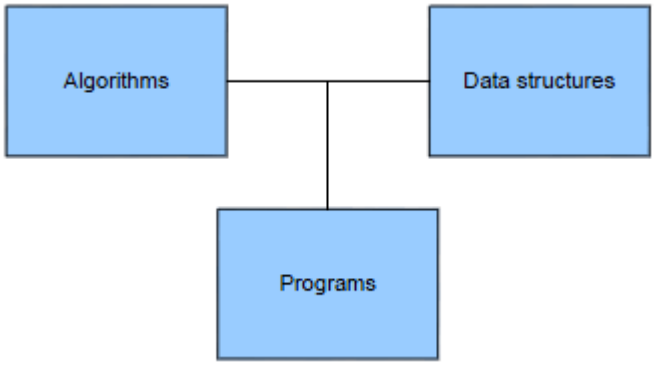
==========

FUNCTIONS

==========

Modular programming

- **Program development**



  - Procedural programming:
    - tend to focus on algorithms
  - Object-oriented programming:
    - tend to focus on data structures

- **Top-down design**
  - Top down design means
    breaking the problem down into components (**modules**) **recursively**.
  - Here want to keep in mind two general principles
    - Abstraction
    - Modularity
  - Architectural design: identifying the building blocks
  - Abstract specification: describe the data/functions and their constraints
  - Interfaces: define how the modules fit together
  - Component design: recursively design each block

  - **Each module**
    - should comprise related data and functions,
    - and the designer needs to specify how these components interact
      - what their dependencies are, and
      - what the interfaces between them are.

  - Minimising dependencies, and making interfaces as simple as possible are both desirable to facilitate modularity.

- By minimising the ways in which modules can interact,
  we greatly limit the overall complexity,
  and hence limit unexpected behaviour, increasing robustness.
- Because a particular module interacts with other modules
  in a carefully defined manner,
  it becomes easier to test/validate, and can become a reusable component.

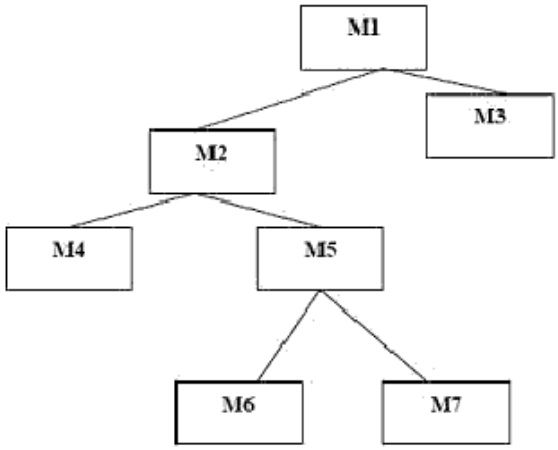
- **Abstraction**
  - The abstraction specifies what operations a module is for,
    without specifying how the operations are performed.

- **Modularity**
  - The aim is to define a set of modules
    each of which transcribes, or encapsulates particular functionality,
    and which interacts with other modules in well defined ways.
  - The more complicated the set of possible interactions between modules,
    the harder it will be to understand.
    - Humans are only capable of understanding and managing
      a certain degree of complexity, and it is quite easy (but bad practice)
      to write software that exceeds this capability.

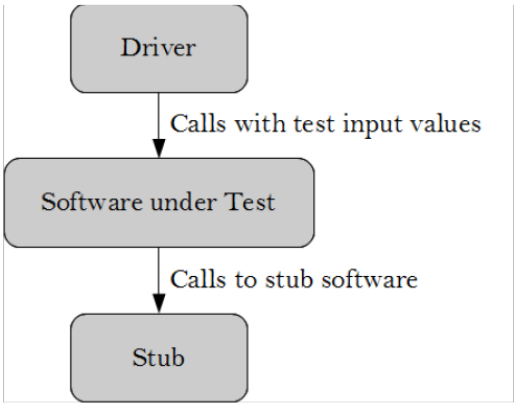
- **Modular design**
  - While the top-down design methodology is a general tool,
    how we approach it will potentially be language dependent.
  - In **procedural programming**
    - the design effort tends to concentrate on
      the functional requirements
    - and recursively subdivides the functionality into
      procedures/functions/subroutines
      until each is a simple, easily understood entity.
      - Examples of procedural languages are
        C, Pascal, Matlab, Fortran
  - In **object-oriented programming**
    - the design emphasis shifts to the data structures
    - and to the interfaces between objects.
      - Examples of object-oriented languages are
        C++ and Java.

- **Top-down, modular design in procedural programming**
  - Break the algorithm into subtasks
  - Break each subtask into smaller subtasks
  - Repeat until the smaller subtasks are easy to implement in the programming language

- **Benefits of modular programming**
  - Programs are
    - easier to write
    - easier to test
    - easier to debug
    - easier to understand
    - easier to change
    - easier for teams to develop
  - Modules can be reused

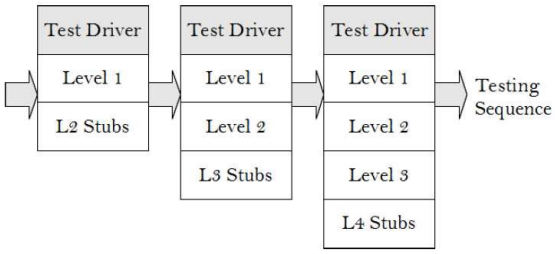


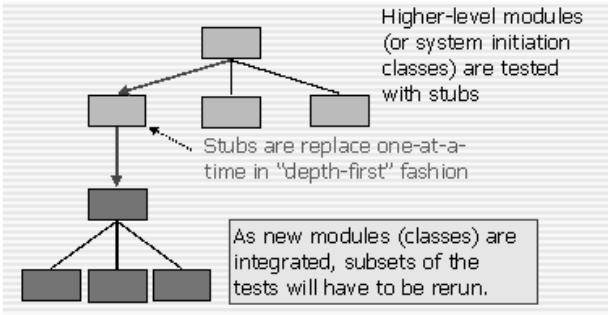
- **Code development**
  - **Top-down**
    - Code high level parts
      using "stubs" with assumed functionality for low-level dependencies
    - Iteratively descend to lower-level modules
  - **Bottom-up**
    - Code and test each low-level component
    - Need "test driver"
      so that low-level can be tested in its correct context
    - Integrate components



- **Stub:**
  is temporary or dummy software
  that is required
  by the software under test
  for it to operate properly.
- **Test driver:**
  calls the software under test,
  passing the test data as inputs.

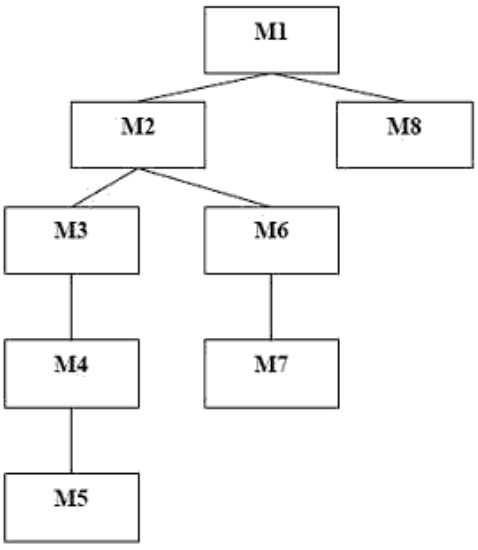
- **Top-down integration**





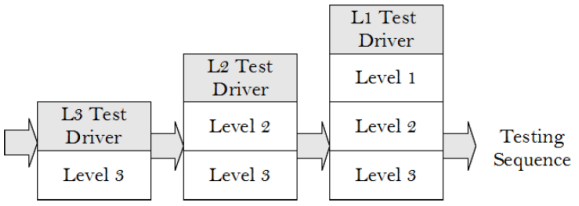
  - The highest-level modules are tested and integrated first.
  - This allows high-level logic and data flow to be tested early in the process
  - **Advantages**:
    - The top layer provides an early outline of the overall program
      - helping to find design errors early on and
      - giving confidence to the team, and possibly the customer, that the design strategy is correct
  - **Disadvantages**:
    - Difficulty with designing stubs that provide a good emulation of the interactivity between different levels
    - If the lower levels are still being created while the upper level is regarded as complete, then sensible changes that could be made to the upper levels that would improve the functioning of the program may be ignored
    - When the lower layers are finally added the upper layers may need to be retested

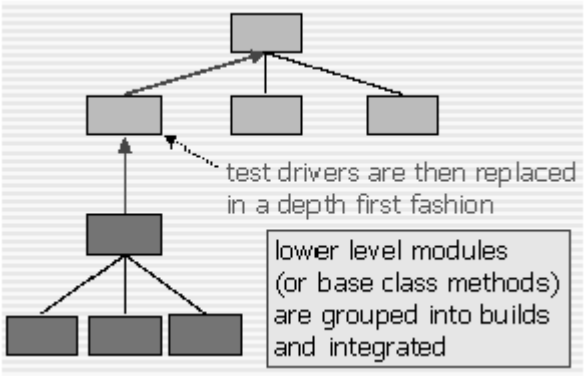
- **Top-down integration example**



  - Depth first approach
    - All modules on a control path are integrated first.
    - The sequence of integration would be (M1, M2, M3), M4, M5, M6, M7, and M8.
  - Breadth first approach
    - All modules directly subordinate at each level are integrated together.
    - The sequence of integration would be (M1, M2, M8), (M3, M6), M4, M7, and M5.

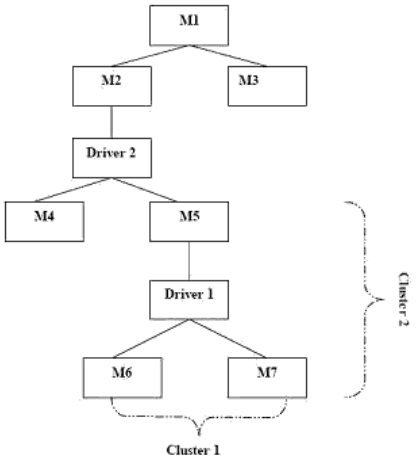
- **Bottom-up integration**





  - The lowest-level units are tested and integrated first.
  - These modules are tested early in the development process
  - The need for stubs is minimized
  - **Advantages**:
    - Overcome the disadvantages of top-down testing.
    - Additionally, drivers are easier to produce than stubs
    - Because the tester is working upwards from the bottom layer, they have a more thorough understanding of the functioning of the lower layer modules and thus have a far better idea of how to create suitable tests for the upper layer modules
  - **Disadvantages**:
    - It is more difficult to imagine the working system until the upper layer modules are complete
    - The important user interaction modules are only tested at the end
    - Drivers must be produced, often many different ones with varying levels of sophistication

- **Top-down integration example**



- **Top-down vs. Bottom-up integration**
  - Top-down integration has the "advantage'' (over bottom-up integration)
    that it starts with the main module, and
    continues by including top level modules -
    so that a `"global view'' of the system is available early on.
  - It has the disadvantage that
    it leaves the lowest level modules until the end,
    so that problems with these won't be detected early.
  - If Bottom-up integration is used
    then these might be found soon after integration testing begins.
  - Top-down integration also has the disadvantage or requiring stubs -
    which can sometimes be more difficult to write than drivers,
    since they must simulate computation of outputs,
    instead of their validation.
  - With Top-down
    - it's harder to test early
      because parts needed may not have been designed yet
  - With Bottom-up,
    - you may end up needing
      things different from how you built them

- **Sandwich Integration**
  - It is a combination of both Top-down and Bottom-up integration testing.
  - A target layer is defined in the middle of the program
  - Testing is carried out
    from the top and bottom layers to converge at this target layer.
  - Advantages:
    - the top and bottom layers can be tested in parallel and
    - can lower the need for stubs and drivers.
  - However,
    - it can be more complex to plan and
    - selecting the 'best' target layer can be difficult.

- **Function call syntax**
  - `function_name(parameter_1, parameter_2, … , parameter_n)`

- **Predefined functions**
  - C++ comes with libraries of predefined functions
  - Example: `sqrt` and `pow` functions
    - `square_side = sqrt(square_area);`
    - `cube_volume = pow(cube_edge, 3.0);`
  - `square_area`, `cube_edge` and `3.0` are the **arguments** of calls
  - The arguments can be variables, constants or expressions
  - A library must be "included" in the program to make the predefined functions available
    - To include the math library containing `sqrt()` and `pow()`:
      `#include <cmath>`

- **Programmer defined functions**
  - Two components of a function definition:
    - **Function declaration** (or **function prototype**)
      - Shows how the function is called
      - Must appear in the code before the function can be called
      - Syntax:

```
type_returned function_name(parameter_1,..., parameter_n);
```

      - Example:

```
double total_price(int num_items, double item_price);
```

      - Tells the **return type** (`double`)
        - the return type can be `void` (*see later*)
      - Tells the name of the function (`total_price`)
      - Tells how many arguments are needed (2)
      - Tells the types of the arguments (`int` and `double`)
      - Tells the formal parameter names (`num_items` and `item_price`)
      - Formal **parameters** are like placeholders for the actual **arguments** used when the function is called
      - Formal parameter names can be any valid identifier
    - **Function definition**
      - Provides the same information as the function declaration
      - Describes how the function does its task
      - Can appear before or after the function is called
      - Syntax:

```
type_returned function_name(parameter_1,..., parameter_n)
   {
     //FUNCTION BODY - code to make the function work
   }
```

      - Example:

```
double total_price(int num_items, double item_price)
{
   double total = num_items * item_price;
   return total; // OR just return num_items * item_price;
}
```

    - **`return` statement**
      - Ends the function call
      - Returns the value calculated by the function

- **The function call**
  - A function call must be preceded by either
    - The function's declaration or the function's definition
    - If the function's definition precedes the call, a declaration is not needed
  - A function call
    - tells the name of the function to use
    - lists the arguments
    - is used in a statement where the returned value makes sense
  - Example:

```
...
double amount_to_pay;
...
amount_to_pay = total_price(10,0.5);
...
```

  - **NOTES**:
    - the type of the arguments is not included in the call
    - the type of the arguments must be compatible with the type of the parameters
    - the number of arguments must be equal to the number of parameters
    - the compiler checks that the types of the arguments are correct and in the correct sequence
    - the compiler cannot check that arguments are in the correct logical order
      - Example of a sintactically correct call that would produce a faulty result:

```
double total_price(int num_items, double item_price);

...
int numItems = 100, itemPrice = 5;
double amount_to_pay;
...
amount_to_pay = total_price(itemPrice, numItems);
```

- **Procedural abstraction**
  - Programs and the "black box" analogy
    - A "black box" refers to something that we know how to use, but the method of operation is unknown
    - A person using a program does not need to know how it is coded
    - A person using a program needs to know what the program does, not how it does it
  - Functions and the "black box" analogy
    - A programmer who uses a function needs to know what the function does, not how it does it
    - A programmer using a function needs to know what will be produced if the proper arguments are put into the box
  - Designing functions as black boxes is an example of information hiding
  - The function can be used without knowing how it is coded
  - The function body can be "hidden from view"
  - Designing with the black box in mind allows us
    - To change or improve a function definition without forcing programmers using the function to change what they have done
  - **To know how to use a function** simply by reading the **function declaration / prototype** and **its comments**
  - **Function comments** should describe:
    - what the function does
    - what is the meaning of each function parameter
    - what is the return value
  - Procedural Abstraction is writing and using functions as if they were black boxes
    - Procedure is a general term meaning a "function like" set of instructions
    - Abstraction implies that when you use a function as a black box, you abstract away the details of the code in the function body

- **Formal parameters**
  - Programmers must choose meaningful names for formal parameters
  - Formal parameter names may or may not match variable names used in the main part of the program
  - Variables used in the function, other than the formal parameters, should be declared in the function body

- **Local variables (/ constants or identifiers, in general)**
  - Variables declared in a function:
    - Are local to that function
      - they cannot be used from outside the function
    - Have the function as their scope
  - Example: variables declared in the **`main`** function of a program:
    - Are local to the main part of the program, they cannot be used from outside the main part
    - Have the **`main`** function as their scope
  - Formal parameters are local variables
    - They are used just as if they were declared in the function body
  - Do NOT re-declare the formal parameters in the function body, they are declared in the function declaration

- **Global variables and constants (or identifiers, in general)**
  - Declared outside any function body
  - Declared before any function that uses it
  - Available to more than one function
  - Should be used sparingly
  - Generally make programs more difficult to understand and maintain
  - Example:

```
...
const double PI = 3.14159; //global constant
...

double sphereVolume(double radius)
{
  return 4.0/3*PI*pow(radius,3);  // ... is visible here
}

double circlePerimeter(double radius)
{
  return 2*PI*radius;   // ... and here
}
```

- **Call-by-value and call-by-reference parameters**
  - **Call-by-value** means that the function parameter receives a copy of the value of the argument in the call
    - if the parameter is modified the value of the argument is not affected
    - the argument of the call may be a variable or a constant
  - **Call-by-reference** means that the function parameter receives the address of the argument in the call
    - if the parameter is modified the value of the argument is modified
    - the argument of the call can not be a constant must be a variable
    - '&' symbol (ampersand) identifies the parameter as a call-by-reference parameter
  - **Example**:
    - *see the* **swap** *functions example in the next pages.*
  - Call-by-value and call-by-reference parameters can be mixed in the same function
  - How do you decide whether a call-by-reference or call-by-value formal parameter is needed?
    - Does the function need to change the value of the variable used as an argument?
      - Yes? => Use a call-by-reference formal parameter
      - No? => Use a call-by-value formal parameter
    - Note: sometimes, the value of a call-by-reference formal parameter is undetermined when the function is called
      - its value is determined (calculated or read) in the function body

TO DO BY STUDENTS:

- Write a function to calculate the square root of a number, using the previously referred Babilonian algorithm.

```
/*
REFERENCE OPERATOR: &
INDEPENDENT REFERENCES (not much useful)
*/
#include <iostream>

using namespace std;

int main()
{
	int x;
	int& y = x; // independent reference
	            // TRY WITH JUST int &y;

	x = 10;
	cout << "x = " << x << "; y = " << y << endl;

	y = 20;
	cout << "x = " << x << "; y = " << y << endl;

	//cout << "&x = " << &x << "; &y = " << &y << endl;
	//UNCOMMENT ABOVE STATEMENT AND EXPLAIN THE RESULT
	//the result is shown in hexadecimal format;
	//modify the program to show it in decimal format

	return 0;
}
```

==========

```
/*
CALL BY VALUE vs. CALL BY REFERENCE
*/
#include <iostream>

using namespace std;

void swap1(int a, int b)   // 'a' and 'b' are value parameters
{
      int tmp = a;
      a = b ;
      b = tmp;

      //showing the memory addresses of the parameters. UNCOMMENT TO SEE
      /*
      cout << endl;
      cout << "swap1():addr. a = " << &a << endl; //show mem.addr. of a
      cout << "swap1():addr. b = " << &b << endl; //show mem.addr. of b
      */
}

void swap2(int &a, int &b)   // 'a' and 'b' are reference parameters
{
      int tmp = a;
      a = b ;
      b = tmp;

      //showing the memory addresses of the parameters. UNCOMMENT TO SEE
      /*
      cout << endl;
      cout << "swap2():addr. a = " << &a << endl; //show mem.addr. of a
      cout << "swap2():addr. b = " << &b << endl; //show mem.addr. of b
      */
}

int main()
{
      int x = 1;
      int y = 2;

     cout << "Before swaps(): x = " << x << "; y = " << y << endl << endl;

      swap1(x,y);
      cout << "After swap1(): x = " << x << "; y = " << y << endl;

      swap2(x,y);
      cout << "After swap2(): x = " << x << "; y = " << y << endl;

      //showing the memory addresses of the variables. UNCOMMENT TO SEE
      /*
      cout << endl;
      cout << "address of x = " << &x << endl; //show mem.addr. of x
      cout << "address of y = " << &y << endl; //show mem.addr. of y
      */

      return 0;
}
=====================================================================
```

```
/*
FUNCTIONS - creating abstractions
PASSING PARAMETERS BY VALUE AND BY REFERENCE
RETURN VALUES
*/
#include <iostream>
#include <iomanip>
#include <cctype>

using namespace std;

/**
Tests if operator is valid (+,-,*,/)
Returns:
@param operation - the operator (THIS PARAMETER IS PASSED BY VALUE)
@function return value: true if operation is valid, false otherwise
*/
bool validOperation(char operation)
{
   return (operation == '+' || operation == '-' || operation == '*' || operation == '/');
}


/**
Reads arithmetic operation in the form "operand1 operation operand2"
@param operation - the operator (+,-,*,/)
@param operand1 - 1st operand
@param operand2 - 2nd operand
@function return value: FALSE when user enters CTRL-Z; TRUE otherwise
NOTE:
this function returns 4 values to the caller
- 3 of them are returned through its parameters (PASSED BY REFERENCE)
- 1 is the return value of the function
*/
bool readOperation(char &operation, double &operand1, double &operand2)
{
      bool anotherOperation = true;
      bool validInput = false;

      do
      {
            bool validOperands = true;
            cout << endl << "x op y (CTRL-Z to end) ? ";
            cin >> operand1 >> operation >> operand2;

            if (cin.fail())
            {
                  validOperands = false;
                  if (cin.eof())
                        anotherOperation = false;
                  // ALTERNATIVE: // return false; OR exit(0); ...
                  // IN THIS LAST CASE main() COULD BE WRITTEN IN A DIF.WAY
                  // HOW ?
                  // BUT WOULD LOOSE LEGIBILITY ...
                  else
                  {
                        cin.clear();
                        cin.ignore(1000,'\n');
                  }
            }
```

```
            else
                  cin.ignore(1000, '\n');
            if (!validOperation(operation))
                  cerr << "Invalid operation !\n";
            validInput = validOperands && validOperation(operation);
      } while (anotherOperation && !validInput);
        //Repeat ... until ((NOT anotherOperation) OR validInput)

      return anotherOperation;
}

/**
Computes result of "operand1 operation operand2"
where operation is (+,-,*,/);   assumes that 'operation' is valid
@param operation - the operator; must be (+,-,*,/)
@param operand1 - 1st operand
@param operand2 - 2nd operand (THESE PARAMETERS ARE PASSED BY ...?)
@function return value: the result of the operation
*/
double computeResult(char operation, double operand1, double operand2)
{
      double result;

      switch (operation)
      {
      case '+':
            result = operand1 + operand2;
            break;
      case '-':
            result = operand1 - operand2;
            break;
      case '*':
            result = operand1 * operand2;
            break;
      case '/':
            result = operand1 / operand2;
            break;
      }
      return result;
}

/**
Shows result of "operand1 operation operand2" where operation is (+,-,*,/)
Returns:
@param operation - the operator; must be (+,-,*,/)
@param operand1 - 1st operand
@param operand2 - 2nd operand
@param result - result of the operation
@function return value: (none)
(THE PARAMETERS ARE PASSED BY ...?; THIS FUNCTION HAS NO RETURN VALUE)
*/
void showResult(char operation, double operand1, double operand2, double
result, unsigned int precision)
{
      cout << fixed << setprecision(precision);
      cout << operand1 << ' ' << operation << ' ' << operand2 <<
            " = " << result << endl;
}
```

```
int main()
{
	const unsigned int NUMBER_PRECISION = 6;

	double operand1, operand2;
	char operation;
	double result; // result of "operand1 operation operand2"
	bool anotherOperation; // FALSE when user enters CTRL-Z; TRUE otherwise

	anotherOperation = readOperation(operation, operand1, operand2);

	while (anotherOperation)
	{
		result = computeResult(operation, operand1, operand2);
		showResult(operation,operand1,operand2,result, NUMBER_PRECISION);
		anotherOperation = readOperation(operation, operand1, operand2);

	};

	return 0;
}
```

==========

```
/*
TESTING USER KNOWLEDGE ABOUT BASIC MATH OPERATIONS
*/

#include <iostream>
#include <iomanip>
#include <cctype>
#include <cstdlib>
#include <ctime>
#include <cmath>

using namespace std;

/**  NOTE: COMMENTS IN JAVADOC FORMAT
Rounds a decimal number to a selected number of places
@param x: number to be rounded
@param n: number of places
@function return value: x rounded to n places
*/
double round(double x, int n)  // SEE PROBLEM 3.4 OF THE PROBLEM LIST
{
      return (floor(x * pow(10.0,n) + 0.5) / pow(10.0,n));

}

/**
Generates random number in the interval [n1..n2]
@param n1: lower limit of the interval
@param n2: upper limit of the interval
@function return value: integer in the interval [n1..n2]
*/
int randomBetween(int n1, int n2)
{
      return n1 + rand() % (n2 - n1 + 1);
     //TODO: IMPROVE SO THAT n2 CAN BE LESS THAN n1
}

/**
Selects arithmetic operator and operands to be used for testing user
knowledge
Returns:
@param operation - the operator (+,-,*,/)
@param operand1 - 1st operand; an integer in the interval [1..10]
@param operand2 - 2nd operand; an integer in the interval [1..10]
@function return value: (none)
*/
void selectOperation(char &operation, int &operand1, int &operand2)
{
      switch (randomBetween(1,4))
      {
      case 1: operation = '+'; break;
      case 2: operation = '-'; break;
      case 3: operation = '*'; break;
      case 4: operation = '/'; break;
      }

      operand1 = randomBetween(1,10);
      operand2 = randomBetween(1,10);
}
```

```
/**
Computes result of "operand1 operation operand2"
where operation is (+,-,*,/); assumes that 'operation' is valid
@param operation - the operator; must be (+,-,*,/)
@param operand1 - 1st operand
@param operand2 - 2nd operand
@function return value: the result of the operation
THIS IS AN EXAMPLE OF CODE REUSE - REMEMBER PREVIOUS PROGRAM */
double computeResult(char operation, int operand1, int operand2)
{
      double result;

      switch (operation)
      {
      case '+':
            result = operand1 + operand2;
            break;
      case '-':
            result = operand1 - operand2;
            break;
      case '*':
            result = operand1 * operand2;
            break;
      case '/':
            result = static_cast<double> (operand1) / operand2;
            break;
      }
      return result;
}

/**
Reads user answer to the  "operand1 operation operand2 ? " question
@param operation - the operator; must be (+,-,*,/)
@param operand1 - 1st operand
@param operand2 - 2nd operand
@param result - result of the operation
@function return value: user answer
*/
double readUserAnswer(char operation, int operand1, int operand2)
{
  double answer;
  // int extraInputChars;

  cout << operand1 << " " << operation << " " << operand2 << " ? ";
  while (! (cin>>answer))
  {
    cin.clear();
    cin.ignore(1000,'\n');
    /*
    extraInputChars = cin.rdbuf()->in_avail();
    if (extra_input_chars > 1)
      cin.ignore(extraInputChars,'\n');
    */

    cout << operand1 << " " << operation << " " << operand2 << " ? ";
  }
  /*
  extraInputChars = cin.rdbuf()->in_avail();
  if (extraInputChars > 1)
    cin.ignore(extraInputChars,'\n');
  */
  return answer;
}
```

```
int main()
{
	const int NUM_OPERATIONS = 10;

	int operand1, operand2;
	char operation;
	double result; // result of "operand1 operation operand2"
	double answer; // user answer to the arithmetic operation question
	int numCorrectAnswers = 0; //n.o of correct answers given by user
	int numIncorrectAnswers = 0; //n.o of incorrect answers given by user

	srand((unsigned) time(NULL)); // call srand() just ONCE, in main()

	for (unsigned i = 1; i <= NUM_OPERATIONS; i++)
	{
		selectOperation(operation, operand1, operand2);

		answer = readUserAnswer(operation, operand1, operand2);
		answer = round(answer,1);

		result = computeResult(operation, operand1, operand2);
		result = round(result,1);

		if (answer == result) //alternative: develop function evaluateAnswer()
		{
			numCorrectAnswers++;
			cout << "Correct.\n";
		}
		else
		{
			numIncorrectAnswers++;
			cout << "Incorrect !  Correct answer was " << result << endl;
		}

		cout << endl;
		cout << "Number of correct answers   = "<< numCorrectAnswers << endl;
		cout << "Number of incorrect answers = "<< numIncorrectAnswers << endl << endl;
	}

	return 0;
}
```

```
/*
TESTING USER KNOWLEDGE ABOUT BASIC MATH OPERATIONS
Separating function declaration from function definition
*/
#include <iostream>
#include <iomanip>
#include <cctype>
#include <cstdlib>
#include <ctime>
#include <cmath>

using namespace std;

//FUNCTION DECLARATIONS
/**
Rounds a decimal number to a selected number of places
@param x: number to be rounded
@param n: number of places
@function return value: x rounded to n places
*/
double round(double x, int n);

/**
Generates random number in the interval [n1..n2]
@param n1: lower limit of the interval
@param n2: upper limit of the interval
@function return value: integer in the interval [n1..n2]
*/
int randomBetween(int n1, int n2);

/**
Selects arithmetic operator and operands to be used for testing user
knowledge
Returns:
@param operation - the operator (+,-,*,/)
@param operand1 - 1st operand; an integer in the interval [1..10]
@param operand2 - 2nd operand; an integer in the interval [1..10]
@function return value: (none)
*/
void selectOperation(char &operation, int &operand1, int &operand2);


/**
Computes result of "operand1 operation operand2"
where operation is (+,-,*,/); assumes that 'operation' is valid
@param operation - the operator; must be (+,-,*,/)
@param operand1 - 1st operand
@param operand2 - 2nd operand
@function return value: the result of the operation
*/
double computeResult(char operation, int operand1, int operand2);

/**
Reads user answer to the  "operand1 operation operand2 ? " question
@param operation - the operator; must be (+,-,*,/)
@param operand1 - 1st operand
@param operand2 - 2nd operand
@param result - result of the operation
@function return value: user answer
*/
double readUserAnswer(char operation, int operand1, int operand2);
```

```
//------------------------------------------------------------------------

int main()
{
	const int NUM_OPERATIONS = 10;

	int operand1, operand2;
	char operation;
	double result; // result of "operand1 operation operand2"
	double answer; // user answer to the arithmetic operation question
	int numCorrectAnswers = 0; //n.o of correct answers given by user
	int numIncorrectAnswers = 0; //n.o of incorrect answers given by user

	srand((unsigned int) time(NULL));

	for (int i = 1; i <= NUM_OPERATIONS; i++)
	{
		selectOperation(operation, operand1, operand2);

		answer = readUserAnswer(operation, operand1, operand2);
		answer = round(answer,1);

		result = computeResult(operation, operand1, operand2);
		result = round(result,1);

		if (answer == result)
		{
			numCorrectAnswers++;
			cout << "Correct.\n";
		}
		else
		{
			numIncorrectAnswers++;
			cout << "Incorrect !  Correct answer was " << result <<
endl;
		}
		cout << "Number of correct answers   = "<< numCorrectAnswers <<
endl;
		cout << "Number of incorrect answers = "<< numIncorrectAnswers
<< endl << endl;
	}
	return 0;
}



//------------------------------------------------------------------------
//FUNCTION DEFINITIONS
/**
Rounds a decimal number to a selecetd number of places
@param x: number to be rounded
@param n: number of places
@function return value: x rounded to n places
*/
double round(double x, int n)
{
	return (floor(x * pow((double)10,n) + 0.5) / pow((double)10,n));

}
```

```
/**
Generates random number in the interval [n1..n2]
@param n1: lower limit of the interval
@param n2: upper limit of the interval
@function return value: integer in the interval [n1..n2]
*/
int randomBetween(int n1, int n2)
{
      return n1 + rand() % (n2 - n1 + 1);
}
/**
Selects arithmetic operator and operands to be used for testing user
knowledge
Returns:
@param operation - the operator (+,-,*,/)
@param operand1 - 1st operand; an integer in the interval [1..10]
@param operand2 - 2nd operand; an integer in the interval [1..10]
@function return value: (none)
*/
void selectOperation(char &operation, int &operand1, int &operand2)
{
      switch (randomBetween(1,4))
      {
      case 1: operation = '+'; break;
      case 2: operation = '-'; break;
      case 3: operation = '*'; break;
      case 4: operation = '/'; break;
      }

      operand1 = randomBetween(1,10);
      operand2 = randomBetween(1,10);
}
/**
Computes result of "operand1 operation operand2"
where operation is (+,-,*,/); assumes that 'operation' is valid
@param operation - the operator; must be (+,-,*,/)
@param operand1 - 1st operand
@param operand2 - 2nd operand
@function return value: the result of the operation
*/
double computeResult(char operation, int operand1, int operand2)
{
      double result;

      switch (operation)
      {
      case '+':
            result = operand1 + operand2;
            break;
      case '-':
            result = operand1 - operand2;
            break;
      case '*':
            result = operand1 * operand2;
            break;
      case '/':
            result = static_cast<double> (operand1) / operand2;
            break;
      }
      return result;
}
```

```
/**
Reads user answer to the  "operand1 operation operand2 ? " question
@param operation - the operator; must be (+,-,*,/)
@param operand1 - 1st operand
@param operand2 - 2nd operand
@param result - result of the operation
@function return value: user answer
*/
double readUserAnswer(char operation, int operand1, int operand2)
{
	double answer;

	cout << operand1 << " " << operation << " " << operand2 << " ? ";
	while (! (cin>>answer))
	{
		cin.clear();
		cin.ignore(1000,'\n');
		cout << operand1 << " " << operation << " " << operand2 << " ? ";
	}

	return answer;
}
```

==========

```
============================================================================
/*
TESTING USER KNOWLEDGE ABOUT BASIC MATH OPERATIONS
Modified version: computing the time the user takes to answer
*/

#include <iostream>
#include <iomanip>
#include <cctype>
#include <cstdlib>
#include <ctime>
#include <cmath>

using namespace std;
//----------------------------------------------------------------------------
/**
Rounds a decimal number to a selecetd number of places
@param x: number to be rounded
@param n: number of places
@function return value: x rounded to n places
*/
double round(double x, int n)
{
      return (floor(x * pow(10.0,n) + 0.5) / pow(10.0,n));

}
//----------------------------------------------------------------------------
/**
Generates random number in the interval [n1..n2]
@param n1: lower limit of the interval
@param n2: upper limit of the interval
@function return value: integer in the interval [n1..n2]
*/
int randomBetween(int n1, int n2)
{
      return n1 + rand() % (n2 - n1 + 1);
}
//----------------------------------------------------------------------------
/**
Selects arithmetic operator and operands to be used for testing user
knowledge
Returns:
@param operation - the operator (+,-,*,/)
@param operand1 - 1st operand; an integer in the interval [1..10]
@param operand2 - 2nd operand; an integer in the interval [1..10]
@function return value: (none)
*/
void selectOperation(char &operation, int &operand1, int &operand2)
{
      switch (randomBetween(1,4))
      {
      case 1: operation = '+'; break;
      case 2: operation = '-'; break;
      case 3: operation = '*'; break;
      case 4: operation = '/'; break;
      }

      operand1 = randomBetween(1,10);
      operand2 = randomBetween(1,10);
}
//----------------------------------------------------------------------------
```

```
/**
Computes result of "operand1 operation operand2"
where operation is (+,-,*,/); assumes that 'operation' is valid
@param operation - the operator; must be (+,-,*,/)
@param operand1 - 1st operand
@param operand2 - 2nd operand
@function return value: the result of the operation
*/
double computeResult(char operation, int operand1, int operand2)
{
	double result;

	switch (operation)
	{
	case '+':
		result = operand1 + operand2;
		break;
	case '-':
		result = operand1 - operand2;
		break;
	case '*':
		result = operand1 * operand2;
		break;
	case '/':
		result = static_cast<double> (operand1) / operand2;
		break;
	}
	return result;
}
//-------------------------------------------------------------------------
/**
Reads user answer to the  "operand1 operation operand2 ? " question
@param operation - the operator; must be (+,-,*,/)
@param operand1 - 1st operand
@param operand2 - 2nd operand
@param result - result of the operation
@function return value: user answer
*/
double readUserAnswer(char operation, int operand1, int operand2)
{
	double answer;

	cout << operand1 << " " << operation << " " << operand2 << " ? ";
	while (! (cin>>answer))
	{
		cin.clear();
		cin.ignore(1000,'\n');
		cout << operand1 << " " << operation << " " << operand2 << " ? ";
	}

	cin.ignore(1000,'\n');

	return answer;
}
//-------------------------------------------------------------------------
```

```
int main()
{
	const int NUM_OPERATIONS = 10;

	int operand1, operand2;
	char operation;
	double result; // result of "operand1 operation operand2"
	double answer; // user answer to the arithmetic operation question
	int numCorrectAnswers = 0; //no. of correct answers given by the user
	int numIncorrectAnswers = 0; //no. of incorrect answers

	time_t time1, time2, elapsedTime;

	srand((unsigned int) time(NULL));

	for (int i = 1; i <= NUM_OPERATIONS; i++)
	{
		selectOperation(operation, operand1, operand2);

		time1 = time(NULL);
		answer = readUserAnswer(operation, operand1, operand2);
		time2 = time(NULL);
		elapsedTime = time2 - time1;
		cout << "Elapsed time = " << elapsedTime << endl;

		answer = round(answer,1);

		result = computeResult(operation, operand1, operand2);
		result = round(result,1);

		// TO DO BY STUDENTS:
		// SCORE ANSWER CORRECTNESS AND ELAPSED TIME
		// You are free to choose the scoring rules


		if (answer == result)
		{
			numCorrectAnswers++;
			cout << "Correct.\n";
		}
		else
		{
			numIncorrectAnswers++;
			cout << "Incorrect !  Correct answer was " << result << endl;
		}
		cout << "Number of correct answers   = "<< numCorrectAnswers << endl;
		cout << "Number of incorrect answers = "<< numIncorrectAnswers << endl << endl;
	}

	return 0;
}
```

==========

```
// GLOBAL VARIABLES
// GLOBAL VARIABLES MAY BE "DANGEROUS"
// IF PROGRAMMER IS NOT ALERT ... !!!

#include <iostream>

using namespace std;


int numTimes; // global variable

void func1(void)
{
      for (numTimes=1; numTimes<=10; numTimes++)
            cout << "numTimes = " << numTimes << endl;
     // should not have used 'numTimes' for loop control ... WHY ?
}

void func2(int numTimes)
{
      int i; // local variable
      for (i=1; i<=numTimes; i++)
            cout << "i = " << i << endl;
}

int main(void)
{

      numTimes = 20;

      func1();
      func2(numTimes);

      return 0;
}
```

==========

INSTEAD OF USING FUNCTION **PARAMETERS** TO PASS THEM INPUT VALUES
ONE **COULD USE GLOBAL VALUES TO PASS** THOSE VALUES ...

WHY **SHOULD** THIS **NOT BE DONE** ?

CONSIDER THE ABOVE SHOWN EXAMPLE.
CONSIDER THE CASE THAT YOU WANT TO REUSE THE FUNCTION IN OTHER PROGRAMS.

```
// SCOPE & LIFETIME (/PERSISTENCE) OF VARIABLES

// GLOBAL AND LOCAL VARIABLES WITH THE SAME NAME. WHICH ONE IS SEEN ?

// LOCAL VARIABLES MUST BE INITIALIZED

// VALUE PARAMETERS ARE LOCAL VARIABLES

// DEFAULT FUNCTION ARGUMENTS

#include <iostream>

using namespace std;

int numTimes;  // global variable

void func1()
{
      for (int i=1; i<=numTimes; i++)
            cout << "func1: i = "<< i << endl;
      cout << endl;

}

void func2()
{
      int numTimes;  //BE CAREFUL !!! uninitialized local variable
                                       // VISUAL STUDIO 2013 will not compile ...
                                       // some compilers may only give a warning …
     // cout << "numTimes = " << numTimes << endl;

      for (int i=1; i<=numTimes; i++)
            cout << "func2: i = "<< i << endl;
      cout << endl;
}

void func3(int numTimes=5) //default function argument
{
      for (int i=1; i<=numTimes; i++)
            cout << "func3: i = "<< i << endl;
      cout << endl;

}

int main()
{
     // cout << "numTimes = " << numTimes << endl;

      numTimes = 10;

      func1();    //UNCOMMENT ONE STATEMENT AT A TIME
      //func2();
      //func3();
      //func3(7);

      return 0;
}
```

==========

- **Static storage**
  - Is storage that exists throughout the lifetime of a program.
  - There are two ways to make a variable static.
    - One is to define it externally, outside a function (global variable).
    - The other is to use the keyword **`static`** when declaring a variable (see next example)
  - A static variable declared inside a function, although having existence during the lifetime of the program is visible only inside the function.

==========

```
// STATIC LOCAL VARIABLES

#include <iostream>

using namespace std;

int getTicketNumber(void)
{
      static int ticketNum = 0; // initialized only once, at program startup

      ticketNum++;              // OR ...

      return ticketNum;         // return ++ticketNum;
}

int main(void)
{

      int i;

      for (i=1; i <= 10; i++)
            cout << "ticket no. = " << getTicketNumber() << endl;

      return 0;
}
```

==========

- **Recursive functions**
  - A **function** definition that includes a call to itself is said to be **recursive**.
  - General outline of a successful recursive function definition is as follows:
    - One or more cases in which the function accomplishes its task by using one or more recursive calls to accomplish one or more smaller versions of the task.
    - One or more cases in which the function accomplishes its task without the use of any recursive calls. These cases without any recursive calls are called base cases or stopping cases.
    - Pitfall (be careful):
      - If every recursive call produces another recursive call, then a call to the function will, in theory, run forever.
      - This is called infinite recursion.
      - In practice, such a function will typically run until the computer runs out of resources and the program terminates abnormally.

  - Example:

```
// A function that writes a number, vertically,
// one digit on each line
void writeVertical(unsigned int n)
{
  if (n < 10)    // BASE CASE
  {
    cout << n << endl;
  }
  else //n is two or more digits long:
  {
    writeVertical(n / 10); // RECURSIVE CALL
    cout << (n % 10) << endl;
  }
}
```

  - TO DO ON THE WHITE BOARD (/ BY STUDENTS):
    - TRACE THE EXECUTION OF THE CALL: writeVertical(123)

```
// RECURSIVE FUNCTIONS
#include <iostream>
#include <iomanip>
using namespace std;

unsigned long factorialIte(unsigned n)  // iterative version
{
      int f = 1;

      for (unsigned i = n; i >= 2; i--)
            f = f * i;

      return f;
}

unsigned long factorialRec1(unsigned n)
// recursive version; a bad example of recursion... why ?
{

      if (n == 0 || n == 1)
            return 1;
      else
      {
            unsigned long f;

            f = n * factorialRec1(n-1);
            return f;
      }
}

unsigned long long factorialRec2(unsigned int n)
// recursive version; a bad example of recursion... why ?
{

      if (n == 0 || n == 1)
            return 1;
      else
            return (n * factorialRec2(n-1));
}


int main(void)
{
      unsigned i;

      cout << " i          factorialIte         factorialRec1
factorialRec2\n";
      cout << "--    ------------------   ------------------   -----------
--------\n";
      for (i=0; i <= 20; i++)
            cout << setw(2) << i << " - " <<
            setw(19) << factorialIte(i) << "   " <<
            setw(19) << factorialRec1(i) << "   " <<
            setw(19) << factorialRec2(i) << endl;

      //cout << ULLONG_MAX << endl; //=> #include <climits>
      return 0;
}
```

==========

Other examples: Fibonacci numbers, GCD, flood fill, Hanoi towers, 8 Queens, ...

```cpp
// EFFICIENCY OF RECURSION
// Computing the n-th element of the Fibonacci sequence
// F(1)=1; F2(2)=1; F(n)=F(n-1)+F(n-2)

#include <iostream>
#include <ctime>
#include <ratio>
#include <chrono>

using namespace std;
using namespace std::chrono;

int fib_recursive(int n)
{
  if (n <= 2) return 1;
  else return fib_recursive(n - 1) + fib_recursive(n - 2);
}

int fib_iterative(int n)
{
  if (n <= 2) return 1;
  int fold1 = 1;
  int fold2 = 1;
  int fnew;
  for (int i = 3; i <= n; i++)
  {
    fnew = fold1 + fold2;
    fold1 = fold2;
    fold2 = fnew;
  }
  return fnew;
}

int main()
{
  int num = 1;

  high_resolution_clock::time_point t1, t2;  // FOR NOW, USE THE TIME MEASUREMENT STATEMENTS
  duration<double> time_elapsed;             // AS A "COOKING RECIPE"

  cout << "Number (1..45) ? "; cin >> num;
  t1 = high_resolution_clock::now();
  cout << "Fibonacci recursive ... (computing)\n";
  cout << "Fibonacci(" << num << ") = " << fib_recursive(num) << endl;
  t2 = high_resolution_clock::now();
  time_elapsed = duration_cast<std::chrono::microseconds>(t2 - t1);
  cout << "fib_recursive - time = " << time_elapsed.count() << endl << endl;

  t1 = high_resolution_clock::now();
  cout << "Fibonacci iterative ... (computing)\n";
  cout << "Fibonacci(" << num << ") = " << fib_iterative(num) << endl;
  t2 = high_resolution_clock::now();
  time_elapsed = duration_cast<std::chrono::microseconds>(t2 - t1);
  cout << "fib_iterative - time = " << time_elapsed.count() << endl << endl;

  return 0;
}
```

- **The efficiency of recursion**
  - Although recursion can be a powerful tool to implement complex algorithms, it can lead to algorithms that perform poorly (execute example above).
  - Each recursive call generally requires a memory address to be pushed to the **stack** (so that later the program could return to that point) as well as the function parameters and local variables
  - Sometimes, the iterative and recursive solution have similar performance.
  - There are some problems that are much easier to solve recursively than iteratively.

```
// A PROGRAM THAT EVALUATES ARITHMETIC EXPRESSIONS
// Examples:
// 2+3*5          NOTE: DOES NOT DEAL WITH 2 + 3 * 5 !!!
// (2+3)*5        CHALLENGE: IMPROVE IT TO DEAL WITH THESE EXPRESSIONS
// (5-3)*(2-3*(1-5))
// Source: Big C++ book

#include <iostream>
#include <cctype>

using namespace std;

double term_value(); //WHY ARE THESE FUNCTION DECLARATIONS NEEDED IN THIS CASE ?
double factor_value();

/*
Evaluates the next expression found in cin
Returns the value of the expression.
*/
double expression_value()
{
  double result = term_value();
  bool more = true;
  while (more)
  {
    char op = cin.peek();
    if (op == '+' || op == '-')
    {
      cin.get();
      double value = term_value();
      if (op == '+') result = result + value;
      else result = result - value;
    }
    else more = false;
  }
  return result;
}

/*
Evaluates the next term found in cin
Returns the value of the term.
*/
double term_value()
{
  double result = factor_value();
  bool more = true;
  while (more)
  {
    char op = cin.peek();
    if (op == '*' || op == '/')
    {
      cin.get();
      double value = factor_value();
      if (op == '*') result = result * value;
      else result = result / value;
    }
    else more = false;
  }
  return result;
}
```

```
/*
Evaluates the next factor found in cin
Returns the value of the factor.
*/
double factor_value()
{
  double result = 0;
  char c = cin.peek();
  if (c == '(')
  {
    cin.get(); // read "("
    result = expression_value();
    cin.get(); // read ")"
  }
  else // assemble number value from digits
  {
    while (isdigit(c))
    {
      result = 10 * result + c - '0';
      cin.get();
      c = cin.peek();
    }
  }
  return result;
}

int main()
{
  cout << "Enter an expression: ";
  cout << expression_value() << endl;
  return 0;
}
```

NOTE:

- This is an example of **mutual recursion** -
  a set of cooperating functions call each other in a recursive fashion

- To compute the value of an expression, we implement three functions **`expression_value()`**, **`term_value()`**, and **`factor_value()`**.
- The **`expression_value()`** function
  - calls **`term_value()`**,
  - checks to see if the next input is **+** or **-**, and
  - if so calls **`term_value()`** again to add or subtract the next term.
- The **`term_value()`** function
  - calls **`factor_value()`** in the same way, multiplying or dividing the factor values.
- The **`factor_value()`** function
  - checks whether the next input is '**(**' or a digit, calling either **`expression_value()`** recursively or returning the value of the digit / number (sequence of digits).
- The termination of the recursion is ensured
  because the **`expression_value()`** function consumes some of the input characters, ensuring that the next time it is called on a shorter expression.

(see chapter on Recursion of the Big C++ book, for in depth explanation)

- **Function overloading**
  - **In C++**, if you have two or more function definitions for the same function name, that is called **overloading**.
  - When you overload a function name, the function definitions must have:
    - different numbers of formal parameters or
    - some formal parameters of different types.
  - When there is a function call, the compiler uses the function definition whose number of formal parameters and types of formal parameters match the arguments in the function call.

```
================================================================================
/*
FUNCTION OVERLOADING

Function sum() is overloaded - different types of parameters
*/

#include <iostream>

using namespace std;


int sum(int x, int y)
{
  //cout << "sum1 was called\n";
  return x+y;
}

double sum(int x, double y)
{
  //cout << "sum2 was called\n";
  return x+y;
}

double sum(double x, double y)   //TO DO: comment the other 2 functions
{                                //       and recompile ...
  //cout << "sum3 was called\n"; //
  return x+y;                     //After that, comment the last 2 functions
}                                //       and recompile ...
                                 //Surprised ...?! Explain.

int main()
{
  cout << sum(2,3) << endl;
  cout << sum(2.5,3.5) << endl;
  cout << sum(2,3.5) << endl;

  return 0;
}
```

```
/*
FUNCTION OVERLOADING

Function sortAscending() is overloaded - different no. of parameters
*/

#include <iostream>

using namespace std;


void sortAscending(int &x, int &y)
{
	if (x > y)
	{
		int tmp = x;
		x = y;
		y = tmp;
	}
	//return;  // not necessary
}

void sortAscending(int &x, int &y, int &z)
{
	int max, min, med;

	if (x <= y && x <= z)
	{
		min = x;
		if (y <= z)
		{
			med = y;
			max = z;
		}
		else
		{
			med = z;
			max = y;
		}
	}
	else if (y <= x && y <= z)
	{
		min = y;
		if (x <= z)
		{
			med = x;
			max = z;
		}
		else
		{
			med = z;
			max = x;
		}
	}
```

```
	else
	{
		min = z;
		if (x <= y)
		{
			med = x;
			max = y;
		}
		else
		{
			med = y;
			max = x;
		}

	}

	x = min;
	y = med;
	z = max;

}

int main()
{

	int a, b, c;

	cout << "input 2 numbers: a b ? ";
	cin >> a >> b;
	sortAscending(a,b);
	cout << "sorted numbers: " << a << " " << b << " " << endl << endl;

	cout << "input 3 numbers: a b c ? ";
	cin >> a >> b >> c;
	sortAscending(a,b,c);
	cout << "sorted numbers: " << a << " " << b << " " << c << endl <<
endl;

	return 0;
}




/* ALTERNATIVE
void sortAscending(int &x, int &y, int &z)
{
	sortAscending(x,y);  // Is this a recursive call ...?
	sortAscending(y,z);
	sortAscending(x,y);
}
*/
```

```
//=========================================================================
/*
FUNCTIONS AS PARAMETERS (calculating the integral using Simpson's rule)
see problem 3.9 of the problem list
*/
#include <iostream>
#include <cmath>
using namespace std;

double func(double x)
{
  double y;

  y = x * x;

  // test with other functions; ex:
  //y = x;
  //y = sqrt( 4 - x * x ); // NOTE:=> parameters for integrateTR -> a=0; b=2

  return y;
}

double integrateTR(double f(double), double a, double b, int n)
{
      double dx; // width of the sub-interval
      double x1, x2, y1, y2; // limits of the sub-interval
      double area; // limits and area of the sub-interval
      double totalArea=0.0; // integral sum

      int i;

      dx = (b-a) / n;

      x1 = a;
      x2 = x1 + dx;

      for (i=1; i<=n; i++)  // WHY USE A COUNTED LOOP?
      {
            y1 = f(x1);
            y2 = f(x2);
            area = dx * (y1 + y2) /2;
            totalArea = totalArea + area;
            x1 = x1 + dx;
            x2 = x2 + dx;
      }

      return totalArea;
}
```

```
int main(void)
{
      double a, b; // limits of the interval
      int n;       // n.o of sub-intervals to be used

      cout << "Integrate y = x*x in the interval [a,b]\n";
      // change this message according to y=... in function func() above
      //  :-(  => recompile program; do you see another alternative ?
      //  func() could return a string indicating the function that it implements ...
      //  strings are coming next ...

      cout << "a b ? ";
      cin >> a >> b;

      cout << endl;

      // repeat 10x the calculation,
      // using different sub-intervals in each calculation
      for (int i=1; i<=10; i++)
      {
        n = 10*i; //calculate with 10, 20, 30, ..., 100 sub-intervals
       // n = (int) pow(2.0,i); //calculate with, 2^1, 2^2, 2^3, ..., 2^10 sub-intervals

        cout << "n =  " << n << endl;
        cout << "integral( a=" << a << ", b=" << b << ", n=" << n << ") = "
             << integrateTR(func,a,b,n) << endl;
      }
}
```