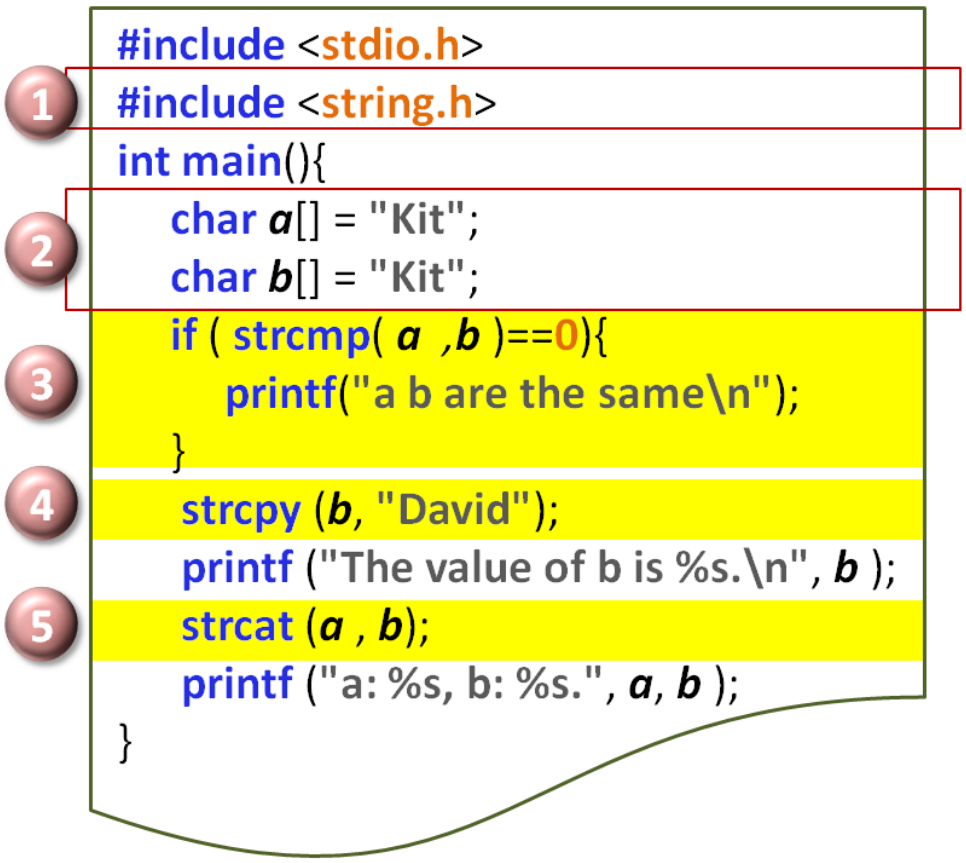
Question:
If there is no **string** class in C, how can we perform **string operations** ☹?

Answer: In C, a number of functions are provided to manipulate the strings.
We need to **#include**< **string.h** > in order to use these functions.

Function	Effect
`strcpy (char s1[], char s2[])`	Copy `char` array **s2** to **s1**. **s2** is not changed.
`strcat (char s1[], char s2[])`	Append **s2** after the end of **s1**. Note that the first character of **s2** will overwrite the null character of **s1**. **s2** is not changed.
`strcmp (char s1[], char s2[])`	Return negative if **s1<s2**, return 0 if **s1==s2**, return positive if **s1>s2**.
`strlen (char s1[])`	Return the length of the string stored in the char array **s1**.

- Consider the file `string1.c` as an illustration.

```
$ gedit string1.c & / vi string1.c
```



Code explanations:

1. We need to include `<string.h>` header file to use the string functions.
2. Create two `char` arrays `a` and `b`, both store the string value "`Kit`".
3. Use `strcmp()` to compare if two strings are the same. Note that the function returns 0 if the two strings are identical.
4. Use `strcpy()` to copy a string value into a `char` array.
   - In this case we assign "`David`" into the content of char array `b`. (**However there are some problems in this line of code, to be explained shortly.**)
5. Use `strcat()` to concatenate two strings, the content of `a` and `b` are concatenated and the result is stored in the first parameter (i.e., the `char` array `a`.)

- Let's try to compile the program and run it. **There could be (not always) some run-time problem in the code.** Can you identify the problem?

```
$ gcc string1.c -o string1
$ ./string1
a b are the same
The value of b is David.
a: KitDavid, b: David.
```

Seems that there is something wrong with **strcpy**() and **strcat**() in **string1.c**. What's wrong in the code?

- When using `strcpy()` or `strcat()`, the programmer needs to ensure that the first input parameter has enough space to hold the final content.
- Otherwise, a runtime error may occur due to **array access out of bound**.
- What we need to do is to provide the initial size of the `char` arrays `a` and `b` to be large enough, say, `100 char` slots.
    - Then `strcpy(b, "David")` will have enough space to store "`David`" (at least 6 `chars`) in `b`.

char *b*[]

D	a	v	i	d	\0	…
0	1	2	3	4	5	…

    - Then `strcat(a,b)` will have enough space to store "`KitDavid`" (at least 9 `chars`) in `a`.

char *a*[]

K	i	t	D	a	v	i	d	\0	…
0	1	2	3	4	5	6	7	8	…

Checkpoint 9.2b (Please submit your answer to Moodle.)

Write a program that reads in a string input by users, and changes all the characters from Upper cases to Lower cases.

- Consider the file `string2.c`

```
$ gedit string2.c & / vi string2.c
```

- `string2.c` contains the following unfinished code

```
#include<stdio.h>
#include<string.h>

// Task2. Build the toLower() function here.
void toLower(char a[]){
    // To be implemented by you.

}

int main(){
    char input[100];
    // Task 1. Read in user input to the char array input.

    // Task 3. Call the toLower function.

    printf("%s",input);
}
```

- **Task1. Read in user input to the `char` array `input`.**

```
scanf("%s", ??? );
```

  - **Important note:** usually we need to pass in the address of a variable into a `scanf()` function because it is pass by reference.
  - However note that `input` is a `char` array. That is to say, `input` itself is already a pointer storing the address of the first slot of the char array. Thus passing an array into a function does not need to use the address-of operator "`&`", it is by default pass by reference.
  - Therefore **we only need to pass in `input` (but not `&input`) if we are reading values into `char` array using a `scanf()` function**.

- **Task2. Build the `toLower()` function.**
  - The logic is to use a `for` loop to scan through the char array. What is the string function to get the length of the char array? (Hints: look at the last row in the table in page 11.)

```
for (int i = 0 ; i< ??? ; i++){
    // Check individual char a[i] here ...

}
```

  - The logic of comparing whether a character `a[i]` is a capital letter is

```
    if(a[i] >= 'A' && a[i]<='Z'){
        // Update a[i] to lower case
    }
```

  - Changing the letter from Upper case to Lower case is left as your own exercise.
    - **Reminder:** From level 1 programming course we learn that `a[i]+'a'-'A'` will change the letter from Upper case to Lower case, do you still remember it? ☺

```
        a[i] = a[i] + 'a' - 'A'
```

- **Task3. Call the `toLower()` function.**
  - Note that passing a `char` array to a function does not require the address-of operator "`&`" because `input` itself is already a pointer pointing to the first slot of the array.

```
toLower(input);
```

- Let's try to compile the program and run it.

```
$ gcc -std=c99 string2.c -o string2
$ ./string2
ABCDEFG
abcdefg
```

Please submit the **string2.c** source file to Moodle.

References

- The C string library `<string.h>`
  http://www.cplusplus.com/reference/cstring/
- C Tutorial – Strings and Text Handling
  http://cplus.about.com/od/learningc/ss/strings.htm
- Cprogramming.com – Lesson 9 : C-string
  http://www.cprogramming.com/tutorial/c/lesson9.html