Computer programming (STQS1313)
Sem 2, Session 2020/2021
Notes - Functions/modules (building blocks)

1) Introduction to function
   - C++ program is a collection of functions/modules/building blocks.
   - The programs that you have written packed all programming instructions into one function.
   - This technique is good only for short programs.
   - It is not practical for long codes → for large programs, we need to break the problem into manageable pieces.
   - Eg: a car,
     - each major car component can be compared to a function.
     - the engine, transmission, and other modules know only their inputs and outputs
     - the driver doesn't need to know the internal operation of the modules

2) Why do we use functions?
   - can focus on a specific function at one time (to construct/write, to debug, to perfect it)
   - different people can work on different functions simultaneously (but they have to know the details (type, input, output etc) of their functions)
   - we only need to write the function once in the code although it is going to be used many times (you are going to see this once you learn arrays/vectors/matrices).
   - Enhance a program's readability, because it reduces the complexity of the main function.

3) Types of function
   - predefined functions: need libraries such as the cmath
   - user-defined functions (**value-returning functions** and **void functions**): because C++ doesn't know every possible functions that you need → need to write your own functions.

4) How to use a function
   - three components: **function declaration** (prototype), **function definition** (function header and body), **function call**

   - **<u>Writing function declaration (prototype)</u>**
     - Can be placed within the main function, or before or after the #include<>. Usually we put it after the include<>.
     - gives information on function's/output's type, function's name, parameters/arguments' names and types, number of parameters. Don't forget the semi colon.
     - Syntax: `returnDataType functionName(list or argument data types);`
     - Example:
       - `int addfun(int x, int y);`
       - `double multfun(double x, double y);`
       - `void display(double a, double b);`
       - `void printline();` ← **function with empty parameter list**

   - **<u>Writing function definition</u>**
     - Can be placed before or after the main function. Usually after the main function.
     - Two main parts: *function header* and *function body*

- The first line is the function header. It is similar to the function declaration/prototype except that it has no semi colon.
- The variables are called **formal parameters.**

- **Writing function call**
  - needs to be placed in the main function
  - Syntax: `functionName(list or actual parameters);`
  - Note, now it is the **actual parameters**.

5) Example of full function code to calculate the **sum** of two numbers:

```
#include<iostream>
using namespace std;
int addfun(int x, int y);     // (1) Function Prototype/Declaration
int main()
{
      int a, b, c;
      a = 1;
      b = 2;
      c = addfun(a, b);       // (3) Function call
      cout << "The sum of the two numbers is: " << c << endl;
      return 0;
}
// (2) Function definition
int addfun(int x, int y)      // Function header
{
      // Function body
      int z;
      z = x + y;
      return z;
}
```

Note: the value of the **actual parameters** *a* and *b* (1 and 2) are passed to the **formal parameters**: *x* and *y* (this is called ***pass by value***).

6) Example of full function code to find the **maximum** of two numbers:

```
#include<iostream>
using namespace std;
int findMax(int, int);        // (1) Function Prototype/Declaration
int main()
{
      int x, y, z;
      cout << "Please enter first number: ";
      cin  >> x;
      cout << "\nPlease enter second number: ";
      cin  >> y;
      z = findMax(x,y);       // (3) Function call
      cout << "\nThe larger number is " << z << endl;
      return 0;
}
// (2) Function definition
int findMax(int a, int b)     // Function header
{
      // Function body
      int c;
      if (a>b)
            c = a;
      else
            c = b;
      return c;
}
```

Note: the value of the **actual parameters** *x* and *y* are passed to the **formal parameters**: *a* and *b* (***pass by value***).

7) Functions usually return a single value only → returning multiple values will be covered in the ***pass by reference***.

8) What can be done to the output of a function?
   - save the value (as shown above): `c = addfun(a, b);`. We have seen this in our previous programs.
   - print the value: `cout << "The sum of the two numbers is " << addfun(a, b);`. For example:

```
#include<iostream>
using namespace std;
int addfun(int x, int y);
int main()
{
      int a, b;
      a = 1;
      b = 2;
      cout << "The sum of the two numbers is: " << addfun(a,b) << endl;
      return 0;
}

int addfun(int x, int y)
{
      int z;
      z = x + y;
      return z;
}
```

where we have removed `c` in the main function since it is not needed.

   - use the value in some calculation: `c = 2*addfun(a, b);`. For example:

```
#include<iostream>
using namespace std;
int addfun(int x, int y);
int main()
{
      int a, b, c;
      a = 1;
      b = 2;
      c = 2*addfun(a,b);
      cout << "The sum of the two numbers when doubled is: " << c << endl;
      return 0;
}

int addfun(int x, int y)
{
      int z;
      z = x + y;
      return z;
}
```

9) Two types of user-defined functions
   - **value-returning functions**. We have seen this in all previous programs where each called function returns a value to the main function.
   - **void functions**. Here, the called function will not return any value to the main function.
   - Example (Code 16)

```
#include<iostream>
using namespace std;
void addFun(int, int);     // (1) Function Prototype/Declaration
int main()
{
      int a, b;
      a = 1;
      b = 2;
      addFun(a,b);         // (3) Function call
      return 0;
}

// (2) Function definition
void addFun(int x, int y)      // Function header
{
      // Function body
      int z;
      z = x + y;
      cout << "\nThe sum of the two numbers is " << z << endl;
}
```

   - ➢ Note that the function call does not have a variable `c` as a placeholder to hold the returned value from the function, since there is no value returned by the function at the first place.
   - ➢ Note also that the command to print the summation is now placed in the void function, not in the main function.

10) Functions with empty parameter lists.
   - Example (Code 17)

```
#include<iostream>
using namespace std;
// void function = no return value, 2) empty parameter list = no input
void printLine();
int main()
{
    printLine();                                          // Function call
    cout << "Fakulti Sains dan Teknologi" << endl;
    printLine();                                          // Function call

    return 0;
}

void printLine()
{
    cout << "==========================" << endl;
}
```

   - Note that the void function with empty parameter list named `printLine` is used twice in the program.

- Example (Code 17)

```
#include<iostream>
using namespace std;
void printLine();
void printLine2();
int main()
{
	printLine();
	cout << "Fakulti Sains dan Teknologi" << endl;
	printLine2();

	return 0;
}

void printLine()
{
	cout << "==========================" << endl;
}

void printLine2()
{
	cout << "**************************" << endl;
}
```

- Note that there are two void functions with empty parameter list named `printLine` and `printLine2` used in the program.

11) Local variable and global variable

- variable defined in a function is local:
  - can be used and changed in that particular function only
  - not accessible to other functions
  - that's why we have separate declaration, and need to return value.
  - Example (see Code 19b)

```
#include<iostream>
using namespace std;
void myFun();                    // void function with empty argument
int main()
{
    int a;
    a = 3;
    cout << "The value of a in main() is " << a << endl;
    myFun();
    cout << "The value of a in main() after changed by myFun() is "
         << a << endl;
    return 0;
}

void myFun()
{
    int a;
    a = 2;
    cout << "The value of a in myFun() is " << a << endl;
}
```

- global variable is defined outside any function → can be used and changed in any function.
- Example of global variable: See Codes 20, 21 and 22.

```
#include<iostream>
using namespace std;
void myFun();                  // void function with no argument
int a=1;                       // global variable
int main()
{
      cout << "The value of a is " << a << endl;
      a = 3;
      cout << "The value of a is " << a << endl;
      myFun();
      cout << "The value of a is " << a << endl;
      return 0;
}

void myFun()
{
      a = 2;
      cout << "The value of a is " << a << endl;
}
```

12) Misuse of global variables
- It's possible to make all variables global.
- But DO NOT DO THIS, because it could be disastrous for large programs, where there are a lot of variables, and user-defined functions → you might not reliase which values are controlled globally.

13) Scope resolution operator
- when the name of a variable is **declared twice**: locally and globally.
- Here, the local variable of a function name takes precedence over the global variable in its function.
- we can still access the global variable by using the ***scope resolution operator* ::**, placed immediately before the variable name.
- Example (Code 23)

```
#include<iostream>
using namespace std;
int a = 1;                    // declared as global variable
int main()
{
      int a = 2;              // declared as local variable
      cout << "The value of a is " << a << endl;
      return 0;
}
```

- Example (Code 24)

```
#include<iostream>
using namespace std;
int a = 1;                  // declared as global variable
int main()
{
      int a = 2;            // declared as local variable
      cout << "The local value of num is " << a << endl;     // local
      cout << "The global value of num is " << ::a << endl; // global
      return 0;
}
```

14) Stub function
- a fake/dummy function
- created because you haven't finalised/completed writing your function (definition)
- used as placeholder for the final function until it's completed
- minimum requirement: a stub function can be compiled and linked to the calling module/code/function.
- Example: Code 25

15) Function overloading and function templates: please read.

16) Returning a single value
- Typical function: the called function receives values from its calling function, and returns at most one value (of course the function will do some manipulation on the values before returning it).
- This is called **pass by value**.

17) Returning multiple values
- can be done by passing the *variable's address* in the calling function to the called functions.
- This will allow the called function to use and change the value of variables defined in the calling function.
- Passing addresses is referred to as **pass by reference**.
- Related topic: pointer.

18) Pass by reference
- method: call a function, and pass an address of a variable.
- How to pass: use **& operator** at function prototype and function header.
- &: "**the address of**"
- Take a look at Code27:

```
#include<iostream>
using namespace std;
void newval(double&,double&);  // (1) Function declaration

int main()
{
    double x, y;
    cout << "Please enter two numbers: ";
    cin  >> x >> y;

    cout << "The value in x is: " << x << endl
         << "The value in y is: " << y << endl;

    newval(x,y);               // (3) Function call

    cout << endl;
    cout << "The value in x is: " << x << endl
         << "The value in y is: " << y << endl;

    return 0;
}

void newval(double& a,double& b)  // (2) Function definition
{
    a = 2; // x
    b = 1; // y
}
```

- **Function header**

```
void newval(double& a, double& b)
```

- "`a` is a **reference parameter** used to **store the address** of a double-precision value", and similarly "`b` is a reference parameter used to store the address of a double-precision value";

- **Function call**

```
newval(x,y)
```

- **connects** the **arguments** used in the function call of the `main` function, `x` and `y`, and the **parameters** used in the header of the `newval` function, `a` and `b`.
- The values in the arguments `x` and `y` can now be **altered from within** by using the reference parameters `a` and `b`.
- The parameters `a` and `b` **don't store copies of the values** in `x` and `y`; instead, they access the locations in memory set aside for these two arguments.

- The value of more than one variable is affected, so the function can't be written as a pass by value function (that only returns a single value).
- Take a look at another example (Code 28):

```
#include<iostream>
using namespace std;
void calc(double, double, double&, double&);   // (1) Function declaration
int main()
{
      double x, y, sum, prod;
      cout << "Enter two numbers: ";
      cin   >> x >> y;
      calc(x, y, sum, prod);                    // (3) Function call

      cout << "\nThe sum of the numbers is: " << sum << endl;
      cout << "The product of the numbers is: " << prod << endl;

      return 0;
}

// (2) Function definition
void calc(double a, double b, double& m, double& n)
{
      m = a + b;
      n  = a*b;
}
```

- In `main()`, the `calc()` function is called with four arguments: `x`, `y`, `sum`, and `prod`. As required, these arguments agree in number and data type with the parameters declared by `calc()`. Of the four arguments passed, only `x` and `y` have been assigned values when the call to `calc()` is made. The remaining two arguments, `sum`, and `prod`, haven't been initialized and are used to receive values back from `calc()`.
- Depending on the compiler used, these arguments initially contain zeros or "garbage" values.

Exercises

1) Write a function that returns the smaller value between `x`, `y`

2) Modify question 2 so that the value of `x` and `y` are entered when the program is running.

3) Modify question 2 so that we can repeat it for `n` times (determined by user, for example, `n=3`).

4) Modify question 2 to determine the smallest between 3 values.

5) Write parameter declarations for the following
   a) A parameter named `amount` that will be a reference to a double-precision value.
   b) A parameter named `price` that will be a reference to a double-precision number.
   c) A parameter named `minutes` that will be a reference to an integer number.
   d) A parameter named `key` that will be a reference to a character.
   e) A parameter named `yield` that will be a reference to a double-precision number.

6) Using reference parameters, write a C++ program that contains a function named `time()` to convert the passed number of seconds into an equivalent number of minutes and seconds.