

PROGRAMMING

MIEIC – 2017/2018

LECTURE MATERIAL

C++ BASICS:

PROGRAM STRUCTURE, TYPES, VARIABLES, EXPRESSIONS, ..., CODING STYLE

Program structure

- A program is basically a collection of functions, one of which must be named **main()**.
- Program execution begins with **main()**.
- If necessary, **main()** can call other functions.

```
/*  
FILE      : p001.cpp                ← OPENNING DOCUMENTATION / COMMENTS  
DATE      : 2018/02/06  
AUTHOR    : JAS  
PROGRAM PURPOSE:  
- To salute the user, on the screen  
*/  
  
#include <iostream>    // ← COMPILER DIRECTIVE(S)  
                        //   FOR INSERTING THE CONTENTS OF A HEADER FILE  
  
int main()             // ← EXECUTION STARTS HERE  
{  
    std::cout << "Hello !" << std::endl; // PROGRAM STATEMENT(S)  
    return 0;          // VALUE 0 IS RETURNED TO THE COMMAND INTERPRETER  
                        // 0 is the 'exit code' of the program (SEE LATER)  
                        // to see 'exit code' on the console: > echo %ERRORLEVEL%  
}
```

```
/*  
FILE      : p002.c                // ← The same code in "pure C"  
...  
*/  
#include <stdio.h>  
  
int main()  
{  
    printf("Hello!\n");  
    return 0;  
}
```

Notes

- C is **case-sensitive**.
- Each **statement** must be terminated by a **semicolon**.
- `std::cout` is a C++ **object** of **class** `ostream` that represents the **standard output stream** oriented to characters (of type `char`). It corresponds to the C stream `stdout`.

Compiler directives

- The most commonly used compiler directives are the **#include directives**.
- They are instructions for the preprocessor to insert the contents of the specified file(s) at this point.
- **#include <filename>** - for standard libraries
- **#include "filename"** - for programmer defined libraries
- The files included are called header files.
 - In C language, header file names usually end with **.h**
 - In C++, you can still use **.h** extension; some people prefer **.hpp**
- These files contain the **declarations** of functions, constants, variables.
- *The use of programmer defined libraries will be introduced later.*

```
/* ..... for simplicity,  
comments will be suppressed in most of the examples  
they will be made verbally */
```

```
#include <iostream>
```

```
using std::cout; //avoids the need of writing std:... in the statements below  
using std::endl;
```

```
int main()  
{  
    cout << "Hello friends !" << endl;  
    return 0;  
}
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()  
{  
    cout << "Hello friends !\n"; // '\n' can be used instead of endl  
    return 0;  
}
```

Namespaces

- Are a means of organizing typenames, variables and functions so as to avoid name conflicts.
- Usually we want to use the standard libraries that are in the **namespace** named **std**.
- This theme will be treated later.

Comments

- Two kinds of comments are allowed in C/C++
- `// comment` - continues to end of line
- `/*
comments
*/` - may extend over several lines

Standard libraries

- **iostream** – basic interactive input/output (I/O) / **stdio.h**, in C
- **iomanip** – format manipulators
- **fstream** – file I/O / **stdio.h**, in C
- **string** – standard C++ strings
- **cstring** – C strings type / **string.h**, in C
- **cmath** – math functions / **math.h**, in C
- **climits** – various properties of the various variable types / **limits.h**, in C
- **cmath** – various platform-dependent constants related to floating point values / **float.h**, in C
- ... and many other

Identifiers and Keywords

- **Identifiers** – a letter (or underscore) followed by any number of letters, digits or underscores.
 - identifiers starting with an underscore are usually reserved for a special purpose
- **C/C++ is case sensitive**
 - **sum**, **Sum**, and **SUM** are different identifiers
- Identifiers may not be any of the language keywords:
 - some examples of **keywords**:
char, double, float, int, unsigned, long, short, bool, true, false, const,
if, else, switch, case, default,
while, do, for, break, continue, goto, return,
class, typename, friend, operator, public, protected, auto, ...
 - For a complete list consult a C++ manual.

- Recommended naming conventions (example):
 - `string bookTitle ;` OR
 - `string book_title;`

Coding style

- Take a look at the links available at the course page in Moodle; many others are available in the web.
 - Example: <https://users.ece.cmu.edu/~eno/coding/CppCodingStandard.html>
- Just "a couple" of recommendations:
 - choose adequate / meaningful identifier names
 - place a comment with each variable declaration explaining the purpose of the variable
 - write one statement in each line
 - indent the code
 - complex sections of code and any other parts of the program that need some explanation should have comments just ahead of them or embedded in them.

```

/*
FILE   : p002.cpp
DATE   : 2018/02/06
AUTHOR : JAS
PROGRAM PURPOSE:
- Read 2 integers and
- compute their sum, difference, product and quotient
BAD CODING STYLE.
*/

#include <iostream>
using namespace std;
int main() {
int x, y, s, d, p; double q; // VARIABLE DECLARATIONS
cout << "x ? "; cin >> x; cout << "y ? "; cin >> y; s = x + y; d = x - y; p
= x * y; q = x / y; cout << "s = " << s << endl << "d = " << d << endl <<
"p = " << p << endl << "q = " << q << endl; return 0;}

```

Variables

- C/C++ is a strongly-typed language, and requires every variable to be declared with its type before its first use.
- This informs the compiler the size to reserve in memory for the variable and how to interpret its value.
- NOTE: Starting with **C++11**, the **auto** keyword can be used in place of the **variable type** to tell the compiler to infer the type of the variable from the type of the initializer; this is called **type inference** or type deduction (*see later*).

```

/*
...
A BETTER CODING STYLE.

BUT SOME PROBLEMS WITH quotient
TEST PROGRAM WITH PAIRS (4,2) (100,5) (357,7) ... AND (1,3)
*/

#include <iostream>

using namespace std;

int main()
{
    int operand1, operand2; // input operands
    int sum;                // sum of input operands
    int difference;         // difference of operands
    int product;            // product of operands
    double quotient;        // quotient of operands

    // input 2 integers
    cout << "operand1 ? "; // C-style
    cin >> operand1;        // scanf("%d",&operand1) for input
    cout << "operand2 ? "; // printf("operand 2 ? ") for output
    cin >> operand2;

    //compute their sum, difference, product and quotient
    sum = operand1 + operand2;
    difference = operand1 - operand2;
    product = operand1 * operand2;
    quotient = operand1 / operand2;

    //show results
    cout << "      sum = " << sum << endl;
    cout << "difference = " << difference << endl;
    cout << "      product = " << product << endl;
    cout << "      quotient = " << quotient << endl;

    return 0;
}

```

NOTE THE RELEVANCE OF TESTING ADEQUATELY YOUR PROGRAM

Fundamental data types

- Variables must be declared before they are used.
- The type of the variable must be chosen.
- **Integers:** `int`
 - integer variations:
 - `short` (OR `short int`), `long` (OR `long int`),
 - `unsigned` (OR `unsigned int`), `long long`
- **Reals:** `float`, `double`, `long double`
- **Characters:** `char` (also has some variations)
- **Booleans:** `bool` (logical values: `true` OR `false`)
- **Void type:** `void`

```

/*
Solving the quotient of 2 integers problem
*/

#include <iostream>

using namespace std;

int main()
{
    int operand1, operand2; // input operands
    int sum;                // sum of input operands
    int difference;         // difference of ...
    int product;            // ...
    double quotient;        // ...

    // input 2 integers
    cout << "operand1 operand2 ? ";
    cin >> operand1 >> operand2;

    //compute their sum, difference, product and quotient
    sum = operand1 + operand2;
    difference = operand1 - operand2;
    product = operand1 * operand2;
    quotient = static_cast<double>(operand1) / operand2;
    //OR
    // quotient = (double) operand1 / operand2; // C-style

    //show results
    cout << "    sum = " << sum << endl;
    cout << "difference = " << difference << endl;
    cout << "    product = " << product << endl;
    cout << "    quotient = " << quotient << endl;

    return 0;
}

```

Declarations

- Variables declarations have the forms:
 - type variable_name*: `int sum;`
 - type list_of_variables*: `int operand1, operand2;`
- VERY IMPORTANT NOTE:**
 When the variables in the above example are declared, they have an undetermined value until they are assigned a value for the first time.
- But it is possible for a variable to have a specific value from the moment it is declared; this is called **variable initialization**:
 - `int x = 0, y = 1;` // C-style initialization
 - other forms of initialization are possible:
 - `int x(0);` // C++ constructor-style initialization
 - `int x{0};` // C++11 uniform initialization

```

/* ...
The results of all the 4 operations are always computed.
Usually one only wants the result of one of the operations...
*/
#include <iostream>

using namespace std;

int main()
{
    int operand1, operand2; // input operands
    // NOTE THE DIFFERENCE FROM PREVIOUS EXAMPLE: other var.s not declared

    // input 2 integers
    cout << "operand1 ? ";
    cin >> operand1;
    cout << "operand2 ? ";
    cin >> operand2;

    //compute and show results
    cout << "      sum = " << operand1 + operand2 << endl;
    cout << "difference = " << operand1 - operand2 << endl;
    cout << "  product = " << operand1 * operand2 << endl;
    cout << "  quotient = " << operand1 / operand2 << endl;

    return 0;
}

```

Input / Output (I/O) expressions

- I/O is carried out using **streams** that connect the program and I/O devices (keyboard, screen) or files.
- **Input expressions**
 - input / extraction operator: `>>`
 - The expression `instream >> variable`
 - extracts a value of the type of *variable* (if possible) from *instream*
 - stores the value in *variable*
 - **returns *instream*** as its result (if successful, else 0)
 - This last property, along the left-associativity (*see later*) of `>>` makes it possible to chain input expressions:
 - `instream >> variable1 >> variable2 >> >> variableN;`
- **Output expressions**
 - output / insertion operator: `<<`
 - The expression `outstream << value`
 - inserts *value* into *instream*;
 - *value* may be a constant, variable or the result of an expression
 - **returns *outstream*** as its result
 - `outstream << value1 << value2 <<<< valueN;` is also possible

Literals / Constants

- **Integers:**
 - use the usual decimal representation: 25, -3, 1000
 - octal numbers begin with 0 (zero)
 - hexadecimal numbers begin with 0x
 - suffixes may be appended to specify the integer type:
 - **u or U** for **unsigned**: 75u
 - **l or L**, for **long**: 1000000000L
 - **ll or LL**, for **long long**
- **Reals:**
 - use the usual decimal representation and **e** or **E**: 19.5, 2e-1, 5.3E3
- **Characters:**
 - single chars are enclosed in single quotes: 'A', 'd', '8', ' '
 - escape sequences are used for special character constants:
 - **'\n'** : newline
 - **'\''** : single quote
 - **'\\'** : backslash
 - ...
- **Strings:**
 - strings are enclosed in double quotes: "Programming course"

=====

Operators

- **Assignment operator :**
 - `int x, y;`
 - `x = 5;`
 - `y = x;`
 - `y = x = 5;` // x equals to 5 and the result of (x = 5) is 5, so ...?
 - `y = 2 + (x = 3);` // POSSIBLE!!! BUT NOT RECOMMENDED ...
 - is equivalent to:
 - `x = 3;`
 - `y = 2 + 3;` // (x = 3) evaluates to 3
- **Arithmetic operators:**
 - `+` - addition
 - `-` - subtraction
 - `*` - multiplication
 - `/` - division (NOTE: be careful when both operands are integers!)
 - `%` - modulo (rest of integer division)
- **Relational and comparison operators:**
 - `==` - equal to (NOTE THIS! BE CAREFUL!!!)
 - `!=` - not equal to
 - `>` - greater than
 - `<` - less than
 - `>=` - greater than or equal to
 - `<=` - less than or equal to
- **Logical operators:**
 - `!` - Boolean NOT
 - `&&` - Boolean AND
 - `||` - Boolean OR
- **To be introduced latter:**
 - Compound assignment :
 - `+=, -=, *=, /=, %=, &=, ^=, |=, ...`
 - Increment and decrement (by one):
 - `++, --`
 - Conditional ternary operator:
 - `? :`
 - Comma operator:
 - `,`
 - Bitwise operators (*not to be confused with logical operators*):
 - `&, |, ^, ~, <<, >>`
 - Explicit type casting operator
 - `sizeof`

THERE ARE OTHER OPERATORS (SEE NEXT PAGE)

Precedence of operators

Level	Precedence group	Operator	Description	Grouping
1	Scope	::	scope qualifier	Left-to-right
2	Postfix (unary)	++ --	postfix increment / decrement	Left-to-right
		()	functional forms	
		[]	subscript	
		. ->	member access	
3	Prefix (unary)	++ --	prefix increment / decrement	Right-to-left
		~ !	bitwise NOT / logical NOT	
		+ -	unary prefix	
		& *	reference / dereference	
		new delete	allocation / deallocation	
		sizeof	parameter pack	
		(type)	C-style type-casting	
4	Pointer-to-member	. * ->*	access pointer	Left-to-right
5	Arithmetic: scaling	* / %	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	+ -	addition, subtraction	Left-to-right
7	Bitwise shift	<< >>	shift left, shift right	Left-to-right
8	Relational	< > <= >=	comparison operators	Left-to-right
9	Equality	== !=	equality / inequality	Left-to-right
10	And	&	bitwise AND	Left-to-right
11	Exclusive or	^	bitwise XOR	Left-to-right
12	Inclusive or		bitwise OR	Left-to-right
13	Conjunction	&&	logical AND	Left-to-right
14	Disjunction		logical OR	Left-to-right
15	Assignment-level expressions	= *= /= %*= += -=>>= <<= &= ^= =	assignment / compound assignment	Right-to-left
		?:	conditional operator	
16	Sequencing	,	comma separator	Left-to-right

source: <http://www.cplusplus.com/doc/tutorial/operators/>

- Examples:
 - `x = 2 + 3 * 4;` `// x = ...?`
 - `y = (2 + 3) * 4;` `// y = ...?`
 - `z = y = x + 10;` `// how is this evaluated ?`
- When an expression has two operators with the same precedence level, grouping determines which one is evaluated first: either left-to-right or right-to-left.
- Enclosing all sub-statements in parentheses (even those unnecessary because of their precedence) improves code readability.

CONTROL STRUCTURES

Control structures

- Are language constructs that allow a programmer to control the flow of execution through the program.
- There are 3 main categories:
 - **sequence**
 - **selection**
 - **repetition**

Sequence

- Sequential execution is implemented by **compound statements** (or **blocks**)
- They consist of statements enclosed between curly braces: { and } .

```
{  
    statement1    ← REMEMBER: statements end with ;  
    statement2  
    ...  
    statementN  
}
```

- Example:

```
{  
    cout << "X and Y values ? ";  
    cin >> x >> y;  
    cout << " x + y = " << x + y;  
}
```

Selection

- Selective execution is implemented by :
 - **if** statements
 - **if ... else** statements
 - **switch ... case** statements

- **if statement**

if (*boolean_expression*)
statement

- The boolean expression must be enclosed in parenthesis.
- The statement may be a compound statement.
- Example 1:

```
if (x > 0)
    cout << "X is positive \n";
```

- Example 2 (what is the result of this compound statement?):

```
if (x > y)
{
    int t = y; // t only exists temporarily, inside this block
    y = x;
    x = t;
}
```

- NOTE 1: in C/C++, the value of any variable or expression may be interpreted as true, if it is different from zero, or false, if it is equal to zero.

```
if (x)    // if x is different from zero
{        // to improve readability do write (x!=0)
    ...
}
```

- NOTE 2: a very common error (not detected by the compiler)

```
if (x = 10)
{
    ...
}
```

- will assign 10 to **x**
- and the value of **(x = 10)** is 10 (**true**)
- so... **BE CAREFUL!**

- NOTE 3: if written in this way the compiler will detect the error 😊

```
if (10 = x)
{
    ...
}
```

- **if ... else statement**

```
if (boolean_expression)  
    statement1  
else  
    statement2
```

- Example 1:

```
if (x==y)  
    cout << "x is equal to y \n"; // note the semicolon  
else  
    cout << "x is not equal to y \n";
```

- **switch ... case statement**

```
switch (integer_expression) // NOTE: must evaluate to an integer  
{  
    case_list_1:  
        statement_list_1  
        break; // usually a break OR return is used after each statement list  
    case_list_2:  
        statement_list_2  
        break;  
    .  
    .  
    .  
    default:  
        default_group_of_statements  
}
```

- Each *case_list_i* is made up of
 - *case constant_value*:
OR
case constant_value_1: ... : *case constant_valueN*:
- **switch** can only test for equality.
- No two **case** constants can have the same value
- The **break** statement causes the execution of the program to continue after the **switch** statement.
- The **default** part is optional;
it only is executed if the value of the *integer_expression* is in no *case_list_i*

```
/*  
A NOT VERY GOOD SOLUTION. WHY ?
```

```
TEST PROGRAM USING:
```

```
2+3, 2 /3, 5 * 10 , ... 2 3 4 ...OR a + 2 (!!!)  
*/
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()  
{
```

```
    double operand1, operand2; // input operands  
    double sum; // sum of input operands  
    double difference; // difference of ...  
    double product; // ...  
    double quotient; // ...  
    char operation; // QUESTION: WHY "OPERATION", NOT "OPERATOR"  
                    // consult the list of reserved identifiers / keywords
```

```
    // input 2 numbers
```

```
    cout << "x op y ? ";
```

```
    cin >> operand1 >> operation >> operand2;
```

```
    //compute their sum, difference, product or quotient
```

```
    if (operation == '+')
```

```
        sum = operand1 + operand2;
```

```
    if (operation == '-')
```

```
        difference = operand1 - operand2;
```

```
    if (operation == '*')
```

```
        product = operand1 * operand2;
```

```
    if (operation == '/')
```

```
        quotient = operand1 / operand2; // BOTH OPERANDS ARE double 😊
```

```
    //show results
```

```
    cout << operand1 << ' ' << operation << ' ' << operand2 << " = ";
```

```
    if (operation == '+')
```

```
        cout << sum;
```

```
    if (operation == '-')
```

```
        cout << difference;
```

```
    if (operation == '*')
```

```
        cout << product;
```

```
    if (operation == '/')
```

```
        cout << quotient;
```

```
    // TRY THE FOLLOWING with and without parenthesis in (quotient = ...)
```

```
    // if (operation == '/')
```

```
    //     cout << (quotient = operand1 / operand2);
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```

```
=====
```

```

/*
1) A LITTLE BETTER ... WHY ?
2) Try with an invalid operation (ex: X instead of *)
*/

#include <iostream>

using namespace std;

int main()
{
    double operand1, operand2; // input operands
    double sum; // sum of input operands
    double difference; // difference of ...
    double product; // ...
    double quotient; // ...
    char operation; // operation; possible values: + - * /

    // input 2 numbers
    cout << "x op y ? ";
    cin >> operand1 >> operation >> operand2;

    //compute their sum, difference, product or quotient
    if (operation == '+')
        sum = operand1 + operand2; //NOTE the semicolon
    else if (operation == '-')
        difference = operand1 - operand2;
    else if (operation == '*')
        product = operand1 * operand2;
    else if (operation == '/')
        quotient = operand1 / operand2;

    //show results
    cout << operand1 << ' ' << operation << ' ' << operand2 << " = ";
    if (operation == '+')
        cout << sum;
    else if (operation == '-')
        cout << difference;
    else if (operation == '*')
        cout << product;
    else if (operation == '/')
        cout << quotient;

    cout << endl;

    return 0;
}

// A little better solution but still bad ...
// ... for several reasons.
// See next solutions.

```

```
=====
```

```

/*
AN EVEN BETTER SOLUTION. WHY ?

GOOD CODING STYLES:
- TRY TO KEEP INPUT, PROCESSING AND OUTPUT SEPARATED FROM EACH OTHER
- USE CONSTANTS INSTEAD OF "MAGIC NUMBERS"
*/
#include <iostream>
#include <iomanip> //needed for stream manipulators: fixed, setprecision

using namespace std;

int main()
{
    const unsigned int RESULT_PRECISION = 3; //for const's use uppercase

    double operand1, operand2; // input operands
    char operation; // operation; possible values: + - * /
    double result; // result of "operand1 operation operand2"
    bool validOperation = false; // operation is not + - * or /

    // input 2 numbers
    cout << "x op y ? ";
    cin >> operand1 >> operation >> operand2;

    // compute result if operation is valid
    if (operation == '+' || operation == '-' || operation == '*' ||
operation == '/') // TO DO: remember operator precedence
    {
        //compute their sum, difference, product or quotient
        if (operation == '+')
            result = operand1 + operand2;
        else if (operation == '-')
            result = operand1 - operand2;
        else if (operation == '*')
            result = operand1 * operand2;
        else if (operation == '/')
            result = operand1 / operand2;
        validOperation = true;
    }

    //show result or invalid input message
    if (validOperation) // equivalent to: if (validOperation == true)
    {
        cout << operand1 << ' ' << operation << ' ' << operand2 << " =
";
        cout << fixed << setprecision(RESULT_PRECISION) << result <<
endl;
    } //TO DO: search for other stream manipulators
    else
        cerr << "Invalid operation !\n"; // NOTE:stream for error output

    return 0;
}
=====

```

TO DO BY STUDENTS: search how to reset precision to default values

```

const std::streamsize oldPrecision = cin.precision();
cout << fixed << setprecision(3) << x << endl << setprecision(oldPrecision) << x;

```



```

/*
USING THE SWITCH STATEMENT
*/

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    const unsigned RESULT_PRECISION = 3;

    double operand1, operand2; // input operands
    char operation; // operation; possible values: + - * /
    double result; // result of "operand1 operation operand2"
    bool validOperation = true; // operation is + - * or /

    // input 2 numbers
    cout << "x op y ? ";
    cin >> operand1 >> operation >> operand2;

    // compute result if operation is valid
    switch (operation)
    {
        case '+':
            result = operand1 + operand2;
            break;
        case '-':
            result = operand1 - operand2;
            break;
        case '*':
            result = operand1 * operand2;
            break;
        case '/':
            result = operand1 / operand2;
            break;
        default:
            validOperation = false;
    }

    //show result or invalid input message
    if (validOperation)
    {
        cout << operand1 << ' ' << operation << ' ' << operand2 << " =
";
        cout << fixed << setprecision(RESULT_PRECISION) << result <<
endl;
    }
    else
        cerr << "Invalid operation !\n";

    return 0;
}
=====

```

CONTROL STRUCTURES (cont.)

Repetition

- When an action (or a sequence of actions) must be repeated, a **loop** is used
- A **loop** is a program construction that repeats a statement or sequence of statements a number of times
- C++ includes several ways to create loops
 - **while** loops
 - the statement(s) in the loop are executed zero or more times
 - **do ... while** loops
 - the statement(s) in the loop are executed at least once
 - **for** loops
 - the statement(s) in the loop are executed zero or more times
- **while** loops and **do ... while** loops are typically used for sentinel-controlled repetition
- **for** loops are typically used for counter-controlled repetition

• **while** statement

```
while (boolean_expression)  
    statement
```

- Example 1 (sum list of positive values)

```
int value, sum = 0;  
cout << "value? ";  
cin >> value;  
while (value > 0)    // when does it end?  
{  
    sum = sum + value; // OR sum += value;  
    cout << "value? ";  
    cin >> value;  
}  
cout << "Sum = " << sum << endl;
```

- **do ... while statement**

```
do
    statement
while (boolean_expression) ;    ← NOTE the semicolon
```

- Example (printing ASCII CODES)

```
char symbol;
const char STOP = '#'; // NOTE: good programming practice

do
{
    cout << "Letter/digit (" << STOP << " to QUIT) ? ";
    cin >> symbol;
    cout << "ASCII(" << symbol << ") = " << (int)symbol << endl;
} while (symbol != STOP);
```

- Question: how to prevent ASCII('#') from being shown?

- **for statement**

```
for (initializing_expr; boolean_expr; step_expr)
    statement
```

- Example 1 (sum of all integer values in the range [1..100])

```
int i;
int sum = 0; // DON'T FORGET INITIALIZATION !!!

for (i=1; i<=100; i=i+1)
    sum = sum + i;

cout << "1 + 2 + ... + 100 = " << sum << endl;
```

- Example 2 (sum of any 5 integer values, read from the keyboard)

```
int sum = 0;

for (int i=1; i<=5; i++) // i is only visible inside the block
{
    int value; // value only visible inside the block
    cout << "value no. " << i << " ? ";
    cin >> value;
    sum = sum + value;
}

cout << "Sum of entered values = " << sum << endl;
```

- NOTE: any of the expressions in a **for** statement can be omitted.

- **break and continue statements**

- **break**

- leaves a loop, even if the condition for its end is not fulfilled
 - It can be used to end an infinite loop
 - can be used with any type of loop

```
for (int divisor=2; divisor<=num; divisor++)  
{  
    if (num % divisor == 0)  
    {  
        cout << "First integer divisor (<= 1) of " << num <<  
            " is " << divisor << endl;  
        break;  
    }  
}
```

- **TO DO BY STUDENTS:** implement using while and do...while loops.

- **continue**

- causes the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached, causing it to jump to the start of the following iteration
 - can be used with any type of loop

```
for (int i=1; i<=100; i++)  
{  
    if (i==13) continue; //... I'm not superstitious, but...  
    cout << i << endl;  
}
```

- **NOTES:**

- If you have a loop within a loop, and a **break** in the inner loop, then the **break** statement will only end the inner loop.
 - The use of **break** and **continue** in loops with many statements, makes the code difficult to read ... 😞

- **"infinite" loops**

- while (true) {...};
 - while (1) {...};
 - do { ... } while (true);
 - do { ... } while (1);
 - for (;;) {...};

- In fact, no loop should be infinite, otherwise the program would never end ...
 - An "infinite" loop should contain a **break** statement somewhere.
 - Sometimes, there are some infinite loops caused by coding errors ... 😞

INPUT – dealing with invalid inputs

Invalid inputs

- Consider the following code:

```
int sum = 0;
for (int i=1; i<=5; i++) // i is only visible inside the block
{
    int value;           // value only visible inside the block
    cout << "Value no. " << i << " ? ";
    cin >> value;
    sum = sum + value;
}
```

- What happens if a careless user inserts value **1o** instead of **10**?
- ... the loop becomes an endless loop!
- In the lectures it will be explained in detail why this happens.
- In summary:
 - the **o** (lowercase letter) is not compatible with an integer value
 - the input will fail and the input stream will enter a failure state;
 - the failure state must be cleared, so that more input can take place
 - even if the failure state is cleared
the **o** could only be extracted to a **char** or **string** variable
 - so, if one wants to continue reading integer values,
the **o** must be removed from the input buffer

Testing for input failure and clearing invalid input state

- The easiest way to test whether a stream is okay is to test its "truth value":
 - `if (cin) ...` // OK to use cin, it is in a valid state
 - `if (! cin) ...` // cin is in an invalid state
 - `while (cin >> value) ...` // OK, read operation successful
- Alternative way:
 - `cin >> value;`
`if (! cin.fail()) ...` // OK; input did not fail
- Other condition states can be tested (*to be presented later*):
 - `cin.good()`, `cin.bad()`, `cin.eof()`
- The **clear** operation puts the I/O stream condition back in its valid state
 - `if (cin.fail()) cin.clear();` // **NOTE: it does not clean the buffer**

Cleaning the input buffer

- Sometimes, it is necessary to remove from the input buffer the input that caused the failure. This is done using the `cin.ignore()` call.
NOTE: must be preceded by `cin.clear()`.
- It can be called in 3 different ways:
 - `cin.ignore()`
 - a single character is taken from the input buffer and discarded
 - `cin.ignore(numChars)`
 - the number of characters specified are taken from the input buffer and discarded;
 - `cin.ignore(numChars, delimiterChar)`
 - discard the number of characters specified, or discard characters up to and including the specified delimiter (whichever comes first):
 - Example: `cin.ignore(10, '\n');`
 - ignore 10 characters or to a newline, whichever comes first
 - The whole buffer contents can be cleaned by calling:
 - `cin.ignore(numeric_limits<streamsize>::max(), '\n');`
`// => #include <ios> AND #include <limits>`
 - **Note: BE CAREFUL!!!**
a call to `cin.ignore()` when the buffer is empty
will stop the execution of the program until something is entered !!!

Other input operations

- `cin.get()`
 - returns the next character in the stream
- `cin.peek()`
 - returns the next character in the stream, but does not remove it from the stream

TO DO BY STUDENTS

- Write a program that uses `cin >>`, `cin.peek()` and `cin.ignore()` to read the integer number contained in "#ABcdE12345\$Esc", that is 12345, discarding all the remaining symbols.

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

int main()
{
    int num = 0;
    char ch;
    ch = cin.peek();
    while (ch != '\n')
    {
        cout << ch;
        if (ch >= '0' && ch <= '9')
            num = num * 10 + ch - '0';
        cin.ignore(1);
        ch = cin.peek();
    }
    cout << endl;
    cout << "num = " << num << endl;
    return 0;
}
```

```

/*
  USING REPETITION STATEMENTS: FOR
*/

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

int main()
{
    const unsigned int NUMBER_PRECISION = 3;
    double operand1, operand2; // input operands
    char operation; // operation; possible values: + - * /
    double result; // result of "operand1 operation operand2"
    unsigned int i;
    unsigned int numOperations;

    cout << "Number of operations to do ? ";
    cin >> numOperations;

    for (i=1; i<=numOperations; i++) // OR for (unsigned int i=1; ...)
    {
        // input 2 numbers
        cout << endl;
        cout << "x op y ? ";
        cin >> operand1 >> operation >> operand2;

        bool validOperation = true; // assume operation is valid
        // compute result if operation is valid
        switch (operation)
        {
            case '+':
                result = operand1 + operand2;
                break;
            case '-':
                result = operand1 - operand2;
                break;
            case '*':
                result = operand1 * operand2;
                break;
            case '/':
                result = operand1 / operand2;
                break;
            default:
                validOperation = false;
        }

        //show result or invalid input message
        if (validOperation)
        {
            cout << fixed << setprecision(NUMBER_PRECISION);
            cout << operand1 << ' ' << operation << ' ' << operand2 <<
                " = " << result << endl;
        }
        else
            cerr << "Invalid operation !\n";
    }
    return 0;
}
=====

```

```

/*
  USING REPETITION STATEMENTS: WHILE
*/
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    const unsigned int NUMBER_PRECISION = 3;
    double operand1, operand2; // input operands
    char operation; // operation; possible values: + - * /
    double result; // result of "operand1 operation operand2"
    unsigned int i;
    unsigned int numOperations;

    cout << "Number of operations to do ? ";
    cin >> numOperations;

    i = 0; // counter //COMMON ERRORS: - forget initialization
    while (i < numOperations) // - off by one loops
    {
        // read operation
        cout << endl;
        cout << "x op y ? ";
        cin >> operand1 >> operation >> operand2;

        bool validOperation = true; // assume operation is valid
        // compute result if operation is valid
        switch (operation)
        {
            case '+':
                result = operand1 + operand2;
                break;
            case '-':
                result = operand1 - operand2;
                break;
            case '*':
                result = operand1 * operand2;
                break;
            case '/':
                result = operand1 / operand2;
                break;
            default:
                validOperation = false;
        }

        //show result or invalid input message
        if (validOperation)
        {
            cout << fixed << setprecision(NUMBER_PRECISION);
            cout << operand1 << ' ' << operation << ' ' << operand2 <<
                " = " << result << endl;
        }
        else
        {
            cerr << "Invalid operation !\n";
            i = i+1; // OR i++; OR i += 1; // DO NOT FORGET ... TO UPDATE
        }
    }
    return 0;
}

```



```

/*
  USING REPETITION STATEMENTS: DO ...WHILE
*/
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    const unsigned int NUMBER_PRECISION = 3;
    double operand1, operand2; // input operands
    char operation; // operation; possible values: + - * /
    double result; // result of "operand1 operation operand2"
    unsigned int i;
    unsigned int numOperations;

    cout << "Number of operations to do ? ";
    cin >> numOperations;

    i = 0; // counter
    do // WHAT HAPPENS IF USER INSERTS numOperations = 0 ?!
    {
        // read operation
        cout << endl;
        cout << "x op y ? ";
        cin >> operand1 >> operation >> operand2; // TRY WITH 'a + 2'
        //cout << "OP = " << operand1 << operation << operand2 << endl;

        bool validOperation = true; // assume operation is valid
        // compute result if operation is valid
        switch (operation)
        {
            case '+':
                result = operand1 + operand2;
                break;
            case '-':
                result = operand1 - operand2;
                break;
            case '*':
                result = operand1 * operand2;
                break;
            case '/':
                result = operand1 / operand2;
                break;
            default:
                validOperation = false;
        }

        //show result or invalid input message
        if (validOperation)
        {
            cout << fixed << setprecision(NUMBER_PRECISION);
            cout << operand1 << ' ' << operation << ' ' << operand2 <<
                " = " << result << endl;
        }
        else
            cerr << "Invalid operation !\n";
        i = i+1; // OR i++; OR i += 1;
    } while (i<numOperations);
    return 0;
}

```

```

=====
/*
USING REPETITION STATEMENTS - user is not asked for the no. of operations
*/
#include <iostream>
#include <iomanip>
#include <cctype> // for using toupper()
using namespace std;
int main()
{
    const unsigned int NUMBER_PRECISION = 3;
    double operand1, operand2; // input operands
    char operation; // operation; possible values: + - * /
    double result; // result of "operand1 operation operand2"
    char anotherOperation;
do
{
    // read operation
    cout << endl;
    cout << "x op y ? ";
    cin >> operand1 >> operation >> operand2;

    // compute result if operation is valid
    bool validOperation = true; // assume operation is valid
    switch (operation)
    {
        case '+':
            result = operand1 + operand2;
            break;
        case '-':
            result = operand1 - operand2;
            break;
        case '*':
            result = operand1 * operand2;
            break;
        case '/':
            result = operand1 / operand2;
            break;
        default:
            validOperation = false;
    }

    //show result or invalid input message
    if (validOperation)
    {
        cout << fixed << setprecision(NUMBER_PRECISION);
        cout << operand1 << ' ' << operation << ' ' << operand2 <<
            " = " << result << endl;
    }
    else
        cerr << "Invalid operation !\n";

    cout << "Another operation (Y/N) ? ";
    cin >> anotherOperation;
    anotherOperation = toupper(anotherOperation);
} while (anotherOperation == 'Y'); //TO DO: alternative ...?
return 0;
}

```

```

=====
/*
USING REPETITION STATEMENTS
- NESTED LOOPS
- DEALING WITH INVALID INPUTS → Teaching note: START WITH SIMPLE EXAMPLE
*/
#include <iostream>
#include <iomanip>
#include <cctype> // for using toupper()

using namespace std;

int main()
{
    const unsigned int NUMBER_PRECISION = 3;

    double operand1, operand2; // input operands
    char operation; // operation; possible values: + - * /
    double result; // result of "operand1 operation operand2"
    bool validOperation; // operation is + - * or /
    char anotherOperation;

do
{
    // input 2 numbers and the operation
    bool validOperands = false; // ONLY VISIBLE INSIDE ABOVE "do" BLOCK
do
{
    cout << endl << "x op y ? ";
    if (cin >> operand1 >> operation >> operand2)
        validOperands = true;
    else
    {
        cin.clear(); // clear error state
        cin.ignore(1000, '\n'); // clean input buffer
    }
} while (!validOperands);

    // compute result if operation is valid
    validOperation = true;

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

```

```

//show result or invalid operator message
if (validOperation)
{
    cout << fixed << setprecision(NUMBER_PRECISION);
    cout << operand1 << ' ' << operation << ' ' << operand2 <<
        " = " << result << endl;
}
else
    cerr << "Invalid operator !\n";

    cout << "Another operation (Y/N) ? ";
    cin >> anotherOperation;
    anotherOperation = toupper(anotherOperation);

} while (anotherOperation == 'Y');
return 0;
}

```

=====

TO DO BY STUDENTS:

- modify the "do ... while (!validOperands);" loop so that it is also repeated when the operator is not valid
ANS: while (!validNumbers || !(operation=='+' || operation=='-' || operation=='*' || operation=='/'))
- modify the "do ... while (!validOperands);" loop so that the initial value of validOperands is true

```

/*
USING REPETITION STATEMENTS
DETECTING END OF INPUT (CTRL-Z)
*/
#include <iostream>
#include <iomanip>
#include <cctype>

using namespace std;

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

    double operand1, operand2;
    char operation;
    double result;
    bool anotherOperation;
    //ANY OF THESE VARIABLES COULD HAVE BEEN DECLARED ELSEWHERE ?

    do
    {
        bool validOperands; // ONLY VISIBLE INSIDE THE 'green' do ...while cycle
        anotherOperation = true;

        do
        {
            cout << endl << "x op y (CTRL-Z to end) ? ";
            cin >> operand1 >> operation >> operand2;
            validOperands = true;
            if (cin.fail())
            {
                validOperands = false;
                if (cin.eof()) // use cin.eof() only after cin.fail() returns TRUE
                anotherOperation = false; //ALTERNATIVE: return 0;
            }
            else
            {
                cin.clear();
                cin.ignore(1000, '\n');
            }
        }
        else
        cin.ignore(1000, '\n'); //clear any additional chars
    } while (anotherOperation && !validOperands);

    //above cycle is equivalent to:
    //REPEAT ... UNTIL ((NOT anotherOperation) OR validOperands)
    //sometimes it is easier to think in terms of REPEAT...UNTIL...
    //then negate REPEAT condition to obtain the WHILE condition

    if (validOperands)
    {
        bool validOperation = true;
        switch (operation)
        {
            case '+':
                result = operand1 + operand2;
                break;
            case '-':

```

```

        result = operand1 - operand2;
        break;
    case '*':
        result = operand1 * operand2;
        break;
    case '/':
        result = operand1 / operand2;
        break;
    default:
        validOperation = false;
    }

    //show result or invalid input message
    if (validOperation)
    {
        cout << fixed << setprecision(NUMBER_PRECISION);
        cout << operand1 << ' ' << operation << ' ' <<
operand2 <<
        " = " << result << endl;
    }
    else
        cerr << "Invalid operation !\n";

}

} while (anotherOperation); // EQUIVALENT TO: repeat ... until (??);
return 0;
}

```

NOTE- alternative way to invoke cin.ignore():

```
std::cin.ignore ( std::numeric_limits<std::streamsize>::max(), '\n' );
```

OR JUST

```
cin.ignore ( numeric_limits<streamsize>::max(), '\n' );
```

This requires you to include the following header files:

```

#include <iostream>
#include <ios>      // for streamsize
#include <limits>   // for numeric_limits

```

```

/*
MORE ON REPETITION (OR ITERATION) STATEMENTS

CALCULATE THE SQUAREROOT USING A BABILONIAN ALGORITHM
SEE PROBLEM 2.14
*/
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    cout << "Please enter a number: ";
    double num;
    cin >> num;

    // VARIABLES CAN BE DECLARED ANYWHERE, IN THE SAME CODE BLOCK,
    // BEFORE THEY ARE USED
    const double EPSILON = 1E-6;
    double xnew = num;
    double xold;

    do
    {
        xold = xnew;
        xnew = (xold + num / xold) / 2;
    }
    while (fabs(xnew - xold) > EPSILON);

    cout << "The square root is " << xnew << "\n";
    return 0;
}

```

num = 2

xold	xnew
2	1.5
1.5	1.41667
1.41667	1.41422
1.41422	1.41421
1.41421	...
...	...

TO DO BY STUDENTS:
- specify a maximum number of iterations (SEE PROBLEM 2.14)

```
/*
```

```
MIXED CODE ☹
```

```
SOME LOOP VARIATIONS
```

```
COMMA OPERATOR
```

```
BREAK AND CONTINUE STATEMENTS
```

```
*/
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int x, y;
```

```
    int i, j;
```

```
    // FOR - 1
```

```
    //
```

```
    cout << "FOR - 1\n";
```

```
    for (int k = 10; k != 0; k = k-2) // WHAT DOES IT DO ?
```

```
        cout << "k = " << k << endl;
```

```
    //COMMA OPERATOR
```

```
    //
```

```
    cout << endl << "COMMA OPERATOR\n";
```

```
    x = (y=3, y+1); // the parentheses are necessary,  
                  // because the comma operator has lower precedence  
                  // than the assignment operator
```

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

```
    // FOR - 2
```

```
    //
```

```
    cout << endl << "FOR - 2\n";
```

```
    for (i=1, j=10; i < j; i++, j--) // note the comma operator
```

```
        cout << "i = " << i << "; j = " << j << endl;
```

```
    // FOR - 3 (infinite...? loop)
```

```
    //
```

```
    cout << endl << "FOR - 3\n";
```

```
    for (;;) 
```

```
    {
```

```
        char letter;
```

```
        cout << "letter (q-quit) ? ";
```

```
        cin >> letter;
```

```
        if (letter == 'q' || letter == 'Q')
```

```
            break;
```

```
        //do something
```

```
        cout << "WORKING HARD !\n";
```

```
        //...
```

```
    }
```

```
    cout << "You entered 'q' or 'Q' \n";
```



```
// WHILE - 1
//-----
cout << endl << "WHILE - 1\n";
char letter;
cout << "letter (q-quit) ? ";
cin >> letter;
while (letter != 'q' && letter != 'Q')
{
    //do something
    cout << "WORKING HARD !\n";
    //...
    cout << "letter (q-quit) ? ";
    cin >> letter;
}
cout << "You entered 'q' or 'Q' \n";
```

```
// WHILE - 2 (infinite...? loop)
//-----
cout << endl << "WHILE - 2\n";
while (true) // OR while (1)
{
    char letter;
    cout << "letter (q-quit) ? ";
    cin >> letter;
    if (letter == 'q' || letter == 'Q')
        break;
    //do something
    cout << "WORKING HARD !\n";
    //...
}
cout << "You entered 'q' or 'Q' \n";
```

```
// FOR - 4 (a strange FOR loop...!) DON'T DO ANYTHING LIKE THIS
//-----
cout << endl << "FOR - 4\n";
int n;
for (cout << "n (0=end)? "; cin >> n, n != 0; cout << "n (0=end)? ")
    cout << n*10 << endl;
```

```
// FOR - 5 - NOT RECOMMENDED
//-----
cout << endl << "FOR - 5\n";
const unsigned int NUM_VALUES = 10;
double sum = 0, value, mean;

i = 1;
for ( ; i <= NUM_VALUES; ) // NOT RECOMMENDED
{
    cout << "n" << i << "? ";
    cin >> value;
    sum = sum + value;
    i++;
}

mean = sum / NUM_VALUES;
cout << "mean value = " << mean << endl;

return 0;
}
```

```

/*
LOOP pitfalls
Be careful !!!

*/
#include <iostream>

using namespace std;

int main()
{
    // Guess what do the loops do:

    // PITFALL 1
    for (int count = 1; count <= 10; count++); // NOTE the semicolon
        cout << "Hello\n";

    // PITFALL 2
    int x = 1;
    while (x != 12)
    {
        cout << x << endl;
        x = x + 2;
    }

    // PITFALL 3
    float y = 0;
    while (y != 12.0)
    {
        cout << y << endl;
        y = y + 0.2;
    }

    return 0;
}

```

TO DO BY STUDENTS:

Write a program to show the multiplication tables, from 2 to 9:

```

2x1 =  2
2x2 =  4
...
2x9 = 18
-----

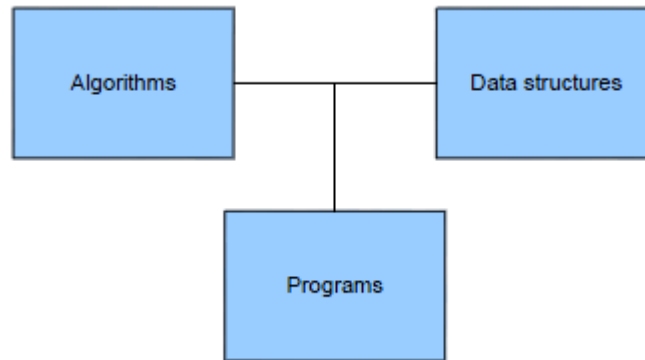
3x1 =  3
...
-----
9x1 =  9
...
9x8 = 72
9x9 = 81

```

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.

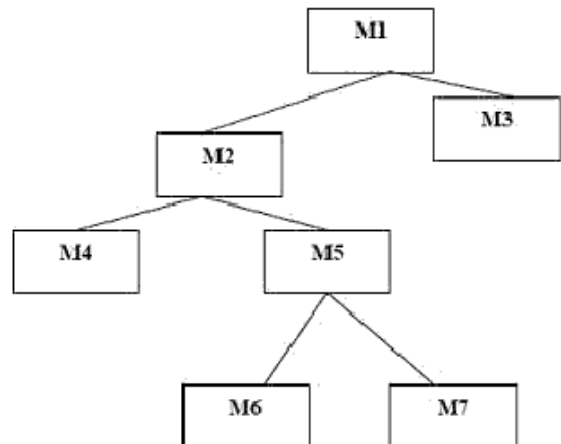
- By minimising the ways in which modules can interact, we greatly limit the overall complexity, and hence limit unexpected behaviour, increasing robustness.
- Because a particular module interacts with other modules in a carefully defined manner, it becomes easier to test/validate, and can become a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



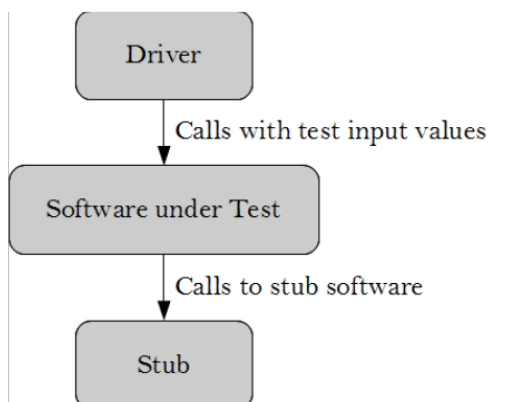
- **Code development**

- **Top-down**

- Code high level parts using “**stubs**” with assumed functionality for low-level dependencies
- Iteratively descend to lower-level modules

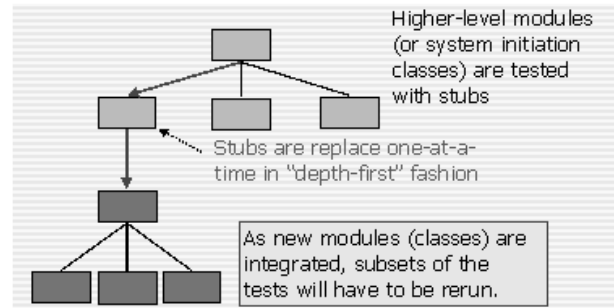
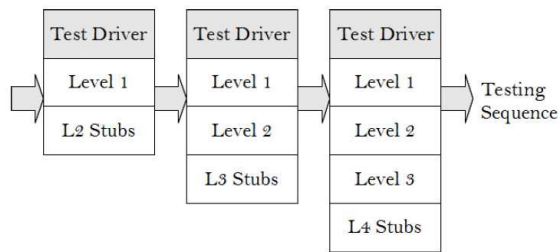
- **Bottom-up**

- Code and test each low-level component
- Need “**test driver**” so that low-level can be tested in its correct context
- Integrate components



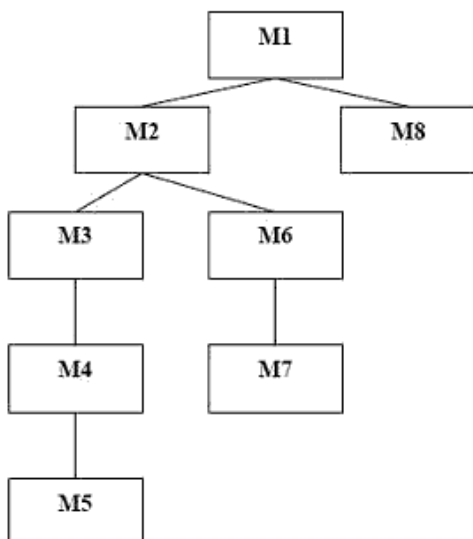
- **Stub:** is temporary or dummy software that is required by the software under test for it to operate properly.
- **Test driver:** calls the software under test, passing the test data as inputs.

- **Top-down integration**



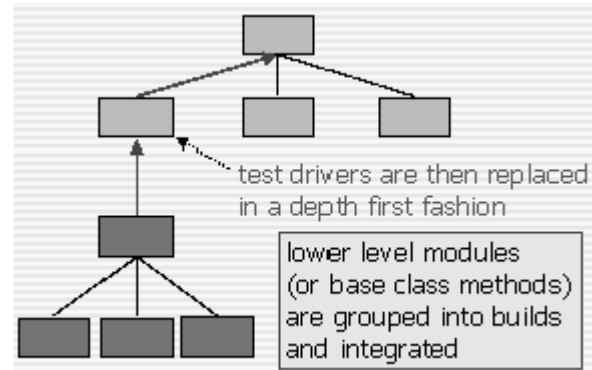
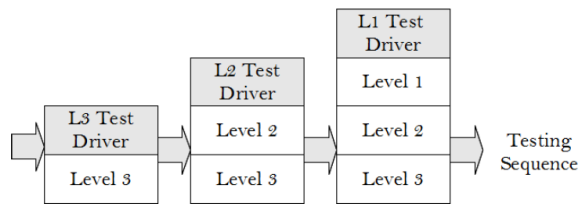
- The highest-level modules are tested and integrated first.
- This allows high-level logic and data flow to be tested early in the process
- **Advantages:**
 - The top layer provides an early outline of the overall program
 - helping to find design errors early on and
 - giving confidence to the team, 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**



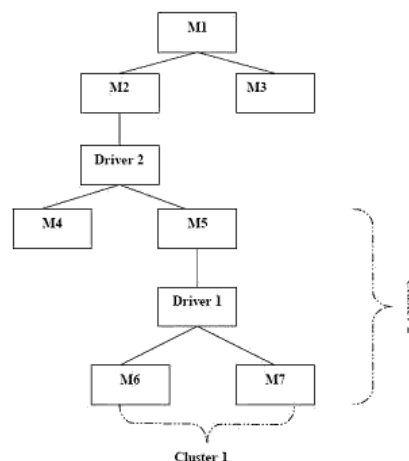
- **Depth first approach**
 - All modules on a control path are integrated first.
 - The sequence of integration would be (M1, M2, M3), M4, M5, M6, M7, and M8.
- **Breadth first approach**
 - All modules directly subordinate at each level are integrated together.
 - The sequence of integration would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

- Alternative (better): each function calculates internally the value of **PI**

```
const double PI = 4 * atan(1.0);
```

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 ...?)
@return 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
@param 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);
//TO DO: 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 selectd 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 selected number of places
@param x: number to be rounded
@param n: number of places
@return 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
@return 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]
@return 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
                  // 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\n";
    cout << "-----\n";
    for (i=0; i <= 20; i++)
        cout << setw(2) << i << " - " <<
            setw(25) << factorialIte(i) << " " <<
            setw(25) << factorialRec1(i) << " " <<
            setw(25) << factorialRec2(i) << endl;

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

```

=====

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

```

// EFFICIENCY OF RECURSION
// Computing the n-th element of the Fibonacci sequence
// F(1)=1; F(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 previous example).
- 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.
- NOTE: 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 !!! (note the additional spaces)

// (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); // ⇔ func(x1)
        y2 = f(x2); // ⇔ func(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 ?

    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;
    }
}

```

ARRAYS

- **Introduction to arrays**

- An array is used to process a collection of **data of the same type**
 - Examples:
 - a list of temperatures
 - a list of names
 - Why do we need to use arrays?
 - ...? (YOUR ANSWER) .

- **Declaring arrays and accessing array elements**

- An array to store the final scores (of type `int`) of the 196 students of a course:
`int score[196];` // **NOTE: the contents is undetermined**
- This is like declaring 196 variables of type `int`:
`score[0], score[1], ... score[195]`
- The value in brackets is called a **subscript** OR an **index**
- The variables making up the array are referred to as
 - indexed variables
 - subscripted variables
 - elements of the array
- **NOTE:**
 - the first index value is zero
 - the largest index is one less than the size
- **Good practice:**
 - Use constants to declare the size of an array:
 - using a constant allows your code to be easily altered for use on a smaller or larger set of data:

`const int NUMBER_OF_STUDENTS = 196;`
`...
int score[NUMBER_OF_STUDENTS];`
- **NOTE:**
 - In C++, variable length arrays are not legal.
`cout << "Enter number of students: ";
cin >> number;
int score[number];` // **ILLEGAL IN MANY COMPILERS**
 - G++ compiler allows this as an "extension" (because C99 allows it)

- **How arrays are stored in memory**

- Declaring the array `int a[6]`
 - reserves memory for 6 variables of type `int`;
 - the variables are stored one after another
- The address of `a[0]` is remembered
 - The addresses of the other indexed variables is not remembered
- To determine the address of `a[3]` the compiler
 - starts at `a[0]`
 - counts past enough memory for three integers to find `a[3]`
- **VERY IMPORTANT NOTE:**
 - A common error is using a nonexistent index.
 - Index values for `int a[6]` are the values 0 through 5 .
 - An index value not allowed by the array declaration is out of range.
 - **Using an out of range index value does not necessarily produce an error message!!!**

- **Initializing arrays**

- Initialization when it is declared
 - `int a[3] = {11, 19, 12};`
 - `int a[] = {3, 8, 7, 1}; //size not needed`
 - `int b[100] = {0}; //all elements equal to zero`
 - `int b[5] = {4 ,7 ,9}; //4th & 5th elements equal to zero`
 - `string name[3] = {"Ana", "Rui", "Pedro"};`
- Initialization after declaration

```
const int NUMBER_OF_STUDENTS = 196;
int score[NUMBER_OF_STUDENTS];
for (int i=0; i<NUMBER_OF_STUDENTS; i++)
    score[i] = 20; // :-)
```

- **Operations on arrays**

- It is not possible to copy all the array elements using a single assignment operation
- It is not possible to read/write/process all the array elements with a "simple" statement

```
int a1[3] = {10,20,30};
int a2[3];
a2 = a1; // COMPILATION ERROR ...
cout << a1 << endl; //NOT AN ERROR! BUT ... WHAT DOES IT SHOW?
```
- Each element must be read/written/processed at a time, using a loop (see previous example)

- **Arrays as function arguments / parameters & as return values**

- Arrays can be arguments to functions.
- A formal parameter can be for an entire array
 - such a parameter is called an array parameter
 - array parameters **behave much like call-by-reference parameters**
- An array parameter is indicated using empty brackets in the parameter list
- The **number of array elements (to be processed) must be indicated as an additional formal parameter** (numStudents, in the example below)

```
void readScores(int score[], int numStudents)
{
    ... // loop for reading student scores
}

...

int studentScore[NUMBER_OF_STUDENTS];
...

readScores(studentScore, NUMBER_OF_STUDENTS); //function call
```

- **const modifier**

- Array parameters allow a function to change the values stored in the array argument **(BE CAREFUL!)**
- If a function should not change the values of the array argument, use the modifier **const**

```
double computeScoreAverage(const int score[], int numStudents)
{
    ... // computes the average; cannot change score[]
}

...
```

- **Returning an array**

- **Functions cannot return arrays, using a **return** statement.**
- However, an array can be returned, if it is embedded in a **struct**. (*see later*)
- A function can return a pointer to an array (*see later*).

- **Multidimensional arrays**

- An array to store the scores of the students for each exam question:

```
int score[NUMBER_STUDENTS][NUMBER_QUESTIONS];
```

- Initialization of a multidimensional array when it is declared:

- ```
int m[2][3] = { {1,3,2}, {5,2,9} }; // or ...
int m[2][3] = { 1,3,2,5,2,9 };
```

- Indexing a multidimensional array:

- `score[0][0]` – score of the 1st student in the 1st question
- `score[0][1]` – score of the 1st student in the 2nd question
- ...
- `score[1][2]` – score of the 2nd student in the 3rd question

- **NOTE:** the "off-by-one offset" in the index ...

- **NOTE:**

When a multidimensional array is used as a formal function parameter, the size of the first dimension is not given, but the remaining dimension sizes must be given in square brackets.

- Since the first dimension size is not given, you usually need an additional parameter of type `int` that gives the size of this first dimension.

```
int readScores(int score[][NUMBER_QUESTIONS], int numStudents)
{
 ...
}
```

## // 1D ARRAYS // How they are stored in memory

```
#include <iostream>

using namespace std;

const int NMAX=3;

void main(void)
{
 int a[NMAX];
 int i;

 for (i=0; i<NMAX; i++)
 a[i] = 10*(i+1);

 for (i=0; i<NMAX; i++)
 cout << "a[" << i << "] = " << a[i]
 << ", &a[" << i << "] = " << (unsigned long) &a[i] << endl;

 cout << "a = " << (unsigned long) a << endl; // 'a' is ...
 cout << "&a[0] = " << (unsigned long) &a[0] << endl; // ... the address of a[0]
}
```

```
a[0] = 10, &a[0] = 1899068
a[1] = 20, &a[1] = 1899072
a[2] = 30, &a[2] = 1899076
a = 1899068
&a[0] = 1899068
```

|       |   |         |         |
|-------|---|---------|---------|
| a =   | → | 1899068 | ...     |
| &a[0] | → | 1899068 | a[0] 10 |
| &a[1] | → | 1899072 | a[1] 20 |
| &a[2] | → | 1899076 | a[2] 30 |
|       |   |         | ...     |

### TO DO BY STUDENTS:

- swap the contents of 2 arrays of the same size



```

// 2D ARRAYS
// How they are stored in memory
// How they are passed to function parameters

#include <iostream>

using namespace std;

const unsigned NLIN=2; // because NLIN & NCOL are globals, the dimensions of the array
const unsigned NCOL=3; // they can be omitted in the function parameters. GOOD SOLUTION...?!

int sumElems(const int m[NLIN][NCOL]) // NLIN could be omitted
{
 int sum=0;

 for (int i=0; i<NLIN; i++)
 for (int j=0; j<NCOL; j++)
 sum = sum + m[i][j];

 return sum;
}

//TO DO: function averageCols() - computes the average of the each of the columns of a[][]

void main(void)
{
 int a[NLIN][NCOL];

 for (int i=0; i<NLIN; i++)
 for (int j=0; j<NCOL; j++)
 a[i][j] = 10*(i+1)+j;

 for (int i=0; i<NLIN; i++)
 for (int j=0; j<NCOL; j++)
 cout << "a[" << i << "][" << j << "] = " << a[i][j]
 << ", &a[" << i << "][" << j << "] = " << (unsigned long) &a[i][j]
 << endl;

 cout << "Sum of elements of a[][] = " << sumElems(a) << endl;
}

```

|   | 0  | 1  | 2  |
|---|----|----|----|
| 0 | 10 | 11 | 12 |
| 1 | 20 | 21 | 22 |

```

a[0][0] = 10, &a[0][0] = 1637144
a[0][1] = 11, &a[0][1] = 1637148
a[0][2] = 12, &a[0][2] = 1637152
a[1][0] = 20, &a[1][0] = 1637156
a[1][1] = 21, &a[1][1] = 1637160
a[1][2] = 22, &a[1][2] = 1637164

```

|          |   |         |            |
|----------|---|---------|------------|
| a =      | → | 1637144 | ...        |
| &a[0][0] | → | 1637148 | a[0][0] 10 |
| &a[0][1] | → | 1637152 | a[0][1] 11 |
| &a[0][2] | → | 1637156 | a[0][2] 12 |
| &a[1][0] | → | 1637160 | a[1][0] 20 |
| &a[1][1] | → | 1637164 | a[1][1] 21 |
| &a[1][2] | → |         | a[1][2] 22 |
|          |   |         | ...        |

```

/*
ARRAYS
Passing arrays as function parameters
Counting number of occurrences of zero values in an array of integers
*/

#include <iostream>
#include <iomanip>

using namespace std;

void initArray(int v[], int size)
{
 for (int i = 0; i < size; i++)
 {
 v[i] = 10 * (i % 3); //try to guarantee the occurrence of some zeros
 }
}

void showArray(int v[], int size)
{
 for (int i = 0; i < size; i++)
 {
 cout << v[i] << endl;
 }
}

int countZeros(int v[], int size) // NOTE: arrays are passed "by reference"
{
 int numZeros=0;
 for (int i=0; i < size ; i++)
 if (v[i] == 0) // BE CAREFUL !!!
 numZeros++;

 return numZeros;
}

int main()
{
 const int MAX_SIZE = 10;
 int a[MAX_SIZE];

 int numElems;
 cout << " Effective number of elements (max = " << MAX_SIZE << ") ? ";
 cin >> numElems;

 initArray(a, numElems);
 showArray(a, numElems);
 cout << "number of zeros = " << countZeros(a, numElems) << endl;
 showArray(a, numElems); // NOTE: interpret the obtained result

 return 0;
}

```

```

/*
ARRAYS
Passing arrays as function parameters.
Using 'const' qualifier.
*/

#include <iostream>
#include <iomanip>

using namespace std;

void initArray(int v[], int size)
{
 for (int i = 0; i < size; i++)
 {
 v[i] = 10 * (i % 3);
 }
}

void showArray(const int v[], int size)
{
 for (int i = 0; i < size; i++)
 {
 cout << v[i] << endl;
 }
}

int countZeros(const int v[], int size) //AFTER ADDING const ...
{
 int numZeros=0;
 for (int i=0; i < size ; i++)
 if (v[i] == 0) //... THE COMPILER DETECTS THIS LOGIC ERROR
 // if (v[i] == 0) //CORRECT FORM ALTERNATIVE: if (0 == v[i])
 numZeros++; // WHAT IS THE ADVANTAGE?

 return numZeros;
}

int main()
{
 const int MAX_SIZE = 10;
 int a[MAX_SIZE];

 int numElems;
 cout << "Effective number of elements (max = " << MAX_SIZE << ") ? ";
 cin >> numElems;

 initArray(a, numElems);
 showArray(a, numElems);
 cout << "number of zeros = " << countZeros(a, numElems) << endl;
 showArray(a, numElems);

 return 0;
}

```

### TO DO BY STUDENTS:

- remove zeros from an array;
- compute mean and standard deviation of array elements;

```

/*
MULTIDIMENSIONAL ARRAYS - 2D ARRAYS
Reads quiz scores for each student into the two-dimensional array grade.
Computes the average score for each student and
the average score for each quiz.
Displays the quiz scores and the averages.
*/

```

```

#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

```

```

const int NUMBER_STUDENTS = 4, NUMBER_QUIZZES = 3;

```

```

void fill_grades(int grade[][NUMBER_QUIZZES]);

```

```

void compute_st_ave(const int grade[][NUMBER_QUIZZES], double st_ave[]);

```

```

void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double quiz_ave[]);

```

```

void display(const int grade[][NUMBER_QUIZZES], const double st_ave[], const
double quiz_ave[]);

```

```

//-----

```

```

int main()
{

```

```

 int grade[NUMBER_STUDENTS][NUMBER_QUIZZES];

```

```

 double st_ave[NUMBER_STUDENTS];

```

```

 double quiz_ave[NUMBER_QUIZZES];

```

```

 fill_grades(grade); // randomly !!!
 compute_st_ave(grade, st_ave);
 compute_quiz_ave(grade, quiz_ave);
 display(grade, st_ave, quiz_ave);
 return 0;
}

```

```

//-----

```

```

void fill_grades(int grade[][NUMBER_QUIZZES]) // fill... RANDOMLY !
{

```

```

 for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
 for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
 grade[st_num-1][quiz_num-1] = 10 + rand() % 11;
}

```

```

//-----

```

```

void compute_st_ave(const int grade[][NUMBER_QUIZZES], double st_ave[])
{

```

```

 for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
 {
 double sum = 0;
 for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
 sum = sum + grade[st_num-1][quiz_num-1];

 st_ave[st_num -1] = sum/NUMBER_QUIZZES;
 }
}

```

```

}
//-----

```

```

void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double quiz_ave[])
{
 for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
 {
 double sum = 0;
 for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
 sum = sum + grade[st_num - 1][quiz_num - 1];

 quiz_ave[quiz_num - 1] = sum/NUMBER_STUDENTS;
 }
}
//-----

void display(const int grade[][NUMBER_QUIZZES], const double st_ave[],
const double quiz_ave[])
{
 cout.setf(ios::fixed);
 cout.setf(ios::showpoint);
 cout.precision(1);

 cout << setw(10) << "Student"
 << setw(5) << "Ave"
 << setw(12) << "Quizzes\n";

 for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
 {
 cout << setw(10) << st_num << setw(5) << st_ave[st_num-1] << " ";
 for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
 cout << setw(5) << grade[st_num-1][quiz_num-1];
 cout << endl;
 }

 cout << "Quiz averages = ";
 for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
 cout << setw(5) << quiz_ave[quiz_num-1];
 cout << endl;
}

```

```

/*
2D ARRAYS -
ALTERNATIVE SOLUTION FOR THE PREVIOUS "PROBLEM" (i.e. index of 1st element is zero)
Reads quiz scores for each student into the two-dimensional array grade.
Computes the average score for each student and
the average score for each quiz.
Displays the quiz scores and the averages.
*/
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

const int NUMBER_STUDENTS = 4, NUMBER_QUIZZES = 3;

void fill_grades(int grade[][NUMBER_QUIZZES]);

void compute_st_ave(const int grade[][NUMBER_QUIZZES], double st_ave[]);

void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double
quiz_ave[]);

void display(const int grade[][NUMBER_QUIZZES], const double st_ave[],
const double quiz_ave[]);
//-----

int main()
{
 int grade[NUMBER_STUDENTS][NUMBER_QUIZZES];
 double st_ave[NUMBER_STUDENTS];
 double quiz_ave[NUMBER_QUIZZES];

 fill_grades(grade);
 compute_st_ave(grade, st_ave);
 compute_quiz_ave(grade, quiz_ave);
 display(grade, st_ave, quiz_ave);
 return 0;
}
//-----

void fill_grades(int grade[][NUMBER_QUIZZES])
{
 for (int st_num = 0; st_num < NUMBER_STUDENTS; st_num++)
 for (int quiz_num = 0; quiz_num < NUMBER_QUIZZES; quiz_num++)
 grade[st_num][quiz_num] = 10 + rand() % 11;
}
//-----

void compute_st_ave(const int grade[][NUMBER_QUIZZES], double st_ave[])
{
 for (int st_num = 0; st_num < NUMBER_STUDENTS; st_num++)
 {
 double sum = 0;
 for (int quiz_num = 0; quiz_num < NUMBER_QUIZZES; quiz_num++)
 sum = sum + grade[st_num][quiz_num];

 st_ave[st_num] = sum/NUMBER_QUIZZES;
 }
}
//-----

```

```

void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double quiz_ave[])
{
 for (int quiz_num = 0; quiz_num < NUMBER_QUIZZES; quiz_num++)
 {
 double sum = 0;
 for (int st_num = 0; st_num < NUMBER_STUDENTS; st_num++)
 sum = sum + grade[st_num][quiz_num];

 quiz_ave[quiz_num] = sum/NUMBER_STUDENTS;
 }
}

//-----

void display(const int grade[][NUMBER_QUIZZES], const double st_ave[],
const double quiz_ave[])
{
 cout.setf(ios::fixed);
 cout.setf(ios::showpoint);
 cout.precision(1);
 cout << setw(10) << "Student"
 << setw(5) << "Ave"
 << setw(15) << "Quizzes\n";
 for (int st_num = 0; st_num < NUMBER_STUDENTS; st_num++)
 {
 cout << setw(10) << st_num + 1
 << setw(5) << st_ave[st_num] << " ";
 for (int quiz_num = 0; quiz_num < NUMBER_QUIZZES; quiz_num++)
 cout << setw(5) << grade[st_num][quiz_num];
 cout << endl;
 }

 cout << "Quiz averages = ";
 for (int quiz_num = 0; quiz_num < NUMBER_QUIZZES; quiz_num++)
 cout << setw(5) << quiz_ave[quiz_num];
 cout << endl;
}

```

## STRUCTS & typedef

- **STRUCTs**

- A structure is a user-definable type.
- It is a derived data type, constructed using objects of other types (ex: int, string, ... or structures !).
- The keyword **struct** introduces the structure definition:

```
struct Person // the new type is named "Person" - C++ syntax
{
 string name; // string type will be introduced later
 char gender;
 unsigned int age;
}; // COMMON ERROR: forgetting the semicolon
```

```
Person p1, p2; // p1 and p2 are variables of type Person
```

- Suggestion: use an uppercase letter for the first character of the type name.
- After you define the type, you can create variables of that type.
- Thus, creating a structure is a two-part process.
  - First, you define a structure description that describes and labels the different types of data that can be stored in a structure.
  - Then, you can create structure variables, or, more generally, structure data objects, that follow the description's plan.

- **Accessing the fields of a structure**

```
cout << p1.name << "-" << p1.gender << << endl;
```

- **typedef**

- The keyword **typedef** provides a mechanism for creating synonyms (or aliases) for (previously defined) data types:

```
typedef unsigned int IdNumber;
IdNumber id;
```

- creates type **IdNumber** that is the same as **unsigned int**
- variable type of **id** is **IdNumber**



- Alternative way to create type Person using **typedef**:

```
typedef struct // the new type is named "Person"- C/C++ syntax
{
 string name;
 char gender;
 unsigned int age;
} Person;
```

```
Person p1 = {"Rui", 'M', 20}; //declaration w/initialization
```

- Using **typedef** to create another user defined type:

```
typedef unsigned int Uint; // Uint is the same as unsigned int
Uint x; // x is of type Uint
```

```

=====
/*
- CREATING NEW TYPES
- USING STRUCTURES FOR RETURNING MULTIPLE VALUES FROM FUNCTIONS
*/

#include <iostream>
#include <string>

using namespace std;

struct Person
{
 string name;
 char gender;
 unsigned int age;
};

const unsigned NUM_MAX_PERSONS = 10;
// Ⓢ different n.o of persons => modify & recompile

Person readPerson() // does not deal with invalid inputs
{
 Person p;
 cout << "Name ? "; // ONLY ONE WORD ... I'll be back to this later
 cin >> p.name;
 cout << "Gender ? ";
 cin >> p.gender;
 cout << "Age ? ";
 cin >> p.age;

 return p; // NOTE: a function can return a struct
}

int main()
{
 Person persons[NUM_MAX_PERSONS];
 size_t numPersons;

 // Read and validate number of persons
 cout << "How many persons ? ";
 cin >> numPersons;
 if (numPersons > NUM_MAX_PERSONS)
 {
 cerr << "Number of persons greater than allocated space ...!\n";
 exit(1);
 }

 // Read all person's data
 for (size_t i = 0; i < numPersons; i++)
 {
 persons[i] = readPerson();
 }
 // TO DO: show all the input data & process it (ex: obtain name and gender of oldest person)
}
=====

```

```

=====
/*
- USING STRUCTURES FOR RETURNING ARRAYS FROM FUNCTIONS
- CREATING NEW TYPES
*/

#include <iostream>
#include <iomanip>

using namespace std;

const int SIZE = 10;

//typedef struct {int a[SIZE];} StructArr; // a new type is created; C-style
struct StructArr {int a[SIZE];}; // a new type is created; C++-style

StructArr initArray(int size)
{
 StructArr s;

 for (int i = 0; i < size; i++)
 {
 s.a[i] = 10 * (i % 3);
 }
 return s;
}

void showArray(const StructArr &s, int size)
{
 for (int i = 0; i < size; i++)
 {
 cout << s.a[i] << endl;
 }
}

int countZeros(const StructArr &s, int size)
{
 int numZeros=0;

 for (int i=0; i < size ; i++)
 if (s.a[i] == 0)
 numZeros++;

 return numZeros;
}

int main()
{
 StructArr sa;

 sa = initArray(SIZE);
 showArray(sa,SIZE);

 cout << "number of zeros = " << countZeros(sa, SIZE) << endl;

 return 0;
}
=====

```

## STL (Standard Template Library) VECTORS

- **Vectors**

- Vectors are a kind of STL container
- Vectors are like arrays that can change size as your program runs :-)
- Vectors, like arrays, have a base type
- To declare an empty vector with base type `int`:

```
vector<int> v1; // be careful! v1 is empty
```

- `<int>` identifies **vector** as a **template class** (see later)

- You can use any base type in a template class:

```
vector<double> v2(10); // v2 has 10 elements of type double
 // all elements equal to zero
```

- **The vector library**

- To use the vector class, include the vector library

```
#include <vector>
```

- Vector names are placed in the standard namespace so the usual using directive is needed:

```
using namespace std;
```

- **Accessing vector elements**

- Vectors elements are indexed in the range `[0.. vector_size-1]`
- `[ ]`'s are used to read or change the value of an item:

```
for (size_t i = 0; i < v.size(); i++)
 cout << v[i] << endl;
```

- The **member function** `size()` returns the number of elements in a vector
- The size of a vector is of type `size_t` ( $\Rightarrow$  `#include <cstdint>`); `size_t` is an unsigned integer type
- The **member function** `at()` can be used instead of **operator** `[ ]` to access the vector elements
  - `v[i]` – can be disastrous if `i` is out of the range `[0.. vector_size-1]`
  - `v.at(i)` - checks whether `i` is within the bounds, throwing an **out\_of\_range exception** if it is not (see exception handling, later)

- Initializing vector elements

- `vector<int> v1; // be careful! v1 is empty`
  - Elements are added to the end of a vector using the member function `push_back()`  
  
`v1.push_back(12);`  
`v1.push_back(3);`  
`v1.push_back(547);`
- `vector<int> v1; // be careful! v1 is empty`  
`v1.resize(3); // additional elements are set to zero`
  - after the above resizing,  
the vector has space for 3 elements  
so you can access `v[i]`, `i=0..2`
  - `resize()` can be also used to shrink a vector !
- `vector<int> v2(10); // v2 has 10 elements equal to 0`
  - elements of number types are initialized to zero
  - elements of other types are initialized using  
the default constructor of the class (see later)
  - `v2.size()` would return 10
- `vector<int> v3(5,1); // v3 has 5 elements equal to 1`
- `vector<int> v4 = {10,20,30}; // possible with C++11 standard`
- NOTE: it is also possible to initialize a vector from an array (see later)

- Multidimensional vectors

- Declaration

```
//2D vector empty vector
vector< vector<int> > v1;
```

```
//2D vector with 5 lines and 3 columns
vector< vector<int> > v2(5, vector<int>(3));
```

- NOTE: in vectors, each row can have a different number of elements  
HOW TO DO THIS ?

- Accessing elements

```
v2[3][1] = 10; // OR ...
```

```
v2.at(3).at(1) = 10;
```

- Other vector methods

- see, for example: <http://www.cplusplus.com/reference/vector/vector/>
- some of them will be introduced later

- **Vectors as function arguments / parameters and as return values**
  - Vector can be used as call-by-value or call-by-reference parameters
    - Large vectors that are not to be modified by the function should be passed as **const** call-by-reference parameters
  - **Functions can return vectors**
    - Large vectors that are to be modified by the function could be passed as call-by-reference parameters

```
=====
// VECTOR
// a kind of STL (Standard Template Library) container

#include <iostream>
#include <vector>
#include <cstdint> // where 'size_t' is defined

using namespace std;

/*
Returns all values within a range
Parameters:
v - a vector of floating-point numbers
low - the low end of the range
high - the high end of the range
returns - a vector of values from v in the given range
*/
vector<double> between(vector<double> v, double low, double high)
// NOTE: vector v is passed by value
{
 vector<double> result;

 for (size_t i = 0; i < v.size(); i++)
 if (low <= v[i] && v[i] <= high)
 result.push_back(v[i]); //a vector can grow ...

 return result; //a vector can be returned, unlike an array
}

int main()
{
 vector<double> salaries(5); //vector with 5 elements of type double
 //try with vector<double> salaries;

 salaries[0] = 35000.0;
 salaries[1] = 63000.0;
 salaries[2] = 48000.0;
 salaries[3] = 78000.0;
 salaries[4] = 51500.0;

 vector<double> midrange_salaries = between(salaries, 45000.0, 65000.0);

 cout << "Midrange salaries:\n";
 for (size_t i = 0; i < midrange_salaries.size(); i++)
 cout << midrange_salaries[i] << "\n";

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

```

=====
/*
USING VECTORS
Read employee salaries
Determine those that have a mid-range salary
Raise their salary by a determined percentage
*/
#include <iostream>
#include <vector>
#include <cstdint>

using namespace std;

/*
Reads salaries
returns - a vector of salary values
*/
vector<double> readSalaries()
{
 int salary;
 vector<double> v;
 // ALTERNATIVES (to v.push_back())
 // 1) ask n.o of employees and do v.resize()
 // 2) declare vector only after asking the n.o of employees

 do
 {
 cout << "Salary (<=0 to terminate) ? ";
 cin >> salary;
 if (salary > 0)
 v.push_back(salary);
 } while (salary > 0);

 return v;
}

/*
Selects elements of vector v whose value belongs to the range [low..high].
Parameters:
v - vector of values
low - low end of the range
high - high end of the range
*/
vector<double> vectorElementsBetween(vector<double> v, double low, double high)
{
 vector<double> result;

 for (size_t i = 0; i < v.size(); i++)
 if (low <= v[i] && v[i] <= high)
 result.push_back(v[i]);

 return result;
}

```

```

void showVector(vector<double> v)
{
 for (size_t i = 0; i < v.size(); i++)
 cout << v[i] << "\n";
}

/*
Raise all values in a vector by a given percentage.
Parameters:
v - vector of values
p - percentage to raise values by; values are raised p/100
*/
void raisesalaries(vector<double> &v, double p)
{
 for (size_t i = 0; i < v.size(); i++)
 v[i] = v[i] * (1 + p / 100);
}

int main()
{
 const double MIDRANGE_LOW = 45000.0;
 const double MIDRANGE_HIGH = 65000.0;
 const double RAISE_PERCENTAGE = 1.0;

 vector<double> salaries;
 vector<double> midrangeSalaries;

 salaries = readSalaries(); //TO DO: return salaries through ref. parameter
 if (salaries.size() > 0)
 {
 cout << "Salaries between " << MIDRANGE_LOW << " and " << MIDRANGE_HIGH << ":\n";
 midrangeSalaries = vectorElemsBetween(salaries, MIDRANGE_LOW, MIDRANGE_HIGH);
 if (midrangeSalaries.size() > 0)
 {
 showVector(midrangeSalaries);
 raisesalaries(midrangeSalaries, RAISE_PERCENTAGE);
 cout << "Raised salaries\n";
 showVector(midrangeSalaries);
 }
 else
 cout << "No salaries to be raised\n";
 }
 else
 cout << "No salaries to be processed\n";

 return 0;
}

```



```

/*
USING VECTORS
Performance tip:
- for very large vectors, pass vectors to functions by reference;
 use qualifier const when vector can't be modified
Quality tip:
- using member function at() from vector class instead of operator []
 signals if the requested position is out of range

```

**Program objective:**  
 Read employee salaries  
 Determine those that have a mid-range salary  
 Raise their salary by a determined percentage  
 \*/

```

#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;

/*
Reads salaries
returns a vector containing the salaries
*/
vector<double> readSalaries()
{
 int salary;
 vector<double> v;

 do
 {
 cout << "Salary (<=0 to terminate) ? ";
 cin >> salary;
 if (salary > 0)
 v.push_back(salary);
 } while (salary > 0);

 return v;
}

/*
Selects elements of vector v whose value belongs to the range [low..high].
Parameters:
 v - vector of values
 low - low end of the range
 high - high end of the range
*/
vector<double> vectorElementsBetween(const vector<double> &v, double low,
double high)
{
 vector<double> result;

 for (size_t i = 0; i < v.size(); i++)
 if (low <= v.at(i) && v.at(i) <= high)
 result.push_back(v.at(i));

 return result;
}

```

```

/*
Shows a vector on screen, one value / line
*/
void showVector(const vector<double> &v)
{
 for (size_t i = 0; i < v.size(); i++)
 cout << v.at(i) << "\n";
}

/*
Raise all values in a vector by a given percentage.
Parameters:
 v - vector of values
 p - percentage to raise values by; values are raised p/100
*/
void raisesalaries(vector<double> &v, double p)
{
 for (size_t i = 0; i < v.size(); i++)
 v.at(i) = v.at(i) * (1 + p / 100);
}

int main()
{
 const double MIDRANGE_LOW = 45000.0;
 const double MIDRANGE_HIGH = 65000.0;
 const double RAISE_PERCENTAGE = 10;

 vector<double> salaries;
 vector<double> midrangeSalaries;
 int numSalaries = 0;

 salaries = readSalaries();
 if (salaries.size() > 0)
 {
 cout << "Salaries between " << MIDRANGE_LOW << " and "
 << MIDRANGE_HIGH << ":\n";
 midrangeSalaries =
 vectorElemsBetween(salaries, MIDRANGE_LOW, MIDRANGE_HIGH);
 if (midrangeSalaries.size() > 0)
 {
 showVector(midrangeSalaries);
 raisesalaries(midrangeSalaries, RAISE_PERCENTAGE);
 cout << "Raised salaries\n";
 showVector(midrangeSalaries);
 }
 else
 cout << "No salaries to be raised\n";
 }
 else
 cout << "No salaries to be processed\n";

 return 0;
}

```

```

/*
VECTORS
The same program as "MULTIDIMENSIONAL ARRAYS - 2D ARRAYS" EXAMPLE,
using vectors instead of arrays

Reads quiz scores for each student into the two-dimensional array grade.
Computes the average score for each student and
the average score for each quiz.
Displays the quiz scores and the averages.
*/

#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <vector>

using namespace std;

const int NUMBER_STUDENTS = 4, NUMBER_QUIZZES = 3;
// TO DO: USER CAN SPECIFY, IN RUNTIME, THOSE NUMBERS - SEE NEXT EXAMPLE

void fill_grades(vector< vector<int>> &grade);
// BE CAREFUL (in some compilers): NOT vector<vector<int>>> . WHY ?
// Microsoft compiler does not care !

void compute_st_ave(const vector< vector<int> > &grade, vector<double> &st_ave);

void compute_quiz_ave(const vector< vector<int> > &grade, vector<double>
&quiz_ave);

void display(const vector< vector<int> > &grade, const vector<double> &st_ave,
const vector<double> &quiz_ave);

//-----

int main()
{
 vector< vector<int> > grade(NUMBER_STUDENTS, vector<int>(NUMBER_QUIZZES));
 vector<double> st_ave(NUMBER_STUDENTS);
 vector<double> quiz_ave(NUMBER_QUIZZES);

 fill_grades(grade);
 compute_st_ave(grade, st_ave);
 compute_quiz_ave(grade, quiz_ave);
 display(grade, st_ave, quiz_ave);
 return 0;
}
//-----

void fill_grades(vector< vector<int> > &grade)
{
 for (int st_num = 0; st_num < NUMBER_STUDENTS; st_num++)
 for (int quiz_num = 0; quiz_num < NUMBER_QUIZZES; quiz_num++)
 grade[st_num][quiz_num] = 10; //10 + rand() % 11;
}
//-----

```

```

void compute_st_ave(const vector< vector<int> > &grade, vector<double> &st_ave)
{
 for (int st_num = 0; st_num < NUMBER_STUDENTS; st_num++)
 {
 double sum = 0;
 for (int quiz_num = 0; quiz_num < NUMBER_QUIZZES; quiz_num++)
 sum = sum + grade[st_num][quiz_num];

 st_ave[st_num] = sum/NUMBER_QUIZZES;
 }
}
//-----

```

```

void compute_quiz_ave(const vector< vector<int> > &grade, vector<double> &quiz_ave)
{
 for (int quiz_num = 0; quiz_num < NUMBER_QUIZZES; quiz_num++)
 {
 double sum = 0;
 for (int st_num = 0; st_num < NUMBER_STUDENTS; st_num++)
 sum = sum + grade[st_num][quiz_num];

 quiz_ave[quiz_num] = sum/NUMBER_STUDENTS;
 }
}
//-----

```

```

void display(const vector< vector<int> > &grade, const vector<double> &st_ave,
 const vector<double> &quiz_ave)
{
 cout.setf(ios::fixed);
 cout.setf(ios::showpoint);
 cout.precision(1);

 cout << setw(10) << "Student"
 << setw(5) << "Ave"
 << setw(15) << "Quizzes\n";

 for (int st_num = 0; st_num < NUMBER_STUDENTS; st_num++)
 {
 cout << setw(10) << st_num + 1
 << setw(5) << st_ave[st_num] << " ";
 for (int quiz_num = 0; quiz_num < NUMBER_QUIZZES; quiz_num++)
 cout << setw(5) << grade[st_num][quiz_num];
 cout << endl;
 }

 cout << "Quiz averages = ";
 for (int quiz_num = 0; quiz_num < NUMBER_QUIZZES; quiz_num++)
 cout << setw(5) << quiz_ave[quiz_num];
 cout << endl;
}
//-----

```

```

/*
VECTORS - a BETTER solution: user chooses vector dimensions, in runtime
The same code as the previous one (using vectors instead of arrays)

Reads quiz scores for each student into the two-dimensional array grade.
Computes the average score for each student and
the average score for each quiz.
Displays the quiz scores and the averages.
*/

#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <vector>
#include <cstdlib>

using namespace std;

void fill_grades(vector< vector<int> > &grade);
void compute_st_ave(const vector< vector<int> > &grade, vector<double> &st_ave);
void compute_quiz_ave(const vector< vector<int> > &grade, vector<double> &quiz_ave);
void display(const vector< vector<int> > &grade, const vector<double> &st_ave, const
vector<double> &quiz_ave);

int main()
{
 size_t numberStudents, numberQuizzes;

 cout << "Number of students ? "; cin >> numberStudents;
 cout << "Number of quizzes ? "; cin >> numberQuizzes;

 vector< vector<int> > grade(numberStudents, vector<int> (numberQuizzes));
 vector<double> st_ave(numberStudents);
 vector<double> quiz_ave(numberQuizzes);

 fill_grades(grade);
 compute_st_ave(grade, st_ave);
 compute_quiz_ave(grade, quiz_ave);
 display(grade, st_ave, quiz_ave);
 return 0;
}

void fill_grades(vector< vector<int> > &grade)
{
 size_t numberStudents = grade.size();
 size_t numberQuizzes = grade[0].size();

 for (size_t st_num = 0; st_num < numberStudents; st_num++)
 for (size_t quiz_num = 0; quiz_num < numberQuizzes; quiz_num++)
 grade[st_num][quiz_num] = 10 + rand() % 11;
}

```

```

void compute_st_ave(const vector< vector<int> > &grade, vector<double>
&st_ave)
{
 size_t numberStudents = grade.size();
 size_t numberQuizzes = grade[0].size();

 for (size_t st_num = 0; st_num < numberStudents; st_num++)
 {
 double sum = 0;
 for (size_t quiz_num = 0; quiz_num < numberQuizzes; quiz_num++)
 sum = sum + grade[st_num][quiz_num];
 st_ave[st_num] = sum/numberQuizzes;
 }
}

```

```

void compute_quiz_ave(const vector< vector<int> > &grade, vector<double>
&quiz_ave)
{
 size_t numberStudents = grade.size();
 size_t numberQuizzes = grade[0].size();

 for (size_t quiz_num = 0; quiz_num < numberQuizzes; quiz_num++)
 {
 double sum = 0;
 for (size_t st_num = 0; st_num < numberStudents; st_num++)
 sum = sum + grade[st_num][quiz_num];

 quiz_ave[quiz_num] = sum/numberStudents;
 }
}

```

```

void display(const vector< vector<int> > &grade, const vector<double>
&st_ave, const vector<double> &quiz_ave)
{
 size_t numberStudents = grade.size();
 size_t numberQuizzes = grade[0].size();

 cout.setf(ios::fixed);
 cout.setf(ios::showpoint);
 cout.precision(1);
 cout << setw(10) << "Student"
 << setw(5) << "Ave"
 << setw(15) << "Quizzes\n";
 for (size_t st_num = 0; st_num < numberStudents; st_num++)
 {
 cout << setw(10) << st_num + 1
 << setw(5) << st_ave[st_num] << " ";
 for (size_t quiz_num = 0; quiz_num < numberQuizzes; quiz_num++)
 cout << setw(5) << grade[st_num][quiz_num];
 cout << endl;
 }

 cout << "Quiz averages = ";
 for (size_t quiz_num = 0; quiz_num < numberQuizzes; quiz_num++)
 cout << setw(5) << quiz_ave[quiz_num];
 cout << endl;
}

```

```

/*
VECTORS - Yet another solution, using push_back()

Reads quiz scores for each student into the two-dimensional array grade.
Computes the average score for each student and
the average score for each quiz.
Displays the quiz scores and the averages.
*/

#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <vector>
#include <cstdlib>

using namespace std;

void fill_grades(vector< vector<int> > &grade, size_t numberStudents,
size_t numberQuizzes);

void compute_st_ave(const vector< vector<int> > &grade, vector<double>
&st_ave, size_t numberStudents, size_t numberQuizzes);

void compute_quiz_ave(const vector< vector<int> > &grade, vector<double>
&quiz_ave, size_t numberStudents, size_t numberQuizzes);

void display(const vector< vector<int> > &grade, const vector<double>
&st_ave, const vector<double> &quiz_ave, size_t numberStudents, size_t
numberQuizzes);

int main()
{
 vector< vector<int> > grade; // how many elements has 'grade' vector ?
 vector<double> st_ave; // and 'st_ave' & 'quiz_ave' vectors ?
 vector<double> quiz_ave;

 size_t numberStudents, numberQuizzes;

 cout << "Number of students ? "; cin >> numberStudents;
 cout << "Number of quizzes ? "; cin >> numberQuizzes;
 fill_grades(grade, numberStudents, numberQuizzes);
 compute_st_ave(grade, st_ave, numberStudents, numberQuizzes);
 compute_quiz_ave(grade, quiz_ave, numberStudents, numberQuizzes);
 display(grade, st_ave, quiz_ave, numberStudents, numberQuizzes);
 return 0;
}

void fill_grades(vector< vector<int> > &grade, size_t numberStudents,
size_t numberQuizzes)
{
 for (size_t st_num = 0; st_num < numberStudents; st_num++)
 {
 vector<int> studentGrade;
 for (size_t quiz_num = 0; quiz_num < numberQuizzes; quiz_num++)
 studentGrade.push_back(10 + rand() % 11);
 grade.push_back(studentGrade);
 }
}

```

```

void compute_st_ave(const vector< vector<int> > &grade, vector<double>
&st_ave, size_t numberStudents, size_t numberQuizzes)
{
 // numberStudents = grade.size(); //alternative to parameters
 // numberQuizzes = grade[0].size(); //alternative to parameters

 for (size_t st_num = 0; st_num < numberStudents; st_num++)
 {
 double sum = 0;
 for (size_t quiz_num = 0; quiz_num < numberQuizzes; quiz_num++)
 sum = sum + grade[st_num][quiz_num];

 st_ave.push_back(sum/numberQuizzes);
 //WHY NOT st_ave[st_num] = sum/numberQuizzes; ???
 }
}

```

```

void compute_quiz_ave(const vector< vector<int> > &grade, vector<double>
&quiz_ave, size_t numberStudents, size_t numberQuizzes)
{
 for (size_t quiz_num = 0; quiz_num < numberQuizzes; quiz_num++)
 {
 double sum = 0;
 for (size_t st_num = 0; st_num < numberStudents; st_num++)
 sum = sum + grade[st_num][quiz_num];

 quiz_ave.push_back(sum/numberStudents);
 }
}

```

```

void display(const vector< vector<int> > &grade, const vector<double>
&st_ave, const vector<double> &quiz_ave, size_t numberStudents, size_t
numberQuizzes)
{
 cout.setf(ios::fixed);
 cout.setf(ios::showpoint);
 cout.precision(1);
 cout << setw(10) << "Student"
 << setw(5) << "Ave"
 << setw(15) << "Quizzes\n";
 for (size_t st_num = 0; st_num < numberStudents; st_num++)
 {
 cout << setw(10) << st_num + 1
 << setw(5) << st_ave[st_num] << " ";
 for (size_t quiz_num = 0; quiz_num < numberQuizzes; quiz_num++)
 cout << setw(5) << grade[st_num][quiz_num];
 cout << endl;
 }

 cout << "Quiz averages = ";
 for (size_t quiz_num = 0; quiz_num < numberQuizzes; quiz_num++)
 cout << setw(5) << quiz_ave[quiz_num];
 cout << endl;
}

```



---

## STRINGS

---

- In computer programming, a **string** is traditionally a sequence of characters,
  - either as a literal constant or
  - as some kind of variable.
- The latter may allow its elements to be mutated and/or the length changed, or it may be constant (after creation).

### C-Strings & C++-strings

- **C-strings** are stored as arrays of characters  
=>  
`#include <cstring>`
- **C++ strings** are objects of the String class, part of the std namespace  
=>  
`#include <string>`  
`using namespace std;`

---

## C - STRINGS

---

### C-string declaration & representation

- To declare a C-string variable, declare an array of characters:
  - `char s[10];`
- C-strings use the null character `'\0'` (character with ASCII code zero) to end a string; the null character immediately follows the last character of the string
- **Be careful**, don't forget to **allocate space for the ending null char**:
  - `char stringName[MAXIMUM_STRING_SIZE + 1];`
  - `MAXIMUM_STRING_SIZE` is some value that you must define
  - `+ 1` reserves the additional character needed by `'\0'`
- Declaring a C-string as `char s[10]` creates space for only nine characters
  - the null character terminator requires one space

## Initializing a C-string

- Initialization of a C-string during declaration (**bad solution**):
  - `char salut[ ] = {'H','i','!','\0'}; // NOTE the ending '\0'`
- Better alternative:
  - `char salut[ ] = "Hi!"; // the null char '\0' is added for you`
- `char anotherSalut[20] = "Hi there!";`  
`// the characters with index 10..19 have an undetermined value`
- **NOTE:**
  - do not to replace the null character when manipulating indexed variables in a C-string
  - If the null character is "lost", the array cannot act like a C-string

## C-string Output

- C-strings can be written with the insertion operator (<<)
- Example:  
`char msg[ ] = "Hello";`  
`cout << msg << " world!" << endl;`

## C-string Input

- C-strings can be read with the extraction operator (>>)
- **NOTE:**
  - whitespace (' ', \n, \t, ...) ends reading of data ;
  - whitespace remains in the input buffer
- Example:  
`char name[80];`  
`cout << "Your name? " << endl;`  
`cin >> name; // enter "Rui Sousa"; " Sousa" remains in the buffer!`

## Reading an entire Line

- Predefined member function `getline()` can read an entire line, including spaces
- `getline()` is a **member function** of all input streams
  - `istream& getline (char *s, streamsize n );`
  - `istream& getline (char *s, streamsize n, char delim );`
- Calling: `streamName.getline(.....); // ex: cin.getline()`

- **cin.getline()**
  - extracts characters from the stream as unformatted input and
  - stores them into **s** as a **C-string**,
  - until either the extracted character is the delimiting character (**'\n'** or **delim**),
  - or **n** characters have been written to **s** (including the terminating null character); in this case, **getline()** stops even if the end of the line has not been reached.
- The delimiting character is:
  - the newline character (**'\n'**) for the first form of **getline()**, and
  - **delim** for the second form.
- When found in the input sequence, the delimiting character,
  - is extracted from the input sequence,
  - but discarded and not written to **s**.
- **NOTE:**
  - If the function stops reading because **n** characters have been read without finding the delimiting character, the failbit internal flag is set (=> **cin.clear()**) but the additional characters are removed from the buffer.

### Accessing string elements

- The elements of a string are accessed just like the elements of an array.
- Example:

```
char s[5]="Hi!";
s[1] = 'o';
cout << s << endl; // what is the output?
```
- **Be careful**, when accessing the elements of a string.
  - Do not access characters past the end of the array of chars!
  - When modifying it, do not forget that the ending null char must be present

### Assignment

- The assignment operator does not work with C-strings
- This statement is illegal:
  - **a\_string = "Hello";**
  - this is an assignment statement, not an initialization.

- A common method to assign a value to a C-string variable is to use function **strcpy()**, defined in the cstring library
- Example:  

```
char msg[10];
strcpy (msg, "Hello"); // places "Hello" followed by '\0' in msg
```
- **NOTE**: **strcpy()** can create problems if not used carefully
  - **strcpy()** does not check the declared length of destination string
  - it is possible for **strcpy()** to write characters beyond the declared size of the array (see **strncpy()** )

## Comparison

- The == operator does not work as expected with C-strings.
- The predefined function **strcmp()** is used to compare C-string variables
  - ```
int strcmp ( const char str1[ ], const char str2[ ] );
```


// why is the number of chars of each string not needed as in other functions that have parameters of type 'array'?
 - As we shall see later, this prototype can also be written as:

```
int strcmp ( const char *str1, const char *str2 );
```
- **strcmp()** returns an integral value indicating the relationship between the strings:
 - a zero value indicates that both strings are equal;
 - a value greater than zero indicates that the first character that does not match has a greater value in **str1** than in **str2** ;
 - a value less than zero indicates the opposite.
- Example:

```
if (strcmp(cstr1, cstr2)) // ⇔ ? if (strcmp(cstr1, cstr2) != ???)
    cout << "Strings are not the same.";
else
    cout << "String are the same.";
```

Converting C-strings to numbers

- "1234" and "12.3" are strings of characters
- 1234 and 12.3 are numbers

- There functions for converting strings to numbers (=> **#include <cstdlib>**)
 - **atoi()** – convert C-string to integer
 - **atol()** – convert C-string to long integer
 - **atof()** – convert C-string to double
- Example:
 - atoi("1234")** returns the integer 1234
 - atoi("#123")** returns 0 because **#** is not a digit
 - atof("9.99")** returns 9.99
 - atof("\$9.99")** returns 0.0 because **\$** is not a digit

Concatenation

- **strcat()** concatenates two C-strings

Other C-string operations

- *see table in the next pages*

C-strings as arguments and parameters

- C-string variables are arrays
- C-string arguments and parameters are used just like arrays

The standard string class (C++ strings)

The standard string class

- The **string** class allows the programmer to treat strings as a basic data type.
- **No need to deal with the implementation as with C-strings.**
- The string class is defined in the **string** library and the names are in the standard namespace
- To use the string class you need these lines:

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

Declaration and assignment of strings

- The default string constructor initializes the string to the empty string
 - *class constructors will be introduced later*
- Another string constructor takes a C-string argument
- Example:

```
string phrase;           // empty string
string name("John");    // calls the string constructor
```
- **Variables of type string can be assigned with the = operator**
- Example:

```
string s1,s2,s3;
...
s1 = "Hello Mom!";
...
s3 = s2;
```

I/O with class string

- The insertion operator << is used to output objects of type string
- Example:

```
string s = "Hello Mom!";
cout << s;
```

- The extraction operator >> can be used to input data for objects of type string
- Example:

```
string s1;
cout << "What is your name ? " ; cin >> s1;
```
- **NOTE:**
 - whitespace (' ', \n, \t, ...) ends reading of data ;
 - whitespace remains in buffer

Accessing string elements

- characters in a string object can be accessed as if they are in an array
- as in an array, index values are not checked for validity!
- **at()** is an alternative to using []'s to access characters in a string.
- **at()** checks for valid index values
- Example:

```
string str("Mary");
cout << str[6] << endl; // INVALID ACCESS; DETECTED IN VStudio Debug mode
cout << str.at(6) << endl; // INVALID ACCESS IS DETECTED
```

Comparison of strings

- Comparison operators work with string objects.
- Objects are compared using lexicographic order (alphabetical ordering using the order of symbols in the ASCII character set.)
- == returns true if two string objects contain the same characters in the same order
 - remember **strcmp()** for C-strings? 😞
- <, >, <=, >= can be used to compare string objects

Strings concatenation

- Variables of type string can be concatenated with the + operator
- Example:

```
s3 = s1 + s2;
```

 - If **s3** is not large enough to contain **s1 + s2**, more space is allocated

String length

- The string class member function **length** returns the number of characters in the string object:
- Example:
`size_t n = s.length();`

Converting C-strings to string objects

- Recall the automatic conversion from C-string to string:
`char cstr[] = "C-string";`
`string str = cstr; // OR string str(cstr);`

Converting strings objects to C-strings

- The string class member function **c_str()** returns the C-string version of a string object
- Example:
`strcpy(cstringVariable, stringVariable.c_str());`

Mixing strings and C-strings

- It is natural to work with strings in the following manner:
`string phrase = "I like " + noun + "!";`
- It is not so easy for C++!
It must either convert the null-terminated C-strings, such as "I like", to strings, or it must use an overloaded **operator+** that works with strings and C-strings

getline for type 'string'

- A **getline()** function exists to read entire lines into a **string** variable
- This version of **getline** is not a member of the **istream** class, it is a non-member function.
- **getline()** declarations:
 - `istream& getline (istream &is, string &str, char delim);`
 - `istream& getline (istream &is, string &str);`
- Extracts characters from **is** (input stream) and stores them into **str** until
 - the delimitation character **delim** is found (1st prototype)
 - or the newline character, **'\n'** (2nd prototype)

- The extraction also stops
 - if the end of file is reached in **is** (*see later*)
 - or if some other error occurs during the input operation.
- If the delimiter is found,
 - it is extracted and discarded, i.e. it is not stored and the next input operation will begin after it.
- Example:

```
cout << "Enter your full name:\n"; //now you can enter "Rui Sousa" ☺
getline(cin, name);
```

Mixing "cin >>" and "getline" (BE CAREFUL !!!)

- Recall **cin >>** skips whitespace to find what it is to read then stops reading when whitespace is found
- **cin >>** leaves the '\n' character in the input stream
- Example:

```
int n;
string line;
cin >> n; // leaves the '\n' in the input buffer
getline(cin, line); // returns immediately;
// 'line' is set equal to the empty string.
```

Other string operations

- *Ex: finding substrings =>*
 - *see the examples in these lecture notes*
 - *consult some manuals / web pages*
- **std::string::npos**
 - This value, when used as the value for a len (or sublen) parameter in string's member functions, means "until the end of the string"
 - As a return value, it is usually used to indicate no matches
 - **npos** is a static member constant value with the greatest possible value for an element of type **size_t**
 - It is defined with a value of -1, which because **size_t** is an unsigned integral type, it is the largest possible representable value for this type

STRINGS

ARRAY OF STRUCTS

```
/*
C STRINGS
are arrays of characters, terminated by a null character, '\0'
*/

#include <iostream>
#include <iomanip>
#include <cstring>

using namespace std;

int main()
{
    const int MAX_NAME_LEN = 10;

    //string declarations C-style
    char name[MAX_NAME_LEN];
    char salutation[] = "Hello "; // string declaration with initialization

    cout << "Your name ? "; //try with "Rui" "Alexandrino" and "Rui Sousa"
    cin >> name;
    cout << salutation << name << "!\n";
    //cout << sizeof(salutation) << endl;

    /*
    //show 'name' characters (not only ...)
    for (unsigned i=0; i<MAX_NAME_LEN; i++) //TRY: for (unsigned i=0; i<strlen(name);
i++)
        cout << setw(4) << (unsigned) name[i] << " - " << name[i] << endl;
    */

    return 0;
}
```

```

/*
C++ STRINGS
*/

#include <iostream>
#include <string>

using namespace std;

int main()
{
    string name; //string declaration; MAX. LENGTH NOT NECESSARY :-)

    cout << "Your name ? "; //try with "Rui" and "Rui Sousa"
    cin >> name;
    cout << "Hello " << name << "!\n";

    return 0;
}

```

```

/*
C++ STRINGS
getline()
string member functions call: length(), find_last_of(), substr()
*/

#include <iostream>
#include <string>
#include <cstdlib>

using namespace std;

int main()
{
    string name, lastName;
    size_t posLastSpace;

    cout << "Your full name ? "; //try with "Rui" and "Rui Sousa"
    getline(cin,name);

    cout << "Hello " << name << "!\n";

    posLastSpace = name.find_last_of(' ');
    if (posLastSpace == string::npos) // no space character was found
        cout << "Your name has only one word ?!\n";
    else
    {
        lastName = name.substr(posLastSpace+1,name.length()-posLastSpace-1);
        cout << "Your last name is: " << lastName << endl;
    }

    return 0;
}

```

TO DO, BY THE STUDENTS:

search for other methods (similar to find_last_of(), substr() and length()) of the string class, for string manipulation.
The class and method concepts will be introduced later.

```

/*
C++ STRINGS
string concatenation
*/

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

int main()
{
    string name, salutation;

    cout << "Your name ? ";
    getline(cin,name);
    salutation = "Hello " + name + "!\n"; //can't do this with C-strings
    cout << salutation;

    return 0;
}

```

```

/*
C++ STRINGS
accessing string elements
passing string parameters by reference
*/

#include <iostream>
#include <iomanip>
#include <string>
#include <cctype> //toupper()
#include <cstdint> //size_t

using namespace std;

void strToUpper(string &str) // NOTE : string reference. WHY ?
{
    for (size_t i=0; i<str.length(); i++)
        str[i] = toupper(str[i]);
    // str[i] = toupper(str.at(i)); //checks for valid index, i
}

int main()
{
    string name, salutation;

    cout << "Your name ? ";
    getline(cin,name);
    strToUpper(name);
    salutation = "Hello " + name + "!\n";
    cout << salutation;

    return 0;
}

```

```

/*
C++ STRINGS
array of strings
*/

#include <iostream>
#include <iomanip>
#include <string>
#include <cctype> //toupper()
#include <cstdint> //size_t

using namespace std;

void readNames(string names[], size_t n)
{
    for (size_t i=0; i < n ; i++)
    {
        cout << "Name [" << i << "] ? ";
        getline(cin,names[i]);
    }
}

//shows names in array names[] right-aligned
void showNames(const string names[], size_t n)
{
    size_t maxNameLen = 0;
    for (size_t i=0; i < n ; i++)
        if (names[i].length() > maxNameLen)
            maxNameLen = names[i].length();

    for (size_t i=0; i < n ; i++)
        cout << setw(maxNameLen) << names[i] << endl;
}

int main()
{
    const unsigned NUM_NAMES = 5;

    string names[NUM_NAMES];

    readNames(names,NUM_NAMES);
    showNames(names,NUM_NAMES);

    return 0;
}

```

TO DO, BY THE STUDENTS:

Do the same using a vector instead of an array.

```

/*
C++ STRINGS
be careful when mixing "getline()" and "cin >> variable"
*/

#include <iostream>
#include <iomanip>
#include <string>
#include <cctype> //toupper()
#include <cstdlib> //size_t

using namespace std;

void readNames(string names[], unsigned ages[], size_t n)
{
    for (size_t i=0; i < n ; i++)
    {
        cout << "Name [" << i << "] ? ";
        getline(cin,names[i]);
        cout << "Age [" << i << "] ? ";
        cin >> ages[i];
        // BE CAREFUL WHEN MIXING getline() AND cin >> variable
        // WHICH IS THE SOLUTION ?
    }
}

//shows names in vector nms[] right-aligned
void showNames(string names[], unsigned ages[], size_t n)
{
    size_t maxNameLen = 0;
    for (size_t i=0; i < n ; i++)
        if (names[i].length() > maxNameLen)
            maxNameLen = names[i].length();

    for (size_t i=0; i < n ; i++)
        cout << setw(maxNameLen) << names[i] <<
            setw(3) << ages[i] << endl;
}

int main()
{
    const unsigned NUM_NAMES = 5;

    string names[NUM_NAMES];
    unsigned ages[NUM_NAMES];

    readNames(names,ages,NUM_NAMES);
    showNames(names,ages,NUM_NAMES);

    return 0;
}

```

```

/*
STRUCTS
ARRAY OF STRUCTS
Using typedef keyword to form an alias for a type
*/
#include <iostream>
#include <iomanip>
#include <string>
#include <cctype>
#include <cstdlib>

using namespace std;

typedef struct
{
    string name;
    unsigned age;
} NameAge;
// typedef works both in C and C++;
// ALTERNATIVE C++, only
// struct NameAge {...};
// parameter types remain the same

void readNames(NameAge namesAndAges[], size_t n)
{
    for (size_t i=0; i < n ; i++)
    {
        cout << "Name [" << i << "] ? ";
        getline(cin, namesAndAges[i].name);
        cout << "Age [" << i << "] ? ";
        cin >> namesAndAges[i].age;
        cin.ignore(1000, '\n'); // solves the "mixing problem"
    }
}

//shows names in vector namesAndAges[] right-aligned
void showNames(NameAge namesAndAges[], size_t n)
{
    size_t maxNameLen = 0;
    for (size_t i=0; i < n ; i++)
        if (namesAndAges[i].name.length() > maxNameLen)
            maxNameLen = namesAndAges[i].name.length();

    for (size_t i=0; i < n ; i++)
        cout << setw(maxNameLen) << namesAndAges[i].name <<
            setw(3) << namesAndAges[i].age << endl;
}

int main()
{
    const unsigned NUM_NAMES = 5;
    NameAge namesAndAges[NUM_NAMES]; // TO DO: use vectors instead of arrays

    readNames(namesAndAges, NUM_NAMES);
    cout << endl;
    showNames(namesAndAges, NUM_NAMES);

    return 0;
}

```

Some Predefined C-String Functions in <cstring> (part 1 of 2)

Function	Description	Cautions
 strcpy(<i>Target_String_Var</i> , <i>Src_String</i>)	Copies the C-string value <i>Src_String</i> into the C-string variable <i>Target_String_Var</i> .	Does not check to make sure <i>Target_String_Var</i> is large enough to hold the value <i>Src_String</i> .
strncpy(<i>Target_String_Var</i> , <i>Src_String</i> , <i>Limit</i>)	The same as the two-argument strcpy except that at most <i>Limit</i> characters are copied.	If <i>Limit</i> is chosen carefully, this is safer than the two-argument version of strcpy. Not implemented in all versions of C++.
strcat(<i>Target_String_Var</i> , <i>Src_String</i>)	Concatenates the C-string value <i>Src_String</i> onto the end of the C string in the C-string variable <i>Target_String_Var</i> .	Does not check to see that <i>Target_String_Var</i> is large enough to hold the result of the concatenation.
strncat(<i>Target_String_Var</i> , <i>Src_String</i> , <i>Limit</i>)	The same as the two-argument strcat except that at most <i>Limit</i> characters are appended.	If <i>Limit</i> is chosen carefully, this is safer than the two-argument version of strcat. Not implemented in all versions of C++.
 strlen(<i>Src_String</i>)	Returns an integer equal to the length of <i>Src_String</i> . (The null character, '\0', is not counted in the length.)	
 strcmp(<i>String_1</i> , <i>String_2</i>)	Returns 0 if <i>String_1</i> and <i>String_2</i> are the same. Returns a value < 0 if <i>String_1</i> is less than <i>String_2</i> . Returns a value > 0 if <i>String_1</i> is greater than <i>String_2</i> (that is, returns a nonzero value if <i>String_1</i> and <i>String_2</i> are different). The order is lexicographic.	If <i>String_1</i> equals <i>String_2</i> , this function returns 0, which converts to <i>false</i> . Note that this is the reverse of what you might expect it to return when the strings are equal.
strncmp(<i>String_1</i> , <i>String_2</i> , <i>Limit</i>)	The same as the two-argument strcmp except that at most <i>Limit</i> characters are compared.	If <i>Limit</i> is chosen carefully, this is safer than the two-argument version of strcmp. Not implemented in all versions of C++.



Member Functions of the Standard Class string

Example

Constructors

<code>string str;</code>	Default constructor creates empty string object <code>str</code> .
<code>string str("sample");</code>	Creates a string object with data "sample".
<code>string str(a_string);</code>	Creates a string object <code>str</code> that is a copy of <code>a_string</code> ; <code>a_string</code> is an object of the class <code>string</code> .

Element access

 <code>str[i]</code>	Returns read/write reference to character in <code>str</code> at index <code>i</code> . Does not check for illegal index.
 <code>str.at(i)</code>	Returns read/write reference to character in <code>str</code> at index <code>i</code> . Same as <code>str[i]</code> , but this version checks for illegal index.
<code>str.substr(position, length)</code>	Returns the substring of the calling object starting at <code>position</code> and having <code>length</code> characters.

Assignment/modifiers

<code>str1 = str2;</code>	Initializes <code>str1</code> to <code>str2</code> 's data,
<code>str1 += str2;</code>	Character data of <code>str2</code> is concatenated to the end of <code>str1</code> .
<code>str.empty()</code>	Returns <i>true</i> if <code>str</code> is an empty string; <i>false</i> otherwise.
<code>str1 + str2</code>	Returns a string that has <code>str2</code> 's data concatenated to the end of <code>str1</code> 's data.
<code>str.insert(pos, str2);</code>	Inserts <code>str2</code> into <code>str</code> beginning at position <code>pos</code> .
<code>str.remove(pos, length);</code>	Removes substring of size <code>length</code> , starting at position <code>pos</code> .

Comparison

<code>str1 == str2 str1 != str2</code>	Compare for equality or inequality; returns a Boolean value.
<code>str1 < str2 str1 > str2</code>	Four comparisons. All are lexicographical comparisons.
<code>str1 <= str2 str1 >= str2</code>	

Finds

<code>str.find(str1)</code>	Returns index of the first occurrence of <code>str1</code> in <code>str</code> .
<code>str.find(str1, pos)</code>	Returns index of the first occurrence of string <code>str1</code> in <code>str</code> ; the search starts at position <code>pos</code> .
<code>str.find_first_of(str1, pos)</code>	Returns the index of the first instance in <code>str</code> of any character in <code>str1</code> , starting the search at position <code>pos</code> .
<code>str.find_first_not_of(str1, pos)</code>	Returns the index of the first instance in <code>str</code> of any character not in <code>str1</code> , starting the search at position <code>pos</code> .

```

/*
C-STRINGS and C++-STRINGS
Converting between each other
Comparing strings
Nobody would use two different types of strings to do this !!!
Just for illustrating string conversions
*/

#include <iostream>
#include <iomanip>
#include <cstring>
#include <string>

using namespace std;

int main()
{
    const int MAX_CODE_LEN = 80;

    char code1[MAX_CODE_LEN];
    string code2;

    cout << "Type your code : "; // ex: A1B2 C3B4 D5E6
    cin.getline(code1,MAX_CODE_LEN); // cin.getline( ) ONLY FOR C-strings
    cout << code1 << endl;

    cout << "Retype your code : ";
    getline(cin,code2); // getline( ) ONLY FOR C++-strings
    cout << code2 << endl;

    // CHECKING WHETHER THE 2 CODES ARE EQUAL ...

    // version 1 - convert both strings to C-style strings
    char code2aux[MAX_CODE_LEN];
    strcpy(code2aux, code2.c_str());
    cout << "test1: ";
    if (strcmp(code1,code2aux) == 0)
        cout << "codes are equal\n";
    else
        cout << "codes are different\n";

    // version 2 - convert both strings to C++-style strings
    string code1aux(code1); //OR: string code1aux = string(code1);
    cout << "test2: ";
    if (code2 == code1aux)
        cout << "codes are equal\n";
    else
        cout << "codes are different\n";

    return 0;
}

```

TO DO BY STUDENTS:
investigate the behaviour of `cin.getline()`
when more than `MAX_CODE_LEN` characters are inserted

```

/*
READING WITHOUT ECHO
Microsoft C/C++ compiler specific
*/

#include <iostream>
#include <iomanip>
#include <cstring>
#include <string>
#include <conio.h> //needed for _getch();

using namespace std;

int main()
{
    const char ENTER = 13;
    char ch;
    string password;

    cout << "Password ? ";

    while (( ch = _getch()) != ENTER) // Microsoft C/C++ compiler specific
    {
        password = password + ch;
        cout << "*";
    }

    cout << endl << password << endl; // you shouldn't do this ... :-)
    return 0;
}
-----
/*
READING WITHOUT ECHO
*/
#include <iostream>
#include <iomanip>
#include <cstring>
#include <string>
#include <conio.h>

using namespace std;

int main()
{
    const char ENTER = 13;
    char ch;

    cout << "chars (end with Z) ? \n";
    do
    {
        ch = _getch(); //note: returns 13 (CARRIAGE RETURN) when ENTER key is typed
        cout << ch << "- " << unsigned(ch) << endl;
        // TRY: cin.get(ch);
    } while (ch != 'Z');

    return 0;
}

```

ARRAYS, POINTERS and REFERENCES

STATICALLY and DYNAMICALLY ALLOCATED MEMORY

Pointers

- A **pointer** is a **variable that holds a memory address**.
- This address is the location of another variable (or object) in memory.
- If one variable contains the address of another variable, the first variable is said to **point to** the second.

Pointer variables

- General form of declaring a pointer variable:

*typeName *varName;*

- Examples:

- `int *ptr1;`
 - **OR**
 - `int * ptr1;`
 - `int* ptr1;`
- **BE CAREFUL!**
 - `int* ptr3, ptr4;`
 - `ptr3` is of type "int pointer", but `ptr4` is of type "int"
- `double *dPtr;`
- `char *chPtr;`

Pointer operators

- There are 2 special pointer **operators**: **&** and ***** (both are unary operators).
 - **&** - returns the memory address of its operand
 - `int *xPtr;`
 - `xPtr = &x; // xPtr receives "the address of x"`
 - assume that the value of `x` is 10 and that this value is stored at address 2000 of the memory; then `xPtr` will have value 2000.
 - ***** - returns the value located at the address that follows
 - `int y;`
 - `y = *xPtr; // y receives the "value at address xPtr" or "the value pointed to by xPtr"`
 - considering the example above `y` takes the value stored at address 2000, that is 10.

- **NOTE:**
 - make sure that your pointer variables point to the correct type of data
 - Example: if you declare a pointer of type `int`, the compiler assumes that any address that it holds points to an integer variable, whether it actually does or not.

Pointer assignments

- As with any **simple variable**, you may use a pointer on the right-hand side of an assignment statement, to assign its value to another pointer.

```
int x;
int *p1, *p2;
p1 = &x;
p2 = p1;
```

Pointer arithmetic

- Only 2 arithmetic operations may be used with pointers:
 - **addition** and **subtraction**
 - operators `++` and `--` can be used with pointers
- Each time a pointer is incremented / decremented it points to the next / previous location of its base type
- When a value `i` is added / subtracted to / from a pointer `p` the pointer value will increase / decrease by the length of `i * sizeof(pointed_data_type)`
 - `int *p1, *p2;`
 - `p1 = 2000; // usually, you shouldn't do this`
 - `p2 = p1 + 3; // if sizeof(int) is 4,`
`// p2 will point to address 2000 + 3*4 = 2012`

Pointer comparison

- You can compare 2 pointers in a relational expression:
 - `if (p1 < p2) cout << "p1 points to lower memory than p2\n";`

Pointers and arrays

- There is a close relationship between pointers and arrays.
- An array name without an index returns the address of the first element of the array.
- So it is possible to assign an array identifier to a pointer provided that they are of the same type.
 - `int a[10];`
 - `int *p;`
 - `p = a;` `// p points to the first element of array a[]`
 - `p = &a[0];` `// equivalent to the previous assignment`
`// taking into account these declarations and`
`// the pointer arithmetic rules (above),`

- `// the following two statements are equivalent`
 - `// (both access the 4th element of the array):`
 - `o a[3] = 27;`
 - `o *(p+3) = 27;`

- Also, the following code is syntactically correct:

```
void showArray(const int *a, size_t size) { ... }
...
int values[10];
... // fill array "values"
showArray(values,10);
```

- The following 2 declarations with initialization are equivalent:
 - `o char s[] = "Hello!"; // s can be modified`
 - `o char *s = "Hello!"; // s can't be modified`
 but the 1st string can be modified
 while the 2nd can't !!!
 (it is stored in non-modifiable memory)

Initializing pointers

- After a local pointer is declared but before it has been assigned a value, it contains an **unknown value**.
- Global pointers are automatically initialized to **NULL** (equal to zero).
 - Address zero can not be accessed by user programs.
 - Programmers frequently assign the **NULL** value to a **pointer**, meaning that it points to nothing and should not be used.
- BE CAREFUL**: Should you try to use the pointer before giving it a valid value, you will probably crash your program.

Multiple indirection

- You can have a pointer that points to another pointer that points to the target value.
 - `o int **p; // p is a pointer to pointer that points to an int`
 - Example:
 - `▪ int x, *p1, **p2;`
 - `▪ x = 5;`
 - `▪ p1 = &x;`
 - `▪ p2 = &p1;`
 - `▪ cout << "x = " << **p2 << endl;`
- You can have multiple levels of indirection.
- See examples in the next pages on
 - 2D arrays with dynamic allocation
 - accessing command line arguments
 - `▪ int main(int argc, char **argv) {.....};`
 - OR
 - `▪ int main(int argc, char *argv[]) {.....};`

Pointers to functions

- Even though a function is not a variable, it still has a physical address in memory that can be assigned to a pointer.
- This address is the entry point of the function.
- Once a pointer points to a function, the functions can be called through that pointer.
- Example:

```
int sum(int x, int y)
{
    return x+y;
}

int main( )
{
    int result;
    int (*p) (int, int); // p is a pointer to a function that has
                        // 2 int parameters and returns int

    p=sum;               // now, p contains the starting address of sum()

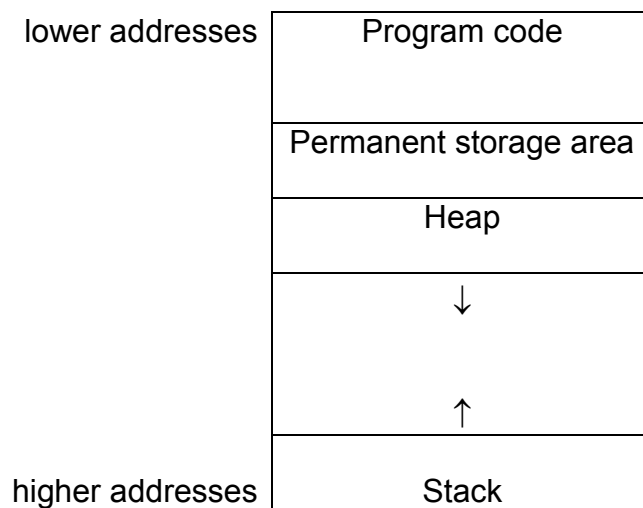
    result = (*p)(2,3); // sum() is called using pointer p !!!
    cout << result << endl;
}
```

References and Pointers

- A **reference** is essentially an **implicit pointer**.
- By far, the most common use of references is
 - to pass an argument to a function using **call-by-reference** (*already seen*)
 - to act as a **return value** from a function (*an example will be seen later*)
- A **reference is a pointer in disguise**
 - When you use references the compiler automatically passes parameters addresses
 - and dereferences the pointer parameters in the function body.
 - For that reason, **in some situations**, references are **more convenient for the programmer than explicit pointers**
- Example:
 - *see example of swap() functions in the next pages.*
- **NOTE:**
 - all independent references must be initialized in declaration
 - `int &r = x; // an independent reference`
 - independent pointers can be declared without being initialized
 - `int *p;`
 - ... but ... don't forget to initialize them before use
 - sometimes they are initialized as the result of a `malloc()` / `new` call (*see next pages*)

Dynamic memory allocation

- **Pointers** provide necessary **support** for C/C++ dynamic memory allocation system.
- **Dynamic memory allocation** is the means by which a program can obtain memory while it is running.
- Global variables are allocated storage at compile time.
- Local variables use the **stack**.
- However, neither global nor local variables can be added during program execution.
- Yet, there will be times where the storage needs of a program cannot be known when the program is being written.
This is why dynamic memory allocation is useful.
- Dynamic memory allocation is particularly useful when you are programming in C.
 - As we have seen, some C++ data structures (ex: strings and vector) can change size dynamically.
- C++ supports **2 dynamic allocations systems**:
 - the one **defined by C**
 - the one **defined by C++**
- Memory allocated by dynamic allocation functions is obtained from the **heap**.



C dynamic memory allocation

- The core of C's allocation system consists of the functions: **malloc()** and **free()**
 - => **#include <stdlib>**
 - **void *malloc(size_t number_of_bytes);**
 - **number_of_bytes** is the number of bytes of memory you wish to allocate
 - the return value is a **void pointer**
 - **in C**, a **void *** can be assigned to another type of pointer; it is automatically converted
 - **in C++**, an explicit type cast is needed when a **void *** is assigned to another type of pointer
 - after a successful call, **malloc()** returns a pointer to the first byte of memory allocated from the heap
 - if there is not enough memory available **malloc()** returns a **NULL pointer**
 - Example:

```
int *p; int n;
cout << "n ? ", cin >> n;
p = (int *) malloc(n*sizeof(int)); // allocate space for
                                   // n consecutive integers
                                   // = an array of integers

if (p == NULL) {
    cout << "Out of heap memory !\n";
    exit(1);
}
```
 - **NOTE:** the contents of the allocated memory is unknown
 - **void free(void *p)**
 - returns previously allocated memory to the system;
 - **p** is a pointer to memory that was previously allocated using **malloc()**.
 - **BE CAREFUL:** never call **free()** with an invalid argument

C++ dynamic memory allocation

- C++ provides two dynamic allocation operators: **new** and **delete**;
 - => **#include <new>**
 - **p_var = new type;**
 - `int *p = new int ; // useful... ?`
 - **p_var = new type(initializer);**
 - `int *p = new int(0); //initialize the int pointed to by p with zero`
 - **delete p_var;**
 - `delete p;`
- Allocating **arrays** with **new**
 - **p_var = new array_type[size];**
 - `int *p = new int[10]; // allocate 10 integers array`
 - **delete [] p_var;**
 - `delete [] p;`

```
// Pointer concept
// JAS
```

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

void main()
{
    int a;
    int *aPtr; // OR int * aptr;    OR int* aptr; // 'aPtr' is a pointer to an integer

    a = 10;
    aPtr = &a; // '&a' means the address of 'a'

    cout << "    &a = " << &a << " (hexadecimal)\n";
    cout << "    &a = " << setw(8) << (unsigned long) &a << " (decimal)\n";
    cout << "&aPtr = " << &aPtr << " (hexadecimal)\n";
    cout << "&aPtr = " << setw(8) << (unsigned long) &aPtr << " (decimal)\n";
    cout << " aPtr = " << aPtr << " (hexadecimal)\n";
    cout << " aPtr = " << setw(8) << (unsigned long) aPtr << " (decimal)\n";
    cout << "    a = " << a << endl;
    cout << " *aPtr = " << *aPtr << endl << endl;

    *aPtr = 99; // *aPtr - dereferencing pointer aPtr, using * operator
    cout << "    a = " << a << endl;
}

&a = 0018FF08 (hexadecimal)
&a = 1638152 (decimal)

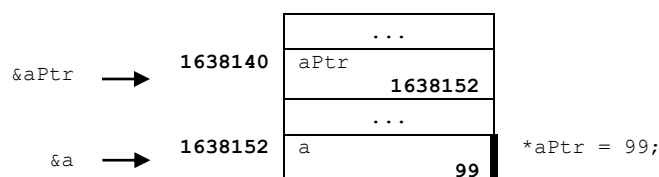
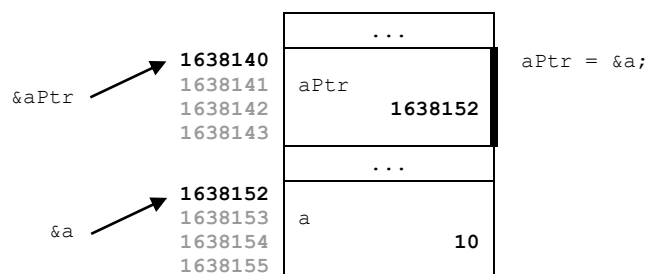
&aPtr = 0018FEFC (hexadecimal)
&aPtr = 1638140 (decimal)

aPtr = 0018FF08 (hexadecimal)
aPtr = 1638152 (decimal)

a = 10
*aPtr = 10

a = 99

Press any key to continue . . .
```



```

// Pointers
// Using pointers - Passing parameters by reference (2 different ways):
// 1) using explicit pointers
// 2) using reference parameters
// JAS - Mar/2011

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

// Using explicit pointers for passing parameters by reference
void swap1(int *x, int *y)
{
    int temp;

    temp = *x; // *x - dereferencing pointer x, using * operator
    *x = *y;
    *y = temp;
}

// Passing reference parameters (as we have seen before)
// A reference is a pointer "in disguise"
// When you use references the compiler automatically passes parameters addresses
// and dereferences the pointer parameters in the function body.
// For that reason, references are more convenient for the programmer
// than explicit pointers
void swap2(int &x, int &y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}

void main(void)
{
    int a, b;

    a=10; b=20;
    cout << "a = " << a << ", b = " << b << endl << endl;

    swap1(&a,&b);
    cout << "after swap1(): a = " << a << ", b = " << b << endl << endl;

    swap2(a,b);
    cout << "after swap2(): a = " << a << ", b = " << b << endl << endl;

}

a = 10, b = 20

after swap1(): a = 20, b = 10

after swap2(): a = 10, b = 20

```

TO DO BY STUDENTS: try to overload swap()

```

// Pointers
// Using pointers - Passing parameters by reference (2 different ways):
// (similar to the last example, but showing more information)
// 1) using explicit pointers
// 2) using reference parameters
// JAS - Mar/2011

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

void swap1(int *x, int *y)
{
    int temp;

    cout << "SWAP1_a\n";
    cout << "&x = " << (unsigned long) &x << ", " <<
    "&y = " << (unsigned long) &y << ", " <<
    "&temp = " << (unsigned long) &temp << " (decimal)\n";
    cout << " x = " << (unsigned long) x << ", " <<
    " y = " << (unsigned long) y << " (decimal)\n";
    cout << " *x = " << *x <<
    " *y = " << *y << ", " <<
    " temp = " << temp << endl;

    temp = *x;
    *x = *y;
    *y = temp;

    cout << "SWAP1_b\n";
    cout << " *x = " << *x <<
    " *y = " << *y << ", " <<
    " temp = " << temp << endl;
}

void swap2(int &x, int &y)
{
    int temp;

    cout << "SWAP2_a\n";
    cout << "&x = " << (unsigned long) &x << ", " <<
    "&y = " << (unsigned long) &y << ", " <<
    "&temp = " << (unsigned long) &temp << " (decimal)\n";
    cout << " x = " << (unsigned long) x << ", " <<
    " y = " << (unsigned long) y << ", " << " (decimal)" <<
    " temp = " << temp << endl;

    temp = x;
    x = y;
    y = temp;

    cout << "SWAP2_b\n";
    cout << " x = " << x << ", " <<
    " y = " << y << ", " <<
    " temp = " << temp << ", " << endl;
}

void main(void)
{
    int a, b;

    a=10; b=20;

    cout << "MAIN\n";
    cout << " &a = " << (unsigned long) &a << ", " <<
    "&b = " << (unsigned long) &b << " (decimal)\n";
    cout << " a = " << a << ", b = " << b << endl << endl;

    swap1(&a,&b);

    cout << "MAIN after swap1(): a = " << a << ", b = " << b << endl << endl;

    swap2(a,b);

    cout << "MAIN after swap2(): a = " << a << ", b = " << b << endl << endl;
}

```

```

MAIN
    &a = 2816340,  &b = 2816336 (decimal)
    a = 10,  b = 20

```

```

SWAP1_a
    &x = 2816320,  &y = 2816324,  &temp = 2816312 (decimal)
    x = 2816340,  y = 2816336 (decimal)
    *x = 10 *y = 20,  temp = -175524

```

```

SWAP1_b
    *x = 20 *y = 10,  temp = 10

```

```

MAIN after swap1(): a = 20,  b = 10

```

```

SWAP2_a
    &x = 2816340,  &y = 2816336,  &temp = 2816320 (decimal)
    x = 20,  y = 10,  (decimal)  temp = 1759800264

```

*temp has same address as x in swap1()
just by chance; memory was reused*

```

SWAP2_b
    x = 10,  y = 20,  temp = 20,
MAIN after swap2(): a = 10,  b = 20

```

```

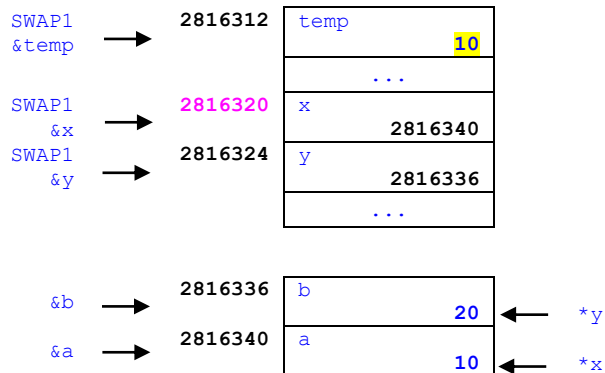
void swap1(int *x, int *y)
{...}

```

```

void main(void)
{
    int a, b;
    a=10; b=20;
    ...
    swap1(&a,&b);
    ...
}

```



```

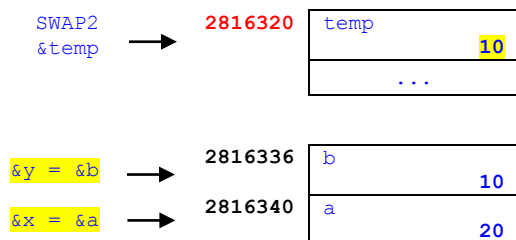
void swap2(int &x, int &y)
{...}

```

```

void main(void)
{
    int a, b;
    ... // after swap1() -> a=20; b=10
    swap2(a,b);
    ...
}

```



```

// Pointers and 1D arrays with static allocation
// Relationship between arrays and pointers
// Pointer arithmetic
// JAS - Mar/2011

#include <iostream>

using namespace std;

#define NMAX 3

void main(void)
{
    int a[NMAX];
    int *aPtr;
    int i;

    for (i=0; i<NMAX; i++)
        a[i] = 10*(i+1);

    for (i=0; i<NMAX; i++)
        cout << "&a[" << i << "] = " << (unsigned long) &a[i]
            << ", a[" << i << "] = " << a[i] << endl;

    aPtr = a; // an array identifier is a pointer to the 1st array element

    cout << "a = " << (unsigned long) a << endl;
    cout << "&a[0] = " << (unsigned long) &a[0] << endl;
    cout << "aPtr = " << (unsigned long) aPtr << endl;

    for (i=0; i<NMAX; i++)
        cout << "(aPtr+" << i << ") = " << (unsigned long) (aPtr+i)
            << ", *(aPtr+" << i << ") = " << *(aPtr+i) << endl;
}

&a[0] = 1899068, a[0] = 10
&a[1] = 1899072, a[1] = 20
&a[2] = 1899076, a[2] = 30
a = 1899068
&a[0] = 1899068
aPtr = 1899068
(aPtr+0) = 1899068, *(aPtr+0) = 10
(aPtr+1) = 1899072, *(aPtr+1) = 20
(aPtr+2) = 1899076, *(aPtr+2) = 30

```

a =	→	1899068	...
&a[0]	→	1899068	a[0] 10
&a[1]	→	1899072	a[1] 20
&a[2]	→	1899076	a[2] 30
			...

```

// Pointers and 1D arrays with static allocation
// Relationship between arrays and pointers
// Passing arrays as function parameters
// JAS - Mar/2011

#include <iostream>

using namespace std;

#define NMAX 3

void showArray1(const int v[], int nElems)
{
    cout << "showArray1()\n";
    for (int i=0; i<nElems; i++)
        cout << "v[" << i << "] = " << v[i] << endl;
}

void showArray2(const int *v, int nElems)
{
    cout << "showArray2()\n";
    for (int i=0; i< nElems; i++)
        cout << "v[" << i << "] = " << v[i] << endl; // v[i] <=> *(v+i)
}

void main(void)
{
    int a[NMAX];

    for (int i=0; i<NMAX; i++)
        a[i] = 10*(i+1);

    showArray1(a,NMAX);

    showArray2(a,NMAX);

    showArray2(&a[0],NMAX);

    // showArray1(&a[0],NMAX); // also possible
}

showArray1()
v[0] = 10
v[1] = 20
v[2] = 30
showArray2()
v[0] = 10
v[1] = 20
v[2] = 30
showArray2()
v[0] = 10
v[1] = 20
v[2] = 30

```

QUESTION:
is it possible to overload showArray(), giving this same name to both functions ? _

```

// Pointers and 1D arrays with dynamic allocation
// Dynamic memory allocation:
// 1) C-style: malloc() & free()
// 2) C++-style: new & delete
// VERY IMPORTANT; NEVER MIX THE 2 KINDS OF DYNAMIC MEMORY ALLOCATION
// JAS - Mar/2011

#include <iostream>
// #include <cstdlib>
#include <new>

using namespace std;

// #define NMAX 3

void main(void)
{
    int *a; // OR int * a; OR int* a;
    int nMax, i;

    cout << "nMax ? "; cin >> nMax;

    cout << "&a = " << (unsigned long) &a << endl;
    cout << "a (before dynamic memory allocation) = " << (unsigned long) a << endl;

    // dynamically allocate memory for array of integers
    // a = (int *) malloc(nMax * sizeof(int)); // C-style
    a = new int[nMax]; // C++-style

    cout << "a (after dynamic memory allocation) = " << (unsigned long) a << endl;

    for (i=0; i<nMax; i++)
        a[i] = 10*(i+1);

    for (i=0; i<nMax; i++)
        cout << "a[" << i << "] = " << a[i]
            << ", &a[" << i << "] = " << (unsigned long) &a[i] << endl;

    // free the dynamically allocate memory
    // free(a); // C-style
    delete [] a; // C++-style

    // a[0] = 100; // should not be done ... why ?
}

```

```

nMax ? 3
&a = 1703544
a (before dynamic memory allocation) = 9449524
a (after dynamic memory allocation) = 3047344
a[0] = 10, &a[0] = 3047344
a[1] = 20, &a[1] = 3047348
a[2] = 30, &a[2] = 3047352

```

&a	→	1703544	a	3047344
			...	
a =		3047344	a[0]	10
&a[0]	→	3047348	a[1]	20
&a[1]	→	3047352	a[2]	30
&a[2]	→		...	

ANOTHER RUN ...

```
nMax ? 5
&a = 2750352
a (before dynamic memory allocation) = 8532020
a (after dynamic memory allocation) = 10121352
a[0] = 10, &a[0] = 10121352
a[1] = 20, &a[1] = 10121356
a[2] = 30, &a[2] = 10121360
a[3] = 40, &a[3] = 10121364
a[4] = 50, &a[4] = 10121368
```

ANOTHER RUN ...

```
nMax ? 7
&a = 2946936
a (before dynamic memory allocation) = 0
a (after dynamic memory allocation) = 688048
a[0] = 10, &a[0] = 688048
a[1] = 20, &a[1] = 688052
a[2] = 30, &a[2] = 688056
a[3] = 40, &a[3] = 688060
a[4] = 50, &a[4] = 688064
a[5] = 60, &a[5] = 688068
a[6] = 70, &a[6] = 688072
```

```
// 2D arrays with static allocation
// JAS - Mar/2011
```

```
#include <iostream>
```

```
using namespace std;
```

```
#define NLIN 2
```

```
#define NCOL 3
```

```
void main(void)
```

```
{
```

```
    int a[NLIN][NCOL];
```

```
    for (int i=0; i<NLIN; i++)
```

```
        for (int j=0; j<NCOL; j++)
```

```
            a[i][j] = 10*(i+1)+j;
```

```
    for (int i=0; i<NLIN; i++)
```

```
        for (int j=0; j<NCOL; j++)
```

```
            cout << "a[" << i << "][" << j << "] = " << a[i][j]
```

```
                << ", " << "&a[" << i << "][" << j << "] = " << (unsigned long) &a[i][j]
```

```
                << endl;
```

```
}
```

	0	1	2
0	10	11	12
1	20	21	22

```
a[0][0] = 10,  &a[0][0] = 1637144
```

```
a[0][1] = 11,  &a[0][1] = 1637148
```

```
a[0][2] = 12,  &a[0][2] = 1637152
```

```
a[1][0] = 20,  &a[1][0] = 1637156
```

```
a[1][1] = 21,  &a[1][1] = 1637160
```

```
a[1][2] = 22,  &a[1][2] = 1637164
```

```
a =
```

```
&a[0][0] → 1637144
```

```
&a[0][1] → 1637148
```

```
&a[0][2] → 1637152
```

```
&a[1][0] → 1637156
```

```
&a[1][1] → 1637160
```

```
&a[1][2] → 1637164
```

...
a[0][0] 10
a[0][1] 11
a[0][2] 12
a[1][0] 20
a[1][1] 21
a[1][2] 22
...

```

// 2D arrays with static allocation
// 2D arrays as function parameters
// JAS - Mar/2011

#include <iostream>

using namespace std;

#define NLIN 2
#define NCOL 3

void showArray(int a[][NCOL], int numLines, int numCols)
// WHY DOES THE COMPILER NEED TO KNOW THE NUMBER OF COLUMNS, "NCOL" ?
{
    for (int i=0; i< numLines; i++)
    {
        for (int j=0; j< numCols; j++)
            cout << a[i][j] << " ";
        cout << endl;
    }
}

void main(void)
{
    int a[NLIN][NCOL];

    for (int i=0; i<NLIN; i++)
        for (int j=0; j<NCOL; j++)
            a[i][j] = 10*(i+1)+j;

    showArray(a, NLIN, NCOL);
}

```

```

10 11 12
20 21 22

```

CHALLENGE

Implement a similar program using 2D dynamically allocated array

```

// Pointers and 2D arrays with ("bidimensional") dynamic allocation
// "C-like": using malloc / free

#include <iostream>
#include <cstdlib>
// #include <new>

using namespace std;

void main(void)
{
    int **a; // <-- NOTE THIS
    int i, j, nLin, nCol;

    cout << "nLin ? "; cin >> nLin;
    cout << "nCol ? "; cin >> nCol;

    // allocate memory for 2D array
    a = (int**)malloc(nLin * sizeof(int *)); // allocate memory for each line pointer
    for (i = 0; i < nLin; i++)
        a[i] = (int*)malloc(nCol * sizeof(int)); // allocate memory for each line contents

    // use the array
    for (i = 0; i < nLin; i++)
        for (j = 0; j < nCol; j++)
            a[i][j] = 10 * (i + 1) + j;

    cout << "&a = " << (unsigned long)&a << endl;
    cout << " a = " << (unsigned long)a << endl;
    for (i = 0; i < nLin; i++)
        cout << "&a[" << i << "] = " << (unsigned long)&a[i] << endl;
    for (i = 0; i < nLin; i++)
        cout << " a[" << i << "] = " << (unsigned long)a[i] << endl;

    for (i = 0; i < nLin; i++)
        for (j = 0; j < nCol; j++)
            cout << "a[" << i << "][" << j << "] = " << a[i][j] << ", &a[" << i << "][" << j << "] = " <<
(unsigned long)&a[i][j] << endl;

    // free all allocated memory (in reverse order of allocation)
    for (i = 0; i < nLin; i++)
        free(a[i]);
    free(a);
}

nLin ? 2
nCol ? 3
&a = 1440120
a = 1514440
&a[0] = 1514440
&a[1] = 1514444
a[0] = 1514496
a[1] = 1514552
a[0][0] = 10, &a[0][0] = 1514496
a[0][1] = 11, &a[0][1] = 1514500
a[0][2] = 12, &a[0][2] = 1514504
a[1][0] = 20, &a[1][0] = 1514552
a[1][1] = 21, &a[1][1] = 1514556
a[1][2] = 22, &a[1][2] = 1514560
Press any key to continue . . .

```

```

// Pointers and 2D arrays with ("bidimensional") dynamic allocation
// "C++-like": using new / delete

#include <iostream>
// #include <cstdlib>
#include <new>

using namespace std;

void main(void)
{
    int **a; // <-- NOTE THIS
    int i, j, nLin, nCol;

    printf("nLin ? "); cin >> nLin;
    printf("nCol ? "); cin >> nCol;

    // allocate memory for 2D array
    a = new int*[nLin]; // allocate memory for each line pointer
    for (i=0; i<nLin; i++)
        a[i] = new int[nCol]; // allocate memory for each line contents

    // use the array
    for (i=0; i<nLin; i++)
        for (j=0; j<nCol; j++)
            a[i][j] = 10*(i+1)+j;

    cout << "&a = " << &a << endl;
    cout << "a = " << a << endl;
    for (i=0; i<nLin; i++)
        cout << "&a[" << i << "] = " << &a[i] << endl;
    for (i=0; i<nLin; i++)
        cout << "a[" << i << "] = " << a[i] << endl;

    for (i=0; i<nLin; i++)
        for (j=0; j<nCol; j++)
            cout << "a[" << i << "][" << j << "] = " << a[i][j] << ", " <<
                "&a[" << i << "][" << j << "] = " << &a[i][j] << endl;

    // free all allocated memory (in reverse order of allocation)
    for (i=0; i<nLin; i++)
        delete[] a[i];
    delete[] a;
}

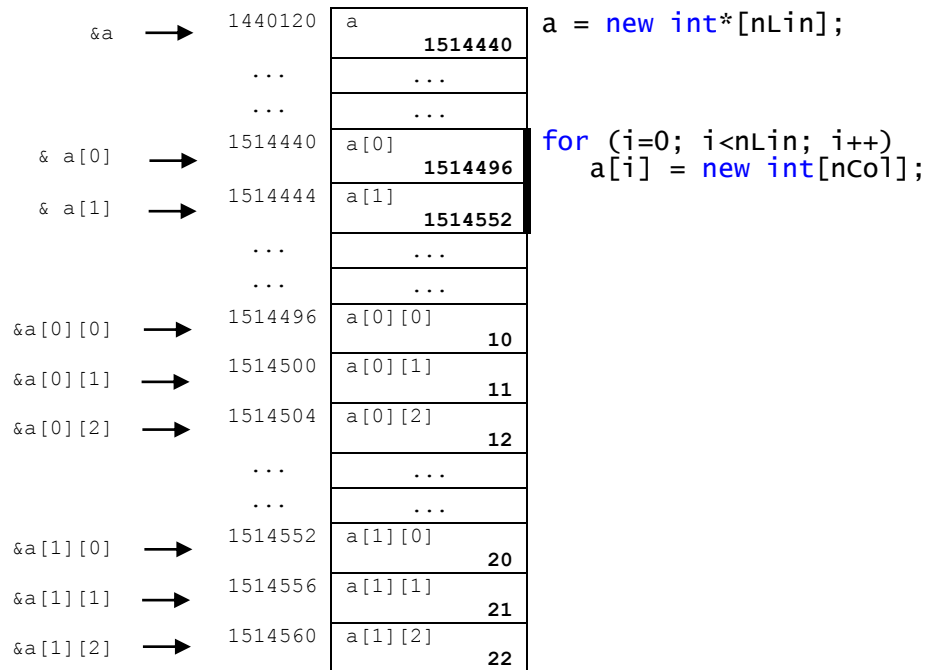
nLin ? 2
nCol ? 3
&a = 1440120
a = 1514440
&a[0] = 1514440
&a[1] = 1514444
a[0] = 1514496
a[1] = 1514552
a[0][0] = 10, &a[0][0] = 1514496
a[0][1] = 11, &a[0][1] = 1514500
a[0][2] = 12, &a[0][2] = 1514504
a[1][0] = 20, &a[1][0] = 1514552
a[1][1] = 21, &a[1][1] = 1514556
a[1][2] = 22, &a[1][2] = 1514560

```

```

nLin ? 2
nCol ? 3
&a = 1440120
  a = 1514440
& a[0] = 1514440
& a[1] = 1514444
  a[0] = 1514496
  a[1] = 1514552
a[0][0] = 10, &a[0][0] = 1514496
a[0][1] = 11, &a[0][1] = 1514500
a[0][2] = 12, &a[0][2] = 1514504
a[1][0] = 20, &a[1][0] = 1514552
a[1][1] = 21, &a[1][1] = 1514556
a[1][2] = 22, &a[1][2] = 1514560

```



```

/*
POINTERS TO STRUCT'S
How to access the members of a struct using a pointer to the struct ?
*/

#include <iostream>
#include <iomanip>
#include <cctype>
#include <string>
#include <sstream>

using namespace std;

struct Fraction
{
    int numerator;
    int denominator;
};

bool readFraction(Fraction &f) // readFraction() is overloaded (see below)
{
    string fractionString;
    char fracSymbol;
    int numerator;
    int denominator;
    bool success;

    cout << "n / d ? ";
    getline(cin, fractionString);

    istringstream fractionStrStream(fractionString);
    if (fractionStrStream >> numerator >> fracSymbol >> denominator)
        if (fracSymbol == '/')
        {
            f.numerator = numerator;
            f.denominator = denominator;
            success = true;
        }
        else
            success = false;
    else
        success = false;
    return success;
}

bool readFraction(Fraction *f) // readFraction() is overloaded (see above)
{
    string fractionString;
    char fracSymbol;
    int numerator;
    int denominator;
    bool success;

    cout << "n / d ? ";
    getline(cin, fractionString);

```

```

istringstream fractionStrStream(fractionString);
if (fractionStrStream >> numerator >> fracSymbol >> denominator)
{
    if (fracSymbol == '/')
    {
        f->numerator = numerator;
        //(*f).numerator = numerator;
        f->denominator = denominator;
        //(*f).denominator = denominator;
        success = true;
    }
    else
        success = false;
else
    success = false;
return success;
}

```

```

Fraction multiplyFractions(Fraction f1, Fraction f2)
{
    Fraction f;

    f.numerator = f1.numerator * f2.numerator;
    f.denominator = f1.denominator * f2.denominator;
    return f;
}

```

```

void showFraction(Fraction f)
{
    cout << f.numerator << "/" << f.denominator;
}

```

```

int main()
{
    Fraction f1, f2, f3;

    cout << "Input 2 fractions:\n";
    //if (readFraction(f1) && readFraction(f2))
    if (readFraction(&f1) && readFraction(&f2))
    {
        f3 = multiplyFractions(f1,f2);

        cout << "Product: ";
        showFraction(f3);
    }
    else
    {
        cout << "Invalid fraction\n";
    }
    cout << endl;

    return 0;
}

```

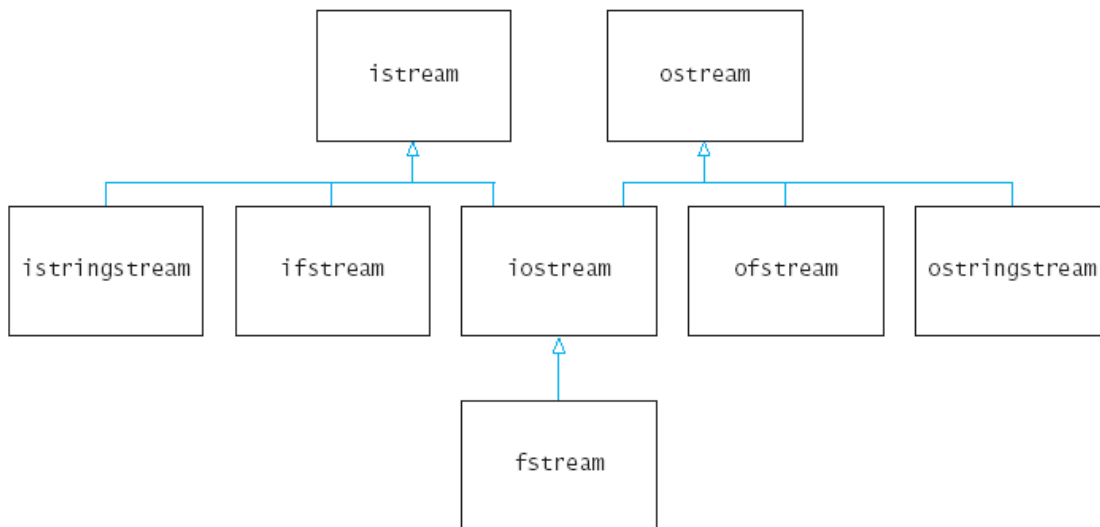

=====

STREAMS / FILES

=====

I/O Streams

- **I/O** refers to program **I**ntput and **O**utput
- I/O is done via stream objects
- A **stream** is a flow of data
- **Input stream**: data flows into the program
 - Input can be from
 - the keyboard
 - a file
- **Output stream**: data flows out of the program
 - Output can be to
 - the screen
 - a file
- **Input and Output stream**: data flows either into or out of the program
 - only possible with files
- **The C++ input/output library** consists of several classes that are related by **inheritance** (*inheritance will be treated later in this course*)
- The inheritance hierarchy of stream classes:



- The standard **cin** and **cout** objects belong to specialized system-dependent classes with nonstandard names.
- You can assume that
 - **cin** belongs to a class that is derived from **istream** and
 - **cout** belongs to a class derived from **ostream**.

cin & cout streams

- **cin**
 - input stream connected to the keyboard
- **cout**
 - output stream connected to the screen
- **cin** and **cout** are declared in the **iostream** header file
 - => **#include <iostream>**
- You can declare your own streams to use with files.

Why use files?

- Files allow you
 - to use input data **over and over**
 - to deal with **large data sets**
 - to store data **permanently**
 - to access output data **after the program ends**

Text files vs. Binary files

- Usually files are classified in two categories:
 - **Text files**
 - and **binary files**.
- While both binary and text files contain data stored as a series of bits,
 - the bits in text files represent characters,
 - while the bits in binary files represent other types of data (int, float, struct, ...)
- **Simple text files** are usually created by using a text editor like **notepad**, **pico**, etc.
(not Word or OpenOffice)
- We work with binary files all the time.
 - **executable files**, **image files**, **sound files**, ... are **binary files**.
- In effect, text files are basically binary files, because they store binary numbers.
- **cin & cout** "behave like" text files.

Accessing file data

- **Open** the file
 - this operation associates the name of a file in disk to a stream object.
 - **NOTE: cin and cout** are open automatically on program start.
- Use **read/write** calls or extraction/insertion operators, to get/put data from/into the file.
- **Close** the file.

Declaring Stream Variables

- Like other variables, a stream variable must be ...
 - declared before it can be used
 - initialized before it contains valid data
 - Initializing a stream means connecting it to a file
- **Input-file streams** are of type **ifstream**
- **Output-file streams** are of type **ofstream**
- These types are defined in the **fstream** library
 - => **#include <fstream>**

- Example:

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

ifstream in_stream;
ofstream out_stream;
```

Connecting a stream to a file / Opening a file

- The opening operation connects a stream to an external file name
 - An external file name is the name for a file that the operating system uses
 - Examples:
 - `infile.txt` and `outfile.txt` used in the following examples
- Once a file is open, it is referred to using the name of the stream connected to it.
- A file can be opened using
 - the `open()` member function associated with streams
 - the **constructor** of the stream classes
- Examples:
 - `ifstream in_stream;`
 - `ofstream out_stream;`
 - `in_stream.open("infile.txt");`
 - connects `in_stream` to `"infile.txt"`
 - `out_stream.open("C:\\Mieic\\Prog\\programs\\outfile.txt");`
 - connects `out_stream` to `"oufile.txt"` that is in directory `"C:\\Mieic\\Prog\\programs"`
 - note the double backslash in the string argument
 - necessary in Windows systems where the directories of the path are separated by `'\\'`
 - Alternatively:
 - `ifstream in_stream("infile.txt");`
 - calls the constructor of `ifstream` class that automatically tries to open the file
- The filename does not need to be a constant, as in the previous examples. Program users can enter the name of a file to use for input or for output.
 - in this case it must be stored in a string variable
 - In `C++11`, you can use a `std::string` as argument to `open()` or to the constructor
 - `std::string filename;`
`cout << "Filename ?"; cin >> filename;`
`myFile.open(filename);`
 - In the previous C++ standard, `open()` only accepts a C-string for the first parameter. The correct way of calling it would then be:
 - `myFile.open(filename.c_str());`
- Note:
 - The name of a text file does not necessarily have the extension `' .txt '`

open() method (C++11)

- `void ifstream::open(const string &filename, ios::openmode mode = ios::in);`
- `void ofstream::open(const string &filename, ios::openmode mode = ios::out);`
- `void fstream::open(const string &filename, ios::openmode mode = ios::in | ios::out);`
 - `filename` is the name of the file (must be a C-string, in pre-C++11 compilers)
 - `mode` determines how the file is opened; can be the OR (|) of several constants
 - `ios::in` – the file is capable of input
 - `ios::out` – the file is capable of output
 - `ios::binary` – causes file to be opened in binary mode;
by default, all files are opened in text mode
 - `ios::ate` – cause initial seek to end-of-file;
I/O operations can still occur anywhere within the file
 - `ios::app` – causes all output to the file to be appended to the end
 - `ios::trunc` – the file is truncated to zero length

Using input/output stream for reading/writing from/to text files

- It is very easy to read from or write to a text file.
- Simply use the `<<` and `>>` operators the same way you do when performing console I/O, except that, instead of using `cin` and `cout`, use a stream that is linked to a file.
- Example 1:

```
ifstream in_stream;
in_stream.open("infile.txt");
int one_number, another_number;
in_stream >> one_number >> another_number;
```
- Example 2:

```
ofstream out_stream;
out_stream.open("outfile.txt");
out_stream << "Resulting data:";
out_stream << one_number << endl << another_number << endl;
```

Closing a file

- After using a file, it should be closed.
This disconnects the stream from the file
 - Example: `in_stream.close();`
- The system will automatically close files if you forget as long as your program ends normally
- Files should be closed:
 - to reduce the chance of a file being corrupted if the program terminates abnormally.
 - if your program later needs to read input from the output file.

Errors on opening files

- Opening a file could fail for several reasons.
Common reasons for open to fail include
 - the file does not exist (or the path is incorrect)
 - the external name is incorrect
 - the file is already open
- Member function `is_open()`, can be used to test whether the file is already open
- May be no error message if the call to open fails.
Program execution continues!
- Member function `fail()`, can be used to test the success of a stream operation (not only the `open()` operation)
 - Example:

```
in_stream.open("numbers.txt");
if( in_stream.fail() )
{
    cerr << "Input file opening failed.\n";
    exit(1) ; // sometimes, it is best to stop the program,
              // with an exit code != 0
}
```

Reading from text files – additional notes

- Stream input is performed with the stream extraction operator `>>`, which
 - skips white space characters (' ', '\t', '\n')
 - returns `false`, after end-of-file (EOF) is encountered
 - Example:

```
double next, sum = 0;
while(in_stream >> next)
{
    sum = sum + next;
}
```
- Stream input causes some stream state flags to be set when an error occurs:
 - `failbit` - improper input (internal logic error of the operation)
 - `badbit` - the operation failed (failure of I/O on the stream buffer)
 - `eofbit` - EOF was reached on the input stream
 - EOF can be tested using the `eof()` member function
 - `while(! in_stream.eof()) ...` (see later)
 - `goodbit` to be set when no error has occurred
- Member function `ignore()` can be used to skip characters, as with `cin` stream.
- NOTE:
 - be careful when mixing operator `>>` and `getline()`
OR operator `>>` and `cin.get()`
 - remember what has been said about this, in the *string* section

How To Test End of File

- In some cases, you will want to know when the end of the file has been reached.
 - For example, if you are reading a list of values from a file, then you might want to continue reading until there are no more values to obtain.
 - This implies that you have some way to know when the end of the file has been reached.
 - C++ I/O system supplies such a function to do this: `eof()`.
 - To detect EOF involves these steps:
 - 1. Open the file being read for input.
 - 2. Begin reading data from the file.
 - 3. After each input operation, determine if the end of the file has been reached by calling `eof()`.
- NOTE:
 - `eof()` returns false only when the program tries to read past the end of the file
- Example:
 - This loop reads each character, and writes it to the screen

```
in_stream.get(next);  
while ( ! in_stream.eof( ) ) // NOTE: first, input must be tried  
{                             // then you can test for EOF  
    cout << next;  
    in_stream.get(next);  
}
```

Formatting output to text files

- As for `cout`, formatting can be done using:
 - `manipulators` (defined in `iomanip` library => `#include <iomanip>`)
 - `setw()`
 - `fixed`
 - `setprecision()`
 - ... and some other
 - using `setf()` member function of output streams
 - `out_stream.setf(ios::fixed);`
 - `out_stream.setf(ios::showpoint);`
 - `out_stream.precision(2);`
 - ... and some other
- Note:
 - A manipulator is a function called in a nontraditional way used after the insertion operator (`<<`) as if the manipulator function call is an output item
 - Manipulators in turn call member functions
 - `setw` does the same task as the member function `width`
 - `setprecision` does the same task as the member function `precision`
 - ...
 - Any flag that is set, may be unset, using the `unsetf` function
 - Example:
`cout.unsetf(ios::showpos);`
causes the program to stop printing plus signs on positive numbers

Stream names as arguments

- Streams can be arguments to a function
- The function's formal parameter for the stream must be call-by-reference
 - Example:

```
void make_neat(istream &messy_file, ostream &neat_file);  
// make_neat() code will be presented in the following pages
```
- Take advantage of the inheritance relationships between the stream classes whenever you write functions with stream parameters.
 - **istream** as well as **cin** are objects of type **istream**
 - **ostream** as well as **cout** are objects of type **ostream**
 - Example:
 - `double get_max(istream &in);`
 - You can now pass parameters of types derived from **istream**, such as an **istream** object or **cin**.
 - `max = get_max(in_stream);`
 - `max = get_max(cin);`
 - both **cin** and **in_stream** can be used as arguments of a call to `get_max()`, whose parameter is of type **istream &**

Binary I/O

- While reading and writing text files is very easy it is not always the most efficient way to handle files.
- There will be times when you need to store information in binary format: **int**'s, **double**'s, **struct**'s, ... or **char**'s
- When performing I/O of binary data be sure to open the file using the **ios::binary** mode specifier
- I/O can be performed using the
 - **get()** and **put()** member functions
 - - `istream & get(char &ch);`
 - `ostream & put(char ch);`
 - NOTE:
 - In a **text stream**, some **character translations** may take place. For example, when the newline character is **output**, using `<<`, it may be converted into a carriage-return / linefeed sequence.
 - The reverse happens when a carriage-return / linefeed sequence is **input** from a file: it is converted into a newline char.
 - No such translations occur on binary files:
 - using `get()` you can "see" the carriage-return/linefeed chars in a text file

- `read()` and `write()` member functions
 - can be used to read/write blocks of binary data
 - `istream & read (char *buf, streamsize num);`
 - reads **num** characters from the invoking stream and puts them into the buffer pointed to by **buf**
 - `ostream & write (const char *buf, streamsize num);`
 - writes **num** characters to the invoking stream from the buffer pointed to by **buf**
- Example: (see next pages)

Random access

- The C++ I/O system manages 2 pointers associated with a file:
 - the **get pointer**, which specifies where in the file the next input operation will occur
 - the **put pointer**, which specifies where in the file the next output operation will occur
- You can perform random access (in a nonsequential fashion) by using the `seekg()` and `seekp()` functions.
- Generally, random access I/O should only be performed on those files opened in binary mode. **WHY?**
- Their most common forms are:
 - `istream& seekg (streamoff offset, ios_base::seekdir origin);`
 - `ostream& seekp (streamoff offset, ios_base::seekdir origin);`
 - **origin** can take one of the values: `ios::beg`, `ios::end`, `ios::cur`
 - **offset** is an integer that specifies the displacement of the get/put pointer relative to the specified **origin**
- `seekg()` and `seekp()` are interchangeable for file streams. However, this is not true for other types of streams (ex: stringstream, see next pages), as they may hold separate pointers for the put and get positions.
- `tellg()` and `tellp()` can be used to obtain the current position of the pointers.
- Note: you can't call `seekp/tellp` on an instance of `ifstream` and you can't call `seekg/tellg` on an instance of `ofstream`. However, you can use both on an instance of `fstream`.

INPUT/OUTPUT – TEXT FILES

```
/**
  INPUT FROM TEXT FILE
  Reads numbers from a file and finds the maximum value
  @param in the input stream to read from
  @return the maximum value or 0 if the file has no numbers

  (from BIG C++ book)
*/
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

double max_value(ifstream &in) //stream parameters must always be passed by reference
{
    double highest;
    double next;
    if (in >> next) // if file contains at least 1 element
        highest = next;
    else
        return 0;    // If file is empty. Not the best solution ...!!!

    while (in >> next)
    {
        if (next > highest)
            highest = next;
    }

    return highest;
}

int main()
{
    string filename;
    cout << "Please enter the data file name: "; // numbers.txt
    //located in C:\Users\jsilva\.....\Project_folder\numbers.txt
    cin >> filename;

    ifstream infile;
    infile.open(filename);

    if (infile.fail()) // OR if (! infile.is_open()) OR if (! infile)
    {
        cerr << "Error opening " << filename << "\n";
        return 1;    // exit(1);
    }

    double max = max_value(infile);
    cout << "The maximum value is " << max << "\n";

    infile.close();
    return 0;
}
```

```

/**
INPUT FROM TEXT FILE OR KEYBOARD
Reads numbers from a file and finds the maximum value
@param in the input stream to read from
@return the maximum value or 0 if the file has no numbers

(adapted from BIG C++ book)
*/

#include <iostream>
#include <string>
#include <fstream>

using namespace std;

double max_value(istream &in) // can be called with 'infile' or 'cin'
{
    double highest;
    double next;
    if (in >> next)
        highest = next;
    else
        return 0;

    while (in >> next)
    {
        if (next > highest)
            highest = next;
    }

    return highest;
}

int main()
{
    double max;

    string input;
    cout << "Do you want to read from a file? (y/n) ";
    cin >> input;

    if (input == "y")
    {
        string filename;
        cout << "Please enter the data file name: ";
        cin >> filename;

        ifstream infile;
        infile.open(filename);

        if (infile.fail())
        {
            cerr << "Error opening " << filename << "\n";
            return 1;
        }

        max = max_value(infile);
        infile.close();
    }
}

```

```

else
{
    cout << "Insert the numbers. End with CTRL-Z." << endl;
    max = max_value(cin);
}

cout << "The maximum value is " << max << "\n";

return 0;
}

```

TO DO BY STUDENTS:

- what is the output when the file is empty or the user types CTRL-Z as first input?
- Modify the program to solve the 'problem'.

// INPUT/OUTPUT - TEXT FILES

// Reads all the numbers in the file **rawdata.dat** and writes the numbers
// to the screen and to the file **neat.dat** in a neatly formatted way.
// Illustrates output formatting instructions.
// Adapted from Savitch book

// DON'T FORGET TO PUT FILE rawdata.txt IN THE PROJECT DIRECTORY
// OR IN THE CURRENT DIRECTORY (IF YOU RUN THE PROGRAM FROM THE COMMAND PROMPT)

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <iomanip>
```

```
using namespace std;
```

```
/*
The numbers are written one per line, in fixed-point notation
with 'decimal_places' digits after the decimal point;
each number is preceded by a plus or minus sign and
each number is in a field of width 'field_width'.
(This function does not close the file.)
*/
```

```
void make_neat(ifstream &messy_file, ofstream &neat_file,
               int field_width, int decimal_places);
```

```
int main( )
{
    const int FIELD_WIDTH = 12;
    const int DECIMAL_PLACES = 5;

    ifstream fin;
    ofstream fout;

    fin.open("rawdata.txt");
    if (fin.fail( )) //Could have tested if(fin.is_open())
    {
        cerr << "Input file opening failed.\n";
        exit(1);
    }

    fout.open("neatdata.txt");
    if (fout.fail( ))
    {
        cerr << "Output file opening failed.\n";
        exit(2);
    }

    make_neat(fin, fout, FIELD_WIDTH, DECIMAL_PLACES);

    fin.close();
    fout.close();

    cout << "End of program.\n";
    return 0;
}
```

```

//Uses iostream, fstream, and iomanip:
void make_neat(ifstream &messy_file, ofstream &neat_file,
               int field_width, int decimal_places)
{
    double next;

    neat_file.setf(ios::fixed);           // not in e-notation
    neat_file.setf(ios::showpoint);       // show decimal point ...
                                           // ... even when fractional part is 0
    neat_file.setf(ios::showpos);         // show + sign
    neat_file.precision(decimal_places);

/*
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.setf(ios::showpos);
    cout.precision(decimal_places);
*/
    while (messy_file >> next)
    {
        //cout << setw(field_width) << next << endl;
        neat_file << setw(field_width) << next << endl;
    }
}

/*
rawdata.txt

10.37      -9.89897
2.313     -8.950  15.0

   7.33333  92.8765
-1.237568432e2

neatdata.txt

+10.37000
-9.89897
+2.31300
-8.95000
+15.00000
+7.33333
+92.87650
-123.75684
*/

```

```

//FILES
//Detecting the end of a file with eof() method
//Copies file code.txt to file code_numbered.txt,
//adding the line number to the beginning of each line.
//Illustrates the use of get() member function of istream/ifstream
//Assumes code.txt is not empty.

#include <fstream>
#include <iostream>
#include <cstdlib>

using namespace std;

int main( )
{
    ifstream fin;
    ofstream fout;

    fin.open("code.txt");
    if (fin.fail( ))
    {
        cerr << "Input file opening failed.\n";
        exit(1);
    }

    fout.open("code_numbered.txt");
    if (fout.fail( ))
    {
        cerr << "Output file opening failed.\n";
        exit(1);
    }

    char next;
    int n = 1;
    fin.get(next); //THE ARGUMENT OF get() IS PASSED BY VALUE OR BY REFERENCE?
    fout << n << " ";
    while (! fin.eof( )) //returns true if the program has read past the end of the input file;
                        //otherwise, it returns false
    {
        fout << next;
        if (next == '\n')
        {
            n++;
            fout << n << ' ';
        }
        fin.get(next); //NOTE: get() READS SPACE AND NEWLINE CHARACTERS
    }

    fin.close( );
    fout.close( );

    return 0;
}

```

TO DO BY STUDENTS:

try with an empty file; see what happens; solve the "problem"

TIP: investigate the use of get()

//Appending data to the end of a text file

```
#include <iostream>
#include <fstream>

using namespace std;

int main( )
{
    ofstream fout;
    fout.open("numbers.txt", ios::app); //TO DO: try with a non-existing file
    fout << "Appended data:\n";
    for (int i=10; i<=19; i++)
        fout << i << endl;
    fout.close( );
    return 0;
}
```

INPUT/OUTPUT – BINARY FILES

```
// A binary file for storing integer values
// JAS - 2015/04/09

#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    fstream f; // read and write stream

    f.open("numbers.dat", ios::out | ios::binary); // create the file

    //streampos place = 5 * sizeof(int); // start writing at position of 5th integer
    //f.seekp(place); // random access

    for (int x = 65; x <= 65 + 25; x++) // write 26 integers, starting with 65
        f.write((char *)&x, sizeof(int));

    f.close();

    //-----

    // USUALLY THIS PART WOULD BE DONE BY ANOTHER PROGRAM ...

    f.open("numbers.dat", ios::in | ios::binary); // open the file for reading

    //streampos place = 7 * sizeof(int); // start reading at position of 7th integer
    //f.seekg(place); // random access

    for (int i = 1; i <= 26; i++)
    {
        int y;
        f.read((char *)&y, sizeof(int));
        cout << "y= " << y << endl;

        // FOR RETRIEVING THE INTEGERS AS CHARS !!!
        // char c;
        // f.read(&c, sizeof(char));
        // cout << "c= " << c << endl;
    }

    f.close();

    return 0;
}
```



```

// Random access to a binary file
// a very crude example
// JAS

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib> // for exit()
using namespace std;

const int MAX_NAMELEN = 10;
const char file[] = "name_age.dat";

//-----
typedef struct
{
    char name[MAX_NAMELEN]; //why not "string name" instead of char name[MAX_NAMELEN] ?
    unsigned int age;
} Person;

//-----
bool fileExists(const char *filename)
{
    bool exists = false;
    ifstream ifile;
    ifile.open(filename);
    if (ifile.is_open())
    {
        exists = true;
        ifile.close();
    }
    return exists;
}

//-----
void showPerson(const Person &p)
{
    cout << setw(MAX_NAMELEN) << p.name << " " << p.age << endl;
}

//-----
bool writeFileRecord(fstream &f, const Person *rec, unsigned int recNum)
{
    streampos place = recNum * sizeof(Person); // convert to streampos type
    //cout << "WRITE: place = " << place << endl;

    f.seekp(place); // random access
    if (f.fail()) return false;

    f.write((char *) rec, sizeof(Person)) << flush;
    // flush the output to guarantee that the file is updated before proceeding
    // NOTE: this syntax is possible because write() returns an "istream &"
    if (f.fail()) return false;
    return true;
}

```

```

//-----
bool readFileRecord(fstream &f, Person *rec, unsigned int recNum)
{
    streampos place = recNum * sizeof(Person); // convert to streampos type
    //cout << "READ: place = " << place << endl;

    if (f.eof())
        f.clear(); // clear flags if last read attempt returned end of file

    f.seekg(place); // random access
    if (f.fail()) return false;

    f.read((char *) rec, sizeof(Person));
    if (f.fail()) return false;

    return true;
}

//-----
void showFileContents(fstream &f)
{
    Person p;
    int n = 0;

    if (f.eof())
        f.clear();

    f.seekg(0); // go to beginning
    cout << "CONTENTS OF THE FILE: \n";

    // could have used readFileRecord() above; TO DO by students
    while (f.read((char *) &p, sizeof(Person))) //compare w/other blue code
    {
        n++;
        if (f.fail())
        {
            cerr << "Error in reading " << file << endl;
            exit(EXIT_FAILURE);
        }
        if (p.age != -1) // SEE HOW p2 WAS INITIALIZED
            cout << n << ": " << setw(MAX_NAMELEN) << p.name << " " << p.age <<
endl;
    }
}

```

// CONTINUES ON NEXT PAGE

```

int main()
{
    Person p1={"Ana", 20}, p2="", -1}, p3={"Rui", 21} ;
    // TO DO by students:
    // 1) use an array of struct's
    // 2) alternatively, read data from keyboard

    Person p;
    unsigned int numRec;

    fstream finout; // read and write streams

    if (!fileExists(file)) // if file does not exist ...
    {
        cout << "File does not exist. An empty file will be created.\n";
        finout.open(file, ios::out | ios::binary); //... create the file
        finout.close();
    }
    // open file in input/output + binary modes
    finout.open(file, ios::in | ios::out | ios::binary);
    if (finout.is_open())
    {
        if (!writeFileRecord(finout,&p1,0)) {cerr << "write error\n"; exit(1);}
        if (!writeFileRecord(finout,&p2,1)) {cerr << "write error\n"; exit(1);}
        if (!writeFileRecord(finout,&p3,2)) {cerr << "write error\n"; exit(1);}
        //if (!writeFileRecord(finout,&p3,10)) {cerr << "Write error\n";
    exit(1);}

    // TO DO: use an array of Persons instead of p1, p2, and p3

    showFileContents(finout);

    cout << "numRec ? "; cin >> numRec; //try with other numRec's
    if (readFileRecord(finout,&p,numRec))
    {
        cout << "numRec = " << numRec << ": ";
        showPerson(p);
    }
    else
    {
        cerr << "Read error\n";
        exit(1);
    }

}
else
{
    cerr << file << " could not be opened\n";
    exit(EXIT_FAILURE);
}

    finout.close();

    return 0;
}

```

QUESTION:

what will happen if you try to see the contents of file name_age.dat using a text editor ?

STRINGSTREAMS

String Streams

- We saw how a stream can be connected to a file.
- A stream can also be connected to a string.
- With stringstreams you can perform input/output from/to a string.
- This allows you to convert numbers (or any type with the << and >> stream operators overloaded) to and from strings.
- To use stringstream =>
 - **#include <sstream>**
- The **istringstream** class reads characters from a string
- The **ostringstream** class writes characters to a string.

Stringstream uses

- A very common use of string streams is:
 - to accept input one line at a time and then to analyze it further.
 - by using stringstreams you can avoid mixing `cin >> ...` and `getline()`
 - *see examples in the following pages*
 - to use standard output manipulators to create a formatted string

istringstream

- Using an **istringstream**, you can read numbers that are stored in a string by using the >> operator:

```
string input = "March 25, 2014";
istringstream instr(input); //initializes 'instr' with 'input'
string month, comma;
int day, year;
instr >> month >> day >> comma >> year;
```

- Note that this input statement yields **day** and **year** as integers. Had we taken the string apart with **substr**, we would have obtained only strings.
- Converting strings that contain digits to their integer values is such a common operation that it is useful to write a helper function for that purpose:

```
int string_to_int(string s)
{
    istringstream instr;
    instr.str(s); // ALTERNATIVE way to initialize 'instr' with 's'
                // to the initialization mode used above
    int n;
    instr >> n;
    return n;
}
```

ostringstream

- By writing to a string stream, you can convert numbers to strings.
- By using the << operator, the number is converted into a sequence of characters.

```
ostringstream ostr;  
ostr << setprecision(5) << sqrt(2);
```

- To obtain a string from the stream, call the `str` member function.
 - `string output = ostr.str();`
- Example: (builds the string "January 23, 1955")

```
string month = "January";  
int day = 23;  
int year = 1955;  
ostringstream ostr;  
ostr << month << " " << day << ", " << year;  
string output = ostr.str();
```

- Converting an integer into a string is such a common operation that is useful to have a helper function for it.

```
string int_to_string(int n)  
{  
    ostringstream ostr;  
    ostr << n;  
    return ostr.str();  
}
```

String ↔ Number conversion in C++11

- C++11 introduced some standard library functions that can directly convert basic types to `std::string` objects and vice-versa.
- These functions are declared in `<string>`.
- `std::to_string()` converts basic numeric types to strings.
 - Example:

```
int number = 123;  
string text = to_string(number);
```
- The set of functions
 - `std::stoi`, `std::stol`, `std::stoll` - convert to integral types
 - `std::stof`, `std::stod`, `std::stold` - convert to floating-point values.
 - Example:

```
text = "456"  
number = stoi(text);
```

```

/**
READ TIME IN SEVERAL FORMATS
ex:
21:30
9:30 pm
10 am
and show it in "military format" (HH:MM) and "am/pm format" (HH:MM am/pm)
*/
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

/**
Converts an integer value to a string, e.g. 3 -> "3".
@param s an integer value
@return the equivalent string
*/
string int_to_string(int n)
{
    ostringstream ostr;
    ostr << n;
    return ostr.str(); //convert stringstream into string
}

/**
Reads a time from standard input
in the format hh:mm or hh:mm am or hh:mm pm
@param hours filled with the hours
@param minutes filled with the minutes
*/
void read_time(int &hours, int &minutes)
{
    string line;
    string suffix;
    char ch;

    getline(cin, line);

    istringstream instr(line); //initialize stringstream from string
    // ALTERNATIVE:
    // istringstream instr;
    // instr.str(line);

    instr >> hours;

    minutes = 0;

    instr.get(ch); // do {instr.get(ch);} while (ch==' '); // EFFECT ?
    // try with 18:45 and 18: 45 and 18 :45 and 18 : 45
    if (ch == ':')
        instr >> minutes;
    else // input was 'HH am' OR 'HH pm'
        instr.unget(); // OR instr.putback(ch);

    instr >> suffix;
    if (suffix == "pm")
        hours = hours + 12;
}

```

```

/**
Computes a string representing a time.
@param hours the hours (0...23)
@param minutes the minutes (0...59)
@param military
    true for military format,
    false for am/pm format,
*/
string time_to_string(int hours, int minutes, bool military)
{
    string suffix;
    string result;

    if (!military)
    {
        if (hours < 12)
            suffix = "am";
        else
        {
            suffix = "pm";
            hours = hours - 12;
        }
        if (hours == 0) hours = 12;
    }

    result = int_to_string(hours) + ":";
    if (minutes < 10) result = result + "0";
    result = result + int_to_string(minutes);

    if (!military)
        result = result + " " + suffix;

    return result;
}

int main()
{
    int hours;
    int minutes;

    do
    {
        cout << "Please enter the time\n";
        cout << "HH[:MM] or HH[:MM] am or HH[:MM] pm (0:0 => END): ";

        read_time(hours, minutes);

        cout << "Military time: "
              << time_to_string(hours, minutes, true) << "\n";
        cout << "Using am/pm: "
              << time_to_string(hours, minutes, false) << "\n";
        cout << endl;

    } while (hours!=0 || minutes!=0); // TO DO by students

    return 0;
}

```

The program is not robust enough... ☹
TO DO: Turn it more robust. ☺

```

/*
Read fractions and do arithmetic operations with them

STRINGSTREAMS
By using STRINGSTREAMS you can avoid mixing cin << ... and getline(cin, ...)
You may always use getline()
*/

/*
TO DO:
Fraction sumFractions(Fraction f1, Fraction f2)
Fraction subtractFractions(Fraction f1, Fraction f2)
Fraction divideFractions(Fraction f1, Fraction f2)
*/

#include <iostream>
#include <iomanip>
#include <cctype>
#include <string>
#include <sstream>

using namespace std;

struct Fraction
{
    int numerator;
    int denominator;
};

/* // READING / WRITING DIRECTLY FROM / TO cin / cout
bool readFraction(Fraction &f)
{
    char fracSymbol;
    int numerator;
    int denominator;
    bool success;

    cout << "n / d ? ";
    cin >> numerator >> fracSymbol >> denominator;
    if (cin.fail())
    {
        cin.clear();
        success = false;
    }
    else
    {
        if (fracSymbol == '/')
        {
            f.numerator = numerator;
            f.denominator = denominator;
            success = true;
        }
        else
            success = false;
    }

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

    return success;
}
*/

```



```

bool readFraction(Fraction &f)
{
    string fractionString;
    char fracSymbol;
    int numerator;
    int denominator;
    bool success;

    cout << "n / d ? "; // should be done before calling readFraction()... ?
    getline(cin, fractionString);

    istringstream fractionStrStream(fractionString);
    if (fractionStrStream >> numerator >> fracSymbol >> denominator)
    {
        if (fracSymbol == '/')
        {
            f.numerator = numerator;
            f.denominator = denominator;
            success = true;
        }
        else
            success = false; // TO DO: write these tests in a different way
    }
    else
        success = false; // suggestion: initialize 'success'
    return success;
}

Fraction multiplyFractions(Fraction f1, Fraction f2)
{
    Fraction f;

    f.numerator = f1.numerator * f2.numerator;
    f.denominator = f1.denominator * f2.denominator;
    return f;
}

void showFraction(Fraction f)
{
    cout << f.numerator << "/" << f.denominator;
}

int main()
{
    Fraction f1, f2, f3;

    cout << "Input 2 fractions:\n";
    if (readFraction(f1) && readFraction(f2))
    {
        f3 = multiplyFractions(f1, f2);

        cout << "Product: ";
        showFraction(f3);
    }
    else
    {
        cout << "Invalid fraction\n";
    }
    cout << endl;

    return 0;
}

```