Module 5 Guidance Notes

Functions

ENGG1340

Computer Programming II

Before We Start

- We will still deal with C++ only in this module.
- Important: We will be using the C++ 11 standard, so make sure that your compiler option is set appropriately. We suggest to use the following command to compile your C++ program:

```
g++ -pedantic-errors -std=c++11 your_program.cpp
```

The -pedantic-errors flag is to make sure that your code conforms to the ISO C/C++ standard. We will enforce this in your assignment submission too.
For more information about C/C++ standards, you may read https://en.wikipedia.org/wiki/ANSI_C and https://isocpp.org/std/the-standard

How to Use this Guidance Notes

- This guidance notes aim to lead you through the learning of the C/C++ materials. It also defines the scope of this course, i.e., what we expect you should know for the purpose of this course. (and which should not limit what you should know about C/C++ programming.)
- Pages marked with "Reference Only" means that they are not in the scope of assessment for this course.
- The corresponding textbook chapters that we expect you to read will also be given. The textbook may contain more details and information than we have here in this notes, and these extra textbook materials are considered references only.

How to Use this Guidance Notes

- We suggest you to copy the code segments in this notes to the coding environment and try run the program yourself.
- Also, try make change to the code, then observe the output and deduce the behavior of the code. This way of playing around with the code can help give you a better understanding of the programming language.

What are we going to learn?

- Top-down design (divide and conquer) approach
- Functions definition
- Function call
- Function declaration
- Scope of Variables
- Parameters passing mechanism
  - Pass-by-value
  - Pass-by-reference

References

- cplusplus.com tutorial
  - Functions
  - Variable Scope
- Textbook Chapters
  - C++: How to program (9th edition)
    Electronic version available from HKU library
  - Ch. 6.1 – 18

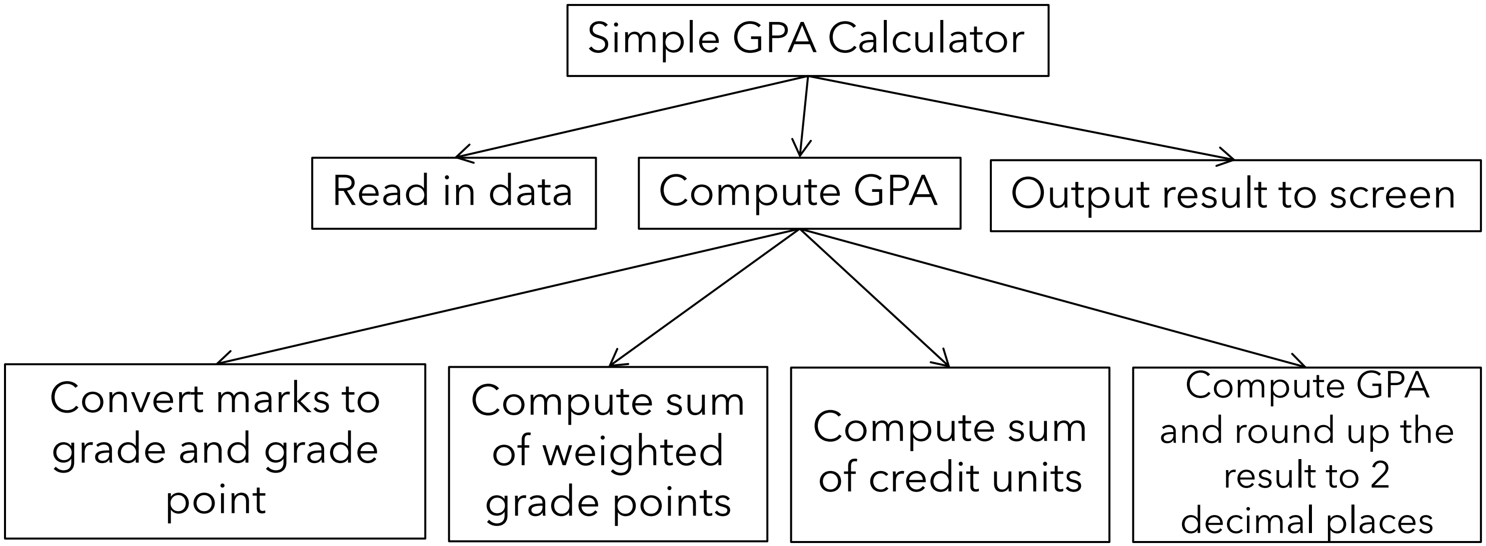
TOP-DOWN DESIGN (DIVIDE AND CONQUER) APPROACH

Top-Down Program Design

- A good way to design a program is to break down the task to be accomplished into a few sub-tasks
- Each sub-task can be further decomposed into smaller sub-tasks, and this process is repeated until all sub-tasks are small enough that their implementations become manageable
- This approach is called top-down design (a.k.a. divide and conquer)

Top-Down Design

- Example: Compute the final score for a student



Each module should perform a single, well-defined task

Functions

- Preserving the top-down design structure in a program will make it easier to understand and modify the program, as well as to write, test, and debug the program
- In C++, sub-tasks are implemented as functions
  - A function is a group of statements that is executed when it is called from some point of the program
  - E.g., the main function `main()` in previous examples
- A program is composed of a collection of functions
- When a program is put into execution, **it always starts at the main function**, which may in turn call other functions

Advantages of using Functions

- May focus on a particular task, easy to construct and debug
- Different people can work on different functions simultaneously
- A function is written once and can be reused multiple times in a program or in different programs
- Improve readability of a program by reducing the complexity of `main()`

Predefined Functions

- Some computations and operations are so common that they are implemented as pre-defined functions that are shared for use
- Consider computing the square root of a number. It would be nice if we have a black box function (i.e., we don't care how the computation is done) to help us do the calculation.

e.g.

```
double x = sqrt(5.29);
```

Here, 5.29 is the function input and the function output 2.3 would be stored to x

- What we need to know is what is required for the computation (i.e., function input) and what is the result of the computation (i.e., function output)

Predefined Functions

- C++ comes with libraries of pre-defined functions that programmers can use in their programs
  - The function definitions (i.e., codes for a function doing the actual computations) are stored in separate files and have been **pre-compiled** into object codes for further linking
  - The function declarations (i.e., what a function accepts as input and returns as output) are stored in files known as the header files
- Hence, to use certain pre-defined functions we will need to include the corresponding header files so the compiler can check if the functions are used correctly.

The math library

- Some commonly used predefined functions

Function	Description	Library Header
`double sqrt(double x)`	Square root of x	`cmath`
`double pow(double x, double y)`	x to the power of y (i.e., x^y)	`cmath`
`double fabs(double x)`	Absolute value of x	`cmath`
`double ceil(double x)`	Round up the value of x	`cmath`
`double floor(double x)`	Round down the value of x	`cmath`
`int abs(int x)`	Absolute value of x	`cstdlib`
`int rand()`	A random integer	`cstdlib`

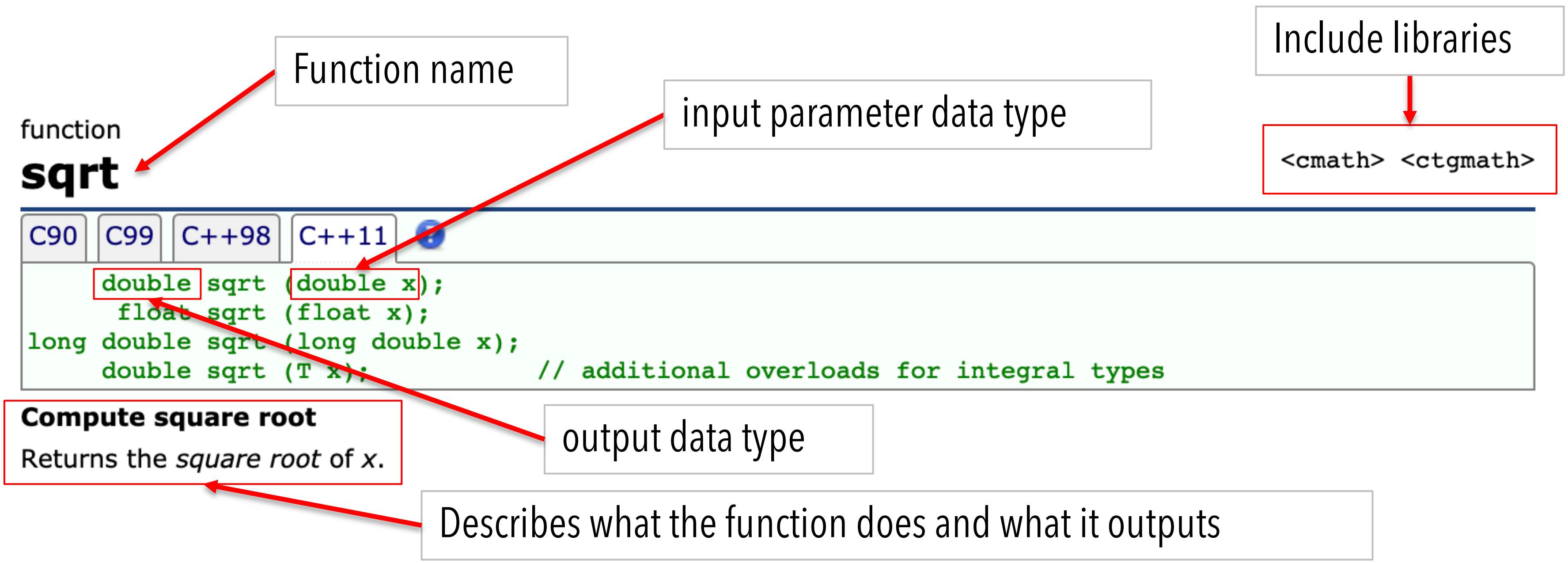
- Always consult the C++ manuals, e.g. www.cplusplus.com, for the details of individual functions.

Using Predefined Functions

- To use a pre-defined function, simply include the corresponding header file using the include directive `#include <…>`
  - e.g., `#include<iostream>` for using `cin`, `cout`, `endl`
  - e.g., `#include<cstdlib>` for using `rand()`, `srand()`
- Take note of the input parameters and the return type for a function and make the function call accordingly.

Example: sqrt()

This is what you can find from the sqrt() function reference from cplusplus.com:



Note that a function may accept different data types as inputs (hence there are 4 different function declarations for sqrt()). This is called function overloading.
In this case, if you provide a double type input to sqrt(), the output returned by the function will also be of a double type.

Check this for some examples in function overloading.

Using Predefined Functions

Example:

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
     // Compute the root mean square of 10 input numbers
     int i;
     double n, sq_sum = 0;

     for(i=0; i<10; i++)
     {
          cout << i+1 << ": ";
          cin >> n;
          sq_sum += pow(n, 2.0);
     }

     cout << "The root mean square is " << sqrt(sq_sum/10) << endl;

     return 0;
}
```

A function may accept one or more input parameters. Check the pow() reference page to see what each parameter mean.

Defining Your Own Functions

Suppose you want to have a function which tells which of the two given floating point numbers is larger.

These are the questions that you should ask (& answer):

Q1. What are the input(s) to the functions? What are their data type?

Two floating-point numbers, data type: double

Q2. What is the output of the function? What is its data type?

One floating-point numbers, data type: double

Q3. What should be done inside the function to make it work?

How do you determine which of the two given numbers are larger?

Defining Your Own Functions

Let's give a name to the function: larger

By answering Q1 & Q2, we can come up with the function header

```
double larger(double x, double y)
```

return data type

input parameters with data type
The two input numbers will be named x and y inside this function

Defining Your Own Functions

To answer Q3, we need the actual computations inside the **function body**:

```
double larger(double x, double y)
{
   double max;
   if (x >= y)
      max = x;
   else
      max = y;

   return max;
}
```

function body embraced by **{}**

function parameters x and y are used in the calculation

return statement

- returns the specified value to the caller
- terminates the execution of the function

max is the return value, and its data type must agree with that specified in the function header (i.e., `double`)

Function Definition

By now, we have completed the function definition for larger().

return type

function name

list of parameters

function header

function body

```
double larger(double x, double y)
{
    double max;
    if (x >= y)
        max = x;
    else
        max = y;
    return max;
}
```

Function Definition

Formally speaking, a function is defined using a function definition which

- Describes how a function computes the value it returns
- Consists of a function header followed by a function body

type of return value

function name

list of parameters (types and names)

function header

function body embraced by **{}**

Syntax

```
type_ret  func_name(type1 par1, type2 par2, ...)
{
      // variable declarations
      ...
      // executable statements
      ...
}
```

Function Call

- How to call (or invoke) a function?
- Think about how you use the pre-specified function sqrt()?
- A function call (i.e., the process of calling a function) is made using the function name with the necessary parameters
  - A function call is itself an expression, and can be put in any places where an expression is expected
  - Example:

```
double z = larger(2.5, 5.0);
```

Return values from `larger()` after function call is assigned to the variable `z`

Function name

Parameters as input to function

Function Call

- Parameters vs. arguments
  - The parameters used in the function definition are called **formal parameters** or simply **parameters**. They are placeholders in the function.

```
double larger(double x, double y)
{
    …
}
```

parameters

  - The actual values passed to a function in a function call are referred to as **actual parameters** or **arguments**. They are the actual values used in the execution of the function to produce the return value.

```
c = larger ( 2.5, 5.0 );
```

arguments

Function Call

- The arguments used in a function call can be constants, variables, expressions, or even function calls, e.g.,

```
double z1 = larger (2.5, 5.0);                 // constants
double z2 = larger (one, two);                 // variables
double z3 = larger (one – 2, two);             // expressions
double z4 = larger (2.5, larger (3, 5.0) );   // a function
```

- In using expressions as arguments, the expressions will be evaluated to produce a value before the function call is made
- Since a function call is also an expression, the mechanism of using function calls as arguments is identical to that of using expressions

Function Declaration

The compiler needs to know about the function prototype (i.e., its name, input parameters, return type) before a function can be used.

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
    ….
    c = larger(a, b);
    ….
}
```

One way to do this is to place the function definition before the function call in the source file

```
#include <iostream>
using namespace std;

double larger(double x, double y);

int main()
{
  ….
  c = larger(a, b);
  ….
}

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}
```

Note the ; here. It is needed since this function declaration is a statement. Compare this with the function header in the example on the left.

Alternatively, the function definition can be placed anywhere in the source file by including a function declaration before the function call

Function Declaration

- A function declaration is similar to a function header except that it must be followed by a semicolon; and the identifiers in the parameter list can be changed or even omitted. It provides all the information needed in making a function call.

Syntax

```
    type_ret  func_name(type1 par1, type2 par2, ...);
or
    type_ret  func_name(type1, type2, ...);
```

Function Declaration

Examples:

```
#include <iostream>
using namespace std;

double larger(double p, double q) ;

int main()
{
  ….
  c= larger(a, b);
  ….
}

double larger(double x, double y)
{
  return (x >= y)? x : y;
}
```

```
#include <iostream>
using namespace std;

double larger(double, double) ;

int main()
{
  ….
  c = larger(a, b);
  ….
}

double larger(double x, double y)
{
  return (x >= y)? x : y;
}
```

Function Call - Flow of Control

- When a program is put into execution
  - It always starts at the main function no matter where its definition is in the source file
  - The statements in the main function are executed sequentially from top to bottom and the control is passed from one statement to another
  - When a function call is encountered, the execution of the current function is suspended
    - The values of the arguments are copied to the formal parameters of the called function, and the control is passed to the called function
    - Likewise, the statements in the called function are executed from top to bottom, and the control is passed from one statement to another
    - When a return statement is encountered, the execution of the function terminates
    - The control is passed back to the calling function together with the return value
  - The main function will resume at the calling statement
  - When a return statement in the main function is encountered, the program ends

Function Call - Flow of Control

The program on the right consists of two functions:

- main(): controls general logic flow and handles I/O
- larger(): determines the larger of two numbers

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Function Call - Flow of Control

- When a program is put into execution, it always starts at the main function no matter where its definition is in the source file

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Function Call - Flow of Control

- The statements in the main function are executed sequentially from top to bottom
- The control is passed from one statement to another

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
   return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Function Call - Flow of Control

- When a function call is encountered, the execution of the current function is suspended

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Function Call - Flow of Control

- The values of the arguments are copied to the formal parameters of the called function
- The control is passed to the called function

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Function Call - Flow of Control

- Likewise, the statements in the called function are executed from top to bottom
- The control is passed from one statement to another

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{                          2.5        5.0
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Function Call - Flow of Control

- When a return statement is encountered, the execution of the function terminates

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

2.5 5.0

Function Call - Flow of Control

- The control is passed back to the calling function together with the return value

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

5.0

Function Call - Flow of Control

- The main function will resume at the calling statement

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Function Call - Flow of Control

- The statements in the main function are executed sequentially from top to bottom
- The control is passed from one statement to another

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

c takes the value 5.0 which is the return value of `larger()`

Function Call - Flow of Control

- The statements in the main function are executed sequentially from top to bottom
- The control is passed from one statement to another

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Function Call - Flow of Control

- When a return statement in the main function is encountered, the program ends

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  double a = 2.5, b = 5.0, c;

  c = larger(a, b);
  cout << c << " is larger." << endl;

  return 0;
}
```

Think about this: the main body is also a function main(), it is called by the operating system when you run the program.

Void Functions

- In some situations, a function simply carries out some operations and produces no return value
- A function with no return value is called a void function
- In this case, the void type specifier, which indicates absence of type, can be used
- The return statement in a void function does not specify any return value. It is used to return the control to the calling function
- If a return statement is missing in a void function, the control will be returned to the calling function after the execution of the last statement in the function

Void Functions

Examples

```
void print_msg(int x)
{
  cout << "This is a void function " << x << endl;
  return;
}
```

A return statement with no return value

```
void print_msg(int x)
{
  cout << "This is a void function " << x << endl;
}
```

No return statement

Both are OK!

Local Variables

- Variables declared within a function, including formal parameters, are private or local to that particular function, i.e., no other function can have direct access to them
- Local variables in a function come into existence only when the function is called, and disappear when the function is exited
  - Do not retain their values from one function call to another
  - Their values must be explicitly set upon each entry
- Local variables declared within the same function must have unique identifiers, whereas local variables of different functions may use the same identifier

Local Variables

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
    double max;
    max = (x >= y)? x : y;

    return max;
}

int main()
{
    double a = 2.5, b = 5.0, max;

    max = larger(a, b);
    cout << max << " is larger." << endl;

    return 0;
}
```

local variables of **larger()**:
x, y, max
i.e., these variables are input parameters or variables defined in the function larger(), and therefore can only be seen or used in larger()

local variables of **main()**:
a, b, max
i.e., these variables are defined in the function main(), and therefore can only be seen or used in main()

The local variables **max** of **larger()** and **max** of **main()** are **unrelated**

Local Variables

```
#include <iostream>
using namespace std;

double larger(double x, double y)
{
   // double max;
   max = (x >= y)? x : y;

   return max;
}

int main()
{
   double a = 2.5, b = 5.0, max;

   max = larger(a, b);
   cout << max << " is larger." << endl;

   return 0;
}
```

There will be a compilation error if we comment out the declaration of **max** in **larger()** because **max** in **main()** is a local variable of **main()** and cannot be seen or used in larger()

Global Variables

- Variables may also be declared outside all functions
- Such variables are called global variables because they can be accessed by all functions, i.e., globally accessible within the file containing the program
- Global variables remain in existence permanently
  - Retain their values even after the functions that set their values have returned and exited
  - Can be used instead of arguments to communicate data between functions, however:
    - The values of global variables can be changed by any functions
    - Hard to trace, especially when something goes wrong
    - Not recommended and should be avoided!
- Frequently used as declared constant (whose values cannot be changed)

Global Variables

```
#include <iostream>
using namespace std;

double a, b;
const double PI = 3.1415;

double larger()
{
   return (a >= b)? a : b;
}

int main()
{
   cout << "Input two integers: ";
   cin >> a >> b;
   cout << larger() << " is larger." << endl;

   double r;
   cout << "Input radius of a circle: ";
   cin >> r;
   cout << "Area of circle = " << PI * r * r << endl;
   return 0;
}
```

global variables:
a, b, PI

The global constant **PI** can be used throughout the file after its declaration.

Avoid using global variables to communicate data between functions

The variables **a, b** should best be changed into input parameters for the function larger(). Can you do that?

Scopes of Variables

- The scope of a variable is the portion of a program that the variable is well-defined and can be used
- A variable cannot be referred to beyond its scope
- The scope of a local / global variable starts from its declaration up to the end of the block / file
  - A block is delimited by a pair of braces { }
  - Variables declared in outer blocks can be referred to in an inner block
- Variables can be declared with the same identifier as long as they have different scopes
  - Variables in an inner block will hide any identically named variables in outer blocks

Scopes of Variables

```
double a;
int func(int x, int y)
{
    …
    if (x > y)
    {
        int k;
        …
    }
    int z;
    …
}

double b;
int main()
{
    int x, y, z;
    …
    if (…)
    {
        int x;
        …
    }
    …
}
```

Scope of global variable **a**:
from declaration to end of block
(in this case, end of file; hence scope of **a** is the entire file)

Scope of formal parameters **x, y**:
entire function

Scope of local variable **k**:
from declaration to end of block
(in this case, end of if statement)

Scope of local variable **z**:
from declaration to end of block
(in this case, end of func)

Scopes of Variables

```
double a;
int func(int x, int y)
{
    …
    if (x > y)
    {
        int k;
        …
    }
    int z;
    …
}

double b;
int main()
{
    int x, y, z;
    …
    if (…)
    {
        int x;
        …
    }
    …
}
```

Scope of global variable **b**:
from declaration to end of block
(in this case, end of file)

Scope of local variables **x, y, z**:
from declaration to end of block
(in this case, end of main function)

Scope of local variable **x** in the inner block:
from declaration to end of block
(in this case, end of if statement)

the outer x is hidden within this block

Scopes of Variables

```
#include <iostream>
using namespace std;

int main()
{
   int i = 0;
   cout << "Outer block: i = " << i << endl;

   {
      int i = 100;
      cout << "Inner block: i = " << i << endl;
   }

   cout << "Outer block: i = " << i << endl;
   return 0;
}
```

1 2 3

Screen output

```
Outer block: i = 0
Inner block: i = 100
Outer block: i = 0
```

Pass-by-Value

- When a function call takes place, the values of the arguments are copied to the formal parameters of the function
- This mechanism of parameter-passing is known as pass-by-value
- Recall that formal parameters are local variables
  - Any changes made to their values are local to the function and will not alter the arguments in the calling function
  - These variables will disappear when the function exits, only the return value of the function will be passed back to the calling function

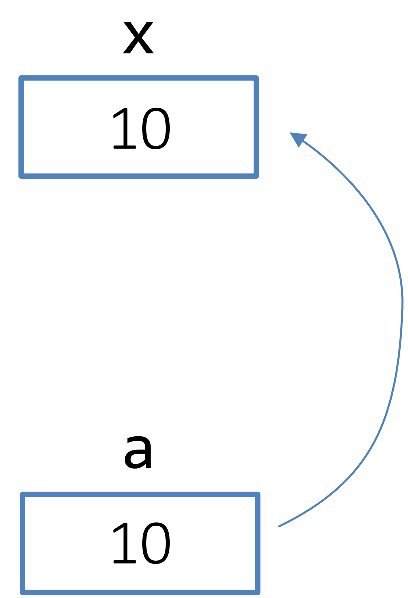
Pass-by-Value

```
#include <iostream>
using namespace std;

// computes the square of an integer
void square( int x )
{
   x *= x;
}

int main()
{
   int a = 10;
   cout << a << " squared: ";
   square( a );
   cout << a << endl;
   return 0; }
```

x is a local variable of the square function



Copying of value of actual argument to formal parameter

Pass-by-Value

```
#include <iostream>
using namespace std;

// computes the square of an integer
void square( int x )
{
    x *= x;
}

int main()
{
    int a = 10;
    cout << a << " squared: ";
    square( a );
    cout << a << endl;
    return 0; }
```

x

100

a

10

Pass-by-Value

```
#include <iostream>
using namespace std;

// computes the square of an integer
void square( int x )
{
    x *= x;
}

int main()
{
    int a = 10;
    cout << a << " squared: ";
    square( a );
    cout << a << endl;
    return 0; }
```

Variable **x** disappears (more precisely, the memory location it occupies is released back to the system) upon function completion.

a

10

Pass-by-Value

```
#include <iostream>
using namespace std;

void swap(int a, int b)
{
    cout << "a = " << a << ", b = " << b << endl;
    int temp = a;
    a = b;
    b = temp;
    cout << "a = " << a << ", b = " << b << endl;
}

int main()
{
    int x = 0, y = 100;
    cout << "x = " << x << ", y = " << y << endl;
    swap(x, y);
    cout << "x = " << x << ", y = " << y << endl;
    return 0;
}
```

2

3

1

4

Suppose we want to swap the values in the variables x and y using the function swap(), what will happen in this program?

Screen output

```
x = 0, y = 100
a = 0, b = 100
a = 100, b = 0
x = 0, y = 100
```

It doesn't work! Why?

Because the variables x and y are passed to swap() using pass-by-value, only the values are transferred to swap(), and swap() can only deal with its local variables a and b.

Pass-by-Reference

- In order to allow a function to modify the arguments (variables) in the calling function, another parameter-passing mechanism known as pass-by-reference should be used
- In pass-by-reference
  - The formal parameters will refer to the same memory cells of the arguments in run-time, and therefore the arguments must be variables
  - Any changes made to the values of the formal parameters will be reflected in the arguments as they share the same memory cells

Pass-by-Reference

- To indicate a formal parameter will be passed by reference, an ampersand sign & is placed in front of its identifier in the function header and function declaration

Syntax (function header)

```
type_ret  func_name(type1 &par1, type2 &par2, ...)
```

Syntax (function declaration)

```
type_ret  func_name(type1 &par1, type2 &par2, ...);
```

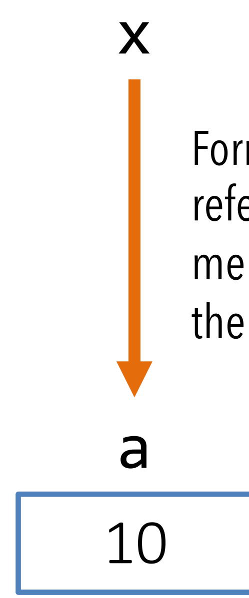
Pass-by-Reference

```
#include <iostream>
using namespace std;

// computes the square of an integer
void square( int &x )
{
   x *= x;
}

int main()
{
   int a = 10;
   cout << a << " squared: ";
   square( a );
   cout << a << endl;
   return 0;
}
```

Note the **&** to indicate that the formal parameter x is pass-by-reference



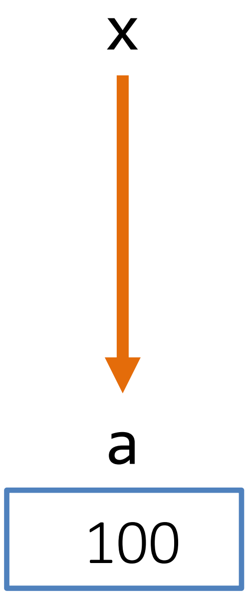
Formal parameter refers to the same memory location as the argument

Pass-by-Reference

```
#include <iostream>
using namespace std;

// computes the square of an integer
void square( int &x )
{
   x *= x;
}

int main()
{
   int a = 10;
   cout << a << " squared: ";
   square( a );
   cout << a << endl;
   return 0;
}
```

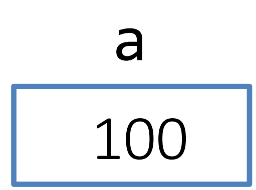


Pass-by-Reference

```
#include <iostream>
using namespace std;

// computes the square of an integer
void square( int &x )
{
   x *= x;
}

int main()
{
   int a = 10;
   cout << a << " squared: ";
   square( a );
   cout << a << endl;
   return 0;
}
```



Pass-by-Reference

What happens if we use pass-by-reference for the swap function?

```
#include <iostream>
using namespace std;

void swap(int &a, int &b)
{
2   cout << "a = " << a << ", b = " << b << endl;
    int temp = a;
    a = b;
    b = temp;
3   cout << "a = " << a << ", b = " << b << endl;
}

int main()
{
    int x = 0, y = 100;
1   cout << "x = " << x << ", y = " << y << endl;
    swap(x, y);
4   cout << "x = " << x << ", y = " << y << endl;
    return 0;
}
```

Screen output

```
x = 0, y = 100
a = 0, b = 100
a = 100, b = 0
x = 100, y = 0
```

The formal parameters a and b in swap() refer to the memory locations of the arguments x and y, respectively.

Pass-by-Reference vs. Value-Returning Function

- Call by Reference: modify the values of the actual parameters in the calling function
- Value-Returning Function: returning a value that can be used by the calling function

Call by Value

```
int squareByValue( int number )
{
      return number *= number;
}
```

Caller's argument not modified,
return result by **return value**

Call by Reference

```
void squareByReference( int &number )
{
      number *= number;
}
```

Caller's argument modified,
result stores in the **reference parameter**

Pass-by-Reference vs. Value-Returning Function

```
int squareByValue( int );
void squareByReference( int & );

int main()
{
     int x = 2;
     int z = 4;

     cout << "x = " << x << " before squareByValue\n";
     cout << "Value returned by squareByValue: "
          << squareByValue( x ) << endl;
     cout << "x = " << x << " after squareByValue\n" << endl;

     cout << "z = " << z << " before squareByReference" << endl;
     squareByReference( z );
     cout << "z = " << z << " after squareByReference" << endl;

     return 0;
}
```

```
x = 2 before squareByValue
Value returned by
squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Screen output

Return value of squareByValue() is used by the cout expresssion.

Result of computation by squareByReference() is updated in z.

Pass-by-Reference vs. Value-Returning Function

- Good programming style:
  - If a function needs to return more than one values, use a void function with reference parameters to return the values

```
const double CONVERSION = 2.54;
const int INCHES_IN_FOOT = 12;
const int CENTIMETERS_IN_METER = 100;

void metersAndCentTofeetAndInches(int mt, int ct, int& f, int& in)
{
    int centimeters;
    centimeters = mt * CENTIMETERS_IN_METER + ct;
    in = (int) (centimeters / CONVERSION);
    f = in / INCHES_IN_FOOT;
    in = in % INCHES_IN_FOOT;
}
```

f and in are the computation results.
Think about how the calling functions can call this function and access the results through the arguments after function call.

Quick Exercise 1

What's the output of the following program?

Try dry run (i.e., trace manually without using the computer to run) the program to obtain the result. Then run it on your computer to check the result.

```
#include <iostream>
using namespace std;

void figureMeOut(int &x, int y, int &z) {
    cout << x << ' ' << y << ' ' << z << endl;
    x = 1;
    y = 2;
    z = 3;
    cout << x << ' ' << y << ' ' << z << endl;
}

int main() {
    int a=10, b=20, c=30;
    figureMeOut(a, b, c);
    cout << a << ' ' << b << ' ' << c << endl;
}
```

Answer to Quick Exercise 1

Screen output:

```
10 20 30
1 2 3
1 20 3
```

flow of control and functions

PROBLEMS

Problem 1

Is there any error in the following program? if no, what is the output? (Try to answer before compiling and running the program.)

```
#include <iostream>
using namespace std;
int main(){
    int a = 0;
    if(a = 0)
        cout << "a is 0";
    else
        cout << "a is not 0";
}
```

Problem 2

What is the output of this program? Can you explain why?

```
#include <iostream>
using namespace std;
int main(){
    int b = 2;
    if(b == 2)
        b++;
    if(b == 2);
        b++;
    cout<< b;
}
```

Problem 3

What is the output of this program? (Try to answer before compiling and running the program.)

```cpp
#include <iostream>
using namespace std;
int main(){
    int c = 3;
    int d = c++;

    if(c++ == 4 && d == 3)
        cout << "1: " << c << " " << d << endl;
    if(++c == 5 && d-- == 3)
        cout << "2: " << c-- << " " << d << endl;
    cout << "3: " << c << " " << d << endl;
}
```

Problem 4

Recall that we wrote a program that reads in three integers and outputs the maximum in Module 3. Draw the flowchart for a program that reads in three integers and outputs the minimum.

Problem 5

Write the corresponding program to the flowchart of problem 4.

Problem 6

Write a program with an **`if-else`** statement that outputs the word **`High`** if the value of the variable score is greater than 100 and **`Low`** if the value of score is at most 100. The variable score is of type **`int`**.

Problem 7

Write a program to determine the outcome of the paper-rock-scissor game. Each of two users types in either P, R, or S. The program then announces the winner as well as the basis for determining the winner: Paper covers rock, Rock breaks scissors, Scissors cut paper, or Nobody wins. Be sure to allow the users to use lowercase as well as uppercase letters.

Problem 8

What is the output of the following program? Can you explain the output?

```cpp
#include <iostream>
using namespace std;

int main() {
    int count = 3;
    for (int i=1; i<=count; i++) {
        for (int j=1; j<=count; j++)
            cout << i << " x " << j << " = " << i*j << endl;
    }

}

```

Problem 9

Write a program that reads in five integers and that outputs the sum of all integers greater than zero, the sum of all the integers less than zero, and the sum of all the integers, whether positive, negative, or zero. The user enters the numbers just once each and the user can enter them in any order. Your program should not ask the user to enter the positive numbers and the negative numbers separately.

Problem 10

Write a program that will output the following pattern. Use two for loops to achieve the output

```
0123456
012345
01234
0123
012
01
0
```

Problem 11

Write a program with a function that takes one argument of type `double`. The function returns the character value `P` if its argument is positive and returns `N` if its argument is zero or negative. In the main function of your program, call this function to test its behavior.

Problem 12

Write a program with a function that takes one argument of type `int` and one argument of type `double`. The function returns a value of type `double` that is the average of the two arguments.

Problem 13

Greek mathematicians took a special interest in numbers that are equal to the sum of their proper divisors (a proper divisor of n is any divisor less than n itself). They called such numbers *perfect numbers*. For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14. Write a function that determines if a given number is a perfect number. Your function should take an `int` as a parameter and return a value of type `bool`.

Problem 14

Write a program that finds all the perfect numbers between two limits entered by the user. Use your function from Problem 13.

Problem 15

- A liter is 0.264179 gallons. Write a program that will read in the number of liters of gasoline consumed by the user's car and the number of miles the car delivered, and will then output the number of miles per gallon the car gets. Your program should allow the user to repeat this calculation as many times as the user wishes. Define a function to compute the number of miles per gallon.
- Modify your program so that it will take input data for two cars and output the number of miles per gallon delivered by each car. Your program will also announce which car has the best fuel efficiency (highest number of miles per gallon).

Problem 16

The area of an arbitrary triangle can be computed using the formula

$$\text{area} = \sqrt{s(s - a)(s - b)(s - c)},$$

where a, b and c are the lengths of the sides, and s is the semiperimeter s=(a+b+c)/2.

Write a `void` function that uses **five** parameters: three *value parameters* that provide the lengths of the edges, and two *reference parameters* for the area and the perimeter. You may use the function `sqrt()` which is available if you write `#include<cmath>`.

Problem 17

What is the output of the following program? Explain.

```
#include <iostream>
using namespace std;

int main()
{
  int a = -15, b;
  a--;
  cout << "a = " << a << endl;
  {
    int b = 7;
    b = 2*a*b;
  }
  int result = a+b;
  cout << "result = " << result << endl;
}
```

Optional.

For those who would like to challenge yourselves.
Even for those of you who are beginners in C++ programming, it's highly recommended for you to take a look at these problems and try to tackle them as well.

You are welcome to discuss these problems in the Moodle forum.

CHALLENGES

Challenge 1

What will be the output of this program? Or will it loop forever?
Try to think about the final value of “cycleCount” before compiling this program.

```cpp
#include <iostream>
using namespace std;

int main(){
  int cycleCount = 0;
  for(int i = 1; ++i < 10; )
    cycleCount++;

  cout << cycleCount << endl;
}
```

Challenge 2

Redo Problem 12. This time, use ONLY one statement in your function to accomplish the same task.

Challenge 3

Write a program to print the following diamond pattern of '*'.

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

Challenge 4

Write a function which checks if a number is prime or not. (What’s the return type of the function?)

Challenge 5

1. Using the function created in Challenge 4, create another function that print all prime numbers p within a given range defined by parameters x, y, where x <= p <= y.
2. Can you think of a faster implementation of the function?

Challenge 6

Extend the program you have written for Problem 14, output also whether the perfect numbers found are prime numbers.