

DATA STRUCTURES AND ALGORITHMS

L.BIO & MEB – 2021/2022

LECTURES MATERIAL

C/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                                ← OPENING DOCUMENTATION / COMMENTS
DATE   : 2021/10/19

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/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 */
```

```
#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; //avoids the need of MANY MANY using 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** – max and min values of integer types
- **cfloat** - max and min values of float types
- ... 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, ...
 - For a complete list consult <https://en.cppreference.com/w/cpp/keyword>

- Recommended naming conventions for variables (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.
 - 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    : 2020/02/14
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.
-

```

/*
...
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; // TO DO: "investigate" usage of scanf() and scanf_s()

    //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 vars 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 sucessful, 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 `outstream`:
 - `value` may be a constant, variable or the result of an expression
 - returns outstream as its result (if sucessful, else 0)
 - `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:
 - `&, |, ^, ~, <<, >>`
- Explicit type casting operator
- `sizeof`

THERE ARE OTHER OPERATORS (SEE NEXT PAGE)

Precedence and associativity (grouping) 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:
 - $x = 2 + 3 * 4;$ // $x = \dots?$
 - $y = (2 + 3) * 4;$ // $y = \dots?$
 - $z = y = x + 10;$ // 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 } .

```
{  
    statement_1    ← REMEMBER: statements end with ;  
    statement_2  
    ...  
    statement_N  
}
```

- 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 ☺ WHY?

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

- **if ... else statement**

```
if (boolean_expression)
    statement_1
else
    statement_2
```

- 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" AND 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 😊

    //output 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;

    //output 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
    {
        validOperation = true;

        //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;
    }

    //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; // also available clog stream
}
=====


```

NOTE: **cerr** is unbuffered (the message is written immediately); **clog** is buffered

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

```

const std::streamsize oldp = cin.precision();
cout << fixed << setprecision(3) << x << endl << setprecision(oldp) << 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
 - can be used with any type of loop
 - can be used to end an infinite loop

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

- **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 lengthy loops,
makes the code difficult to read ... :-)

- **"infinite" loops**

- **while (true) { ... };** // "..." represents the statement(s)
 - **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**!
- Let us see in detail why this happens → **explanation**
- 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.
- 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 <iostream> 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
- `cin >> ws`
 - `ws` is an input-only I/O manipulator that discards leading whitespace (space, tab or enter) from an input 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 num = 0;
char ch;
ch = cin.peek();
if(ch == '#') {
    ch = cin.get();
    if(ch == 'A' || ch == 'B' || ch == 'C' || ch == 'D') {
        ch = cin.get();
        if(ch == 'E') {
            ch = cin.get();
            if(ch == '$') {
                ch = cin.get();
                cout << endl << "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
    while (i < numOperations) // COMMON ERRORS: - forget initialization
    { // 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'); //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 the initial value of validOperands is true
- 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=='/'))`

```

/*
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); // NOTE THE CONDITION

        //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;
}

```

NOTES:

1 – to continue reading from `cin` after the user types **CTRL-Z** it is necessary to clear the error flags associated to the input stream, using `cin.clear()`.

2 – 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 BABYLONIAN 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); // fabs() returns absolute value

    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)

```

/*
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') // TO DO: alternative condition
{
    //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 - NOT RECOMMENDED
//-----
cout << endl << "FOR - 5\n";
const unsigned int NUM_VALUES = 10;
double sum = 0, value, mean;

i = 1;
for ( ;i <= NUM_VALUES; ) // DON'T DO ANYTHING LIKE THIS
{
    cout << "\n" << i << "? ";
    cin >> value;
    sum = sum + value;
    i++;
}

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

// FOR - 5 (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;

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

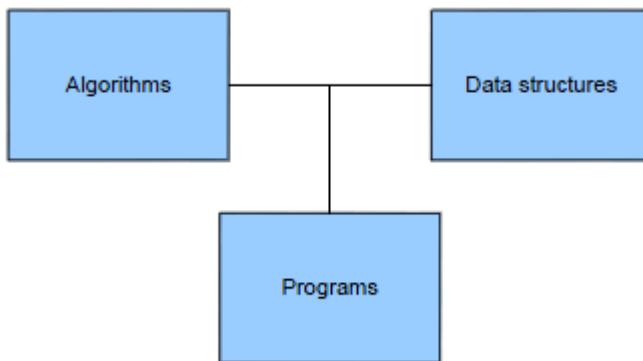
...
2x9 = 18

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

FUNCTIONS

Modular programming

- Program development



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

- Top-down design

- Top down design means
breaking the problem down into components (modules) recursively.
- Here we 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**

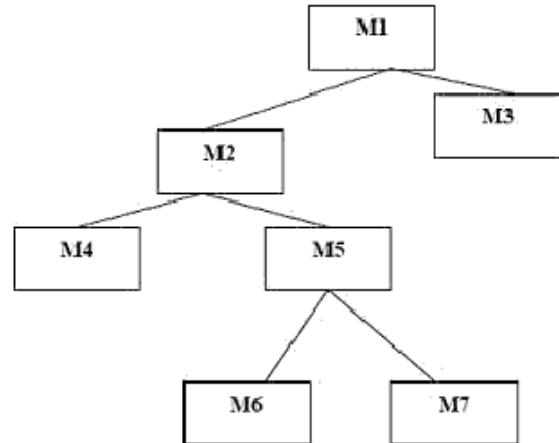
- While the top-down design methodology is a general tool, how we approach it will potentially be language dependent.
- In **procedural programming**
 - the design effort tends to concentrate on the functional requirements
 - and recursively subdivides the functionality into procedures/functions/subroutines until each is a simple, easily understood entity.
 - Examples of procedural languages are C, Basic, Pascal, 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



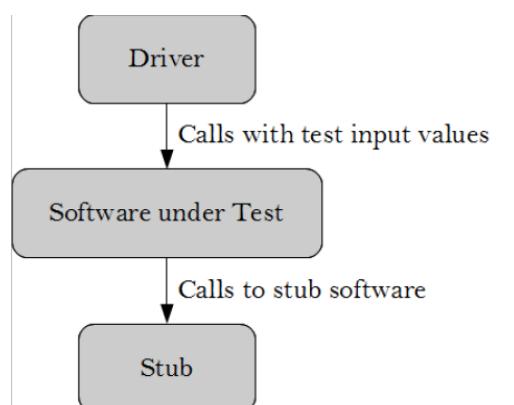
- **Code development**

- **Top-down**

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

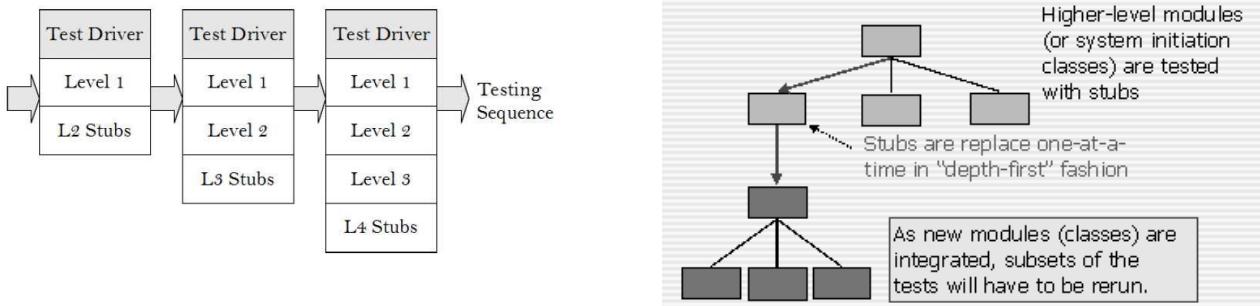
- **Bottom-up**

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



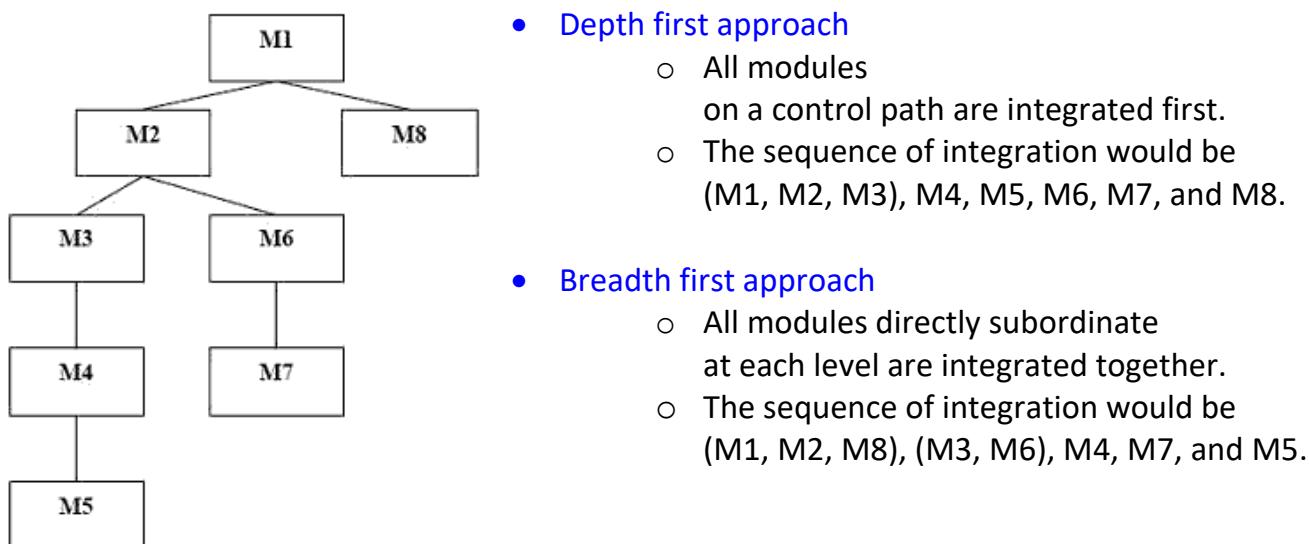
- **Stub:**
is temporary or dummy software that is required by the software under test for it to operate properly.
- **Test driver:**
calls the software under test, passing the test data as inputs.

- **Top-down integration**

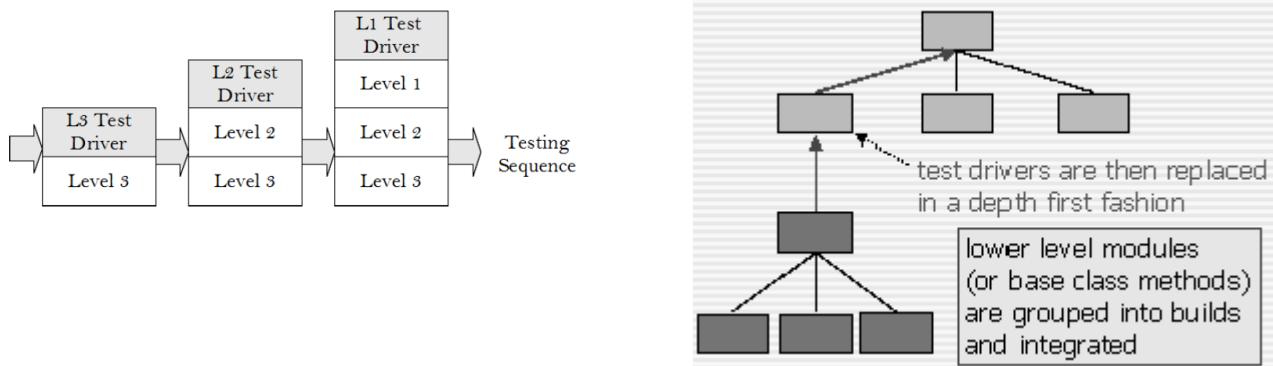


- The highest-level modules are tested and integrated first.
- This allows high-level logic and data flow to be tested early in the process
- **Advantages:**
 - The top layer provides an early outline of the overall program
 - helping to find design errors early on and
 - giving confidence to the team,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

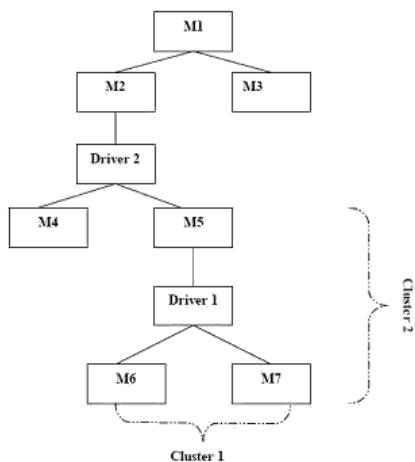


- **Bottom-up integration**



- 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 syntactically correct call
that would produce a faulty result:

```
double total_price(int num_items, double item_price)
{
    ...
}

int numItems = 100, itemPrice = 5;
double amount_to_pay;

amount_to_pay = total_price(itemPrice, numItems);
```

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

- **Local variables (/ constants or identifiers, in general)**
 - Variables declared in a function:
 - Are local to that function
 - they cannot be used from outside the function
 - Have the function as their scope
 - Example: variables declared in the **main** function of a program:
 - Are local to the main part of the program, they cannot be used from outside the main part
 - Have the **main** function as their scope
 - Formal parameters are local variables
 - They are used just as if they were declared in the function body
 - Do NOT re-declare the formal parameters in the function body, they are declared in the function declaration
- **Global variables and constants (or identifiers, in general)**
 - Declared outside any function body
 - Declared before any function that uses it
 - Available to more than one function
 - **Should be used sparingly**
 - Generally make programs more difficult to understand and maintain
 - Example:

```

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

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

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

```

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

TO DO BY STUDENTS:

- Write a function to calculate the square root of a number, using the previously referred Babylonian 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)
@return 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
@return 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 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
@return (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
@return 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
@return 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]
@return (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
@return 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
@return 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 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++)
    {
        int operand1, operand2;
        char operation;
        double result; // result of "operand1 operation operand2"
        double answer; // user answer to the arith. operation question
        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
@return 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
@return 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]
@return (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
@return 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
@return user answer
*/
double readUserAnswer(char operation, int operand1, int operand2);

```

```

//-----
int main()
{
    const int NUM_OPERATIONS = 10;

    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++)
    {
        int operand1, operand2;
        char operation;
        double result; // result of "operand1 operation operand2"
        double answer; // user answer to the arithm. operation question

        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 selected number of places
@param x: number to be rounded
@param n: number of places
@return 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
@return 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 (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
@return 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
@return 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 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 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 (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
@return 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
@return 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 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++)
    {
        int operand1, operand2;
        char operation;
        double result; // result of "operand1 operation operand2"
        double answer; // user answer to the arithm. operation question

        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: TRACE THE EXECUTION OF THE CALL: writeVertical(123)**

```

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

unsigned long factorialIte(unsigned n) // iterative version
{
    int f = 1;
    for (unsigned i = n; i >= 2; i--)
        f = f * i;
    return f;
}

unsigned long factorialRec1(unsigned n)
// recursive version; a bad example of recursion... why ?
{
    if (n == 0 || n == 1)
        return 1;
    else
    {
        unsigned long f;
        f = n * factorialRec1(n-1);
        return f;
    }
}

unsigned long long factorialRec2(unsigned int n)
// recursive version; a bad example of recursion... why ?
{
    if (n == 0 || n == 1)
        return 1;
    else
        return (n * factorialRec2(n-1));
}

int main(void)
{
    unsigned i;
    cout << " i           factorialIte      factorialRec1
factorialRec2\n";
    cout << "-----\n";
    for (i=0; i <= 20; i++)
        cout << setw(2) << i << " - " <<
            setw(19) << factorialIte(i) << "   " <<
            setw(19) << factorialRec1(i) << "   " <<
            setw(19) << factorialRec2(i) << endl;
    //cout << ULONG_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 <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;
    // n1 n2 n3 are 3 consecutive numbers in the Fibonacci sequence
    // n3 = n1 + n2
    int n1 = 1;
    int n2 = 1;
    int n3;
    for (int i = 3; i <= n; i++)
    {
        n3 = n1 + n2;
        n1 = n2;
        n2 = n3;
    }
    return n3;
}

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;
}

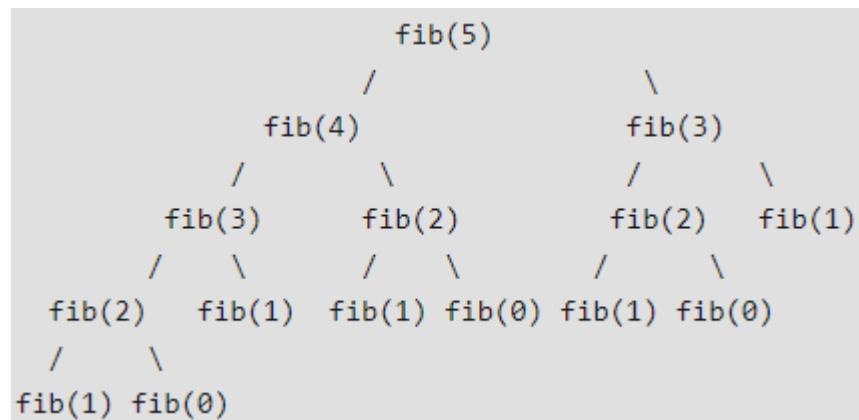
```

- **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.

NOTE:

- The recursive version for computing the n-th element of the Fibonacci sequence does a lot of repeated work!



```

// 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'; // NOTE the conversion of each digit
            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 3.14 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

    dx = (b-a) / n;

    x1 = a;
    x2 = x1 + dx;

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

    return totalArea;
}

```

```

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

    cout << "Integrate y = x*x in the interval [a,b]\n";
    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; // COMPILE 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);

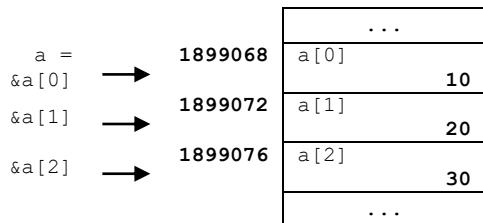
    for (i=0; i<NMAX; i++)
        cout << "a[" << i << "] = " << a[i]
        << ", &a[" << i << "] = " << (unsigned long) &a[i] << endl;

    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

```



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

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;
}

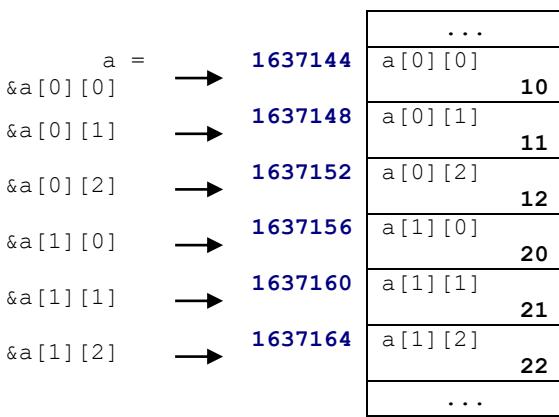
```

	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

```



```

/*
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) //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/NOMBRE 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;
}

```

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** (=>**#include <cstddef>**); **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 after 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 <cstddef> // 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 <cstddef>

using namespace std;

/*
Reads salaries
returns - a vector of salary values
*/
vector<double> readsSalaries()
{
    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> vectorElemsBetween(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();

    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 <cstddef>

using namespace std;

/*
Reads salaries
returns a vector containing the salaries
*/
vector<double> readsSalaries() // TODO: modify so that the result is returned through a parameter
{
    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> vectorElemsBetween(const vector<double> &v, double low,
double high) // TODO: modify so that the result is returned through a parameter
{
    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/NOMBRE 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/NOMBRE_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 <cstddef>

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(); // but could have diff. no. of grades for each student
    ...

    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 <cstddef>

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;
}

```

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 12777440 of the memory;
then `xPtr` will have value 12777440.
 - ***** - 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 12777440, 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 = 12777440; // you shouldn't do this, WHY?`
 - `p2 = p1+3; // if sizeof(int) is 4,`
`// p2 will point to address 12777440+3*4=12777452`

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):
○ a[3] = 27;
○ *(p+3) = 27;

```

- Also, the following code is syntactically correct:

```

void showArray(const int *a, size_t size) { ... }
...
int values[10];
...
showArray(values, 10);

```

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, in C).
 - 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.
 - In C++, use **nullptr** instead of **NULL** (see *why in the examples, in the next pages*)
- 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.
 - 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.

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

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

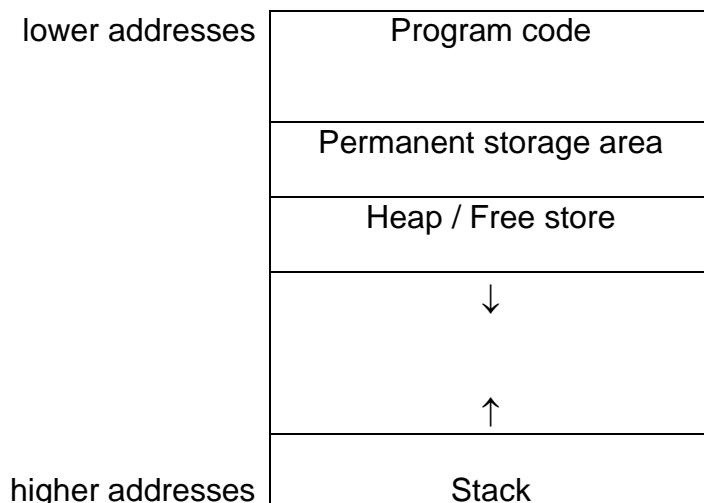
- Note that **function pointer syntax is flexible**; it can either look like most other uses of pointers, with & and *, or you may omit that part of syntax.

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 (*examples 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](#): using [functions malloc\(\)](#) and [free\(\)](#) - [see next pages](#)
 - the one [defined by C++](#): using [operators new](#) and [delete](#) - [see next pages](#)
- Memory allocated by dynamic allocation functions is obtained from the [heap/free store](#).
 - [Heap](#): A dynamic memory area that is allocated/freed by the [malloc\(\) / free\(\)](#).
 - [Free Store](#): A dynamic memory area that is allocated/freed by [new / delete](#).
 - Its possible for [new](#) and [delete](#) to be implemented in terms of [malloc\(\)](#) and [free\(\)](#); so, technically, they could be the same memory area. However, as the standard doesn't specify this, its [best](#) to treat them separately, and [not to mix malloc\(\)/delete\(\) or new/free](#).



C dynamic memory allocation

- The core of C's allocation system consists of the functions: `malloc()` and `free()`
 - => `#include <cstdlib>`
 - `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;`

NOTE:

- **malloc()** allows to change the size of buffer using **realloc()** while **new** doesn't

See the following slides (with animation) in Moodle:

POINTERS & DYNAMIC MEMORY ALLOCATTION

PROGRAM
OUTPUT

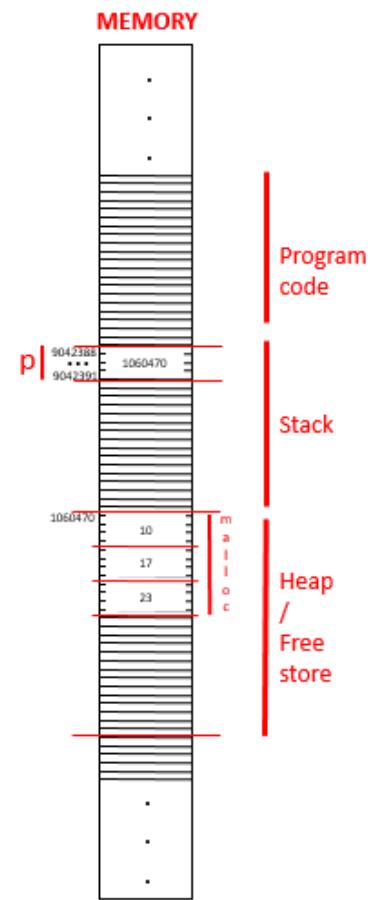
```
#include <iostream>
#include <cstdlib>
using namespace std;

void main(void)
{
    int *p;
    cout << (long)&p << endl;
    cout << p << endl;

    p = (int*)malloc(3 * sizeof(int));
    cout << (long)p << endl;

    *p = 10;
    p[1] = 17;
    *(p + 2) = 23;
    for (size_t i = 0; i < 3; i++) cout << p[i] << endl;

    free(p);
    cout << (long)p << endl;
    for (size_t i = 0; i < 3; i++) cout << p[i] << endl;
}
```



```

// 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 = 0018FEEC (hexadecimal)
&aPtr = 1638140 (decimal)

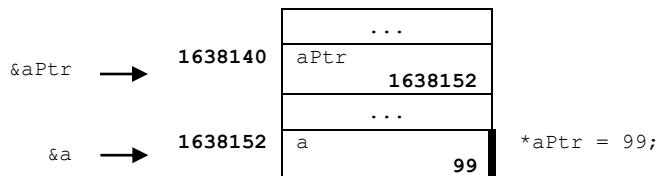
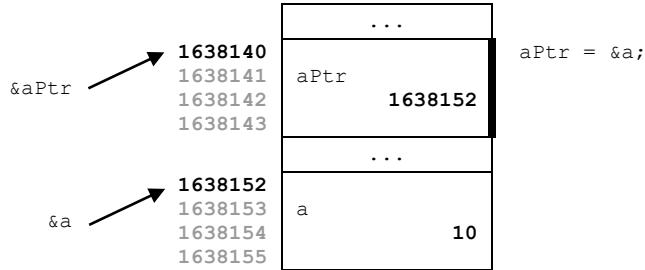
aPtr = 0018FF08 (hexadecimal)
aPtr = 1638152 (decimal)

    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 << endl << endl;

    swap1(&a,&b); // NOTE: the &'s

    cout << "MAIN after swap1(): a = " << a << ", " << b << endl << endl;
    swap2(a,b);

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

```

```

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

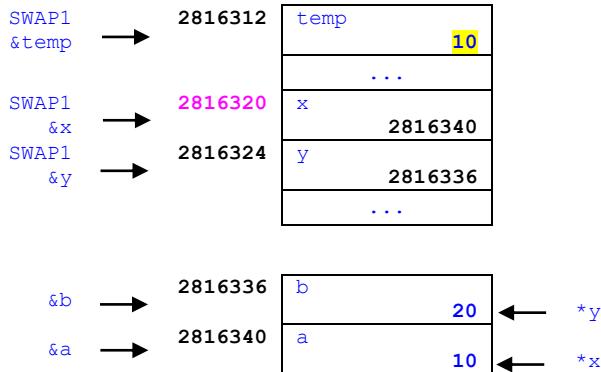
SWAP1_a
  &x = 2816320,  &y = 2816324,  &temp = 2816312 (decimal)
    x = 2816340,  y = 2816336 (decimal)
    *x = 10 *y = 20,  temp = -175524
SWAP1_b
  *x = 20 *y = 10,  temp = 10
MAIN after swap1(): a = 20,  b = 10

SWAP2_a
  &x = 2816340,  &y = 2816336,  &temp = 2816320 (decimal)      temp has same address as x in swap1()
    x = 20,  y = 10,  (decimal)  temp = 1759800264                just by chance; memory was reused
SWAP2_b
  x = 10,  y = 20,  temp = 20,
MAIN after swap2(): a = 10,  b = 20

```

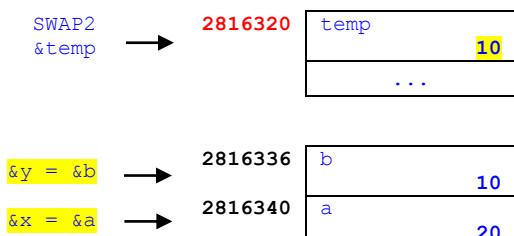
```
void swap1(int *x, int *y)
{...}
```

```
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;

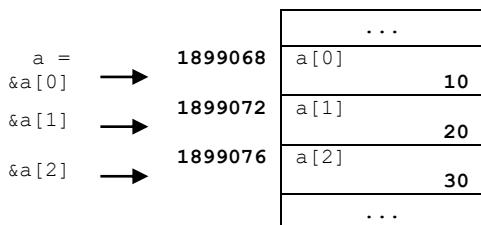
    aPtr = a; // an array identifier is a pointer to the 1st array element

    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

```



```

// 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 allocated memory
    // free(a); // C-style
    delete [] a; // C++-style

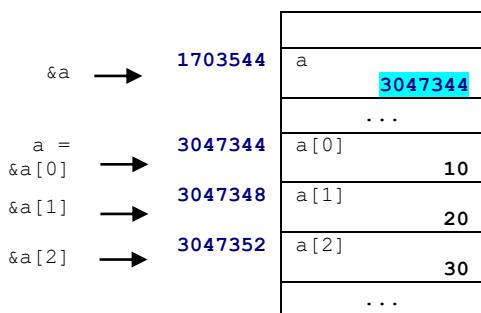
    // a[0] = 100; // should not be done ... why ?
}

```

```

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

```



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;
}

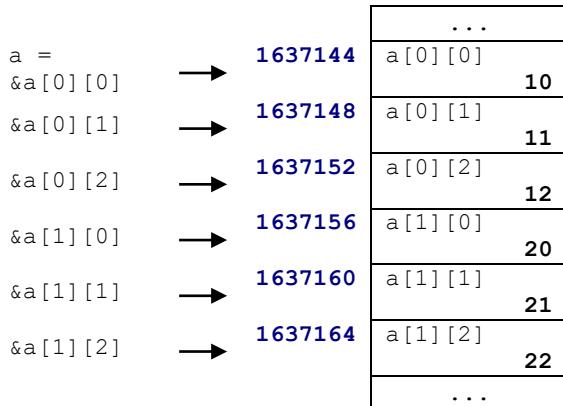
```

	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

```



```

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

    // 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];
    for (i=0; i<nLin; i++)
        a[i] = new int[nCol]; // allocate memory for each line of the array

    // 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;
}

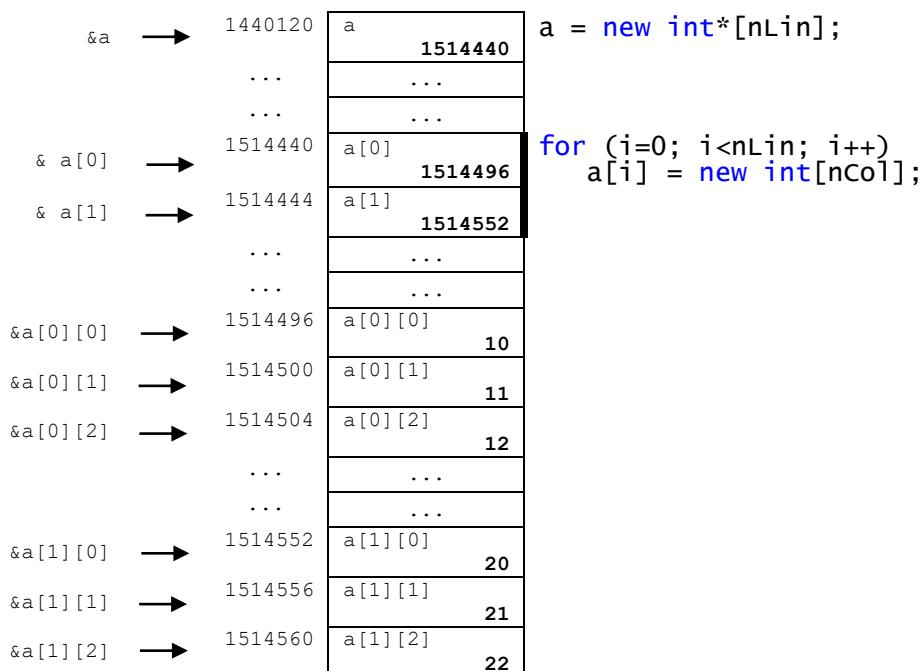
nLin ? 2
nCol ? 3
&a = 1440120
 a = 1514440
& a[0] = 1514440
& a[1] = 1514444
 a[0] = 1514496
 a[1] = 1514552
a[0][0] = 10, &a[0][0] = 1514496
a[0][1] = 11, &a[0][1] = 1514500
a[0][2] = 12, &a[0][2] = 1514504
a[1][0] = 20, &a[1][0] = 1514552
a[1][1] = 21, &a[1][1] = 1514556
a[1][2] = 22, &a[1][2] = 1514560

```

```

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

```



```
// IN C++, WHY USE nullptr INSTEAD OF NULL
// 1- Distinguishing between null and zero
// 2-Programmers have often requested that the null pointer constant have a name (rather than just 0)
// source: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2431.pdf

#include <iostream>
#include <cstddef> // where NULL is defined -> #define NULL 0

using namespace std;

void f1(int *p) // overloading
{
    cout << "version w/pointer parameter called\n";
}

void f1(int x) // overloading
{
    cout << "version w/int parameter called\n";
}

int main()
{
    f1(5);
    f1(NULL); // NULL = 0, version w/int parameter called !!!
    f1(nullptr);
}
```

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
=>
`in C → #include <string.h>` `in C++ → #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 name[MAX_NAME_SIZE + 1];`
 - `MAX_NAME_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

- NOTE:
 - do not 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

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`
- The following 2 declarations with initialization are equivalent:
 - `char s[] = "Hello!"; // s can be modified`
 - `char *s = "Hello!"; // s can't be modified`

but the 1st string can be modified (BUT ... **CAN ITS SIZE BE MODIFIED ?**)
 while the 2nd can't; it is stored in non-modifiable memory.

C-string Output

- In C, C-strings can be written using `printf("%s",)` or `printf_s("%s",)`
- In C++, C-strings can be written with the insertion operator (`<<`)
 - Example:
`char msg[] = "Hello";`
`cout << msg << " world!" << endl;`

C-string Input

- In C, C-strings can be read using `scanf("%s",...)` or `scanf_s("%s",...)`
 - Example : `scanf_s("%s", name, MAX_NAME_SIZE)`
- In C++, 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(.....);`
- **`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:
 - `msg = "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

- Strings are compared in the following way:
 - the characters in similar positions are compared until one of the following conditions happens:
 - one of the characters is before the other, taking into account the corresponding character codes; the string containing this character is considered to be "before" the other one, in lexicographical order
 - the end of one of the strings is found; this string is considered to "before" the other one
- The == operator does not work as expected with C-strings.
- The C-library 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'?*
 - 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))
    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,
but it is not necessary to pass the number os elements os the (string) array. **WHY?**

Printing strings in C language

- To print strings using the **printf()** or **printf_s()** library functions, the **%s** formater must be used.
- Example:

```
char salutation[] = "Hello world!";
printf ("%s", salutation);
printf_s ("%s", salutation, strlen(salutation));
```

Safe versions of string manipulation functions

- Some string manipulation functions (ex: **strcpy()** and **strcat()**) are unsafe functions.
 - For example, when you try to copy a string using **strcpy()**, to a buffer which is not large enough to contain it, it will cause a buffer overflow.
- Those unsafe functions have security enhanced versions (having the same name with suffix **_s**):
 - **strcpy_s()** and **strcat_s()** are enhanced versions of **strcpy()** and **strcat()**.
- With **strcpy_s()** you can specify the size of the destination buffer to avoid buffer overflows during copies.
- Example:

```
char s1[10]; // a buffer which holds 10 chars including the null character
char s2[] = "A string longer than 10 chars";
strcpy(s1, s2); // this will corrupt memory because of the buffer overflow
strcpy_s(s1, 10, s2); // strcpy_s can not write more than 10 chars;
                      // this will cause an execution error
```

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 (like when used with vectors)
- Example:
`string str("Mary");`
`cout << str[6] << endl; // INVALID ACCESS ... DETECTED ...?`
`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 functions **length()** or **size()** return the number of characters in the string object:

- Example:

```
size_t n = s.length();
```

Converting C-strings to string objects

- The conversion is automatic:

```
char cstr[ ] = "C-string";
string str = cstr;
```

Converting strings 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+** (see later) 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

- There are many functions that can be used to manipulate strings, for example, for finding characters or substrings (the first, or the last occurrence) or erasing characters
 - consult some manuals / web pages
 - see following examples
- The **find** functions return a a special value **string::npos** to indicate that no occurrence was found (*in fact, it is an integer value equal to **ULONG_MAX***).
 - Example:


```
string s = "Hello world!";
if (s.find("hello") != string::npos)
    cout << "'hello' was found" << endl;
else
    cout << "'hello' was not found" << endl;
```

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

Function	Description	Cautions
<code>strcpy(Target_String_Var, Src_String)</code>	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> .
<code>strncpy(Target_String_Var, Src_String, Limit)</code>	The same as the two-argument <code>strcpy</code> 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 <code>strcpy</code> . Not implemented in all versions of C++.
<code>strcat(Target_String_Var, Src_String)</code>	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.
<code>strncat(Target_String_Var, Src_String, Limit)</code>	The same as the two-argument <code>strcat</code> 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 <code>strcat</code> . Not implemented in all versions of C++.
<code>strlen(Src_String)</code>	Returns an integer equal to the length of <i>Src_String</i> . (The null character, '\0', is not counted in the length.)	
<code>strcmp(String_1, String_2)</code>	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 <code>false</code> . Note that this is the reverse of what you might expect it to return when the strings are equal.
<code>strncmp(String_1, String_2, Limit)</code>	The same as the two-argument <code>strcat</code> 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 <code>strcmp</code> . Not implemented in all versions of C++.

Member Functions of the Standard Class `string`

Example	Remarks
Constructors	
<code>string str;</code>	Default constructor creates empty <code>string</code> object <code>str</code> .
<code>string str("sample");</code>	Creates a <code>string</code> object with data "sample".
<code>string str(a_string);</code>	Creates a <code>string</code> 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 <code>true</code> if <code>str</code> is an empty string; <code>false</code> 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</code> <code>str1 != str2</code>	Compare for equality or inequality; returns a Boolean value.
<code>str1 < str2</code> <code>str1 > str2</code>	Four comparisons. All are lexicographical comparisons.
<code>str1 <= str2</code> <code>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> .

STRINGS - examples

```
/*
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 + 1];
    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);
    {
        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 <cstddef>

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 <cstddef> //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 <cstddef> //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 <cstddef> //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;
}
```

```
=====
/*
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(cin, ... ) 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()); //QUESTION: WHAT DOES .c_str() RETURN ?
    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

=====

ACCESSING COMMAND LINE ARGUMENTS

=====

```
// Program (test.c) that shows its command line arguments.  
// Command line arguments are passed to the program as an array of C-strings  
  
// NOTE: run this program from the command prompt  
// EX: C:\Users\username> test abc 123  
  
#include <iostream>  
  
using namespace std;  
  
void main(int argc, char **argv) // OR void main(int argc, char *argv[])  
{  
    for (int i=0; i<argc; i++)  
        cout << "argv[" << i << "] = " << argv[i] << endl;  
}  
  
  
//=====  
  
// Program (sum.c) that shows the command line arguments  
// Command line arguments are passed to the program as an array of C-strings  
  
// NOTE: run this program from the console / terminal window (cmd in windows)  
// EX: C:\Users\username> sum 123 456  
  
#include <iostream>  
#include <string>  
#include <sstream>  
  
using namespace std;  
  
void main(int argc, char *argv[]) // OR void main(int argc, char **argv)  
{  
    int n1, n2, n3;  
  
    if (argc != 3)  
    {  
        cout << "USAGE: " << argv[0] << " integer1 integer2\n";  
        exit(1);  
    }  
    n1 = atoi(argv[1]);  
    n2 = atoi(argv[2]);  
    n3 = n1 + n2;  
    cout << n1 << " + " << n2 << " = " << n3;  
}
```

STRUCT type & typedef

- **STRUCTs**

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

```
struct Person // the new type is named "Person" - C++ syntax
{
    string name;
    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 / Pointers to structs**

- (*Considering the declarations above*)

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

- (*Considering the following declarations*)

```
Person * ptr;
cout << ptr->name << "-" << ptr->gender << ... ... << endl;
```

the alternative to `ptr->name` would be:

`(*ptr).name`

WHY NOT *ptr.name ?

- **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**

- 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
```

- 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 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 ... see next examples
    cin >> p.name;
    cout << "Gender ? ";
    cin >> p.gender;
    cout << "Age ? ";
    cin >> p.age;

    return p; // NOTE: a function can return a struct
    // TODO: modify to void readPerson(Person &p) => modify main()
}

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 (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;
}
=====
```

```
=====
/*
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 ? ";

    if (cin >> numerator >> fracSymbol >> denominator) // COMPARE WITH NEXT
        if (fracSymbol == '/') // readFraction() VERSION
    {
        f.numerator = numerator;
        f.denominator = denominator;
        success = true;
    }
    else
        success = false;
    else
        success = false;
    if (!success)
    {
        cin.clear();
        cin.ignore(1000, '\n');
    }
    return success;
}

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

```

cout << "n / d ? ";

if (cin>>numerator>>fracSymbol>>denominator, fracSymbol == '/') //comma exp.
{
    f->numerator = numerator;
    //(*f).numerator = numerator;
    f->denominator = denominator;
    //(*f).denominator = denominator;
    success = true;
}
else
    success = false;
if (!success)
{
    cin.clear();
    cin.ignore(1000, '\n');
}

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)) // WHICH readFraction() IS CALLED
    if (readFraction(&f1) && readFraction(&f2)) // IN EACH CASE ?
    {
        f3 = multiplyFractions(f1,f2);

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

    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 <cstddef>
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

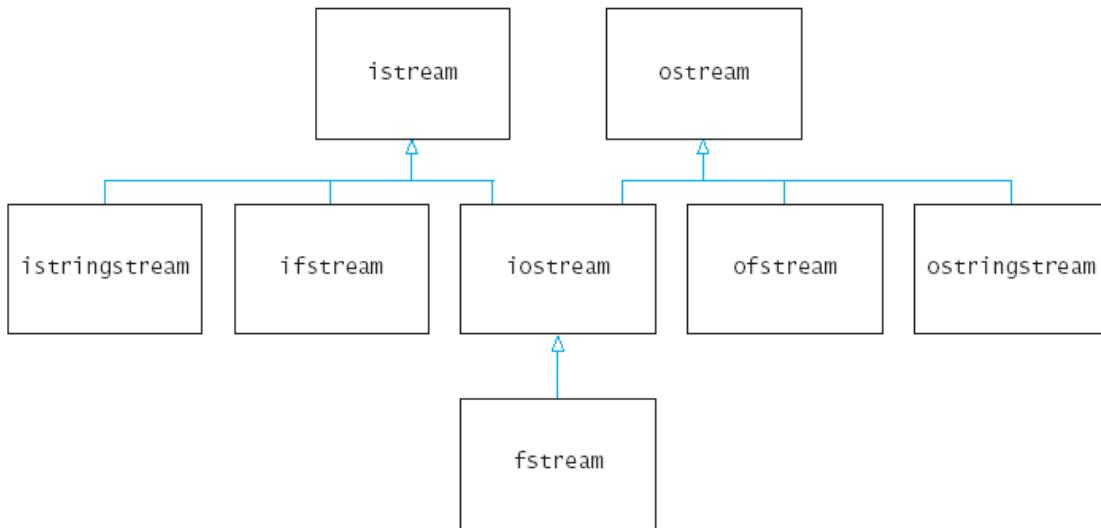
    readNames(namesAndAges,NUM_NAMES);
    cout << endl;
    showNames(namesAndAges,NUM_NAMES);

    return 0;
}
```

STREAMS / FILES

I/O Streams

- I/O refers to program Input and Output
- 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 access output data **after the program ends**
 - to store data **permanently**

Text files vs. Binary files

- Usually files are classified in two categories:
 - **text files** (ASCII, Unicode, UTF-8, UTF-16, ...)
 - 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 inStream;
ofstream outStream;
```

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 inStream;**
 - **ofstream outStream;**
 - **inStream.open("infile.txt");**
 - connects **in_stream** to "infile.txt"
 - **outStream.open("C:\Mieic\Prog\programs\outfile.txt");**
 - connects **outStream** to "outfile.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 inStream("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
 - Since **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
 - **mode** determines how the file is opened;
can be the bitwise 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 inStream;
inStream.open("infile.txt");
int oneNumber, anotherNumber;
inStream >> oneNumber >> anotherNumber;
```
- Example 2:

```
ofstream outStream;
outStream.open("outfile.txt");
outStream << "Resulting data:";
outStream << oneNumber << endl << anotherNumber << endl;
```

Closing a file

- After using a file, it should be closed.
This disconnects the stream from the file
 - Example: `inStream.close();`
- The system will automatically close files if you forget,
but ...
- 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:

```
inStream.open("numbers.txt");
if( inStream.fail( ) ) // OR if( !inStream.is_open( ) )
{
    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(inStream >> 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(! inStream.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

```
inStream.get(next);           // NOTE: first, input must be tried
while ( ! inStream.eof( ) ) // then you can test for EOF
{
    cout << next;
    inStream.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
 - **outStream.setf(ios::fixed);**
 - **outStream.setf(ios::showpoint);**
 - **outStream.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 as function call 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(ifstream &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.
 - **ifstream** as well as **cin** are objects of type **istream**
 - **ofstream** 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 **ifstream** object or **cin**.
 - **max = get_max(inStream);**
 - **max = get_max(cin);**
 - both **cin** and **inStream** can be used as arguments of a call to **get_max()**, whose parameter is of type **istream &**

Binary I/O (not lectured in 2021/2022)

Random access (not lectured in 2021/2022)

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, by JAS)
*/
#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,
//but adds a 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') //NOTE: get() READS SPACE AND NEWLINE CHARACTERS
        {
            n++;
            fout << n << ' ';
        }
        fin.get(next);
    }

    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;
}
```

STRINGSTREAMS

String Streams

- We saw how a stream can be connected to a file.
- A stream can also be connected to a string.
- With stringstream 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 stringstream 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 outstr;
outstr << setprecision(5) << sqrt(2);
```

- To obtain a string from the stream, call the `str` member function.
 - `string output = outstr.str();`
- Example: (builds the string "January 23, 1955")

```
string month = "January";
int day = 23;
int year = 1955;
ostringstream outstr;
outstr << month << " " << day << "," << year;
string output = outstr.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 outstr;
    outstr << n;
    return outstr.str();
}
```

"String \leftrightarrow Number" conversion since 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(number);`

```


/** 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 outstr;
    outstr << n;
    return outstr.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==' ');
    // try with 18:45 and 18: 45 and 18 :45 and 18 : 45
    if (ch == ':')
        instr >> minutes;
    else
        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;
}

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;
}


```

```

/*
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 ? ";
    getline(cin, fractionString);

    istringstream fractionStrStream(fractionString);
    if (fractionStrStream >> numerator >> fracSymbol >> denominator)
        if (fracSymbol == '/')
    {
        f.numerator = numerator;
        f.denominator = denominator;
        success = true;
    }
    else // TO DO: write these tests in a different way
        success = false; // suggestion: initialize 'success'
    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))
    {
        f3 = multiplyFractions(f1, f2);

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

    return 0;
}

```

CLASSES

C++ the Object Based Paradigm

Object Oriented Programming

- Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability.
- Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.
- The important features of object-oriented programming are:
 - Bottom-up approach in program design
 - Programs organized around objects, grouped in classes
 - Focus on data with methods to operate upon object's data
 - Interaction between objects through functions
 - Reusability of design through creation of new classes by adding features to existing classes
- Some examples of object-oriented programming languages are:
 - C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

(source: http://www.tutorialspoint.com/object_oriented_analysis_design/ooad_object_oriented_paradigm.html)

Object:

- An object has state,
- exhibits some well-defined behaviour,
- and has a unique identity.

Class:

- A class describes a set of objects that share a common structure, and a common behaviour.
- A single object is an instance of a class.

Object-Oriented Analysis

- Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.
- The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions.
They are modelled after real-world objects that the system interacts with.
In traditional analysis methodologies, the two aspects - functions and data - are considered separately.
- The primary tasks in object-oriented analysis (OOA) are:
 - Identifying objects
 - Organizing the objects by creating object model diagram
 - Defining the internals of the objects, or object attributes
 - Defining the behavior of the objects, i.e., object actions
 - Describing how the objects interact
- The common models used in OOA are use cases and object models.

Object-Oriented Design

- Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis.
In OOD, concepts in the analysis model, which are technology-independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.
- The implementation details generally include:
 - Restructuring the class data (if necessary),
 - Implementation of methods, i.e., internal data structures and algorithms,
 - Implementation of control, and
 - Implementation of associations.

(source: http://www.tutorialspoint.com/object_oriented_analysis_design/ooad_object_oriented_paradigm.html)

An example: a **class Date**

```
#include ...
...
class Date
{
public: // access specifier; users can only access the PUBLIC members
    Date(); // constructor; constructors have the name of the class
    Date(unsigned int y, unsigned int m,unsigned int d);
    Date(string yearMonthDay); // constructors can be overloaded
    void setYear(unsigned int y) ; // member function OR method
    void setMonth(unsigned int m) ;
    void setDay(unsigned int d) ;
    void setDate(unsigned int y, unsigned int m, unsigned int d) ;
    unsigned int getYear() ;
    unsigned int getMonth() ;
    unsigned int getDay() ;
    string getStr(); // get (return) date as a string
    void show();
private: // PRIVATE data & function members are hidden from the user
    unsigned int year; // data member
    unsigned int month;
    unsigned int day;
    // the date could have been represented internally as a string
    // the internal representation is hidden from the user
}; // NOTE THE SEMICOLON

Date::Date() // constructors do not have a return type
{
// ... CONSTRUCTOR DEFINITION
}

Date::Date(unsigned int y, unsigned int m, unsigned int d)
{
    year = y;
    month = m;
    day = d;
}
//... DEFINITION OF OTHER MEMBER FUNCTIONS

void Date::show() //scope resolution is needed; other classes could have a show() method
{
// ...
}

int main()
{
    Date d1;
    Date d2(2011,03,18);
    Date d3("2011/03/18");

    d2.setDay(19);

    d2.show();

    string d2_str = d2.getStr();
    cout << d2_str << endl;
}
```

Classes in C++

- A class is a user-defined type.
- A class declaration specifies
 - the representation of objects of the class
 - and the set of operations that can be applied to such objects.

A *class* comprises:

- ***data members*** (or ***fields***) :
each object of the class has its own copy of the data members (local state)
- ***member functions*** (or ***methods***):
applicable to objects of the class

Data members

- describe the ***state*** of the objects
- they have a type, and are declared as:
`type dataMemberId`

Member functions

- denote a ***service*** that objects offer to their clients
- the ***interface*** to such a service is specified by
 - its return type
 - and formal parameter(s):
`returnType memberFuncId(formalParams)`
- In particular, a function with ***void*** return type
usually indicates a function which modifies/shows the state of the object.

Access specifier

- a class may have several ***private*** and ***public*** sections
- keyword ***public*** marks the beginning of each public section
- keyword ***private*** marks the beginning of each private section
- by default, members (data and functions) are ***private***
- normally, the ***data members*** are placed in ***private*** section(s)
and the ***function members*** in ***public*** section(s)
- public members can be accessed by both member and nonmember functions

Constructor

- ***special function*** that is a member of the class and has the ***same name as the class***
- ***does not have a return type***
- ***is automatically called*** when an object of that class is created

```

/*
CLASSES
Fraction class (partial implementation)
TO DO:
- implement other arithmetic operations
*/

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

using namespace std;

class Fraction
{
public: //access-specifier
    Fraction(); // default constructor; constructors have the same name of the class
    Fraction(int num, int denom); // constructor overloading; parameterized constructor
    // ~Fraction(); // destructor (sometimes not necessary, as in this case)
    void read();
    void setNumerator(int num); // member function OR class method
    void setDenominator(int num); // mutator function
    int getNumerator() const; //const member functions can't modify the object that invokes it
    int getDenominator() const; // accessor function
    bool isValid() const;
    void setValid(bool v);
    void show() const;
    void showAll() const;
    Fraction multiply(const Fraction &f);
    // Fraction divide(const Fraction &f);
    // Fraction sum(const Fraction &f);
    // Fraction subtract(const Fraction &f);
    void reduce();
private: //access-specifier
    int numerator; // data member OR attribute
    int denominator;
    bool valid; // fractions with denominator = 0 or that
                // were not read in the format "n/d" are considered invalid !!!
    int gcd(int x, int y) const; // can only be invoked inside class methods
}; // NOTE THE SEMICOLON

// -----
// MEMBER FUNCTIONS DEFINITIONS
// -----


// Constructs a fraction with numerator=0 and denominator=1
// Constructors DO NOT HAVE A RETURN TYPE
Fraction::Fraction() // :: is named the scope resolution operator
{
    numerator = 0;
    denominator = 1;
    valid = true;
}

```

```

// Constructs a fraction with numerator=num and denominator=denom
Fraction::Fraction(int num, int denom)
{
    numerator = num;
    denominator = denom;
    valid = (denominator != 0);
}
//-
/* //UNCOMMENT AND INTERPRET WHAT HAPPENS
Fraction::~Fraction()
{
    cout << "fraction destroyed" << endl;
}
*/
//-
// Reads a fraction; fraction must have format 'numerator' / 'denominator'
void Fraction::read()
{
    string fractionString;
    char fracSymbol;
    int num;
    int denom;

    cout << "n / d ? "; // should not be done here ... TO DO BY STUDENTS: modify
    getline(cin,fractionString);

    istringstream fractionStrStream(fractionString);
    valid = false;
    if (fractionStrStream >> num >> fracSymbol >> denom)
    {
        numerator = num;
        denominator = denom;
        valid = (fracSymbol == '/' && denom !=0);
    }
}
//-
// Set fraction numerator to 'n'
void Fraction::setNumerator(int n)
{
    numerator = n;
}
//-
// Set fraction denominator to 'n'
void Fraction::setDenominator(int n)
{
    denominator = n;
    valid = (denominator != 0);
}
//-
// Set the valid fraction information
void Fraction::setValid(bool v)
{
    valid = v;
}
//-
// Returns the fraction numerator
int Fraction::getNumerator() const
{
    return numerator;
}

```

```

//-
// Returns the fraction denominator
int Fraction::getDenominator() const
{
    return denominator;
}
//-
// Returns the valid fraction information
bool Fraction::isValid() const
{
    return valid;
}
//-
// Multiply current fraction by fraction 'f'
Fraction Fraction::multiply(const Fraction &f)
{
    Fraction result;

    result.setNumerator(numerator * f.getNumerator());
    result.setDenominator(denominator * f.getDenominator());
    result.setValid(valid && f.isValid());

    result.reduce();

    return result;
}
//-
void Fraction::reduce()
{
    if (valid)
    {
        int n = gcd(numerator,denominator);
        numerator = numerator / n;
        denominator = denominator / n;
    }
}
//-
// Show fraction; format is 'numerator / denominator'
void Fraction::show() const
{
    cout << numerator << "/" << denominator;
}
//-
// Show fraction; format is 'numerator / denominator' followed by
// 'valid/invalid' information
void Fraction::showAll() const
{
    show();
    cout << (valid ? " valid" : " invalid") << endl << endl;
    // ?: is a C operator, named conditional operator (it is a ternary operator)
}
//-

```

```

// Compute greatest common divisor between 'x' and 'y'
// using Euclid's algorithm
int Fraction::gcd(int x, int y) const
{
    x = abs(x);  y=abs(y); // for dealing with negative numbers
    if (valid)
    {
        while (x != y)
        {
            if (x < y)
                y = y - x;
            else
                x = x - y;
        }
        return x;
    }
    else
        return 0; // impossible to calculate gcd
}
//-----
// Defines and reads several fractions
// and executes some multiplication operations with them
int main()
{
    Fraction f1, f2(2,5), f3(5,7), f4(0,0), f5(1,0), f6, f7, f8;

    cout << "f1" << endl;
    f1.show(); cout << endl; //later on we'll see how can we do: cout << f1; ◎
    f1.showAll();

    // f1.Fraction(1,2); // can't invoke constructor on existing object
    f1 = Fraction(1,2); // ... but can do this :-) EXPLICIT CONSTRUCTOR CALL
    cout << "f1 new" << endl;
    f1.show(); cout << endl;
    f1.showAll();

    cout << "f2" << endl;
    f2.show(); cout << endl;
    f2.showAll();

    cout << "f3" << endl;
    f3.show(); cout << endl;
    f3.showAll();

    cout << "f4" << endl;
    f4.show(); cout << endl;
    f4.showAll();

    cout << "f5" << endl;
    f5.show(); cout << endl;
    f5.showAll();

    cout << "f6 = f2 * f3" << endl;
    f6 = f2.multiply(f3); // assignment is defined for objects; comparison (==) is not
    f6.show(); cout << endl;
    f6.showAll();

    f6.reduce();
    cout << "f6 reduced" << endl;
    f6.show(); cout << endl;
    f6.showAll();
}

```

```

cout << "f7 = f2 * f4" << endl;
f7 = f2.multiply(f4);
f7.show(); cout << endl;
f7.showAll();

cout << "f8 = ";
f8.read();
cout << "f8" << endl;
f8.show(); cout << endl;
f8.showAll();

cout << "f8 = f6 * f8" << endl;
f8 = f6.multiply(f8);
f8.show(); cout << endl;
f8.showAll();

return 0;
}

```

NOTES:

- **INCLUDE A DEFAULT CONSTRUCTOR IN YOUR CLASSES
SPECIALLY WHEN YOU DO CONSTRUCTOR OVERLOADING**
 - If you define no constructor the compiler will define a default constructor that does nothing
 - But if you only define a constructor with arguments, ex:
`Fraction(int num, int denom);`
 no default constructor will be defined by the compiler;
 so, the following declaration will be illegal
`Fraction f1;`
- **... UNLESS YOU DON'T WANT TO HAVE A DEFAULT CONSTRUCTOR**
 - Ex: what should a default constructor for the `Date` class do ...?
- To call a constructor without arguments do
`this → Fraction f1;`
not this → Fraction f1();
- A constructor behaves like a function that returns an object of its class type. That is what happens when you do
`f1 = Fraction(3,5);`

```
/*
Classes
```

```
Fraction class (partial implementation)
```

SOLUTION SIMILAR TO THE PREVIOUS ONE BUT USING the **this** POINTER

NOTE that by using **this->**
parameters can have the same name as the data members of the class
(not particularly useful...)

TO DO:

- implement reduceFraction
- implement other arithmetic operations

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

using namespace std;

class Fraction
{
public:
    Fraction(); // default constructor
    Fraction(int numerator, int denominator); // alternative constructor
    void read();
    void setNumerator(int numerator);
    void setDenominator(int numerator);
    int getNumerator() const;
    int getDenominator() const;
    bool isValid() const;
    void setValid(bool v);
    void show() const;
    void showAll() const;
    Fraction multiply(const Fraction &f);
    // Fraction divide(const Fraction &f);
    // Fraction sum(const Fraction &f);
    // Fraction subtract(const Fraction &f);
    // Fraction subtract(const Fraction &f);
    void reduce();
private:
    int numerator;
    int denominator;
    bool valid;
    int gcd(int x, int y) const;
};

//-----
// Constructs a fraction with numerator=0 and denominator=1
Fraction::Fraction()
{
    numerator = 0;
    denominator = 1;
    valid = true;
}
```

```

// Constructs a fraction with numerator and denominator equal to the
parameter values
Fraction::Fraction(int numerator, int denominator)
{
    this->numerator = numerator; // when member data & parameters
    this->denominator = denominator; // have the same name(s)
    this->valid = (denominator != 0);
}
//-----
// Reads a fraction; fraction must have format 'numerator' / 'denominator'
void Fraction::read()
{
    string fractionString;
    char fracSymbol;
    int numerator;
    int denominator;

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

    istringstream fractionStrStream(fractionString);
    this->valid = false;
    if (fractionStrStream >> num >> fracSymbol >> denom)
    {
        this->numerator = numerator;
        this->denominator = denominator;
        this->valid = (fracSymbol == '/') && denom !=0);
    }
}
//-----
// Set fraction numerator to 'numerator' value
void Fraction::setNumerator(int numerator)
{
    this->numerator = numerator;
}
//-----
// Set fraction denominator to 'denominator' value
void Fraction::setDenominator(int denominator)
{
    this->denominator = denominator;
    valid = (denominator != 0);
}
//-----
// Set the valid fraction information
void Fraction::setValid(bool valid)
{
    this->valid = valid;
}
//-----
// Returns the fraction numerator
int Fraction::getNumerator() const
{
    return numerator;
}
//-----
// Returns the fraction denominator
int Fraction::getDenominator() const
{
    return denominator;
}

```

```

//-
// Returns the valid fraction information
bool Fraction::isValid() const
{
    return valid;
}
//-
// Multiply current fraction by fraction 'f'
Fraction Fraction::multiply(const Fraction &f)
{
    Fraction result;

    result.setNumerator(this->numerator * f.getNumerator());
    result.setDenominator(this->denominator * f.getDenominator());
    result.setValid(valid && f.isValid());

    return result;
}
//-
void Fraction::reduce()
{
    if (valid)
    {
        int n = gcd(numerator, denominator);
        numerator = numerator / n;
        denominator = denominator / n;
    }
}
//-
// Show fraction; format is 'numerator / denominator'
void Fraction::show() const
{
    cout << numerator << "/" << denominator;
}
//-
// Show fraction; format is 'numerator / denominator' followed by
// 'valid/invalid' information
void Fraction::showAll() const
{
    show();
    cout << (valid ? " valid" : " invalid") << endl << endl;
}
//-
// Compute greatest common divisor between 'x' and 'y'- Euclid's algorithm
int Fraction::gcd(int x, int y) const
{
    x = abs(x);  y=abs(y);
    if (valid)
    {
        while (x != y)
        {
            if (x < y)
                y = y - x;
            else
                x = x - y;
        }
        return x;
    }
    else
        return 0; // impossible to calculate gcd
}

```

```

//-----
// Defines and reads several fractions
// and executes some multiplication operations with them
int main()
{
    Fraction f1, f2(2,5), f3(5,7), f4(0,0), f5(1,0), f6, f7, f8;

    cout << "f1" << endl;
    f1.show(); cout << endl;
    f1.showAll();

    cout << "f2" << endl;
    f2.show(); cout << endl;
    f2.showAll();

    cout << "f3" << endl;
    f3.show(); cout << endl;
    f3.showAll();

    cout << "f4" << endl;
    f4.show(); cout << endl;
    f4.showAll();

    cout << "f5" << endl;
    f5.show(); cout << endl;
    f5.showAll();

    cout << "f6 = f2 * f3" << endl;
    f6 = f2.multiply(f3);
    f6.show(); cout << endl;
    f6.showAll();

    f6.reduce();
    cout << "f6 reduced" << endl;
    f6.show(); cout << endl;
    f6.showAll();

    cout << "f7 = f2 * f4" << endl;
    f7 = f2.multiply(f4);
    f7.show(); cout << endl;
    f7.showAll();

    cout << "f8 - ";
    f8.read();
    cout << "f8" << endl;
    f8.show(); cout << endl;
    f8.showAll();

    cout << "f8 = f6 * f8" << endl;
    f8 = f6.multiply(f8);
    f8.show(); cout << endl;
    f8.showAll();

    return 0;
}

```

```

/*
Application for library management

class Book and Library - preliminary definition and implementation
class User - not yet defined

Using static class attributes
*/

#include <iostream>
#include <string>
#include <vector>
#include <cstddef>

using namespace std;

typedef unsigned long IdentNum;

//-----

class Book
{
public:
    Book(); //default constructor
    Book(string bookName); //another constructor
    void setName(string bookName);
    IdentNum getId() const;
    string getName() const;
    void show() const;
private:
    static IdentNum numBooks; //static => only one copy for all objects
                            // no storage is allocated for numBooks
                            // numBooks must be defined outside the class
    IdentNum id; // each object has data members id and name
    string name;
};

//-----

class Library
{
public:
    Library(); //only the default constructor is declared
    void addBook(Book book);
    void showBooks() const;
private:
    vector<Book> books;
};

```

```

//-
// CLASS Book - MEMBER FUNCTIONS IMPLEMENTATION
//-

IdentNum Book::numBooks = 0; //static variable definition and initialization
//-

Book::Book()
{
    numBooks++;
    id = numBooks;
    name = "UNKNOWN BOOK NAME";
}

//-

Book::Book(string bookName)
{
    numBooks++;
    id = numBooks;
    name = bookName;
}

//-

IdentNum Book::getId() const
{
    return id;
}

//-

void Book::setName(string bookName)
{
    name = bookName;
}

//-

string Book::getName() const
{
    return name;
}

```

```

//-
// CLASS Library - MEMBER FUNCTIONS IMPLEMENTATION
//-

Library::Library()
{
    books.clear(); // clear() is a method from vector class
}

//-
void Library::addBook(Book b)
{
    books.push_back(b);
}

//-
void Library::showBooks() const
{
    for (size_t i=0; i<books.size(); i++)
        cout << books[i].getId() << " - " << books[i].getName() << endl;
}

//-
//-
int main()
{
    Library lib;

    Book b1; // which constructor is used in each case ?
    Book b2("My First C++ Book");

    lib.addBook(b1);
    lib.addBook(b2);

    Book b3;

    string bookName;
    cout << "Book name ? ";
    getline(cin, bookName);
    b3.setName(bookName);

    lib.addBook(b3);
    lib.showBooks();
}

```

- what happens to the books when the application ends?

```

/*
Application for library management
class Book and Library - preliminary definition and implementation
class User - not yet defined
Using static attributes and methods in class declaration
Saving library books in a file
*/
#include <iostream>
#include <string>
#include <vector>
#include <cstddef>
#include <fstream>
#include <sstream>
using namespace std;

//-----
// AUXILIARY TYPES - DEFINITION
//-----

typedef unsigned long IdentNum;

//-----
// CLASS Book - DEFINITION
//-----
class Book
{
public:
    Book(); // default constructor
    Book(string bookName); //another constructor
    void setId(IdentNum num);
    void setName(string bookName);
    IdentNum getId() const;
    string getName() const;
    void show() const;
    static void setNumBooks(IdentNum n); //static method
    static IdentNum getNumBooks();
    // NOTE: can't be "static IdentNum getNumBooks() const;"  

    // static methods can only refer other static members of the class
private:
    static IdentNum numBooks; //static attribute declaration  

                            //static => only one copy for all objects  

                            // no storage is allocated for numBooks  

                            // numBooks must be defined outside the class
    IdentNum id;
    string name;
};

//-----
// CLASS Library - DEFINITION
//-----
class Library
{
public:
    Library();
    void addBook(Book book);
    void showBooks() const;
    void saveBooks(string filename);
    void loadBooks(string filename);
private:
    vector<Book> books;
};

```

```

//-----
// UTILITARY FUNCTIONS
// Note: since C++11 there are function for converting numbers <-> strings
//       (see previous notes)
//-----

int string_to_int (string intStr)
{
    int n;
    istringstream intStream(intStr);
    intStream >> n;
    return n;
}

//-----

/*
string int_to_string(int n)
{
    ostringstream outstr;
    outstr << n;
    return outstr.str();
}
*/
//-----

// CLASS Book - STATIC ATTRIBUTE DEFINITION AND INITIALIZATION
//-----

IdentNum Book::numBooks = 0;
// static variables MUST BE DEFINED (space is reserved), outside the class body;
// in this case, initialization is optional; by default, integers are initialized to zero

//-----
// CLASS Book - IMPLEMENTATION
//-----
```

Book::Book()

```

{
    // suggestion: do not increment numBooks in this case
    // useful for instantiating temporary books
    id = 0;
    name = "VOID"; // OR "" OR "UNKNOWN" ...
}
```

Book::Book(string bookName)

```

{
    numBooks++;
    id = numBooks;
    name = bookName;
}
```

IdentNum Book::getId() const

```

{
    return id;
}
```

```

//-----
string Book::getName() const
{
    return name;
}

//-----

IdentNum Book::getNumBooks() // NOTE: not "static IdentNum Book::getNumBooks()"
{
    return numBooks;
}

//-----

void Book::setNumBooks(IdentNum n) // NOTE: not "static void Book::setNumBooks(IdentNum n)"
{
    numBooks = n;
}

//-----

void Book::setId(IdentNum num)
{
    id = num;
}

//-----

void Book::setName(string bookName)
{
    name = bookName;
}

//-----

void Book::show() const
{
    cout << id << " - " << name << endl;
}

//----- CLASS Library - IMPLEMENTATION -----
//-----
```

```

Library::Library()
{
    books.clear();
}

//-----

void Library::addBook(Book b)
{
    books.push_back(b);
}
```

```

void Library::showBooks() const
{
    cout << "\n-----BOOKS-----\n";
    for (size_t i=0; i<books.size(); i++)
        cout << books[i].getId() << " - " << books[i].getName() << endl;
    cout << "\n-----\n";
}

//-----

void Library::saveBooks(string filename)
{
    ofstream fout;
    fout.open(filename);
    if (fout.fail())
    {
        cout << "Output file opening failed.\n";
        exit(1);
    }

    fout << Book::getNumBooks() << " (last book ID)" << endl << endl;

    for (size_t i=0; i<books.size(); i++)
    {
        fout << books[i].getId() << endl;
        fout << books[i].getName() << endl << endl;
    }

    cout << books.size() << " books saved in file " << filename << endl;
    fout.close();
}

void Library::loadBooks(string filename)
{
    ifstream fin;
    IdentNum numBooks;
    string bookIdStr;
    string emptyLine;
    //IdentNum bookId;
    string bookName;

    // a static method may be called independent of any object,
    // by using the class name and the scope resolution operator
    // but may also be called in connection with an object (see end of main() function)
    Book::setNumBooks(0);

    books.clear();

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

```

```

fin >> numBooks; fin.ignore(100, '\n');
getline(fin, emptyLine);
cout << "'numBooks' obtained from file " << filename << ":" <<
numBooks << endl;

for (size_t i=0; i<numBooks; i++)
{
    getline(fin, bookIdStr); //NOTE: compare with Library::saveBooks()
//bookId = string_to_int(bookIdStr);
    getline(fin, bookName);
    getline(fin, emptyLine);

    Book b(bookName);
    books.push_back(b);
}

cout << books.size() << " books loaded from file " << filename <<
endl;
fin.close();
}

-----
-----

int main()
{
    Library lib;

    Book b1("My First C++ Book");
    Book b2("My Second C++ Book");
    lib.addBook(b1);
    lib.addBook(b2);
    cout << "2 books added to the library\n";

    lib.showBooks();

    lib.saveBooks("bookfile.txt");

    lib.loadBooks("bookfile.txt");

    Book b3("Big C++");
    lib.addBook(b3);
    cout << "1 book added to the library\n";

    lib.showBooks();
    //cout << "numBooks = " << b1.getNumBooks() << endl; // b1.getNumBooks() is a valid call
}

```

```
/*
Application for library management
class Book and Library - preliminary definition and implementation
class User - not yet defined
```

NOTE:
THESE ARE JUST EXAMPLES OF CLASS USE,
NOT THE WAY YOU SHOULD IMPLEMENT A LIBRARY PROJECT

Using two different constructors for Library class
Using a destructor - NOT USUALLY USED FOR THE ILLUSTRATED PURPOSE
*/

```
#include <iostream>
#include <string>
#include <vector>
#include <cstddef>
#include <fstream>
#include <sstream>

using namespace std;

//-
// AUXILIARY TYPES - DEFINITION
//-

typedef unsigned long IdentNum;

//-
// CLASS Book - DEFINITION
//-

class Book
{
public:
    Book();
    Book(string bookName);
    IdentNum getId() const;
    string getName() const;
    void setId(IdentNum num);
    void setName(string bookName);
    void show() const;
    static IdentNum getNumBooks();
    static void setNumBooks(IdentNum n);
private:
    static IdentNum numBooks;
    IdentNum id;
    string name;
};
```

```

//-
// CLASS Library - DEFINITION
//-

class Library
{
public:
    Library();           //default constructor
    Library(string filename); //another constructor
    ~Library();          //destructor
    void addBook(Book book);
    void showBooks() const;
    void saveBooks(string filename);
    void loadBooks(string filename);
private:
    string libraryFilename;
    vector<Book> books;
};

//-
// UTILITARY FUNCTIONS
//-

int string_to_int (string intStr)
{
    int n;
    istringstream intStream(intStr);
    intStream >> n;
    return n;
}

bool fileExists(string filename)
{
    ifstream fin(filename);
    if (fin.is_open())
    {
        fin.close();
        return true;
    }
    else
        return false;
}

//-
// CLASS Book - STATIC ATTRIBUTE INITIALIZATION
//-

```

IdentNum Book::numBooks = 0;

```

//-
// CLASS Book - IMPLEMENTATION
//-

Book::Book()
{
    id = 0;
    name = "VOID";
}
//-

Book::Book(string bookName)
{
    numBooks++;
    id = numBooks;
    name = bookName;
}
//-

IdentNum Book::getId() const
{
    return id;
}

//-

string Book::getName() const
{
    return name;
}
//-

IdentNum Book::getNumBooks() // NOTE: not "static IdentNum Book::getNumBooks()"
{
    return numBooks;
}
//-

void Book::setId(IdentNum num)
{
    id = num;
}
//-

void Book::setName(string bookName)
{
    name = bookName;
}
//-

void Book::show() const
{
    cout << id << " - " << name << endl;
}
//-

void Book::setNumBooks(IdentNum n) // NOTE: not "static void Book::setNumBooks(IdentNum n)"
{
    numBooks = n;
}

```

```

//-
// CLASS Library - IMPLEMENTATION
//-

Library::Library()
{
    books.clear();
}

//-
Library::Library(string filename)
{
    libraryFilename = filename;

    if (FileExists(libraryFilename))
        loadBooks(libraryFilename);
    else
        books.clear();

}

//-
Library::~Library() // DESTRUCTOR - not commonly used for this purpose ...
{
    // cout << "Library destructor was called.\n";
    saveBooks(libraryFilename);
}

//-
void Library::addBook(Book b)
{
    books.push_back(b);
}

//-
void Library::showBooks() const
{
    cout << "\n-----BOOKS-----\n";
    for (size_t i=0; i<books.size(); i++)
        books[i].show();
    cout << "\n-----\n\n";
}

//-
void Library::saveBooks(string filename)
{
    ofstream fout;
    fout.open(filename);
    if (fout.fail())
    {
        cout << "Output file opening failed.\n";
        exit(1);
    }
}

```

```

fout << Book::getNumBooks() << " (last book ID)" << endl << endl;
for (size_t i=0; i<books.size(); i++)
{
    fout << books[i].getId() << endl;
    fout << books[i].getName() << endl << endl;
}
cout << books.size() << " books saved in file " << filename << endl;
fout.close();
}

//-----
void Library::loadBooks(string filename)
{
    ifstream fin;
    IdentNum nBooks;
    string bookIdStr;
    string emptyLine;
    IdentNum bookId;
    string bookName;
    books.clear();

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

    fin >> nBooks; fin.ignore(100, '\n');
    getline(fin,emptyLine);
    cout << "'numBooks' obtained from file " << filename << ":" << endl;
    << nBooks << endl;

    Book::setNumBooks(nBooks);
    // a static method may be called independently of any object,
    // by using the class name and the scope resolution operator

    for (size_t i=0; i<nBooks; i++)
    {
        getline(fin,bookIdStr);
        bookId = string_to_int(bookIdStr);
        getline(fin,bookName);
        getline(fin,emptyLine);

        Book b;
        b.setId(bookId);
        b.setName(bookName);
        books.push_back(b);
    }

    cout << books.size() << " books loaded from file " << filename << endl;
    fin.close();
}

```

```

//-
// main()
//-
int main()
{
    //books are loaded by Library constructor
    Library lib("bookfile.txt");

    Book b1("My First C++ Book");
    Book b2("My Second C++ Book");
    lib.addBook(b1);
    lib.addBook(b2);
    cout << "2 books added to the Library\n";

    lib.showBooks();

    Book b3("Accelerated C++");
    lib.addBook(b3);
    cout << "1 book added to the library\n";

    lib.showBooks();

    //books are saved by Library destructor !!!
}

```

In Library class, an alternative implementation could define:
`vector<*Book> books;`

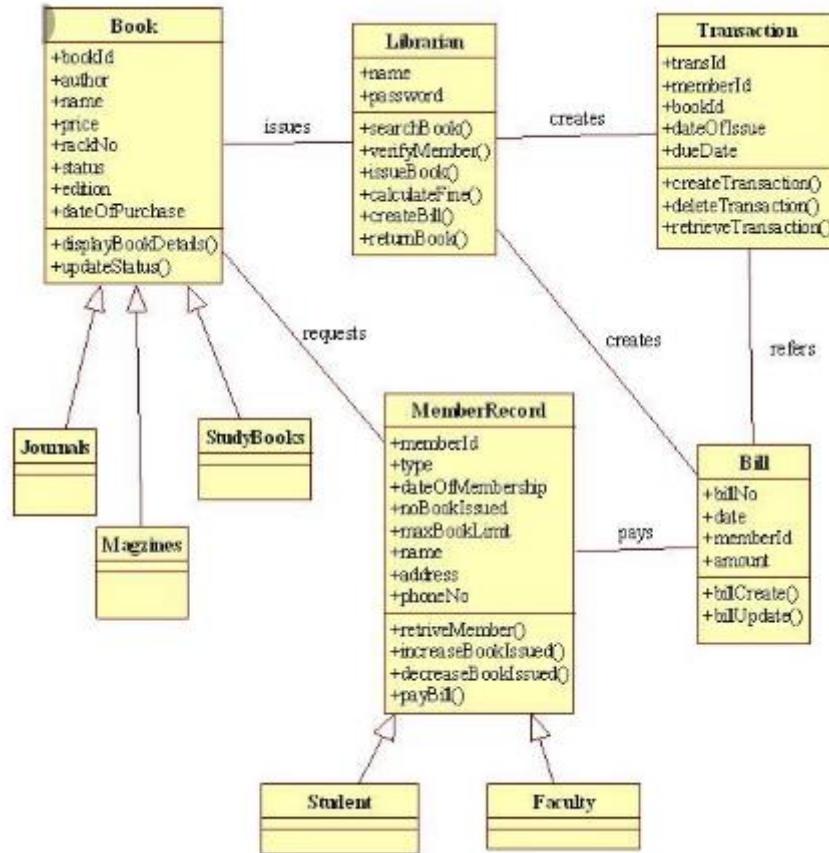
Do you see any advantage / disadvantage ?

Think what happens when you add a User class.

DESTRUCTORS:

- The name of a destructor is a `~Name_of_the_Class`
- A destructor is a member function of a class that is called automatically when an object of the class goes out of scope
- This means that if an object of the class type is a local variable for a function, then the destructor is automatically called as the last action before the function call ends.
- Destructors are used to eliminate any dynamic variables that have been created by the object, so that the memory occupied by these dynamic variables is returned to the freestore.
- Destructors may perform other cleanup tasks as well.

Class diagram for a library management program (an example)



- **Journal**, **Magazine** and **StudyBook** are classes derived from class **Book**
- **Student** and **Faculty** are classes derived from class **MemberRecord** (could have been named **User** ...)
 - Derived classes will be introduced later

Separate compilation & Abstract Data Types (ADTs)

(not for evaluation in 2021/2022)

Until now ... small programs

- code placed into a single file
- typical layout
 - initial comments – what is the program purpose
 - included header files
 - constants
 - typedef's and classes
 - function prototypes (if any)
 - global variables (if any)
 - function / class implementation (+ comments)

When programs get larger or you work in a team ...

- need to separate code into separate source files
- reasons for separating code
 - only those files that you changed need to be recompiled
 - each programmer is solely responsible for a separate set of files
(editing of common files is avoided)

C++ allows you to divide a program into parts

- each part can be stored into a separate file
- each part can be compiled separately
- a class definition can be stored separately from a program
- this allows you to use the class in multiple programs

Header files (interface)

- files that define types or functions that are needed in other files
- are a path of communication between the code
- contain
 - definitions of constants
 - definitions of types / classes
 - declarations of non-member functions
 - declarations of global variables

Implementation files

- contain
 - definitions of member functions
 - definitions of nonmember functions
 - definitions of global variables

Abstract Data Types (ADTs)

- An ADT is a class defined to separate the interface and the implementation
- All member variables are private
- The class definition along with the function and operator declarations are grouped together as the interface of the ADT
- Group the implementation of the operations together and make them available to the programmer using the ADT
- The public part of the class definition is part of the ADT interface
- The private part of the class definition is part of the ADT implementation
 - This hides it from those using the ADT
- C++ does not allow splitting the public and private parts of the class definition across files
- The entire class definition is usually in the interface file

Example: a Book ADT interface

- The Book ADT interface is stored in a file named book.h
- The .h suffix means this is a header file
- Interface files are always header files
- A program using book.h must include it using an include directive
 - #include "book.h"

#include < > OR #include " " ?

- To include a predefined header file use < >
 - #include <iostream>
- < > tells the compiler to look where the system stores predefined header files
- To include a header file you wrote use "....."
 - #include "book.h"
- " " usually causes the compiler to look in the current directory for the header file

The Implementation File

- Contains the definitions of the ADT functions
- Usually has the same name as the header file but a different suffix
- Since our header file is named book.h,
the implementation file is named book.cpp
- The implementation file requires an include directive to include the interface file:
 - #include "book.h"

The Application File

- The application file is the file that contains the program that uses the ADT
 - It is also called a driver file
 - Must use an include directive to include the interface file:
 - #include "book.h"

Running The Program

- Basic steps required to run a program:
(details vary from system to system)
 - Compile the implementation file
 - Compile the application file
 - Link the files to create an executable program using a utility called a linker
 - Linking is often done automatically

Compile book.h ?

- The interface file is not compiled separately
 - The preprocessor replaces any occurrence of `#include "book.h"` with the text of book.h before compiling
 - Both the implementation file and the application file contain `#include "book.h"`
 - The text of book.h is seen by the compiler in each of these files
 - There is no need to compile book.h separately

Why Three Files?

- Using separate files permits
 - The ADT to be used in other programs without rewriting the definition of the class for each
 - Implementation file to be compiled once even if multiple programs use the ADT
 - Changing the implementation file does not require changing the program using the ADT

Reusable Components

- An ADT coded in separate files can be used over and over
- The reusability of such an ADT class
 - Saves effort since it does not need to be
 - Redesigned
 - Recoded
 - Retested
 - Is likely to result in more reliable components

Multiple Classes

- A program may use several classes
 - Each could be stored in its own interface and implementation files
 - Some files can "include" other files, that include still others
 - It is possible that the same interface file could be included in multiple files
 - C++ does not allow multiple declarations of a class
 - The `#ifndef directive` can be used to prevent multiple declarations of a class

Using `#ifndef` directive

- Consider this code in the interface file

```
#ifndef BOOK_H
#define BOOK_H
// the Book class definition goes here
#endif
```

- To prevent multiple declarations of a class, we can use these directives:
 - `#define BOOK_H`
 - adds BOOK_H to a list indicating BOOK_H has been seen
 - `#ifndef BOOK_H`
 - checks to see if BOOK_H has been defined
 - `#endif`
 - if BOOK_H has been defined, skip to #endif
- The first time a `#include "book.h"` is found, BOOK_H and the class are defined
- The next time a `#include "book.h"` is found, all lines between `#ifndef` and `#endif` are skipped
- **NOTE:**
`#pragma once` is a **non-standard** but widely supported preprocessor directive designed to cause the current source file to be included only once in a single compilation; as it is non-standard (yet) its use is not recommended.

Why `BOOK_H` ?

- `BOOK_H` is the normal convention for creating an identifier to use with `#ifndef`
 - it is the file name in all caps
 - use '`_`' instead of '`.`'
- You may use any other identifier, but will make your code more difficult to read

Defining Libraries

- You can create your own libraries of functions
 - You do not have to define a class to use separate files
 - If you have a collection of functions...
 - Declare them in a header file with their comments
 - Define them in an implementation file
 - Use the library files just as you use your class interface and implementation files

Separate compilation - an example : Proj_sep_comp

```
=====  
// FUNCTIONS.H  
=====  
//-----  
// SOME UTILITARY FUNCTIONS  
//-----  
  
#ifndef FUNCTIONS_H  
#define FUNCTIONS_H  
  
#include <string>  
using namespace std;  
  
//-----  
// Converts string to integer  
// 'intStr' - string representing a valid integer  
int string_to_int (string intStr);  
  
//-----  
// Converts integer to string  
// 'n' - an integer value  
string int_to_string(int n);  
  
//-----  
// Tests if a file exists  
// 'filename' - file whose existence is being tested  
bool fileExists(string filename);  
  
//-----  
// Returns a temporary filename  
string getTmpFilename();  
  
#endif  
  
=====  
// FUNCTIONS.CPP  
=====  
//-----  
// SOME UTILITARY FUNCTIONS  
//-----  
  
#include <string>  
#include <fstream>  
#include <sstream>  
  
#include "functions.h"
```

```

//-
// Converts string to integer
// 'intStr' - string representing a valid integer

int string_to_int (string intStr)
{
    int n;
    istringstream intStream(intStr);
    intStream >> n;
    return n;
}

//-
// Converts integer to string
// 'n' - an integer value

string int_to_string(int n)
{
    ostringstream outstr;
    outstr << n;
    return outstr.str();
}

//-
// Tests if a file exists
// 'filename' - file whose existence is being tested

bool fileExists(string filename)
{
    ifstream fin(filename);
    if (fin.is_open())
    {
        fin.close();
        return true;
    }
    else
        return false;
}

//-
// Returns a temporary filename

string getTmpFilename() //Microsoft compiler specific
{
    char fnameC[L_tmpnam_s];
    errno_t err;
    err = tmpnam_s( fnameC, L_tmpnam_s ); //safe version of tmpnam()
    if (err)
        return "";
    else
    {
        string fname(fnameC);
        // tmpnam_s returns names like "\s52k.", "\s1sc.", ...
        // in the following instruction the '\' and the '.' are removed
        fname = fname.substr(1,fname.length()-2);
        return fname;
    }
}

```

```
//=====
// MAIN.CPP (only used for testing the developed functions)
//=====
```

```
#include <iostream>
#include "functions.h"
using namespace std;
int main(void)
{
    cout << string_to_int ("2011") << endl;
    cout << int_to_string(10) << endl;
    cout << "fileExists(\"book.txt\") = " << fileExists("book.txt") <<
endl;
    cout << "temporary file name = " << getTmpFilename() << endl;
    return 0;
}
```

NOTE:

- To compile the functions without having a `main()` function, in Visual Studio, use Build > Compile (`Ctrl+F7`)

Separate compilation - another example

```
//=====
// DEFS.H  (no DEFS.CPP)
//-----

#ifndef DEF_S_H
#define DEF_S_H

typedef unsigned int IdentNum;

#endif
//=====
//=====

// USER.H
//-----
```

```
#ifndef USER_H
#define USER_H

#include <string>
#include <vector>
#include "defs.h"

using namespace std;

class User {

private:
    static IdentNum numUsers; //total number of users - used to obtain ID of each new user
    IdentNum ID; // unique user identifier (unsigned integer)
    string name; // user name
    bool active; // only active users can request books
    vector<IdentNum> requestedBooks; // books presently loaned to the user

public:
    //constructors
    User();
    User(string name);

    //get methods
    IdentNum getID() const;
    string getName() const;
    bool isActive() const;
    vector<IdentNum> getRequestedBooks() const;

    bool hasBooksRequested() const;

    //set methods
    void setID(IdentNum userID);
    void setName (string userName);
    void setActive(bool status);
    void setRequestedBooks(const vector<IdentNum> &booksRequestedByUser);
    static void setNumUsers(IdentNum num);

    void borrowBook(IdentNum bookID);
    void returnBook(IdentNum bookID);
};

#endif
```

```

//=====

//=====
// BOOK.H
//=====

#ifndef BOOK_H
#define BOOK_H

#include <string>
#include "defs.h"
using namespace std;

class Book {

private:
    static IdentNum numBooks; //total number of books - used to obtain ID of each new book
    IdentNum ID; // unique book identifier (unsigned integer)
    string title; // book title
    string author; // book author OR authors
    unsigned int numAvailable; // number of available items with this title

public:
    //constructors
    Book();
    Book(string bookTitle, string bookAuthor, unsigned int bookQuantity);

    //get methods
    IdentNum getID() const;
    string getTitle() const;
    string getAuthor() const;
    unsigned int getNumAvailable() const;

    //set methods
    void setID(IdentNum bookID);
    void setTitle(string bookTitle);
    void setAuthor(string bookAuthor);
    void setNumAvailable(unsigned int numBooks);
    static void setNumBooks(IdentNum num);

    void addBooks(int bookQuantity);
    void loanBook();
    void returnBook();
};

#endif
//=====

```

```

//=====
// LIBRARY.H
//=====

#ifndef LIBRARY_H
#define LIBRARY_H

#include <string>
#include <vector>

#include "defs.h"
#include "book.h"
#include "user.h"

using namespace std;

class Library {
private:
    vector<User> users; // all users that are registered or were registered in the library
    vector<Book> books; // all books that are registered or were registered in the library
    string filenameUsers; // name of the file where users are saved at the end of each program run
    string filenameBooks; // name of the file where books are saved at the end of each program run

public:
    // constructors
    Library();
    Library(string fileUsers, string fileBooks);

    // get functions
    User getUserByID(IdentNum userID) const;
    Book getBookByID(IdentNum bookID) const;

    // user management
    void addUser(User);

    // book management
    void addBook(Book);

    // loaning management
    void loanBook(IdentNum, IdentNum);
    void returnBook(IdentNum, IdentNum);

    // file management methods
    void loadUsers();
    void loadBooks();
    void saveUsers();
    void saveBooks();

    // information display
    void showUsers() const;
    void showUsers(string str) const;
    void showBooks() const;
    void showBooks(string str) const;
    void showAvailableBooks() const;
};

#endif
//=====

```

```

//=====
// USER.CPP
//-
#include "user.h"

    // to do ...

//=====
// BOOK.CPP
//-
#include "library.h"

    // to do ...

//=====
// LIBRARY.CPP
//-
#include "Library.h"

Library::Library(string fileUsers, string fileBooks)
{
    // to do ...
}

//=====

// MAIN.CPP
//-
#include <iostream>

#include "Library.h"

using namespace std;

int main ()
{
    Library library("users.txt","books.txt");

    do
    {
        //show menu
        cout << "#### Menu ####\n\n";
        cout << "1 - New user\n";
        cout << "2 - New book\n";
        / ...
        cout << "0 - Exit\n";
        // TO DO
        // read user option
        // execute user option
    } while (...);

}
//=====

```

Separate compilation – yet another example

Declaring and defining global variables in multiple source file programs

- First of all, it is important to understand the **difference** between defining a variable and declaring a variable:
 - A variable is **defined** when the compiler allocates the storage for the variable.
 - A variable is **declared** when the compiler is informed that a variable exists (and which is its type); it does not allocate the storage for the variable at that point.
- You may declare a variable multiple times (though once is sufficient); you may only define it once within a given scope.

Using the extern keyword

- Using extern is useful when the program you're building consists of multiple source files linked together, where some of the variables defined, for example, in source file **file1.c** need to be referenced in other source files, such as **file2.c**.
- The best way to declare and define global variables is
 - to use a header file **file3.h** to contain an **extern** declaration of the variable.
 - The header is included by the source file that defines the variable (ex: **file3.cpp**) and by all the source files that reference the variable.
 - For each program, **one source file (and only one source file) defines the variable.** Similarly, **one header file (and only one header file) should declare the variable.**

```

//=====
// aux1.h

#ifndef AUX1_H
#define AUX1_H

int globalVar = 1000;

void f1();

#endif

//=====
// aux1.cpp

#include <iostream>
#include "aux1.h"

using namespace std;

void f1()
{
    cout << "globalVar = " << globalVar << endl;
}

//=====
// main.cpp

#include <iostream>
#include "aux1.h"

using namespace std;

int main()
{
    cout << "Global Var = " << globalVar << endl;
    f1();
}

//=====

```

Error List					
	Description	File	Line	Column	Project
✖ 1	error LNK2005: "int globalVar" (?globalVar@@@3HA) already defined in main.obj	aux1.obj			Extern_01
✖ 2	error LNK1169: one or more multiply defined symbols found	Extern_01.exe	1	1	Extern_01

```
//=====
// aux1.h

#ifndef AUX1_H
#define AUX1_H

extern int globalVar;

void f1();

#endif

//=====
// aux1.cpp

#include <iostream>
#include "aux1.h"

using namespace std;

int globalVar = 1000;

void f1()
{
    cout << "globalVar = " << globalVar << endl;
}

//=====
// main.cpp

#include <iostream>
#include "aux1.h"

using namespace std;

int main()
{
    cout << "Global Var = " << globalVar << endl;
    f1();
}

=====
```

LINKED LISTS (not lectured in 2021/2022)

more on

STRUCTS, POINTERS, DYNAMIC MEMORY ALLOCATION, CLASSES, DESTRUCTORS, ...

/*
POINTERS, STRUCTS & DINAMIC MEMORY ALLOCATION
IMPLEMENTATION OF A LINKED LIST CLASS FOR STORING INT VALUES

An example where a DESTRUCTOR is REQUIRED

see, for example:
http://www.codeproject.com/KB/cpp/linked_list.aspx
uses malloc() and free()
*/

```
// #define NDEBUG // see comment on assert() in LinkedList::clear()
#include <iostream>
#include <cstddef>
#include <cassert>

using namespace std;

class LinkedList{
private:
    struct node{           // node is a TYPE !!! could also have defined a Node class
        int data;          // the elements of the list are integers (only) ...
        node *next;
    } *p;
    size_t listSize;
public:
    LinkedList();
    size_t size() const;
    void insertEnd(int value);
    void insertBegin(int value);
    bool insertAfter(size_t index, int value);
    bool remove(int value);
    void clear();
    void display() const;
    ~LinkedList();
};

//-----// constructor
LinkedList::LinkedList()
{
    p = NULL;
    listSize = 0;
}

//-----// return the list size
size_t LinkedList::size() const
{
    return listSize;
}
```

```

//-----  

//insert a new node at the beginning of the linked list  

void LinkedList::insertBegin(int value)  

{  

    node *q;  

    q = new node;  

    q->data = value; //note: access to a struct field through a struct pointer  

    q->next = p;  

    p = q;  

    listSize++;  

}  

//-----  

// insert a new node at the end of the linked list  

void LinkedList::insertEnd(int value)  

{  

    node *q,*t;  

    //if the list is empty  

    if (p == NULL) //alternative: if (listSize==0)  

    {  

        p = new node;  

        p->data = value;  

        p->next = NULL;  

        // listSize++;  

    }  

    else  

    {  

        q = p;  

        while(q->next != NULL)  

            q = q->next;  

        t = new node;  

        t->data = value;  

        t->next = NULL;  

        q->next = t;  

        // listSize++;  

    }  

    listSize++;  

}  

//-----  

// insert a node at a specified location  

// 'index' - location  

// 'value' - contents of the node  

// return value - indicates if insertion was successful  

bool LinkedList::insertAfter(size_t index, int value)  

{  

    node *q, *t;  

    size_t i;  

    if (index > listSize-1) //if (index > size()-1)  

        return false;  

    else  

    {  

        q = p;  

        for (i = 0; i < index; i++)  

            q = q->next;  

        t = new node;  

        t->data = value;  

        t->next = q->next;  

        q->next = t;  

        listSize++;  

        return true;  

    }
}

```

```

//-----  

// deletes the specified value from the linked list  

// 'value' - contents of the node to be deleted  

// return value - indicates if removal was successful  

bool LinkedList::remove(int value)  

{  

    node *q,*r;  

    q = p;  

    //if node to be deleted is the first node  

    if (q->data == value)  

    {  

        p = p->next;  

        delete q;  

        listSize--;  

        return true;  

    }  

    r = q;  

    while(q != NULL)  

    {  

        if(q->data == value)  

        {  

            r->next = q->next;  

            delete q;  

            listSize--;  

            return true;  

        }  

        r = q;  

        q = q->next;  

    }  

    return false;  

}  

//-----  

// deletes all the list elements  

void LinkedList::clear()  

{  

    node *q;  

    if( p == NULL )  

        return;  

    while( p != NULL )  

    {  

        q = p->next;  

        delete p;  

        listSize--;  

        p = q;  

    }  

    //assert(listsize==0); //define NDEBUG before #include <cassert>  

    //is equivalent to commenting assert's  

}  

//-----  

// shows all the list elements  

void LinkedList::display() const  

{  

    node *q;  

    cout << "(" << listsize << ") : ";  

    for(q = p; q != NULL; q = q->next)  

        cout << " " << q->data;  

    cout << endl << endl;  

}

```

```

//-----  

// destructor  

// MUST BE IMPLEMENTED WHEN DYNAMIC MEMORY WAS ALLOCATED  

// to free all the memory allocated for the List nodes

LinkedList::~LinkedList()
{
    clear(); //see LinkedList::clear()
}

//-----  

void main()
{
    LinkedList list;
    int index;
    int value;

    cout << "insertBegin() 1, 2, 3, 4, 5\n";
    for (value=1; value<=5; value++)
    {
        list.insertBegin(value);
        list.display();
    }

    //list.clear();
    //list.display();

    value = 6;
    cout << "insertEnd() " << value << "\n";
    list.insertEnd(value);
    list.display();

    index = 0; value = 7;
    cout << "insertAfter(" << index << "," << value << ")\n";
    if (!list.insertAfter(index,7))
        cout << "there is no such node index: " << index << endl;
    list.display();

    index = 10; value = 8;
    cout << "insertAfter(" << index << "," << value << ")\n";
    if (!list.insertAfter(index,value))
        cout << "there is no such node index: " << index << endl;
    list.display();

    value = 2;
    cout << "remove(" << value << ")\n";
    if (!list.remove(value))
        cout << "there is no such node value: " << value << endl;
    list.display();

    value = 9;
    cout << "remove(" << value << ")\n";
    if (!list.remove(value))
        cout << "there is no such node value: " << value << endl;
    list.display();

    cout << endl;
}

```

TO DO/BY STUDENTS:
implement method `bool LinkedList::removeNode(size_t index)`
 to remove the node at a specified location, 'index', if it exists

TEMPLATES – GENERIC PROGRAMMING

FUNCTION TEMPLATES / GENERIC FUNCTIONS

```
/*
FUNCTION OVERLOADING (remembering ...)
*/
#include <iostream>
using namespace std;

//-
void swapValues(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}

//-
void swapValues(double &x, double &y)
{
    double temp = x;
    x = y;
    y = temp;
}
//-
void swapValues(char &x, char &y)
{
    char temp = x;
    x = y;
    y = temp;
}

//-
void main()
{
    int i1 = 1, i2 = 2;
    double d1 = 1.5, d2 = 2.5;
    char c1 = 'A', c2 = 'B';

    swapValues(i1,i2);
    swapValues(d1,d2);
    swapValues(c1,c2);

    cout << "i1 = " << i1 << ", i2 = " << i2 << endl;
    cout << "d1 = " << d1 << ", d2 = " << d2 << endl;
    cout << "c1 = " << c1 << ", c2 = " << c2 << endl;
}
```

```
/*
FUNCTION TEMPLATES- example 1 (swapping values)
```

Compare with previous example: function overloading

When the operations are the same for each overloaded function, they can be expressed more compactly and conveniently using function templates

Generic programming involves writing code in a way that is independent of any particular type

```
#include <iostream>
#include <string>

using namespace std;

//-----
template <class T> // OR template <typename T>
void swapValues(T &x, T &y)
{
    T temp = x;
    x = y;
    y = temp;
}

//-----
void main()
{
    int     i1 = 1,     i2 = 2;
    double  d1 = 1.5,   d2 = 2.5;
    char    c1 = 'A',   c2 = 'B';
    string s1="ABC",   s2="DEF";

    // NOTE:
    // the type attached to the template arguments
    // is inferred from the value argument list
    swapValues(i1,i2);
    swapValues(d1,d2);
    swapValues(c1,c2);
    swapValues(s1,s2);

    cout << "i1 = " << i1 << ", i2 = " << i2 << endl;
    cout << "d1 = " << d1 << ", d2 = " << d2 << endl;
    cout << "c1 = " << c1 << ", c2 = " << c2 << endl;
    cout << "s1 = " << s1 << ", s2 = " << s2 << endl;
}
```

```

/*
FUNCTION TEMPLATES: example 2 (printing arrays)
*/

#include <iostream>
#include <cstddef>
#include <string>

using namespace std;

//-----
template <typename T> // OR template <class T> //suggestion: use typename
void printArray(ostream &out, const T data[], size_t count)
{
    out << "[";
    for (size_t i = 0; i < count; i++)
    {
        if (i > 0)
            out << ", ";
        out << data[i];
    }
    out << "]";
}

//-----
void main()
{
    int a[] = {10, 20, 30, 40, 50}; // an example of array initialization

    // call integer function-template specialization
    printArray(cout,a,5);
    cout << endl;

    double b[] = {1.1, 1.2, 1.3};
    // call double function-template specialization
    printArray (cout,b,3);
    cout << endl;

    string c[] = {"Mary", "John", "Fred"};
    // call string function-template specialization
    printArray (cout,c,3);
    cout << endl;
}

```

CLASS TEMPLATES / GENERIC CLASSES (TEMPLATES FOR DATA ABSTRACTION)

```
/*
TEMPLATE CLASSES

IMPLEMENTATION OF A GENERIC "LINKED LIST" CLASS

Compare with previous example: Linked list of integer values
Common solution to develop a Template function/class:
- develop a function/class with a 'fixed' type, then create the Template

*/
#include <iostream>
#include <cstddef>
#include <cassert>

using namespace std;

template <class T> //instead of <class T> could have used <typename T>
class LinkedList {
private:
    struct node{
        T data;
        node *next;
    } *p;
    size_t listSize;
public:
    LinkedList();
    size_t size() const;
    void insertEnd(T value);
    void insertBegin(T value);
    bool insertAfter(size_t index, T value);
    bool remove(T value);
    void clear();
    void display() const,
    ~LinkedList();
};

//-----
// constructor
template <class T> //instead of <class T> could have used <typename T>
LinkedList<T>::LinkedList()
{
    p = NULL;
    listSize = 0;
}

//-----
// return the list size
template <class T>
size_t LinkedList<T>::size() const
{
    return listSize;
}
```

```
-----  
//insert a new node at the beginning of the linked list
```

```
template <class T>  
void LinkedList<T>::insertBegin(T value)  
{  
    node *q;  
    q = new node;  
    q->data = value;  
    q->next = p;  
    p = q;  
    listSize++;  
}
```

```
-----  
// insert a new node at the end of the linked list
```

```
template <class T>  
void LinkedList<T>::insertEnd(T value)  
{
```

```
    node *q, *t;  
    //if the list is empty  
    if(p == NULL)  
    {  
        p = new node;  
        p->data = value;  
        p->next = NULL;  
        listSize++;  
    }  
    else  
    {  
        q = p;  
        while(q->next != NULL)  
            q = q->next;  
        t = new node;  
        t->data = value;  
        t->next = NULL;  
        q->next = t;  
        listSize++;  
    }  
}
```

```
-----  
// insert a node at a specified location
```

```
// 'index' - location
```

```
// 'value' - contents of the node
```

```
template <class T>  
bool LinkedList<T>::insertAfter(size_t index, T value)  
{
```

```
    node *q, *t;  
    size_t i;  
  
    if (index > size()-1)  
        return false;  
    else  
    {  
        q = p;  
        for (i = 0; i < index; i++)  
            q = q->next;  
        t = new node;  
        t->data = value;  
        t->next = q->next;
```

```

        q->next = t;
        listSize++;
    }
    return true;
}

//-----
// deletes the specified node from the linked list
// 'value' - contents of the node to be deleted
template <class T>
bool LinkedList<T>::remove(T value)
{
    node *q,*r;
    q = p;
    //if node to be deleted is the first node
    if (q->data == value)
    {
        p = p->next;
        delete q;
        listSize--;
        return true;
    }
    r = q;
    while(q != NULL)
    {
        if(q->data == value)
        {
            r->next = q->next;
            delete q;
            listSize--;
            return true;
        }
        r = q;
        q = q->next;
    }
    return false;
}

//-----
// deletes all the list elements
template <class T>
void LinkedList<T>::clear()
{
    node *q;
    if( p == NULL )
        return;

    while( p != NULL )
    {
        q = p->next;
        delete p;
        p = q;
        listSize--;
    }
    //assert(listsize==0);
}

```

```

//-
// shows all the linked list elements
template <class T>
void LinkedList<T>::display() const
{
    node *q;

    cout << "(" << listSize << "): ";
    for(q = p; q != NULL; q = q->next)
        cout << " " << q->data;
    cout << endl << endl;
}

//-
// destructor
// MUST BE IMPLEMENTED WHEN DYNAMIC MEMORY HAS BEEN ALLOCATED
// to free all the memory allocated for the list nodes
template <class T>
LinkedList<T>::~LinkedList()
{
    clear();
}

//-
void main()
{
    LinkedList<char> list;
    int index;
    char value;

    cout << "insertBegin() 'A', 'B', 'C', 'D', 'E'\n";
    for (value='A'; value<='E'; value++)
    {
        list.insertBegin(value);
        list.display();
    }

    //list.clear();
    //list.display();

    value = 'F';
    cout << "insertEnd() '" << value << "'\n";
    list.insertEnd(value);
    list.display();

    index = 0; value = 'G';
    cout << "insertAfter(" << index << ", '" << value << "')\n";
    if (!list.insertAfter(index,value))
        cout << "there is no such node index: " << index << endl;
    list.display();

    index = 10; value = 'H';
    cout << "insertAfter(" << index << ", '" << value << "')\n";
    if (!list.insertAfter(index,value))
        cout << "there is no such node index: " << index << endl;
    list.display();
}

```

```
value = 'B';
cout << "remove(" << value << ")\\n";
if (!list.remove(value))
    cout << "there is no such node value: " << value << endl;
list.display();

value = 'J';
cout << "remove(" << value << ")\\n";
if (!list.remove(value))
    cout << "there is no such node value: " << value << endl;
list.display();

cout << endl;
}
```

```

// Class Templates
// Implementing a (circular) queue based on an array
// example: https://en.wikipedia.org/wiki/Circular\_buffer
// JAS

#include <iostream>
#include <cstddef>
#include <string>
#include <sstream>

using namespace std;

template <typename T> // OR template <class T>
// template <typename T = int> // T defaults to 'int' => "Queue <> q;" is possible
class Queue
{
public:
    static const size_t MAXSIZE = 10; //all queues have a max. size of 10; not flexible ...
    Queue();
    bool insertLast(T value); // insert element into the queue
    bool removeFirst(T &value); // remove element from the queue
    size_t getNumElems() const; // get number of queue elements
private:
    T v[MAXSIZE]; // array elements are of type T
    size_t first; // index of first queue element
    size_t last; // index of last queue element
    size_t nElems; // number of elements in queue
};

// ----

template <typename T>
Queue<T>::Queue()
{
    first = 0;
    last = 0;
    nElems = 0;
}

// ----

template <typename T>
bool Queue<T>::insertLast(T value) //returns true if value was inserted
{
    if (nElems == 0)
    {
        v[last] = value;
        nElems = 1;
        return true;
    }
    else if (nElems < MAXSIZE)
    {
        last = (last + 1) % MAXSIZE;
        v[last] = value;
        nElems++;
        return true;
    }
    return false;
}
// -----

```

```

template <typename T>
bool Queue<T>::removeFirst(T &value) //returns true if queue is not empty
{
    if (nElems > 0)
    {
        value = v[first];
        nElems--;
        if (nElems !=0)
            first = (first + 1) % MAXSIZE;
        return true;
    }
    return false;
}

// ----

template <typename T>
size_t Queue<T>::getNumElems() const
{
    return nElems;
}

// ----

int main()
{
    cout << "Max. queue size is " << Queue<string>::MAXSIZE << endl;
    //cout << "Max. queue size is " << Queue< >::MAXSIZE << endl; //possible when
T has a default type, e.g. int; see alternative template, in class Queue definition

//-----
cout << "INTEGER QUEUE:\n";

Queue<int> q;

for (size_t i=1; i<=3; i++)          // try with other numbers of insertions
{                                     // or a different MAXSIZE
    if (q.insertLast(i))
        cout << i << " inserted\n";
    else
        cout << "full\n";
}

for (size_t i=1; i<=5; i++)
{
    int value;
    if (q.removeFirst(value))
        cout << "removed " << value << "\n";
    else
        cout << "empty\n";
}

```

CONTINUES →

```
cout << endl;

cout << "STRING QUEUE:\n";

Queue<string> qs;

for (size_t i=1; i<=5; i++)
{
    string s = "value_" + to_string(n);
    if (qs.insertLast(s))
        cout << s << " inserted\n";
    else
        cout << "full\n";
}

for (size_t i=1; i<=6; i++)
{
    string value;
    if (qs.removeFirst(value))
        cout << "removed " << value << "\n";
    else
        cout << "empty\n";
}

cout << endl;

return 0;
}
```

```

// Class Templates
// Implementing a (circular) queue based on an array
// Notes:
// - the template class has 2 parameters & accepts the size as a parameter
// - the 2nd parameter is a numeric value (defaults to 10), not a type
// JAS

#include <iostream>
#include <cstddef>
#include <string>
#include <sstream>

using namespace std;
// -----
template <typename T, size_t MAXSIZE = 10>
class Queue
{
public:
    Queue();
    bool insertLast(T value);
    bool removeFirst(T &value);
    size_t getNumElems() const;
private:
    T v[MAXSIZE];
    size_t first;
    size_t last;
    size_t nElems;
};

// -----
template <typename T, size_t MAXSIZE>
Queue<T,MAXSIZE>::Queue()
{
    first = 0;
    last = 0;
    nElems = 0;
}

// -----
template <typename T, size_t MAXSIZE>
bool Queue<T,MAXSIZE>::insertLast(T value)
{
    if (nElems == 0)
    {
        v[last] = value;
        nElems = 1;
        return true;
    }
    else if (nElems < MAXSIZE)
    {
        last = (last + 1) % MAXSIZE;
        v[last] = value;
        nElems++;
        return true;
    }
    return false;
}
// -----

```

```

template <typename T, size_t MAXSIZE>
bool Queue<T,MAXSIZE>::removeFirst(T &value)
{
    if (nElems > 0)
    {
        value = v[first];
        nElems--;
        if (nElems !=0)
            first = (first + 1) % MAXSIZE;
        return true;
    }
    return false;
}

// ----

template <typename T, size_t MAXSIZE>
size_t Queue<T,MAXSIZE>::getNumElems() const
{
    return nElems;
}

// -----
// Converts integer to string
// 'n' - an integer value

string int_to_string(int n)
{
    ostringstream outstr;
    outstr << n;
    return outstr.str();
}

// ----

int main()
{
    cout << "INTEGER QUEUE:\n";
    Queue<int,5> q;

    int n=1;
    for (size_t i=1; i<=3; i++)
    {
        if (q.insertLast(n))
        {
            cout << n << " inserted\n";
            n++;
        }
        else
            cout << "full\n";
    }

    for (size_t i=1; i<=2; i++)
    {
        int value;
        if (q.removeFirst(value))
            cout << "removed " << value << "\n";
        else
            cout << "empty\n";
    }
}

```

```

for (size_t i=1; i<=5; i++)
{
    if (q.insertLast(n))
    {
        cout << n << " inserted\n";
        n++;
    }
    else
        cout << "full\n";
}

for (size_t i=1; i<=10; i++)
{
    int value;
    if (q.removeFirst(value))
        cout << "removed " << value << "\n";
    else
        cout << "empty\n";
}

//-----
cout << endl;
cout << "STRING QUEUE:\n";
Queue<string,5> qs;

for (size_t i=1; i<=5; i++)
{
    string s = "value_" + int_to_string(i);
    if (qs.insertLast(s))
        cout << s << " inserted\n";
    else
        cout << "full\n";
}

for (size_t i=1; i<=6; i++)
{
    string value;
    if (qs.removeFirst(value))
        cout << "removed " << value << "\n";
    else
        cout << "empty\n";
}

cout << endl;

//-----
cout << "DOUBLE QUEUE:\n";
Queue<double> qd; // NOTE: MAXSIZE defaults to 10

for (size_t i=1; i<=3; i++)
{
    if (qd.insertLast(i/1.2))
        cout << i << " inserted\n";
    else
        cout << "full\n";
}

```

```
for (size_t i=1; i<=5; i++)
{
    double value;
    if (qd.removeFirst(value))
        cout << "removed " << value << "\n";
    else
        cout << "empty\n";
}

cout << endl;

return 0;
}
```

```

// Class Templates
// Implementing a (circular) queue
// using dynamically allocated memory
// JAS

#include <iostream>
#include <cstdlib> // malloc() + free()
#include <cstddef>
#include <string>
#include <sstream>

using namespace std;

template <typename T> // OR template <typename T>
class Queue
{
public:
    Queue();
    Queue(size_t capac);
    ~Queue(); // destructor; must be implemented in this case
               // because memory is allocated dinamically
    bool insertLast(T value); // insert elemento into the queue
    bool removeFirst(T &value); // remove element from the queue
    size_t getNumElems() const; // get number of queue elements
    size_t getCapacity() const; // get queue capacity
private:
    static const size_t MAXSIZE = 10; //all queues have a default size of 10;
    T *v; // pointer to elements of type T
    size_t first; // index of first queue element
    size_t last; // index of last queue element
    size_t nElems; // number of elements in queue
    size_t capacity; // capacity of the queue
};

// ----

template <typename T>
Queue<T>::Queue()
{
    first = 0;
    last = 0;
    nElems = 0;
    v = new T [MAXSIZE]; // v = (T *) malloc (MAXSIZE * sizeof(T));
    capacity = MAXSIZE;
}

// ----

template <typename T>
Queue<T>::Queue(size_t capac)
{
    first = 0;
    last = 0;
    nElems = 0;
    v = new T [capac]; // v = (T *) malloc (capac * sizeof(T));
    capacity = capac;
}

// -----

```

```

template <typename T>
Queue<T>::~Queue()
{
    delete[] v; // free(v);
}

// ----

template <typename T>
bool Queue<T>::insertLast(T value) //returns true if value was inserted
{
    if (nElems == 0)
    {
        v[last] = value;
        nElems = 1;
        return true;
    }
    else if (nElems < capacity)
    {
        last = (last + 1) % capacity;
        v[last] = value;
        nElems++;
        return true;
    }
    return false;
}
// ----

template <typename T>
bool Queue<T>::removeFirst(T &value) //returns true if queue is not empty
{
    if (nElems > 0)
    {
        value = v[first];
        nElems--;
        if (nElems != 0)
            first = (first + 1) % capacity;
        return true;
    }
    return false;
}
// ----

template <typename T>
size_t Queue<T>::getNumElems() const
{
    return nElems;
}

// ----
// Converts integer to string
// 'n' - an integer value
string int_to_string(int n)
{
    ostringstream outstr;
    outstr << n;
    return outstr.str();
}
// ----

```

```

int main()
{
//-----
cout << "INTEGER QUEUE:\n";
Queue<int> q(2);

for (size_t i=1; i<=3; i++)
{
    if (q.insertLast(i))
        cout << i << " inserted\n";
    else
        cout << "full\n";
}

for (size_t i=1; i<=5; i++)
{
    int value;
    if (q.removeFirst(value))
        cout << "removed " << value << "\n";
    else
        cout << "empty\n";
}

//-----
cout << endl;
cout << "DOUBLE QUEUE:\n";
Queue<double> qd;

for (size_t i=1; i<=5; i++)
{
    double s = i * 1.25;
    if (qd.insertLast(s))
        cout << s << " inserted\n";
    else
        cout << "full\n";
}

for (size_t i=1; i<=6; i++)
{
    double value;
    if (qd.removeFirst(value))
        cout << "removed " << value << "\n";
    else
        cout << "empty\n";
}

cout << endl;

//-----
cout << endl;
cout << "STRING QUEUE:\n";
Queue<string> qs(5);

for (size_t i=1; i<=5; i++)
{
    string s = "value_" + int_to_string(i);
    if (qs.insertLast(s))
        cout << s << " inserted\n";
    else
        cout << "full\n";
}

```

```
for (size_t i=1; i<=6; i++)
{
    string value;
    if (qs.removeFirst(value))
        cout << "removed " << value << "\n";
    else
        cout << "empty\n";
}

cout << endl;

return 0;
}
```

TO DO BY STUDENTS:

- implement a circular queue based on a vector

```

/*
TEMPLETE CLASSES
Parameterization with more than one type
*/
#include <iostream>
#include <string>

using namespace std;

template <typename T1, typename T2>
class Pair
{
public:
    Pair(const T1 &f, const T2 &s);
    T1 getFirst() const;
    T2 getSecond() const;
    void show() const;
private:
    T1 first;
    T2 second;
};

//-----
// constructor
template <typename T1, typename T2>
Pair<T1,T2>::Pair(const T1 &f, const T2 &s)
{
    first = f;
    second = s;
}
//-----

template <typename T1, typename T2>
T1 Pair<T1,T2>::getFirst() const
{
    return first;
}
//-----
template <typename T1, typename T2>
T2 Pair<T1,T2>::getSecond() const
{
    return second;
}
//-----

template <typename T1, typename T2>
void Pair<T1,T2>::show() const
{
    cout << first << " - " << second << endl;
}
//-----
void main()
{
    Pair<string,int> p1("John", 19); // John' s grade
    Pair<int,string> p2(1,"F.C.Porto"); // 1st in football rank
    Pair<int,int> p3(2017,365); // Number of days of year

    p1.show();
    p2.show();
    p3.show();
}

```

STL – STANDARD TEMPLATE LIBRARY

- C++ Standard Library
- Standard Template Library (STL)
 - ▶ Recognizing that many data structures and algorithms are commonly used, the C++ standard committee added the Standard Template Library (STL) to the C++ Standard Library.
 - ▶ The STL defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures.
 - ▶ In the previous examples, we built linked lists; object space was allocated dynamically and objects were linked together with pointers.
 - ▶ Pointer-based code is complex, and the slightest omission or oversight can lead to serious memory-access violations and memory-leak errors, with no compiler complaints.
 - ▶ Implementing additional data structures, such as deques, priority queues, sets and maps, requires substantial extra work.
 - ▶ An advantage of the STL is that you can reuse the STL containers, iterators and algorithms to implement common data representations and manipulations.
 - ▶ Programming with the STL will enhance the portability of your code.

STL Containers

- data structures capable of storing objects of almost any data type
- 3 styles of containers
 - first-class containers
 - adapters
 - near containers

STL Iterators

- used by programs to manipulate the STL container elements
- have properties similar to those of pointers
- 5 categories

STL Algorithms

- functions that perform common data manipulation (ex: search, sort, ...)
- ~ 70 algorithms
- each algorithm has minimum requirements for the type of iterators that can be used with it
- a container's supported iterator type determines whether the container can be used with a specific algorithm.

Generic programming

- STL approach allows general program to be written so that the code does not depend on the underlying container

CONTAINERS

- First class containers
 - Sequence containers - represent linear data structures
 - **vector**
 - **deque**
 - **list**
 - *since C++11: array, forward_list (see next pages).*
 - *NOTE: string - supports the same functionality as a sequence container, but stores only character data*
 - Associative containers -
nonlinear containers that typically can locate elements quickly;
can store sets of values or key - value pairs
 - **set**
 - **multiset**
 - **map**
 - **multimap**
 - *since C++11: unordered_set, unordered_map, ... (see next pages).*
- Container adapters
 - **stack**
 - **queue**
 - **priority-queue**
- Near containers (non-STL, but with STL-like characteristics and behaviors)
 - **strings**
 - **valarrays**
 - **...**

Standard library container classes

Standard Library container class	Description
<i>Sequence containers</i>	
vector	Rapid insertions and deletions at back. Direct access to any element.
deque	Rapid insertions and deletions at front or back. Direct access to any element.
list	Doubly linked list, rapid insertion and deletion anywhere.
<i>Associative containers</i>	
set	Rapid lookup, no duplicates allowed.
multiset	Rapid lookup, duplicates allowed.
map	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
multimap	One-to-many mapping, duplicates allowed, rapid key-based lookup.
<i>Container adapters</i>	
stack	Last-in, first-out (LIFO).
queue	First-in, first-out (FIFO).
priority_queue	Highest-priority element is always the first element out.

Common member functions fro most STL containers

Member function	Description
<code>default constructor</code>	A constructor to create an empty container. Normally, each container has several constructors that provide different initialization methods for the container.
<code>copy constructor</code>	A constructor that initializes the container to be a copy of an existing container of the same type.
<code>destructor</code>	Destructor function for cleanup after a container is no longer needed.
<code>empty</code>	Returns <code>true</code> if there are no elements in the container; otherwise, returns <code>false</code> .
<code>insert</code>	Inserts an item in the container.
<code>size</code>	Returns the number of elements currently in the container.
<code>operator=</code>	Assigns one container to another.
<code>operator<</code>	Returns <code>true</code> if the first container is less than the second container; otherwise, returns <code>false</code> .
<code>operator<=</code>	Returns <code>true</code> if the first container is less than or equal to the second container; otherwise, returns <code>false</code> .
<code>operator></code>	Returns <code>true</code> if the first container is greater than the second container; otherwise, returns <code>false</code> .

Member function	Description
<code>operator>=</code>	Returns <code>true</code> if the first container is greater than or equal to the second container; otherwise, returns <code>false</code> .
<code>operator==</code>	Returns <code>true</code> if the first container is equal to the second container; otherwise, returns <code>false</code> .
<code>operator!=</code>	Returns <code>true</code> if the first container is not equal to the second container; otherwise, returns <code>false</code> .
<code>swap</code>	Swaps the elements of two containers.
<i>Functions found only in first-class containers</i>	
<code>max_size</code>	Returns the maximum number of elements for a container.
<code>begin</code>	The two versions of this function return either an <code>iterator</code> or a <code>const_iterator</code> that refers to the first element of the container.
<code>end</code>	The two versions of this function return either an <code>iterator</code> or a <code>const_iterator</code> that refers to the next position after the end of the container.
<code>rbegin</code>	The two versions of this function return either a <code>reverse_iterator</code> or a <code>const_reverse_iterator</code> that refers to the last element of the container.

Member function	Description
<code>rend</code>	The two versions of this function return either a <code>reverse_iterator</code> or a <code>const_reverse_iterator</code> that refers to the next position after the last element of the reversed container.
<code>erase</code>	Erases one or more elements from the container.
<code>clear</code>	Erases all elements from the container.

Standard library container header files

Standard Library container header files

<vector>	
<list>	
<deque>	
<queue>	Contains both queue and priority_queue.
<stack>	
<map>	Contains both map and multimap.
<set>	Contains both set and multiset.
<valarray>	
<bitset>	

New C++11 containers

- Sequence containers:
 - **array**
 - **forward_list**
- Unordered associative containers:
 - **unordered_set**
 - **unordered_multiset**
 - **unordered_map**
 - **unordered_multimap**
- **Unordered containers** provide faster access to individual elements by their key, although they are generally less efficient for range iteration through a subset of their elements.
- Internally, the elements in the unordered containers are not sorted in any particular order, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their values or key values (with a constant average time complexity on average).
 - A hash function is any function that can be used to map data of arbitrary size onto data of a fixed size. The values returned by a hash function are called hash values, hash codes, digests, or simply hashes. Hash functions are often used in combination with a hash table, a common data structure used in computer software for rapid data lookup (from: Wikipedia).
 - The hash function returns always the same value for the same argument. The value returned shall have a small likelihood of being the same as the one returned for a different argument.

typedefs found in first-class containers

typedef	Description
allocator_type	The type of the object used to allocate the container's memory.
value_type	The type of element stored in the container.
reference	A reference to the type of element stored in the container.
const_reference	A constant reference to the type of element stored in the container. Such a reference can be used only for <i>reading</i> elements in the container and for performing <code>const</code> operations.
pointer	A pointer to the type of element stored in the container.
const_pointer	A pointer to a constant of the container's element type.
iterator	An iterator that points to an element of the container's element type.
const_iterator	A constant iterator that points to the type of element stored in the container and can be used only to <i>read</i> elements.
reverse_iterator	A reverse iterator that points to the type of element stored in the container. This type of iterator is for iterating through a container in reverse.

typedef	Description
const_reverse_iterator	A constant reverse iterator that points to the type of element stored in the container and can be used only to <i>read</i> elements. This type of iterator is for iterating through a container in reverse.
difference_type	The type of the result of subtracting two iterators that refer to the same container (operator <code>-</code> is not defined for iterators of <code>lists</code> and associative containers).
size_type	The type used to count items in a container and index through a sequence container (cannot index through a <code>list</code>).

ITERATORS

- An iterator is an object (like a pointer) that points to an element inside the container.
- We can use iterators to move through the contents of the container.
- ***it** - dereferences iterator **it**
 - **it++** - moves **it** to the next element of the container
 - other available operators: **--, ==, !=**
(<, <=, > or >= only for random-access iterators, see below)
- container member-functions
 - **begin()** – returns an iterator located at the 1st element in the container
 - **end()** – returns an iterator located one beyond the last element in the container
- Basic outline of how an iterator can cycle through all elements of a container:

```
STL_container<type>::iterator p;  
for (p = container.begin(); p != container.end(); p++)  
    process_element_at_location p;
```

Ex:

```
vector<int> v1;  
// FILL v1 WITH SOME VALUES ...  
vector<int>::iterator p;  
for (p = v1.begin(); p != v1.end(); p++)  
    cout << *p << endl; // ... AND SHOW THE VALUES
```

Iterator categories

Category	Description
<i>input</i>	Used to read an element from a container. An input iterator can move only in the forward direction (i.e., from the beginning of the container to the end) one element at a time. Input iterators support only one-pass algorithms—the same input iterator cannot be used to pass through a sequence twice.
<i>output</i>	Used to write an element to a container. An output iterator can move only in the forward direction one element at a time. Output iterators support only one-pass algorithms—the same output iterator cannot be used to pass through a sequence twice.
<i>forward</i>	Combines the capabilities of input and output iterators and retains their position in the container (as state information).
<i>bidirectional</i>	Combines the capabilities of a forward iterator with the ability to move in the backward direction (i.e., from the end of the container toward the beginning). Bidirectional iterators support multipass algorithms.
<i>random access</i>	Combines the capabilities of a bidirectional iterator with the ability to directly access any element of the container, i.e., to jump forward or backward by an arbitrary number of elements.

Iterator types supported by each container

Container	Type of iterator supported
<i>Sequence containers (first class)</i>	
vector	random access
deque	random access
list	bidirectional
<i>Associative containers (first class)</i>	
set	bidirectional
multiset	bidirectional
map	bidirectional
multimap	bidirectional
<i>Container adapters</i>	
stack	no iterators supported
queue	no iterators supported
priority_queue	no iterators supported

- **NOTE (VERY IMPORTANT):** The **iterator category** that each **container** supports determines whether the container can be used with specific **algorithms** in the **STL**
 - An algorithm that requires only forward iterators can be used with any container that supports forward, bidirectional or random-access iterators
 - But an algorithm that requires random-access iterators can be used only with containers that support random-access iterators

Iterator **typedefs**

Predefined typedefs for iterator types	Direction of ++	Capability
<code>iterator</code>	forward	read/write
<code>const_iterator</code>	forward	read
<code>reverse_iterator</code>	backward	read/write
<code>const_reverse_iterator</code>	backward	read

Constant iterators

- a constant iterator is an iterator that does not allow you to change the element at its location

Reverse iterators

- a reverse iterator is an iterator that can be used to cycle through all elements of a container, in reverse order, provided that the container has bidirectional iterators;
- basic outline of how a reverse iterator can cycle through all elements of a container:

```
STL_container<type>::reverse_iterator p;
for (p = container.rbegin(); p != container.rend(); p++)
    process_element_at_location p;
```

Ex:

```
vector<int> v1;
// FILL v1 WITH SOME VALUES ...
vector<int>::reverse_iterator p;
for (p = v1.rbegin(); p != v1.rend(); p++)
    cout << *p << endl; // ... AND SHOW THE VALUES
```

- **rbegin()** – member-function that returns an iterator located at the **last element**
- **rend()** – member-function that returns a sentinel that marks the "end" of the elements in reverse order
- **note** that for a **reverse_iterator**, the increment operator, **++**, moves **backward** through the elements

Iterators and element insertion/removal

- note that when you **insert or remove** an element into or from a container, that **can affect the other iterators**
- in general, there is no guarantee that the iterators will be located at the same element after an addition or deletion
- **some containers** do, however, **guarantee that the iterators will not be moved** by additions or deletions, except if the iterator is located at an element that is removed (ex: **list**; **vector and deque make no such guarantee**)

Iterator operations for each type of iterator

Iterator operation	Description
<i>All iterators</i>	
++p	Preincrement an iterator.
p++	Postincrement an iterator.
<i>Input iterators</i>	
*p	Dereference an iterator.
p = p1	Assign one iterator to another.
p == p1	Compare iterators for equality.
p != p1	Compare iterators for inequality.
<i>Output iterators</i>	
*p	Dereference an iterator.
p = p1	Assign one iterator to another.
<i>Forward iterators</i>	
Forward iterators provide all the functionality of both input iterators and output iterators.	

Iterator operation	Description
<i>Bidirectional iterators</i>	
--p	Predecrement an iterator.
p--	Postdecrement an iterator.
<i>Random-access iterators</i>	
p += i	Increment the iterator p by i positions.
p -= i	Decrement the iterator p by i positions.
p + i or i + p	Expression value is an iterator positioned at p incremented by i positions.
p - i	Expression value is an iterator positioned at p decremented by i positions.
p - p1	Expression value is an integer representing the distance between two elements in the same container.
p[i]	Return a reference to the element offset from p by i positions
p < p1	Return true if iterator p is less than iterator p1 (i.e., iterator p is before iterator p1 in the container); otherwise, return false.

Iterator operation	Description
p <= p1	Return true if iterator p is less than or equal to iterator p1 (i.e., iterator p is before iterator p1 or at the same location as iterator p1 in the container); otherwise, return false.
p > p1	Return true if iterator p is greater than iterator p1 (i.e., iterator p is after iterator p1 in the container); otherwise, return false.
p >= p1	Return true if iterator p is greater than or equal to iterator p1 (i.e., iterator p is after iterator p1 or at the same location as iterator p1 in the container); otherwise, return false.

ALGORITHMS

Generic algorithms

- Generic Algorithms are template functions that use iterators as template parameters.

Classification 1

- Nonmodifying Algorithms
 - ex: find, max_elem, min_elem
- Modifying Algorithms
 - ex: copy, remove
- Sorting and Related Algorithms
 - ex: sort, merge
- Numeric Algorithms
 - ex: accumulate

Classification 2

- Initialization Algorithms
 - ex: fill, copy, generate
- Transformations
 - ex: sort, transform, reverse, random_shuffle
- Searching Algorithms
 - find, max_elem, min_elem, binary_search
- Removal and Replacement Algorithms
 - remove (\Rightarrow container.erase() on the next program instruction), replace
- Other Algorithms
 - ex: count, count_if, accumulate

NOTE

- **Algorithms act on container elements;** they don't act on containers
- **parameters are iterators not containers;**
- the container properties (ex: size) remain the same

Example:

```
remove(numbers.begin(), numbers.end(), 0); //remove zeros
```

- does not change the size of **numbers**;
- rather it moves the elements of **numbers** that are not equal to zero to the beginning of **numbers**
and returns an iterator that points to the first element after them
- if one wants to discard the zeros, must do it explicitly, using **erase** method

```
numbers.erase(remove(numbers.begin(), numbers.end(), 0), numbers.end());
```

Nonmodifying algorithms

- Algorithms that do not modify the container they operate upon.
- The declaration of the generic function **find** algorithm:

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T &value);
```

- The declaration tells us that **find** works with any container that provides an iterator at least "as strong as" an **input iterator**.
- Type T objects must be equality comparable.

Iter find(Iter first, Iter last, const T &value);

- The generic algorithm **find()** locates an element within a sequence. It takes three arguments.
- The first two specify a range: [start, end], the third specifies a target value for the search.
- If requested value is **found**, **find()** returns an iterator that points to the **first element** that is identical to the sought-after value.
- If the requested value is **not found**, **find()** returns an iterator pointing **one element past the final element** in the sequence (that is the **Last** iterator, see above parameters).

More nonmodifying algorithms

- **count**
 - counts occurrences of a value in a sequence
- **equal**
 - asks: are elements in two ranges equal ?
- **search**
 - looks for the first occurrence of a match sequence within another sequence
- **binary_search**
 - searches for a value in a sorted container.
This is an efficient search for sorted sequences with random access iterators.

Modifying algorithms

- Container modifying algorithms change the content of the elements or their order.
- **copy**
 - copies from a source range to a destination range;
this can be used to shift elements in a container to the left provided that the first element in the source range is not contained in the destination range.
- **remove**
 - removes all elements from a range equal to the given value.
○ must be followed by erase()
- **random_shuffle**
 - shuffles the elements of a sequence.

Sorting algorithms

- **sort**
 - sorts elements in a range in nondescending order,
or in an order determined by a user-specified binary predicate.
- **merge**
 - merges two sorted source ranges into a single destination range.

Numeric algorithms

- **accumulate**
 - sums the elements in a container.

NOTE:

in general, generic algorithms do not alter the size of the containers they operate on
(see example about `remove()` algorithm)

STL – STANDARD TEMPLATE LIBRARY

ITERATORS

```
// STL - ITERATORS
// Pointers and iterators are similar
// An iterator is a generalization of a pointer

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    const int SIZE = 6;

    int a[SIZE] = {1,2,3,4,5,6};

    vector<int> v(a,a+SIZE); // initializing a vector from an array, using one of the vector constructors (*)  
// scanning an array, using a pointer
    cout << "a[] = { ";
    for (int *aPtr = a; aPtr != a + SIZE; aPtr++)
        cout << *aPtr << " ";
    cout << "}" << endl;

    // scanning a vector, using an iterator
    // note the similarity with the previous cycle
    cout << "v[] = { ";
    for (vector<int>::iterator vPtr = v.begin(); vPtr != v.end(); vPtr++)
        cout << *vPtr << " "; // could / should be const_iterator ???  
cout << "}" << endl;

    return 0;
}
```

(*) NOTE:

The use of an initializer list
(a list of initializers inside brackets ({ }))
vector<int> v = {1,2,3,4,5};
is legal since C++11 but not legal in previous C++ standards

```

// STL - ITERATORS (another version of the previous program, using typedef)
// Pointers and iterators are similar
// An iterator is a generalization of a pointer
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    const int SIZE = 6;

    int a[SIZE] = {1,2,3,4,5,6};

    vector<int> v(a,a+SIZE);

typedef vector<int>::const_iterator vecIntIterator;

    // scanning an array, using a pointer
    cout << "a[] = { ";
    for (int *aPtr = a; aPtr != a + SIZE; aPtr++)
        cout << *aPtr << " ";
    cout << "}" << endl;

    // scanning a vector, using an iterator
    // note the similarity with the previous cycle
    cout << "v[] = { ";
    for (vecIntIterator vPtr = v.begin(); vPtr != v.end(); vPtr++)
        cout << *vPtr << " ";
    cout << "}" << endl;

    return 0;
}

```

NOTE: Iterators (C++11)

Since C++11, the **auto** keyword makes this a little easier:

```

for (auto vPtr = v.begin(); vPtr != v.end(); vPtr++)
    cout << *vPtr << endl;

```

Range-based for() loops

An even simpler syntax to allow us to iterate through sequences, called a range-based for statement (or “for each”):

```

for (auto x: v) // OR for (const auto &x: v) => x can't be modified
    cout << x << endl;

```

You can translate this as “for each value of x in v”.

If you want to modify the value of x, you can make x a reference

```

for (auto &x: v) // x can modified
    x = 10 * x ;

```

This syntax **works for C-style arrays** and anything that supports an iterator via **begin()** and **end()** functions. This includes all standard template library container classes (including string).

```

// STL - ITERATORS
// ::iterator and ::const_iterator
// Another example of template functions

#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename T>
void showVector(string vName, const vector<T> &v)
{
    cout << vName << "[] = { ";
    for (vector<T>::const_iterator vPtr = v.begin(); vPtr != v.end(); vPtr++)
        cout << *vPtr << " ";
    cout << "}\n";
}

// TO DO BY STUDENTS:
// - rewrite the code using new C++11 allowed syntax for the cycle.

int main()
{
    const int SIZE = 5;
    int a[SIZE] = {1,2,3,4,5};
    vector<int> v1(a,a+SIZE); // initializing a vector from an array
    vector<double> v2(10,0.1); // 10 elements, all equal to 0.1

    showVector("v1",v1);
    for (vector<int>::iterator vPtr = v1.begin(); vPtr != v1.end(); vPtr++)
        *vPtr = *vPtr * 10;
    showVector("v1",v1);

    showVector("v2",v2);
    // what is the result of this cycle ?
    for (vector<double>::iterator vPtr = v2.begin() + 1; vPtr != v2.end();
    vPtr++)
        *vPtr = *vPtr + *(vPtr-1); //vector supports RANDOM ITERATORS
    showVector("v2",v2);
    return 0;
}

```

```

// STL - ITERATORS
// Using a reverse_iterator, rbegin() and rend()

#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename T>
void showVector(string vName, const vector<T> &v)
{
    cout << vName << "[] = { ";
    for (vector<T>::const_iterator vPtr = v.begin(); vPtr != v.end(); vPtr++)
        cout << *vPtr << " ";
    cout << "}\n";
}

int main()
{
    const int SIZE = 5;

    int a[SIZE] = {1,2,3,4,5};
    vector<int> v1(a,a+SIZE);

    showVector("v1",v1);

    // what is the result of this loop ? (see previous program)
    for (vector<int>::iterator vPtr = v1.begin() + 1;
         vPtr != v1.end(); vPtr++)
        *vPtr = *vPtr + *(vPtr-1);

    showVector("v1",v1);

    // what is the result of this loop ?
    // compare with previous loop

    for (vector<int>::reverse_iterator vPtr = v1.rbegin() + 1;
         vPtr != v1.rend(); vPtr++)
        *vPtr = *vPtr + *(vPtr-1);

    showVector("v1",v1);
    return 0;
}

```

STL – STANDARD TEMPLATE LIBRARY

CONTAINERS

SEQUENCE CONTAINERS - VECTOR

```
// STL - SEQUENCE CONTAINERS - VECTOR
// insert(), erase() and clear() methods

#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename T>
void showVector(string vName, const vector<T> &v)
{
    typedef vector<T>::const_iterator constIterator;
    cout << vName << "[] = { ";
    for (constIterator vPtr = v.begin(); vPtr != v.end(); vPtr++)
        cout << *vPtr << " ";
    cout << "}\n";
}

int main()
{
    const int SIZE1 = 10;
    const int SIZE2 = 2;
    const int SIZE3 = 3;

    vector<int> v1(SIZE1);
    vector<int> v2(SIZE2);
    int a[SIZE3] = {10,20,30};

    showVector("v1",v1);
    showVector("v2",v2);

    for (size_t i=0; i<v1.size(); i++)
        v1[i] = i;
    cout << endl;
    showVector("v1",v1);

    cout << "\nv1 status after ... \n\n";

    // inserting a value
    v1.insert(v1.begin()+2,-1);
    showVector("insert(v1.begin()+2, -1)",v1);
```

```

// inserting a vector into another one (v2 into v1)
// NOTE: v2.end() points past the end of the vector
v1.insert(v1.begin()+5, v2.begin(), v2.end());
showVector("insert(v1.begin()+5, v2.begin(), v2.end())",v1);

// inserting an array into a vector (a into v1)
// NOTE: a+SIZE3 points past the end of the array
v1.insert(v1.begin(), a, a+SIZE3);
showVector("insert(v1.begin(), a, a+SIZE3)",v1);

cout << endl;

// erasing an element
v1.erase(v1.begin()+5);
showVector("erase(v1.begin()+5)",v1);

// erasing an element sequence
v1.erase(v1.begin(),v1.begin()+3); // WHICH ELEMENTS ARE ERASED?
showVector("erase(v1.begin(), v1.begin()+3)",v1);

// clearing the vector
v1.clear();
showVector("clear()",v1);
cout << endl;

return 0;
}

```

SEQUENCE CONTAINERS - LIST

```
// STL - SEQUENCE CONTAINERS - LIST
// Some methods:
// push_front(), push_back(), insert(), remove() and sort()
// Compare with previous program for manipulating "linked lists"

#include <iostream>
#include <list>
#include <string>

using namespace std;

template <typename T>
void showList(string lstName, const list<T> &lst)
{
    cout << lstName << " = { ";
    for (auto elem : lst)
        cout << elem << " ";
    cout << "}\n\n";
}

void main()
{
    list<int> lst; // why not list<int> list; ??? the compiler accepts it ...
    size_t index;
    int value;

    showList("lst", lst);

    cout << "push_front() 1, 2, 3, 4, 5\n";
    for (value=1; value<=5; value++)
    {
        lst.push_front(value);
        showList("lst", lst);
    }

    value = 6;
    cout << "push_back(" << value << ")\n";
    lst.push_back(value);
    showList("lst", lst);
```

```

index = 2; value = 7;
cout << "insertAt(" << index << "," << value << ")\n";
//lst.insert(lst.begin() + index, value); //NOT ALLOWED lst.begin() + index
if (index > lst.size())
    cout << "there is no such node index: " << index << endl;
else
{ // TO DO : implement as a function
    list<int>::iterator lstIterator = lst.begin();
    for (size_t i = 0; i < index; i++)
        lstIterator++; // alternative: use advance algorithm
    lst.insert(lstIterator, value);
}
showList("lst", lst);

index = 10; value = 8;
cout << "insertAt(" << index << "," << value << ")\n";
//lst.insert(lst.begin() + index, value); //NOT ALLOWED lst.begin() + index
if (index > lst.size())
    cout << "there is no such node index: " << index << endl;
else
{ // TO DO : implement as a function
    list<int>::iterator lstIterator = lst.begin();
    for (size_t i = 0; i < index; i++)
        lstIterator++; // alternative: use advance() algorithm
    lst.insert(lstIterator, value);
}
showList("lst", lst);

value = 2;
cout << "remove(" << value << ")\n";
lst.remove(value); // returns 'void' ! No way to see if 'value' exists ...
                    // Do not confuse with remove() algorithm
showList("lst", lst);

cout << "sort()\n";
lst.sort(); // NOTE: vector class does not have a sort() method
showList("lst", lst);

cout << endl;
}

```

NOTE:

- list supports bidirectional iterators, not random ones ...
- list::erase() - erases elements by their position (iterator)
 - complexity of erase() is O(1) for lists and O(n) for vectors
 - <http://www.cs.northwestern.edu/~riesbeck/programming/c++/stl-summary.html>
 - <https://www.cs.cmu.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html>
- list::remove() - removes elements by their value;

do not confuse with remove() algorithm

ASSOCIATIVE CONTAINERS – MAP & MULTIMAP

```
// STL - Vectors and maps

#include <iostream>
#include <vector>
#include <map>

using namespace std;

int main()
{
    int n;

    vector<int> v;
    //v[0] = 10; // UNCOMMENT and interpret what happens
    //v[9] = 90;

    cout << "VECTOR\n";
    n=0;

    for (vector<int>::const_iterator vi=v.begin(); vi!=v.end(); vi++)
    {
        n++;
        cout << n << " - " << *vi << endl;
    }

    map <int,int> m;

    //m[0] = 10; //UNCOMMENT and interpret what happens
    //m[9] = 90;

    cout << "MAP\n";
    n=0;

    for (map<int,int>::const_iterator mi=m.begin(); mi!=m.end(); mi++)
    {
        n++;
        cout << n << " - " << mi->first << ", " << mi->second << endl;
    }

    /* ALTERNATIVE CODE:
    for (auto p : m)
    {
        n++;
        cout << n << " - " << p.first << ", " << p.second << endl;
        // NOTE: each element 'p' of 'm' is a pair!
    }
    */
    return 0;
}
```

```

// STL - maps

#include <iostream>
#include <string>
#include <map>
#include <utility>

using namespace std;

int main()
{
    map <string,int> m;

    string key = "Porto", k;
    cout << m[key] << endl; //NOTE: 'key' automatically inserted into map 'm'

    cout << "Key to search? "; cin >> k;
    if (m.find(k) != m.end())
        cout << "key '" << k << "' found in map 'm'\n";
    else
        cout << "key '" << k << "' NOT found in map 'm'\n";

    return 0;
}

//=====
// STL - Vectors, maps and pairs

#include <iostream>
#include <string>
#include <map>
#include <utility> // needed for 'pair'

using namespace std;

int main()
{
    map<string,int> m;
    pair<string,int> p; // a pair is a templated struct

    m["Porto"]=1;
    m["Benfica"]=2;

    cout << "MAP\n";
    int n=0;
    for (map<string,int>::const_iterator mi=m.begin(); mi!=m.end(); mi++)
    {
        n++;
        p = *mi; // each element of a "map" is a "pair"
        cout << n << " - " << p.first << ", " << p.second << endl;
    }

    //NOTE the order by which elements were presented

    return 0;
}

```

```

#include <iostream>
#include <string>
#include <map>
#include <utility>

using namespace std;

int main()
{
    map <int, string> phone_user;

    phone_user.insert(pair<int, string> (1234, "Mary"));
    phone_user.insert(pair<int, string> (1234, "John")); //NOTE: key already used !
    phone_user.insert(pair<int, string> (2345, "Ann"));

    for (auto x : phone_user)
        cout << x.first << " - " << x.second << endl;

    return 0;
}
=====

#include <iostream>
#include <string>
#include <map>
#include <utility>

using namespace std;

int main()
{
    map <int, string> phone_user;
    int phoneNumber;
    string phoneUser;

    //create 'phone - user' map
    while (cout << "Phone number & User name (CTRL-Z to end)? ", //NOTE THIS
          cin >> phoneNumber >> phoneUser) // !!!
    {
        pair <map <int, string>::iterator, bool> p; // WHAT IS p ?!!!
        p = phone_user.insert(pair<int, string>(phoneNumber, phoneUser));
        if (p.second == false) // if insertion failed
            cout << "Phone number already associated to another user !\n";
        // TO DO BY STUDENTS: show the name of the user the phone is associated with
        // cout << "Phone number already associated to user " << (*p.first).second << "\n";
    }

    // show 'phone - user' map contents
    for (auto x : phone_user)
        cout << x.first << " - " << x.second << endl;

    return 0;
}

```

```
//=====

#include <iostream>
#include <string>
#include <map>
#include <utility>

using namespace std;

int main()
{
    multimap<int, string> phone_user;

    phone_user.insert(pair<int, string>(1234, "Mary"));
    phone_user.insert(pair<int, string>(2345, "John"));
    phone_user.insert(pair<int, string>(1234, "Ann"));
    //phone_user.insert(pair<int, string>(1234, "Ann")); // TRY THIS

    for (const auto & x : phone_user)
        cout << x.first << " - " << x.second << endl;

    return 0;
}
```

NOTES:

- `for (auto x : phone_user)`
will create a temporary copy of each element; usually less efficient
- `for (auto & x : phone_user)`
won't create copies, and allows you to modify the elements
if the underlying container allows that (that is, it is not 'const').
- `for (const auto & x : phone_user)`
won't create copies and won't allow you to do any modifications;
this can prevent accidental modifications

TO DO BY STUDENTS

Modify the program above in order to do the following:

- after creating the phone directory, ask the user for a phone number and show the list of all users that use that phone number.

Suggestion: investigate the use of the `equal_range()` method of `multimap`

```

// STL - ASSOCIATIVE CONTAINERS - MAPS AND MULTIMAPS
// Telephone directory
// Adapted from Big C++ book

#include <iostream>
#include <string>
#include <map>

using namespace std;

//-----
// TelephoneDirectory maintains a map of name/number pairs.
class TelephoneDirectory
{
public:
    void add_entry(string name, unsigned int number);
    unsigned int find_number(string name) const;
    void print_all(ostream& out) const;
    void print_by_number(ostream& out) const;
private:
    map<string, unsigned int> database;
    typedef map<string, unsigned int>::const_iterator MapIterator;
};

//-----
/** 
Add a new name/number pair to database.
@param name the new name
@param number the new number
*/
void TelephoneDirectory::add_entry(string name, unsigned int number)
{
    database[name] = number;
}

//-----
/** 
Find the number associated with a name.
@param 'name' the name being searched
@return the associated number, or zero if not found in database
*/
unsigned int TelephoneDirectory::find_number(string name) const
{
    /*
    for (MapIterator p = database.begin(); p != database.end(); p++)
        if (p->first == name)
            return p->second;
    return 0; // not found
    */
    MapIterator p = database.find(name);
    if (p != database.end()) // if name was found
        return p->second;
    else
        return 0;
}

```

```

//-----
/** Print all entries on given output stream
in 'name:number' format, ordered by 'name'.
@param 'out' the output stream
*/
void TelephoneDirectory::print_all(ostream& out) const
{
    MapIterator current = database.begin();
    MapIterator stop = database.end();
    while (current != stop)
    {
        out << current->first << " : " << current->second << "\n";
        ++current;
    }
}

//-----
/** Print all entries on given output stream
in 'name:number' format, ordered by 'number'.
@param 'out' the output stream
*/
void TelephoneDirectory::print_by_number(ostream& out) const
{
    multimap<unsigned int, string> inverse_database;
    typedef multimap<unsigned int, string>::iterator MMapIterator;

    MapIterator current = database.begin();
    MapIterator stop = database.end();
    while (current != stop)
    {
        inverse_database.insert(
            pair<unsigned int, string> (current->second, current->first));
        // ALTERNATIVE: using make_pair()
        // pair<unsigned int, string> p = make_pair(current->second, current->first);
        // inverse_database.insert(p);
        ++current;
    }

    MMapIterator icurrent = inverse_database.begin();
    MMapIterator istop = inverse_database.end();
    while (icurrent != istop)
    {
        out << icurrent->first << " : " << icurrent->second << "\n";
        ++icurrent;
    }
}

```

```

//-----
int main()
{
    TelephoneDirectory data;
    data.add_entry("Sarah", 227235591);
    data.add_entry("Mary", 223841212);
    // data.add_entry("Mary", 223841213); // UNCOMMENT AND INTERPRET RESULT
    data.add_entry("Fred", 223841212); //NOTE: repeated number

    cout << "Number for Mary " << data.find_number("Mary") << "\n";
    cout << "Number for John " << data.find_number("John") << "\n";

    cout << "\nPRINTING BY NAME \n";
    data.print_all(cout);

    cout << "\nPRINTING BY NUMBER \n";
    data.print_by_number(cout);
    return 0;
}

//=====

// STL - ASSOCIATIVE CONTAINERS - MAPS
// what does this program do ...?!

#include <iostream>
#include <string>
#include <map>

using namespace std;

int main()
{
    map <string, unsigned int> m;
    typedef map <string, unsigned int>::const_iterator MapIterator;

    string word;

    cout << "Write a text; end with <ENTER> followed by <CTRL-Z>\n";
    while (cin >> word)
        m[word]++;
}

for (MapIterator i = m.begin(); i != m.end(); i++)
    cout << i->first << ":" << i->second << endl;
//cout << (*i).first << " - " << (*i).second << endl; //alternative

return 0;
}

// TEXT: a tooter who tooted a flute ...
a tutor who tooted a flute tried to tutor two tooters to toot
said the two to their tutor
is it harder to toot or to tutor two tooters to toot

```

TO DO BY STUDENTS: use a `for()` range-based loop

```

// STL - ASSOCIATIVE CONTAINERS - MAPS
// A program for searching words in a map

#include <iostream>
#include <string>
#include <map>

using namespace std;

int main()
{
    map <string, unsigned int> word_count;
    string word;

    cout << "Write a text; end with <ENTER> followed by <CTRL-Z>\n";
    while (cin >> word)
        word_count[word]++;
    
    cout << endl;
    cout << "Word list:\n";
    for (auto w:word_count)
        cout << w.first << ":" << w.second << endl;

    // Searching for a 'word' in 'word_count' map
    // BE CAREFUL !!! What happens when the 'word' does not exist in the map ?!

    cin.clear(); // WHY? To be able to continue reading after CTRL-Z

    cout << endl;
    cout << "Word to search ? ";
    cin >> word;

    cout << "word_count[" << word << "] = " << word_count[word] << endl;
    // WHAT HAPPENS IF word DOES NOT BELONG THE THE MAP ...?

    cout << endl;
    cout << "Word list:\n"; //IMPLEMENT AS A FUNCTION ...? (see above code)
    for (auto w:word_count)
        cout << w.first << ":" << w.second << endl;

    return 0;
}

```

```

// STL - ASSOCIATIVE CONTAINERS - MAPS
// Solution for the previous problem (searching for a word that does not exist)

#include <iostream>
#include <string>
#include <map>

using namespace std;

int main()
{
    map <string, unsigned int> word_count;
    typedef map <string, unsigned int>::const_iterator MapIterator;

    string word;

    cout << "write a text; end with <ENTER> followed by <CTRL-Z>\n";
    while (cin >> word)
        word_count[word]++;
}

cout << endl;
cout << "Word list:\n";
for (MapIterator i = word_count.begin(); i != word_count.end(); i++)
    cout << i->first << ":" << i->second << endl;
//cout << (*i).first << " - " << (*i).second << endl; //alternative

cin.clear(); // To be able to continue reading after CTRL-Z
cout << endl;
cout << "Word to search ? ";
cin >> word;

//SOLUTION FOR THE PREVIOUS PROBLEM
MapIterator itaux = word_count.find(word);
if (itaux != word_count.end())
    cout << endl << "word_count[" << word << "] = " << word_count[word] << endl;
//cout << endl << "word_count[" << word << "] = " << itaux->second << endl;

else
    cout << "\\" << word << "\" not found !\n";

cout << endl;
cout << "Word list:\n";
for (MapIterator i = word_count.begin(); i != word_count.end(); i++)
    cout << i->first << ":" << i->second << endl;

return 0;
}

```

STL – STANDARD TEMPLATE LIBRARY

ALGORITHMS

<http://www.sgi.com/tech/stl/>

<http://msdn.microsoft.com/en-us/library/c191tb28.aspx>

<http://msdn.microsoft.com/en-us/library/yah1y2x8.aspx>

// STL – ALGORITHMS (some usage examples)

```
#include <iostream>
#include <cstddef>
#include <iomanip>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
#include <ctime>

using namespace std;

void displayVec(string title, const vector<int> &v)
{
    cout << title << ":" \n";
    for (size_t i=0; i<v.size(); i++)
        cout << setw(3) << v.at(i) << " ";
    cout << endl;
}

int myRand()
{
    return rand() % 10 + 1;
}

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

    vector<int> v1(10);
    vector<int> v2(10);

    fill(v1.begin(), v1.end(),1);
    displayVec("v1 - fill(v1.begin(), v1.end(),1)",v1);

    // void fill_n (OutputIterator first, size n, const T& val);
    fill_n(v1.begin()+4, 3, 2);
    displayVec("v1 - fill_n (v1.begin()+4, 3, 2)",v1);

    generate(v2.begin(),v2.end(),myRand);
    displayVec("v2 - generate(...,myRand)",v2);

    return 0;
}
```

```

// STL - ALGORITHMS

#include <iostream>
#include <cstddef>
#include <iomanip>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
#include <ctime>
#include <iterator> //ostream_iterator iterator

using namespace std;

void displayVec(string title, const vector<int> &v)
{
    cout << setw(37) << title << ":";
    for (size_t i=0; i<v.size(); i++)
        cout << setw(3) << v.at(i) << " ";
    cout << endl << endl;
}

int myRand()
{
    return rand() % 10 + 1;
}

int calcsquare(int value) //calculates de square of 'value'
{
    return value * value;
}

bool equalsTwo(int value)
{
    return value == 2;
}

int main() {
    vector<int> v1(10);
    vector<int> v2(10);

    //fill(v1.begin(), v1.end(),1);
    //displayVec("fill(v1.begin(), v1.end(),1)",v1);

    //fill_n(v1.begin()+4, 3, 2);
    //displayVec("fill_n(v1.begin()+4, 3, 2)",v1);

    srand((unsigned) time(NULL));

    generate(v2.begin(),v2.end(),myRand);
    displayVec("v2: generate(...,myRand)",v2);

    sort(v2.begin(),v2.end());
    displayVec("sort(v2.begin(),v2.end())",v2);

    random_shuffle(v2.begin(),v2.end());
    displayVec("random_shuffle(v2.begin(),v2.end())",v2);

    sort(v2.rbegin(), v2.rend());
    displayVec("sort(v2.rbegin(),v2.rend())", v2);

    vector<int> v3(v2.size());
    transform(v2.begin(),v2.end(),v3.begin(),calcsquare);
    displayVec("v2->v3: transform(...,calculateSquare)",v3);
}

```

```

vector<int> v4(v2.size());
copy(v2.begin(), v2.begin() + 5, v4.begin() + 2);
displayVec("v4: copy(v2.begin(), v2.begin() + 5, v4.begin() + 2)", v4);

// COMMENT BEFORE NEXT STEP: MERGE
//reverse(v2.begin(), v2.end());
//displayVec("reverse(v2.begin(), v2.end())", v2);

//vector<int> v3(v1.size() + v2.size());
//sort(v1.begin(), v1.end());
//merge(v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin()); //vectors must be sorted
//displayVec("merge(v1..., v2..., v3.begin())", v3);

cout << "value to remove from v2 ? ";
int x; cin >> x;
vector<int>::iterator p1 = remove(v2.begin(), v2.end(), x);
displayVec("after remove()", v2); //NOTE: the elements past the new end ...
// ... of the vector can still be accessed but their values are unspecified
v2.erase(p1, v2.end());
displayVec("after remove() + erase()", v2);

//vector<int>::iterator p2 = remove_if(v2.begin(), v2.end(), equalsTwo);
//v2.erase(p2, v2.end());
//displayVec("after remove_if(...equalsTwo) + erase()", v2);

return 0;
}

```

SINCE C++11

ALTERNATIVE TO THE USE OF `calcsquare()`:

```
transform ( v2.begin(), v2.end(), v3.begin(), calcsquare );
```

USE A LAMBDA EXPRESSION (a function defined on the fly):

```
transform ( v2.begin(), v2.end(), v3.begin(), [](int x) -> int {return x*x; } );
```

In C++11 and later, a **lambda expression**—often called a lambda—is a convenient way of defining an **anonymous function object** (a *closure*) right at the location where it is invoked or passed as an argument to a function. Typically lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous methods.

SINTAXE FOR LAMBDA EXPRESSIONS (simplified form)

```
[ lambda-introducer ] (parameters) -> return-type
{
    lambda body
}
```

Generally return-type in lambda expression are evaluated by compiler itself and we don't need to specify that explicitly and **-> return-type** part can be ommited.

<https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=vs-2019>

STL – STANDARD TEMPLATE LIBRARY

Summary

- **STL** - a library of classes that represent **containers** that occur frequently in computer programs in all application areas.
- Each class in the STL supports a relatively small set of operations. Basic functionality is extended through the use of **generic algorithms**.
- The **3 fundamental data structures** are the **vector**, **list**, and **deque**.
- **Vector** and **deque** are indexed data structures; they support efficient access to each element based on an integer key.
- A **list** supports efficient insertion into or removal from the middle of a collection. Lists can also be merged with other lists.
- A **set** maintains elements in order. Permits very efficient insertion, removal, and testing of elements.
- A **map** is a keyed container. Entries in a map are accessed through a **key**, which can be any ordered data type. Associated with each key is a **value**.
- **Multimaps** and **multisets** allow more than one value to be associated with a key.
- **Maps/multimaps** and **sets/multisets** are named **associative containers**. What makes sets/multisets associative is the fact that their elements are referenced by their key and not by their absolute position in the container.
- **Stacks, queues, and priority queues** are **adapters** built on top of the fundamental collections. A stack enforces the LIFO protocol, while the queue uses FIFO.

Some references

STL: <https://docs.microsoft.com/en-us/previous-versions/cscc687y%28v%3dvs.140%29>

ALGORITHMS: <https://docs.microsoft.com/en-us/cpp/standard-library/algorithms?view=vs-2017>

FRIEND FUNCTIONS (not lectured in 2021/2022)

- Class operations are typically implemented as member functions
- Some operations are better implemented as ordinary (nonmember) functions

```
//Program to demonstrate the use of function equal() that compares 2 dates
```

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

class Date
{
public:
    Date(); //Initializes the date to January 1st.
    Date(int y, int m, int d);
    void input();
    void output() const;
    int get_year() const;
    int get_month() const;
    int get_day() const;
private:
    int year;
    int month;
    int day;
};

//-----
Date::Date()
{
    year = 1970;
    month = 1;
    day = 1;
}

//-----
Date::Date(int y, int m, int d)
{
    year = y;
    month = m;
    day = d;
}

//-----
int Date::get_year() const
{
    return year;
}

//-----
int Date::get_month() const
{
    return month;
}
```

```

//-
int Date::get_day( ) const
{
    return day;
}

//-
void Date::input( )
{
    cout << "year (number) ? ";
    cin >> year;
    cout << "month (number) ? ";
    cin >> month;
    cout << "day (number) ? ";
    cin >> day;
}

//-
void Date::output( )
{
    cout << year << "/" << month << "/" << day << endl;
}

//-
bool equal(const &Date date1, const Date &date2) // NOTE: external to class Date
{
    return (
        date1.get_year() == date2.get_year() &&
        date1.get_month() == date2.get_month() &&
        date1.get_day() == date2.get_day()
    );
}

//-
int main( )
{
    Date date1(2014,5,14), date2;

    cout << "Date1 is ";
    date1.output( );

    cout << endl;
    cout << "Enter Date2:\n";
    date2.input( );
    cout << "Date2 is ";
    date2.output( );

    if (equal(date1, date2))
        cout << "date1 is equal to date2\n";
    else
        cout << "date1 is NOT equal to date2\n";
    return 0;
}

```

```

// Program to demonstrate the use of function equal() that compares 2 dates
// A more efficient version that declares
// equal() as a friend function of class Date

#include <iostream>
using namespace std;

class Date
{
    friend bool equal(const Date &date1, const Date &date2);
public:
    Date(int y, int m, int d);
    Date(); //Initializes the date to January 1st.
    void input();
    void output() const;
    int get_year() const;
    int get_month() const;
    int get_day() const;
    bool equal()
private:
    int year;
    int month;
    int day;
};

//-----
Date::Date()
{
    year = 1970;
    month = 1;
    day = 1;
}

//-----
Date::Date(int y, int m, int d)
{
    year = y;
    month = m;
    day = d;
}

//-----
int Date::get_year() const
{
    return year;
}

//-----
int Date::get_month() const
{
    return month;
}

//-----
int Date::get_day() const
{
    return day;
}

```

```

//-
void Date::input( )
{
    cout << "year (number) ? ";
    cin >> year;
    cout << "month (number) ? ";
    cin >> month;
    cout << "day (number) ? ";
    cin >> day;
}

//-
void Date::output( ) const
{
    cout << year << "/" << month << "/" << day << endl;
}

//-
bool equal(const Date &date1, const Date & date2)
// NOTE:
// 1) DOES NOT include friend nor Date::
// 2) can access the private members (data and functions) of Date class
{
    return (
        date1.year == date2.year &&
        date1.month == date2.month &&
        date1.day == date2.day
    );
}

//-
int main( )
{
    Date date1(2014,5,14), date2;

    cout << "Date1 is ";
    date1.output( );

    cout << endl;
    cout << "Enter Date2:\n";
    date2.input( );
    cout << "Date2 is ";
    date2.output( );

    if (equal(date1, date2))
        cout << "date1 is equal to date2\n";
    else
        cout << "date1 is NOT equal to date2\n";
    return 0;
}

```

FRIEND FUNCTIONS

- Friend functions are not members of a class, but can access private member variables of the class
- A friend function is declared using the keyword friend in the class definition
- A friend function is not a member function
- A friend function is an ordinary function

FRIEND FUNCTION DECLARATION, DEFINITION & CALLING

- A friend function is declared as a friend in the class definition
- A friend function is defined as a nonmember function without using the "::" operator
- A friend function is called without using the '.' operator

ARE FRIEND FUNCTIONS NECESSARY ?

- Friend functions can be written as non-friend functions using the normal accessor and mutator functions that should be part of the class
- The code of a friend function is **simpler** and it is **more efficient**

WHEN TO USE FRIEND FUNCTIONS ?

- How do you know when a function should be a friend or a member function?
- In general, use a member function if the task performed by the function involves only one object
- In general, use a nonmember function if the task performed by the function involves more than one object
- Choosing to make the nonmember function a friend is a decision of efficiency and personal taste

FRIEND CLASSES

- Classes may also be declared friend of other classes.
- Declaring a class as a friend means that the friend class and all of its member functions have access to the private members of the other class
- General outline of how you set things:

```
class F; //forward declaration

class C
{
    public:
        ...
    friend class F;
        ...
};

class F
{
    ...
}

• Example:

class LinkedList;

class Node
{
    friend class LinkedList; // LinkedList can access private members of Node
    // ...
private:
    int value;
    Node *next;
};

class LinkedList
{
    // ...
}

• NOTE: friend classes may be "dangerous",  
because the designer of a class usually knows what friends are going to do,  
but cannot predict what a derived class might do (derived classes will be studied later).
```

OPERATOR OVERLOADING

```
/*
    OPERATOR OVERLOADING example / MAIN.CPP
    An example with fractions
*/
#include "fraction.h"

int main()
{
    // Testing constructors
    Fraction a; // Value is 0/1
    Fraction b(4); // Value is 4/1
    Fraction c(6,8); // Value is 6/8, which is converted to 3/4

    // Overloading output operator
    cout << "overloading output operator\n";
    cout << " a = " << a << endl;
    cout << " b = " << b << endl;
    cout << " c = " << c << endl;
    cout << endl;

    // Using default assignment operator and the default copy constructor
    cout << "using default copy constructor & assignment operator\n";

    Fraction d(c); // d is copy of c
    cout << " Fraction d(c): d = " << d << endl;
    // the copy constructor for class Fraction is invoked
    // unless the programmer provides one,
    // the compiler will automatically generate a copy constructor

    Fraction e;
    e = c; // the assignment operator is automatically generated
    cout << " e = c: e = " << e << endl;
    cout << endl;

    // Testing Overloaded arithmetic operators
    cout << "testing arithmetic operators\n";

    e = b + c;
    cout << " e = b + c = " << e << endl;

    Fraction f;
    f = b - c;
    cout << " f = b - c = " << f << endl;

    Fraction g = (b + (-c)); //unary arithmetic operator (minus)
    cout << " g = (b + (-c)) = " << g << endl;
    cout << endl;

    // Testing Overloaded comparison operators
    cout << "testing comparison operators\n";
    if (f == g)
        cout << " f == g; comparison test successful\n";
    else
        cout << " comparison test failed\n";
```

```

a = Fraction(6,8); //note 'a' already defined above
b = Fraction(16,8); //note 'b' already defined above
cout << "a = " << a << endl;
cout << "b = " << b << endl;
if (a < b)
    cout << " a < b ; comparison test successful\n";
else
    cout << " a < b ; comparison test failed\n";

// comparing a fraction and an integer
// NOTE: the Big C++ book is wrong when saying that one could write: if (b == 2)
if (b == Fraction(2))
    cout << " b == Fraction(2) ; comparison test successful\n";
else
    cout << " b == Fraction(2) ; comparison test failed\n";
cout << endl;

// Testing Overloaded input (and output)
cout << "overloading input (and output) operator\n";
cout << " fraction c ? ";
cin >> c;
cout << " c = " << c << endl;

cout << " fraction d ? ";
cin >> d;
cout << " d = " << d << endl;
cout << endl;

// Testing Overloaded increment operators
cout << "testing increment operators\n";
e = c++;
cout << " c = " << c << " ; e = c++ = " << e << endl;

f = ++d;
cout << " d = " << d << " ; f = ++d = " << f << endl;
cout << endl;

// Testing Overloaded 'conversion to double' operator
cout << "testing 'conversion to double' operator\n";
cout << "double(a) = " << double(a) << endl;

return 0;
}

```

```

/*
FRACTION.H
OPERATOR OVERLOADING examples
Adapted from Big C++ book
*/
#ifndef FRACTION_H
#define FRACTION_H

#include <iostream>
using namespace std;

class Fraction
{
public:
    Fraction(); // construct fraction 0/1
    Fraction(int t); // construct fraction t/1
    Fraction(int t, int b); // construct fraction t/b
    int numerator() const; //return numerator value
    int denominator() const; //return denominator value
    void display() const; // displays fraction

    // Updates a fraction by adding in another fraction, 'right'
    // returns the update fraction
    Fraction& operator+=(const Fraction& right);

    // Increment fraction by 1.
    Fraction& operator++(); // Prefix form      : ++(++frac) is allowed!
    Fraction operator++(int unused); // Postfix form : but not (frac++)++
    // These operators, in addition to producing a result,
    // alter their argument value; for this reason they are
    // defined as member functions, not as ordinary functions.

    // Converts a fraction into a floating-point value.
    // returns the converted value
    operator double() const; // NOTE: do not specify a return type
                            // return type is implicit in the name

    // Compare one fraction value to another.
    // Result is negative if less than right,
    // zero if equal, and positive if greater than 'right'.
    int compare(const Fraction& right) const;

private:
    // Place the fraction in least common denominator form.
    void normalize();

    // Compute the greatest common denominator of two integers.
    int gcd(int n, int m);

    int top; // fraction numerator
    int bottom; //fraction denominator
};

// other operators defined as ordinary functions
// ... but they can also be defined as member functions (see later)
Fraction operator+(const Fraction& left, const Fraction& right);
Fraction operator-(const Fraction& left, const Fraction& right);
Fraction operator*(const Fraction& left, const Fraction& right);
Fraction operator/(const Fraction& left, const Fraction& right);
Fraction operator-(const Fraction& value); // unary minus

```

```

bool operator==(const Fraction& left, const Fraction& right);
//bool operator==(const Fraction& left, int intValue);
bool operator!=(const Fraction& left, const Fraction& right);
bool operator<(const Fraction& left, const Fraction& right);
bool operator<=(const Fraction& left, const Fraction& right);
bool operator>(const Fraction& left, const Fraction& right);
bool operator>=(const Fraction& left, const Fraction& right);

// These two operators CAN'T BE defined as member functions. WHY?
// Compare the first parameters of the above functions and those of the following ones
ostream& operator<<(ostream& out, const Fraction& value);
istream& operator>>(istream& in, Fraction& value);

#endif

```

QUESTION:

why is compare() a public method of class Fraction?

ANSWER: because it is used in the operator overloading functions that are not members os class Fraction

```

/*
FRACTION.CPP
OPERATOR OVERLOADING examples
Adapted from Big C++ book
*/
#include "fraction.h"
#include <string>
#include <iostream>
#include <cassert> // #define NDEBUG before #include <cassert> ⇔ comment assert() calls
//#include <stdexcept>

//-----
// example of constructor with Field Initializer List
Fraction::Fraction() : top(0), bottom(1) { }

//-----
Fraction::Fraction(int t) : top(t), bottom(1) { }

//-----
Fraction::Fraction(int t, int b) : top(t), bottom(b)
{
    normalize();
}

//-----
// When function bodies are very short, the function may be declared 'inline'
// Alternatively the body of the function
// may be inserted directly into the class declaration (without 'inline')
// Although usually running more efficiently, they consume more storage
// NOTE: THE COMPILER MAY IGNORE THE "inline" HINT ...
inline int Fraction::numerator() const
{
    return top;
}

//-----
inline int Fraction::denominator() const
{
    return bottom;
}

//-----
inline void Fraction::display() const
{
    cout << top << "/" << bottom;
}

//-----
void Fraction::normalize()
{
    // Normalize fraction by
    // (a) moving sign to numerator
    // (b) ensuring numerator and denominator have no common divisors

    int sign = 1;

    if (top < 0)
    {
        sign = -1;
        top = - top;
    }
}

```

```

    if (bottom < 0)
    {
        sign = - sign;
        bottom = - bottom;
    }

    assert(bottom != 0);

    int d = 1;
    if (top > 0) d = gcd(top, bottom);
    top = sign * (top / d);
    bottom = bottom / d;
}

//-----
int Fraction::gcd(int n, int m)
{
    // Euclid's Greatest Common Divisor algorithm

    assert((n > 0) && (m > 0));

    while (n != m)
    {
        if (n < m)
            m = m - n;
        else
            n = n - m;
    }
    return n;
}

//-----
Fraction operator+(const Fraction& left, const Fraction& right)
{
    Fraction result(
        left.numerator() * right.denominator() +
        right.numerator() * left.denominator(),
        left.denominator() * right.denominator());
    return result;
}
// ALTERNATIVE: no local variable is created;
// the result is constructed as an unnamed temporary
/*
Fraction operator+(const Fraction& left, const Fraction& right)
{
    return Fraction ( left.numerator() * right.denominator() +
        right.numerator() * left.denominator(),
        left.denominator() * right.denominator());
}
*/
//-----
Fraction operator-(const Fraction& left, const Fraction& right)
{
    Fraction result(
        left.numerator() * right.denominator() -
        right.numerator() * left.denominator(),
        left.denominator() * right.denominator());
    return result;
}

```

```

//-----
Fraction operator*(const Fraction& left, const Fraction& right)
{
    Fraction result(
        left.numerator() * right.numerator(),
        left.denominator() * right.denominator());
    return result;
}

//-----
Fraction operator/(const Fraction& left, const Fraction& right)
{
    Fraction result(
        left.numerator() * right.denominator(),
        left.denominator() * right.numerator());
    return result;
}

//-----
Fraction operator-(const Fraction& value) // Unary minus
{
    Fraction result(-value.numerator(), value.denominator());
    return result;
}

//-----  

// NOTE: the comparison operators, below, are written using 'compare'
int Fraction::compare(const Fraction& right) const
{
    return
        numerator() * right.denominator() -
        denominator() * right.numerator();
    // Return the numerator of the difference
}

//-----
bool operator==(const Fraction& left, const Fraction& right)
{
    return left.compare(right) == 0;
}

/*
// To allow comparison of a Fraction and an integer; see comment in main()
bool operator==(const Fraction& left, int intValue)
{
    return ((static_cast<double> (left.numerator()) / left.denominator()) ==
            (static_cast<double> (intValue)));
}
*/

//-----
bool operator!=(const Fraction& left, const Fraction& right)
{
    return left.compare(right) != 0;
}

```

```

//-
bool operator<(const Fraction& left, const Fraction& right)
{
    return left.compare(right) < 0;
}

//-
bool operator<=(const Fraction& left, const Fraction& right)
{
    return left.compare(right) <= 0;
}

//-
bool operator>(const Fraction& left, const Fraction& right)
{
    return left.compare(right) > 0;
}

//-
bool operator>=(const Fraction& left, const Fraction& right)
{
    return left.compare(right) >= 0;
}

//-
// NOTE:
// The operators << and >> return the stream value as the result
// This allows "complex" stream expressions like "cout << frac1 << endl";
// (see examples in main() )
//-
ostream& operator<<(ostream& out, const Fraction& value)
{
    out << value.numerator() << "/" << value.denominator();
    return out; //NOTE THE RETURN VALUE. Why is this done ?
}
// This function could have been declared 'friend' of class Fraction
// Would it have any advantage ?
// class Fraction {
//     friend ostream& operator<<(ostream& out, const Fraction& value);

/*
ostream& operator<<(ostream& out, const Fraction& value)
{
    out << value.top << "/" << value.bottom;
    return out;
}
*/
//-
istream& operator>>(istream& in, Fraction& r) // NOTE: 'r' is non-const
{
    string fractionString;
    char fracSymbol;
    int num;
    int denom;

    getline(in,fractionString); // must be in format 'numerator/denominator'

    istringstream fractionStrStream(fractionString);
    fractionStrStream >> num >> fracSymbol >> denom;
}

```

```

assert(fracSymbol == '/'); // input must be inserted correctly !!!
assert(denom != 0); // otherwise KABOOM !!! ... \ / ...

r = Fraction(num, denom);
return in;
}

//-
//NOTE: do not specify a return type; it is implicit in the name
Fraction::operator double() const
{
    // Convert numerator to double, then divide
    return static_cast<double>(top) / bottom;
}

//-
Fraction& Fraction::operator++() // Prefix form
{
    top += bottom;
    normalize();
    return *this;
    //NOTE: returns the fraction after modification
    // as a reference to the current fraction
    // This enables a preincremented Fraction object
    // to be used as an 'lvalue';
    // ex: +++fraction; // equivalent to ++(++fraction);
    // OR
    // ++fraction *= 2; !!! ⇌ fraction = 2 * (fraction + 1)
    // to be consistent with C++ syntax
    // SEE EXAMPLE OF FUNCTIONS THAT RETURN REFERENCES IN THE NEXT PAGES
}

//-
//NOTE: the additional dummy parameter
Fraction Fraction::operator++(int unused) // Postfix form
{
    Fraction clone(top, bottom);
    top += bottom;
    normalize();
    return clone; //NOTE: returns the fraction before modification
}

//-
Operator()

//NOTE: the assignment operator will be automatically generated
// but +=, -=, *= and /= will not

Fraction& Fraction::operator+=(const Fraction& right)
{
    top = top * right.denominator() + bottom * right.numerator();
    bottom *= right.denominator();
    normalize();
    return *this;
}

```

Some binary operators (ex: operator+) could have been declared inside class Fraction

Instead of (declaration outside class Fraction) ...

```
//-----
// PREVIOUS IMPLEMENTATION (outside class Fraction)

class Fraction
{
public:
...

private:
...
    int top; // fraction numerator
    int bottom; //fraction denominator
};

//-----

Fraction operator+(const Fraction& left, const Fraction& right);
Fraction operator-(const Fraction& left, const Fraction& right);
Fraction operator-(const Fraction& value); // unary minus
...

//-----
Fraction operator+(const Fraction& left, const Fraction& right)
{
    Fraction result(
        left.numerator() * right.denominator() +
        right.numerator() * left.denominator(),
        left.denominator() * right.denominator());
    return result;
}

Fraction operator-(const Fraction& left, const Fraction& right)
{
    Fraction result(
        left.numerator() * right.denominator() -
        right.numerator() * left.denominator(),
        left.denominator() * right.denominator());
    return result;
}

...

Fraction operator-(const Fraction& value) // Unary minus
{
    Fraction result(-value.numerator(), value.denominator());
    return result;
}

//=====
```

... one could have (declaration inside class Fraction)

```
class Fraction
{
public:
    ...
    Fraction operator+(const Fraction& right);
    Fraction operator-(const Fraction& right);
    Fraction operator-(); // unary minus
    ...

private:
    ...
    int top; // fraction numerator
    int bottom; //fraction denominator
};

//-----

Fraction Fraction::operator+(const Fraction& right)
{
    Fraction result(
        top * right.denominator() +
        right.numerator() * bottom,
        bottom * right.denominator());
    return result;
}

Fraction Fraction::operator-(const Fraction& right)
{
    Fraction result(
        top * right.denominator() -
        right.numerator() * bottom,
        bottom * right.denominator());
    return result;
}

...

Fraction Fraction::operator-() // Unary minus
{
    Fraction result(-top, bottom);
    return result;
}
```

NOTE:

```
Fraction f1, f2, f3;  
...
```

```
f3 = add(f1,f2);
```

where

```
Fraction add(const Fraction& left, const Fraction& right)  
{  
    Fraction result(  
        left.numerator() * right.denominator() +  
        right.numerator() * left.denominator(),  
        left.denominator() * right.denominator());  
    return result;  
}
```

is equivalent to:

```
f3 = f1 + f2;
```

where

```
Fraction operator+(const Fraction& left, const Fraction& right)  
{  
    Fraction result(  
        left.numerator() * right.denominator() +  
        right.numerator() * left.denominator(),  
        left.denominator() * right.denominator());  
    return result;  
}
```

It is only a question of syntax ...

//-----
It is easier to read

```
f3 = f1 + f2*f2;
```

than

```
f3 = add(f1,multiply(f2,f2));
```

//-----

NOTES:

- `f3 = f1 + f2;`

will be interpreted by the compiler as (... one could have written the code like this !)
`f3 = operator+(f1,f2);` if operator+ is not a member function of class Fraction
or as

`f3 = f1.operator+(f2);` if operator+ is a member function of class Fraction

- overloaded () , [] , -> and assignment operators must be declared as class members.

THE "this" POINTER

- When defining member functions for a class, you sometimes want to refer to the calling object.
- The *this* pointer is a predefined pointer that **points to the calling object**
- Example:

```
class Fraction
{
public:
    ...
    Fraction operator+(const Fraction& right);
    Fraction operator-(const Fraction& right);
    Fraction operator-(); // unary minus
    ...
private:
    ...
    int top; // fraction numerator
    int bottom; //fraction denominator
};

//-----
```

```
Fraction Fraction::operator+(const Fraction& right)
{
    Fraction result(
        top * right.denominator() +
        right.numerator() * bottom,
        bottom * right.denominator());
    return result;
}
```

Could be written:

```
Fraction Fraction::operator+(const Fraction& right)
{
    Fraction result(
        (*this).numerator() * right.denominator() +
        right.numerator() * (*this).denominator(),
        (*this).denominator() * right.denominator());
    return result;
}
```

or as ...

```
Fraction Fraction::operator+(const Fraction& right)
{
    Fraction result(
        this->numerator() * right.denominator() +
        right.numerator() * this->denominator(),
        this->denominator() * right.denominator());
    return result;
}
```

- Another use:
when a parameter of a function member
has the same name as an attribute of the class
- (can be easily avoided)

```
class Fraction
{
public:
    Fraction(); // construct fraction 0/1
    Fraction(int t); // construct fraction t/1
    Fraction(int top, int bottom); // construct fraction t/b
    ...
private:
    ...
    int top; // fraction numerator
    int bottom; //fraction denominator
};
```

```
-----  
Fraction::Fraction(int top, int bottom)
{
    this->top = top;
    this->bottom = bottom;
    normalize();
}
```

The following code avoids the use of `this->top` and `this->bottom`.
It is syntactically correct but ...

```
-----  
Fraction::Fraction(int top, int bottom) : top(top), bottom(bottom)
{
    normalize();
}
```

- Yet another use:
as we saw, it was not necessary to overload the assignment operator, **operator=**,
for class **Fraction**
- But, when **operator=** is overloaded, it must return the ***this** object

```

class Fraction
{
public:
    Fraction(); // construct fraction 0/1
    Fraction(int t); // construct fraction t/1
    Fraction(int t, int b); // construct fraction t/b
    ...
    // operator= has to be a member of the class (because the language requires so)
    // it can't be a friend of the class
    Fraction& operator=(const Fraction& right);
    ...

private:
    ...
    int top; // fraction numerator
    int bottom; //fraction denominator
};

Fraction & Fraction::operator=(const Fraction &right)
{
    top = right.numerator();
    bottom = right.denominator();
    return *this;
}

```

The primary use of **this** pointer is

- to return the current object,
- or to pass the object to a function.

Returning the left hand object is necessary if one wants to do
multiple assignment operations
(returned as a reference for better efficiency)

f1 = f2 = f3;

// RETURNING POINTERS AND REFERENCES

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

class Date {
public:
    Date();
    Date & setDay(int d);
    Date & setMonth(int m);
    Date & setYear(int y);
    void show() const;
private:
    int day, month, year;
};

Date::Date()
{
    day = month = year = 1;
}

// updates 'day' and returns a reference to 'day' ...
Date & Date::setDay(int d)
{
    day = d;
    return *this;
}

Date & Date::setMonth(int m)
{
    month = m;
    return *this;
}

Date & Date::setYear(int y)
{
    year = y;
    return *this;
}

void Date::show() const
{
    cout << day << "/" << month << "/" << year << endl;
}

void main()
{
    Date d;

    // ... thus enabling the use of cascaded 'set_operations':
    d.setDay(10).setMonth(5).setYear(2016);
    d.show();
}
```

TO DO:

- REPLACE Date & BY Date AND INTERPRET RESULT
- THEN, TRY d.setDay(4).setMonth(5).setYear(2020).show();

```

// MORE EXAMPLES OF FUNCTIONS THAT RETURN REFERENCES OR POINTERS TO OBJECTS

#include <iostream>

using namespace std;

class Position
{
public:
    Position(int x, int y);
    int getX() const;
    int getY() const;
    void setX(int x);
    void setY(int y);
private:
    int x, y;
};

Position::Position(int x, int y)
{
    this->x = x;
    this->y = y;
}

int Position::getX() const
{
    return x;
}

int Position::getY() const
{
    return y;
}

void Position::setX(int x)
{
    this->x = x;
}

void Position::setY(int y)
{
    this->y = y;
}

ostream& operator<<(ostream& out, const Position &p)
{
    out << "(" << p.getX() << "," << p.getY() << ")";
    return out;
}

//-----
int main()
{
    Position p1(1,1);
    cout << "p1 = " << p1 << endl;
    p1.setX(10);
    p1.setY(20);
    cout << "p1 = " << p1 << endl;
}

```

```

//=====

#include <iostream>

using namespace std;

class Position
{
public:
    Position(int x, int y);
    int getX() const;
    int getY() const;
    Position get() const; // IN FACT, NOT NEEDED; TO GET A COPY, JUST DO p2=p1 ...
    void setX(int x);
    void setY(int y);
private:
    int x, y;
};

Position::Position(int x, int y)
{
    this->x = x;
    this->y = y;
}

int Position::getX() const
{
    return x;
}

int Position::getY() const
{
    return y;
}

Position Position::get() const
{
    return *this;
}

void Position::setX(int x)
{
    this->x = x;
}

void Position::setY(int y)
{
    this->y = y;
}

ostream& operator<<(ostream& out, const Position &p)
{
    out << "(" << p.getX() << "," << p.getY() << ")";
    return out;
}

```

```
//-----
int main()
{
    Position p1(1, 1);
    cout << "p1 = " << p1 << endl;
    p1.setX(10);
    p1.setY(20);

    Position p2 = p1.get(); // ⇔ Position p2 = p1;
    cout << "p2 = " << p2 << endl;
    p2.setX(30);
    p2.setY(40);

    cout << endl;
    cout << "p1 = " << p1 << endl;
    cout << "p2 = " << p2 << endl;
}
```

```

//=====
#include <iostream>
using namespace std;

class Position
{
public:
    Position(int x, int y);
    int getX() const;
    int getY() const;
    Position get() const;
    void setX(int x);
    void setY(int y);
private:
    int x, y;
};

Position::Position(int x, int y)
{
    this->x = x;
    this->y = y;
}
int Position::getX() const
{
    return x;
}
int Position::getY() const
{
    return y;
}
Position Position::get() const
{
    return *this;
}
void Position::setX(int x)
{
    this->x = x;
}
void Position::setY(int y)
{
    this->y = y;
}
ostream& operator<<(ostream& out, const Position &p)
{
    out << "(" << p.getX() << "," << p.getY() << ")";
    return out;
}

//-----
int main()
{
    Position p1(1, 1);
    cout << "p1 = " << p1 << endl;

    p1.get().setX(100); // NOTE THIS
    p1.get().setY(200); // NOTE THIS
    cout << endl;
    cout << "p1 = " << p1 << endl;
}

```

```

//=====
#include <iostream>

using namespace std;

class Position
{
public:
    Position(int x, int y);
    int getX() const;
    int getY() const;
    Position& get(); // NOTE: not const
    void setX(int x);
    void setY(int y);
private:
    int x, y;
};

Position::Position(int x, int y)
{
    this->x = x;
    this->y = y;
}
int Position::getX() const
{
    return x;
}
int Position::getY() const
{
    return y;
}
Position& Position::get()
{
    return *this;
}
void Position::setX(int x)
{
    this->x = x;
}
void Position::setY(int y)
{
    this->y = y;
}
ostream& operator<<(ostream& out, const Position &p)
{
    out << "(" << p.getX() << "," << p.getY() << ")";
    return out;
}

//-----
int main()
{
    Position p1(1, 1);
    cout << "p1 = " << p1 << endl;

    p1.get().setX(100); // NOTE THIS
    p1.get().setY(200); // NOTE THIS

    cout << endl;
    cout << "p1 = " << p1 << endl;
}

```

```

//=====
#include <iostream>

using namespace std;

class Position
{
public:
    Position(int x, int y);
    int getX() const;
    int getY() const;
    Position& setX(int x);
    Position& setY(int y);
private:
    int x, y;
};

Position::Position(int x, int y)
{
    this->x = x;
    this->y = y;
}

int Position::getX() const
{
    return x;
}

int Position::getY() const
{
    return y;
}

Position Position::setX(int x)
{
    this->x = x;
    return *this;
}

Position& Position::setY(int y)
{
    this->y = y;
    return *this;
}

ostream& operator<<(ostream& out, const Position &p)
{
    out << "(" << p.getX() << "," << p.getY() << ")";
    return out;
}

//-----
int main()
{
    Position p1(1, 1);
    cout << "p1 = " << p1 << endl;
    p1.setX(30).setY(40);
    cout << "p1 = " << p1 << endl;
}

```

```

//=====
#include <iostream>

using namespace std;

class Position
{
public:
    Position(int x, int y);
    int getX() const;
    int getY() const;
    Position* setX(int x);
    Position* setY(int y);
private:
    int x, y;
};

Position::Position(int x, int y)
{
    this->x = x;
    this->y = y;
}

int Position::getX() const
{
    return x;
}

int Position::getY() const
{
    return y;
}

Position* Position::setX(int x)
{
    this->x = x;
    return this;
}

Position* Position::setY(int y)
{
    this->y = y;
    return this;
}

ostream& operator<<(ostream& out, const Position &p)
{
    out << "(" << p.getX() << "," << p.getY() << ")";
    return out;
}

//-----
int main()
{
    Position p1(1, 1);
    cout << "p1 = " << p1 << endl;
    Position *p1Ptr = &p1;
    p1Ptr->setX(30)->setY(40);
    //(*p1Ptr).setX(30).setY(40);
    cout << "p1 = " << p1 << endl;
}

```

CONTAINERS & OPERATOR OVERLOADING

```
// CONTAINERS: a set of random 'int's

#include <iostream>
#include <cstdlib>
#include <ctime>
#include <set>

using namespace std;

//-
int main()
{
    set<int> s;

    srand((unsigned) time(NULL));

    for (int i=1; i<=20; i++)
        s.insert(rand()%10);

    for (set<int>::const_iterator i=s.begin(); i!=s.end(); i++)
        cout << *i << endl;

    // NOTES:
    // 1- the number os elements in the set may be less than 20
    // 2- the elements of the set are ordered
    //
    // For that the operator < must be defined.
    // It is already defined for 'int'
}
```

TO DO BY STUDENTS:

- Generate a single bet in EuroMillions (5 + 2 numbers) using sets.

```

// CONTAINERS & THE NEED FOR OPERATOR OVERLOADING
// CONTAINERS: a set of 'Person'
// TRY TO COMPILE THIS PROGRAM AND SEE WHAT HAPPENS

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

class Person
{
public:
    Person();
    Person(string pName, unsigned pAge);
    string getName() const { return name; };
    unsigned getAge() const { return age; };
    void setName(string pName) { name=pName; };
    void setAge(unsigned pAge) { age=pAge; };
private:
    string name;
    unsigned age;
};

//-----
Person::Person()
{
    name = "";
    age = 0;
}

//-----
Person::Person(string pName, unsigned pAge)
{
    name = pName;
    age = pAge;
}

//-----
int main()
{
    set<Person> s;
    Person p;
    string name;
    unsigned age;

    for (int i=1; i<=3; i++)
    {
        cout << "name age " << i << " ? ";
        cin >> name >> age;

        p.setName(name);
        p.setAge(age);
        s.insert(p);
    }

    cout << endl;
    for (set<Person>::const_iterator i=s.begin(); i!=s.end(); i++)
        cout << setw(10) << left << i->getName() << " " << i->getAge()
    << endl;
}

```

```

// THE PREVIOUS PROGRAM GENERATES A COMPILER ERROR
// BECAUSE OPERATOR < IS NOT DEFINED FOR CLASS Person
#include <iostream>
#include <iomanip>
#include <string>
#include <set>

using namespace std;

class Person
{
    friend bool operator<(const Person& left, const Person& right);
public:
    Person();
    Person(string pName, unsigned pAge);
    string getName() const { return name; } //const because of const_iterator in main
    unsigned getAge() const { return age; } //const because of const_iterator in main
    void setName(string pName) { name=pName; };
    void setAge(unsigned pAge) { age=pAge; };
private:
    string name;
    unsigned age;
};

//-----
Person::Person()
{
    name = "";
    age = 0;
}

//-----
Person::Person(string pName, unsigned pAge)
{
    name = pName;
    age = pAge;
}

//-----
bool operator<(const Person& left, const Person& right)
{
    return left.name < right.name; // OR left.age < right.age; you decide
}

//-----
int main()
{
    set<Person> persons;

    for (int i = 1; i <= 3; i++)
    {
        string name;
        unsigned age;
        cout << "name age " << i << " ? ";
        cin >> name >> age;
        persons.insert(Person(name,age));
    }
    cout << endl;
    for (auto p : persons)
        cout << setw(10) << left << p.getName() << " " << p.getAge() << endl;
}

```

NOTES:

- the comparison function, that implements **operator<**, must yield false when we compare a key with itself.
Moreover,
 - if we compare two keys, they cannot both be "less than" each other,
 - and if k1 is "less than" k2, which in turn is "less than" k3,
then k1 must be "less than" k3
- it is **not** necessary to define **operator==** and **operator!=**

ANOTHER ALTERNATIVE:

```
...
// THE SAME CODE AS BEFORE
...
//-----
bool comparePersons(const Person &p1, const Person &p2)
{
    return p1.getName() < p2.getName();
};

//-----
int main()
{
    set<Person, bool (*)(const Person &p1, const Person &p2)> s(&comparePersons);
    //                      function pointer as "compare"
    Person p;
    string name;
    unsigned age;

    ...
}
```

NOTE:

- an identical compiling error would occur if, for example, you wanted to declare a map whose key is a Person.

OVERLOADING THE () FUNCTION CALL OPERATOR / FUNCTION OBJECTS

(not lectured in 2021/2022)

- A **function object** (or **functor**) is an instance of a class (an object) that defines the **function call operator**: **operator()**
- Once the object is created, it **can be invoked as you would invoke a function** that's why it is termed a function object.
- Function objects are **used** extensively **by various generic STL algorithms**.
- The function call operator **can only be defined as a member function**.
- The same happens with the assignment operator, **operator=** (later, we shall see an example of **operator=** implementation, for our own **String** class)

```

// Overloading the function call operator

#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

class RandomInt
{
public:
    RandomInt(int a, int b); // constructor
    int operator(); // function call operator
private:
    int limInf, limSup; // interval limits: [limInf..limSup]
};

//-----
RandomInt::RandomInt(int a, int b)
{
    limInf = a; limSup = b;
}

//-----
int RandomInt::operator()
{
    return limInf + rand() % (limSup - limInf + 1);
}

//-----
int main()
{
    //srand((unsigned) time(NULL));

    RandomInt r(1,10); // create an object of type RandomInt (a FUNCTION OBJECT),
                        // initializing the limits of the interval to 1 and 10

    // once the object is created,
    // it can be invoked as you would invoke a function
    // that's why it is termed a FUNCTION OBJECT

    for (int i=1; i<=10; i++)
        cout << r() << endl;
}

return 0;
}

```

A **FUNCTION OBJECT**
is an instance of a class that defines the function call operator

```

// Overloading the function call operator
// Generalizing the random number generator from the previous example

#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

class RandomInt
{
public:
    RandomInt(int a, int b);
    int operator()();
    int operator()(int b); // overloading the function call operator
    int operator()(int a, int b);

private:
    int limInf, limSup; // interval limits: [limInf..limSup]
};

//-----
RandomInt::RandomInt(int a, int b)
{
    limInf = a; limSup = b;
}

//-----
int RandomInt::operator()
{
    return limInf + rand() % (limSup - limInf + 1);
}

//-----
int RandomInt::operator()(int b)
{
    return limInf + rand() % (b - limInf + 1);
}

//-----
int RandomInt::operator()(int a, int b)
{
    return a + rand() % (b - a + 1);
}

//-----
int main()
{
    srand((unsigned) time(NULL));

    RandomInt r(1,10);

    cout << r() << endl;
    cout << r(100) << endl;
    cout << r(20,25) << endl;
    cout << r() << endl;
}

```

- Function objects are used extensively by various generic STL algorithms.

- In a previous example, we saw how to generate a sequence of random numbers in the interval [1..10].

```
int myRand()
{
    return 1 + rand() % 10;
}

...
int main()
{
    ...
    vector<int> v(10);
    generate(v.begin(), v.end(), myRand);
    displayVec("generate(...,myRand)", v);
    ...
}
```

- Suppose that one would like to generate a sequence in which the limits of the interval are set at run time.

- One could be tempted to do

```
int myRand(int a, int b)
{
    return a + rand() % (b - a + 1);
}

int main()
{
    vector<int> v(10);
    int limInf, limSup;
    cout << "limInf ? "; cin >> limInf;
    cout << "limSup ? "; cin >> limSup;
    generate(v.begin(), v.end(), myRand(limInf, limSup));
    ...
}
```

- This is **syntactically incorrect**; it will generate a compile error ...

- One could define 'limInf' and 'limSup' as global variables
but this is **not** a recommended solution

```
// STL - ALGORITHMS

#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cstdlib>

using namespace std;

int limInf, limSup; // GLOBAL VARIABLES :-(
void displayVec(string title, const vector<int> &v)
{
    cout << title << ":";
    for (size_t i=0; i<v.size(); i++)
        cout << setw(3) << v.at(i) << " ";
    cout << endl << endl;
}

int myRand()
{
    return limInf + rand() % (limSup - limInf + 1); // :-(

}

int main() {
    srand((unsigned) time(NULL));
    vector<int> v(10);

    cout << "limInf ? "; cin >> limInf;
    cout << "limSup ? "; cin >> limSup;

    generate(v.begin(), v.end(), myRand);
    displayVec("random numbers", v);

    return 0;
}
```

- The most commonly used solution is to use a **function object** as 3rd parameter to the **generate()** algorithm:

```
// FUNCTION OBJECTS & STL ALGORITHMS

#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cstdlib>

using namespace std;

//-----
//-----

class RandomInt
{
public:
    RandomInt(int a, int b);
    int operator();
private:
    int limInf, limSup; // interval limits: [limInf..limSup]
};

//-----
RandomInt::RandomInt(int a, int b)
{
    limInf = a; limSup = b;
}

//-----
int RandomInt::operator()
{
    return limInf + rand() % (limSup - limInf + 1);
}

//-----
//-----
void displayVec(string title, const vector<int> &v)
{
    cout << title << ":" ;
    for (size_t i=0; i<v.size(); i++)
        cout << setw(3) << v.at(i) << " ";
    cout << endl << endl;
}

//-----
//-----
int main() {
    srand((unsigned) time(NULL));
    vector<int> v(10);
    int limInf, limSup;
```

```

cout << "limInf ? "; cin >> limInf;
cout << "limSup ? "; cin >> limSup;

RandomInt r(limInf,limSup); //instantiates object and sets limits
generate(v.begin(),v.end(),r);

// ALTERNATIVE:
// using an unnamed temporary that will be destroyed at the end of the call
// generate(v.begin(),v.end(),RandomInt(limInf,limSup));

displayVec("random numbers",v);

return 0;
}

```

- Now, each time `generate()` calls its function parameter, it uses the call operator from object 'r'.

```

// CONTAINERS & FUNCTION OBJECTS
// An alternative way for sorting the set<Person> by name (or by age)
// (see previous example, about sets and operator< overloading for Person)
// is to create a function object, SortPersonByName,
// that defines the ordering, instead of overloading operator< for Person

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

using namespace std;

class Person
{
public:
    Person();
    Person(string pName, unsigned pAge);
    string getName() const { return name; }
    unsigned getAge() const { return age; }
    void setName(string pName) { name=pName; }
    void setAge(unsigned pAge) { age=pAge; }
private:
    string name;
    unsigned age;
};

//-
class SortPersonByName
{
public:
    bool operator()(const Person &left, const Person &right) const;
};

bool SortPersonByName::operator()(const Person &left, const Person &right) const
{
    return left.getName() < right.getName();
}

//-
Person::Person()
{
    name = "";
    age = 0;
}

//-
Person::Person(string pName, unsigned pAge)
{
    name = pName;
    age = pAge;
}

//-
int main()
{
    set<Person, SortPersonByName> s;
    Person p;
    string name;
    unsigned age;
}

```

```

for (int i=1; i<=3; i++)
{
    cout << "name age " << i << " ? ";
    cin >> name >> age;

    p.setName(name);
    p.setAge(age);

    s.insert(p);
}

cout << endl;
for (set<Person,SortPersonByName>::const_iterator i=s.begin();
i!=s.end(); i++)
    cout << setw(10) << left << i->getName() << " " << i->getAge()
<< endl;
}

```

LINK:

<http://www.cplusplus.com/reference/stl/set/set/>

```

template < class T,
          class Compare = less<T>,
          class Alloc = allocator<T>
        > class set;                                // set::key_type/value_type
                                                // set::key_compare/value_compare
                                                // set::allocator_type

```

T

Type of the elements.

Each element in a `set` container is also uniquely identified by this value (each value is itself also the element's key).

Compare

A binary predicate that takes two arguments of the same type as the elements and returns a `bool`.

The expression `comp(a,b)`, where `comp` is an object of this type and `a` and `b` are key values, shall return `true` if `a` is considered to go before `b` in the *strict weak ordering* the function defines.

The `set` object uses this expression to determine both the order the elements follow in the container and whether two element keys are equivalent

(by comparing them reflexively: they are equivalent if `!comp(a,b) && !comp(b,a)`).

No two elements in a `set` container can be equivalent.

This can be a function pointer or a function object (see [constructor](#) for an example).

This defaults to [`less<T>`](#), which returns the same as applying the *less-than operator* (`a < b`)

Alloc

Type of the allocator object used to define the storage allocation model. By default, the [`allocator`](#) class template is used, which defines the simplest memory allocation model and is value-independent.

NOTE:

- in a `map` declaration it is also possible to indicate a function object that is used for specifying the ordering of the elements of the `map`

```

template < class Key,                                // map::key_type
          class T,                                    // map::mapped_type
          class Compare = less<Key>,                // map::key_compare
          class Alloc = allocator<pair<const Key, T> > // map::allocator_type
        > class map;

```

MORE ON ... (not lectured in 2021/2022)

... OVERLOADING: copy constructor / operator= / operator[]

... DESTRUCTORS: necessary when dynamic memory is allocated

- **COPY CONSTRUCTORS**

- By default, when an object is used to **initialize** another, C++ performs a **bitwise copy**, that is an identical copy of the initializing object is created in the target object
ex:
 - MyClass obj1 = obj2;
 - MyClass obj1(obj2);
- Although this is perfectly adequate for many cases – and generally exactly what you want to happen –
there are situations in which a bitwise copy should not be used.
- One of the most common is when an object allocates memory dynamically when it is created.
- A **copy constructor** is a constructor that takes as **parameter** a **constant reference** to an object of the same class

```
// mystring.h
// a class from emulating C++ strings
// JAS

#ifndef _MYSTRING
#define _MYSTRING

// using namespace std; // should be avoided in header files because it implies
// that the namespace will be included in every file that includes this header file

class String
{
    friend std::ostream& operator<<( std::ostream& out, const String& right);
    friend bool operator==(const String& left, const String& right);
    friend String operator+(const String& left, const String& right);

public:
    String(); // default constructor
    String(const char s[]); // Simple constructor
    String(const String& right); // Copy constructor
    ~String(); // Destructor
    String& operator=(const String& right); // Assignment operator
    char& operator[](int index); // WHEN IS EACH VERSION OF operator[] USED ?
    char operator[](int index) const;
    int length() const;
private:
    char* buffer; // space to be allocated must include '\0' string terminator
    int len; // perhaps, could be avoided ...?
};

#endif
```

```

// mystring.cpp

// a class from emulating C++ strings (implementation)
// JAS

#include <iostream>
#include <cassert>
#include "mystring.h"

using namespace std;

//-----//
// DEFAULT CONSTRUCTOR (constructs an empty string)
String::String()
{
    cout << "DEFAULT CONSTRUCTOR\n"; //JUST FOR EXECUTION TRACKING

    len = 0;
    buffer = NULL; // No need to allocate space for empty strings
}

//-----//
// SIMPLE CONSTRUCTOR (constructs string from array of chars)
String::String(const char s[])
{
    cout << "SIMPLE CONSTRUCTOR from array of chars |" << s << "|\\n";

    // Determine number of characters in string (alternative:strlen(s))
    len = 0;
    while (s[len] != '\\0')
        len++;

    // Allocate buffer array, remember to make space for the '\\0' character
    buffer = new char[len + 1];

    // Copy new characters (ALTERNATIVE: strcpy( buffer, s ))
    for (int i = 0; i < len; i++)
        buffer[i] = s[i];
    buffer[len] = '\\0'; //terminator could be avoided ... why ? ...
}

//-----//
// COPY CONSTRUCTOR
String::String(const String& right)
{
    cout << "COPY CONSTRUCTION from |" << right << "| " << endl;

    int n = right.length(); // ... right.len
    buffer = new char[n + 1];
    for (int i = 0; i < n; i++)
        buffer[i] = right[i];
    buffer[n] = '\\0';
    len = n;
}

```

```

//-----  

// ASSIGNMENT OPERATOR  

String& String::operator=(const String& right)  

{  

    cout << "OPERATOR= | " << right << " | " << endl;  

    if (this != &right) // NOTE THIS TEST (not "this" pointer...)  

    {  

        delete[] buffer; // Get rid of old buffer of 'this' object  

        len = right.length();  

        buffer = new char[len + 1];  

        for (int i = 0; i < len; i++)  

            buffer[i] = right[i];  

        buffer[len] = '\0';  

    }  

    return *this; // WHY IS THIS DONE ?  

    // NOTE: COULD RETURN 'String' INSTEAD OF 'String&'... but...  

    // ...MODIFY AND ANALYSE THE "cout" MESSAGES  

} // RETURN TYPE FROM OPERATOR= SHOULD BE THE SAME AS FOR THE BUILT-IN TYPES (C++Primer, 4th 3d, p.493)  

//-----  

// SUBSCRIPT OPERATOR FOR const OBJECTS (returns rvalue)  

char String::operator[](int index) const  

{  

    assert((index >= 0) && (index < len));  

    return buffer[index];  

}  

//-----  

// SUBSCRIPT OPERATOR FOR non-const OBJECTS (returns lvalue)  

char& String::operator[](int index)  

{  

    assert((index >= 0) && (index < len));  

    return buffer[index];  

} // NOTE: returns reference to class data members  

//-----  

// STRING LENGTH member function  

int String::length() const  

{  

    return len;  

}  

//-----  

// DESTRUCTOR - in this case, it is fundamental to have a destructor  

String::~String()  

{  

    if (buffer != NULL)  

        cout << "DESTRUCTION OF | " << buffer << " | " << endl;  

    else  

        cout << "NOTHING TO DESTRUCT\n";  

    if (buffer != NULL)  

        delete[] buffer;  

}

```

```

//-----  

// EQUALITY OPERATOR  

bool operator==(const String& left, const String& right)  

{  

    if (left.length() != right.length()) // OR left.len ... right.len  

        return false;  

    for (int i=0; i<left.length(); i++)  

        if (left.buffer[i] != right.buffer[i])  

            return false;  

    return true;  

}  

//-----  

String operator+(const String& left, const String& right)  

{  

    //if (right.length() == 0)  

    //    return left;  

    cout << "OPERATOR+ (" << left << "," << right << ")\n";  

    int newlen = left.length() + right.length();  

    // allocate space for temporary resulting string  

    char *tmpCStr = new char[newlen + 1]; // C-string  

    // concatenate the 2 strings  

    int pos = 0;  

    for (int i=0; i<left.length(); i++)  

        tmpCStr[pos++] = left.buffer[i];  

    for (int i=0; i<right.length(); i++)  

        tmpCStr[pos++] = right.buffer[i];  

    tmpCStr[pos] = '\0';  

    // create String object from temporary C-string  

    String tmpStr(tmpCStr); // invoke String simple constructor  

    // destroy temporary string  

    delete[] tmpCStr; // C-string  

    return tmpStr;  

}  

//-----  

// STRING OUTPUT OPERATOR  

std::ostream& operator<<(std::ostream& out, const String& right)  

{  

    int n = right.length();  

    for (int i=0, i<n; i++)  

        out << right[i];  

    return out;  

}

```

```
// My STRING CLASS  
// a class from emulating C++ strings (implementation)  
// JAS
```

```
// A program for testing my "String class"  
// main.cpp
```

```
#include <iostream>  
#include "mystring.h"
```

```
using namespace std;
```

```
-----  
int main(void)  
{
```

```
    cout << "String s0; - "  
    String s0;
```

```
    cout << "String s1 = \"ABC\"; - "  
    String s1 = "ABC";
```

```
    cout << "String s2(\"DEF\"); - "  
    String s2("DEF");
```

```
    char s[] = "GHI";  
    cout << "String s3(s); - "  
    String s3(s);
```

```
    cout << "String s4 = s1; - "  
    String s4 = s1;
```

```
// UNCOMMENT AND INTERPRET WHAT HAPPENS
```

```
/*  
cout << "s0 = s1; - "  
s0 = s1;  
*/
```

```
// UNCOMMENT AND INTERPRET WHAT HAPPENS
```

```
//cout << "-----\n";  
//cout << "s0 = s1 + s2; - "  
//s0 = s1 + s2;
```

```
cout << "-----\n";  
cout << "s0 = " << s0 << endl;  
cout << "s1 = " << s1 << endl;  
cout << "s2 = " << s2 << endl;  
cout << "s3 = " << s3 << endl;  
cout << "s4 = " << s4 << endl;  
cout << "-----\n";
```

```
cout << "s4[0] = " << s4[0] << endl;
```

```
cout << "modifying s4[0] = a\n";
```

```
s4[0] = 'a';
```

```
cout << "s4 = " << s4 << endl;
```

```
if (s1 == s4)
```

```
    cout << "s1 EQUAL TO s4\n";
```

```
else
```

```
    cout << "s1 NOT EQUAL TO s4\n";
```

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

```
}
```

- WHEN IS A DESTRUCTOR NEEDED ?

- If no destructor is provided, a default destructor will be automatically generated.
The **default destructor** has an empty body, that is, it **performs no actions**.
- A **destructor** is only **necessary** if an object requires some kind of resource management.
- The most common housekeeping task is to avoid a memory leak by releasing any **dynamically allocated memory**.

- TWO SITUATIONS WHERE THE VALUE OF ONE OBJECT IS GIVEN TO ANOTHER

- C++ defines **2 distinct types of situations in which the value of one object is given to another:**
 - **initialization**
 - **assignment**
- **initialization** (=> **copy constructor is invoked**)
can occur any of **3 ways**
 - when an object explicitly initializes another, such in a declaration
 - `MyClass x = y;`
 - `MyClass x(y);`
 - when a copy of an object is made to be passed to a function
 - `func(y);`
 - when a temporary object is generated
(most commonly, as a return value)
 - `y = func(); // y receiving a temporary returned object`
 - note: in this case **assignment** operator is **also invoked**
- **assignment** (=> **operator= is invoked**)
 - `MyClass x;`
 - `MyClass y;`
 - `...`
 - `x = y;`

- THE "BIG THREE "

- The **assignment operator**, **copy constructor** and **destructor** are collectively called "the "big three".
- A simple **rule of thumb** is that if you define a **destructor** then you should always provide a **copy constructor** and an **assignment operator**, and make all three perform in a similar fashion.
 - Analyse what would happen if in the just implemented String class we had defined a **destructor** but had forgotten to define the **copy constructor** (a **copy constructor** would be automatically generated for us):

```
String a = "Peter";
...
{
    String b = a; // memberwise copy;
                   // buffer[] for a and b is the same
    ...
} //destructor b.~String is invoked, a.buffer[] is deleted
```

- You must implement them for any class that manages heap memory.
- The equivalence of a **copy constructor** and the **assignment operator** is clear:
 - both are initializing a new value using an existing value.
- But the **assignment operator** is both **deleting** an old value and **creating** a new one.
You must make sure the first part of this task matches the action of the **destructor**.

NAMESPACES

Namespace concept

- A namespace is a **collection of name definitions**, such as **class definitions**, **function definitions** and **variable declarations**
- If a program uses **classes and functions written by different programmers**, it may be that the **same name** is **used for different things**
- Namespaces help us deal with this problem

The "using" directive

- `#include <iostream>` places names such as **cin** and **cout** in the **std** namespace
- The program does not know about names in the **std** namespace until you add `using namespace std;`
- if you do not use the **std** namespace, you can define **cin** and **cout** to behave differently !!!

The global namespace

- Code that you write is in a namespace
 - it is in the **global namespace** unless you specify a namespace
 - the global name space is **referred to using just ::**
- The global namespace does not require the using directive

Creating and using a namespace

- To place code in a namespace, use a **namespace grouping**

```
namespace Namespace_Name
{
    // Some_Code
}
```

- To use the namespace created, use the appropriate **using directive**

```
using namespace Namespace_Name;
```

Declaring and defining functions in a namespace

- To add a function to a namespace,
declare the function in a namespace grouping

```
namespace ns1
{
    void greeting();
}
```

- To define a function declared in a namespace,
define the function in a namespace grouping

```
namespace ns1
{
    void greeting()
    {
        cout << "Hello!\n";
    }
}
```

Using a function

- To use a function defined in a namespace, include the **using directive** in the program where the namespace is to be used
- Call the function as the function would normally be called

```
int main( )
{
    {
        using namespace ns1;
        greeting();
    }
    //...
}
```

Name conflicts

- If the **same name** is used **in two namespaces** the namespaces cannot be used at the same time
- Example: If `my_function` is defined in namespaces `ns1` and `ns2`, the two versions of `my_function` **could be used** in one program by using local using directives this way:

{ using namespace ns1; my_function(); }	{ using namespace ns2; my_function(); }
--	--

- NOTE:
 - A using directive potentially introduces a name
 - If **ns1** and **ns2** both define **my_function()**

```
using namespace ns1;
using namespace ns2;
```

is **OK**, provided **my_function()** is never used!
 - Suppose you have the namespaces below:
- | | |
|---|---|
| <pre>namespace ns1 { fun1(); my_function(); }</pre> | <pre>namespace ns2 { fun2(); my_function(); }</pre> |
|---|---|
- Is there a way to use both namespaces considering that **my_function()** is in both?
 - Using declarations (not directives) allow us to select individual functions to use from namespaces
 - `using ns1::fun1; // makes only fun1 in ns1 available`
 - The scope resolution operator - `:` - identifies a namespace here
 - Means we are using only namespace ns1's version of fun1
 - If you only want to use the function once, call it like this:
 - `ns1::fun1();`
 - A using declaration (ex: `using std::cout;`) makes only one name available from the namespace
 - A using directive makes all the names in the namespace available
 - A using declaration introduces a name into your code: no other use of the name can be made

<code>using ns1::my_function;</code> <code>using ns2::my_function;</code>
--

 is **illegal**, even if **my_function** is never used.

Naming Namespaces

- To avoid choosing a name for a namespace that has already been used
 - Add your name initials to the name of the namespace (...?)
 - or, use some other unique, long string (the name of your company, ...)
 - You can define a short **alias for a long namespace** in the following way:
 - `namespace cv = Computer_Vision_Library_by_FEUPvisionary;`

Additional remarks

- Unlike classes, namespaces are *open*.
You can add as many items to a namespace as you like, simply by starting another namespace block.

```
namespace ns1
{
    class string
    {
        ...
    };

namespace ns1
{
    class map
    {
        ...
    };
}
```

- Writing programs with using directives introduces all the problems inherent in name collisions when using multiple libraries.
- While **using directives** can appear in both header and implementation files, some style guides suggest they **be avoided in header files**.
 - Including a namespace in a header file means the same namespace will be included in every file that incorporates the header file.
 - However, deciding which namespaces to use should be left to the final application developer, not the developer of the interface file.
- A using statement is subject to the same scope rules as declaration statements.

```
using namespace std;

void f()
{
    using ns1::string; // ns1:string version now shadows std::string
    ...
}
```

```

// NAMESPACES - usage example

#include <iostream>
using std::cout;

//-----
// Namespace declaration

namespace ns1
{
    void hello();
}

namespace ns2
{
    void hello();
}

//-----
int main()
{
    ns1::hello();
    ns2::hello();
}

//-----
// Namespace implementation

namespace ns1
{
    void hello()
    {
        cout << "Hello 1 !\n";
    }
}

namespace ns2
{
    void hello()
    {
        cout << "Hello 2 !\n";
    }
}

```

```

// NAMESPACES - usage example

#include <iostream>
#include <string>

using std::cout;
using std::endl;
using std::string;

// Namespace declaration & function implementation

namespace ns1
{
    string creator="Mary";

    void hello()
    {
        cout << "hello from " << creator << "!\n";
    }
}

namespace ns2
{
    string creator="John";

    void hello()
    {
        cout << "hello from " << creator << "!\n";
    }
}

//-----
int main()
{
    cout << "ns1 creator: " << ns1::creator << endl;
    cout << "ns2 creator: " << ns2::creator << endl << endl;

    ns1::hello();
    ns2::hello();
}

// =====

ALTERNATIVE main():

int main()
{
    {
        using namespace ns1;
        hello();
    }

    {
        using namespace ns2;
        hello();
    }
}

```

OBJECT ORIENTED LANGUAGES

To support the principles of object-oriented programming (OOP), all OOP languages have **three traits in common**:

Encapsulation :

- the mechanism that **binds** together **code** and the **data** it manipulates and keeps both **safe from outside interference and misuse**.
 - code and data may be combined in such a way that a self-contained "**black-box**" is created

Inheritance :

- the process by which **one object can acquire the properties of another object**
- this is important because it supports the **concept of classification**
 - most knowledge is made manageable by hierarchical classifications
 - ex: a Human is a Primate
a Primate is a Mammal
a Mammal is an Animal
- allows us to define a class in terms of another class, providing an opportunity to reuse the code functionality which makes it easier to create and maintain an application

Polymorphism:

- In programming languages, polymorphism means that **some code or operations or objects behave differently in different contexts**.
- Helps reduce the complexity by allowing the **same interface** to be used **to access a general class of actions**
- In C++, both **compile-time (static)** and **run-time (dynamic)** polymorphism are supported
 - Forms of **compile-time** polymorphism in C++ (*already studied*)
 - **overloading**
 - **templates**
 - Forms of **run-time** polymorphism in C++
 - **inheritance + virtual functions** (*following themes*)

- **Polymorphism (run-time)**
 - describes a set of **objects** of different classes with similar behavior.
 - **Inheritance** is used to express the commonality between the classes, and **virtual functions** enable variations in behavior.

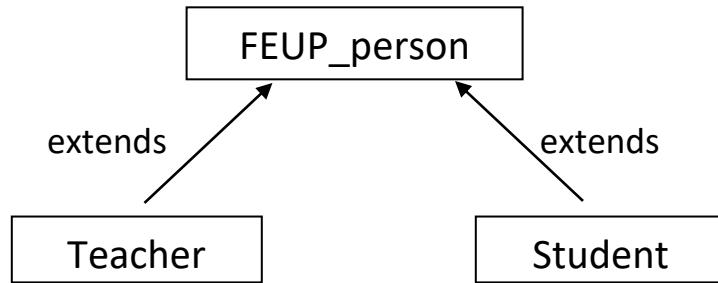
Virtual functions

- In contrast to **all other function calls that are statically bound**
(the compiler determines which function is called
by looking only at the type of the implicit parameter variable)
virtual functions are dynamically bound
(when a virtual function is called,
the **actual type of the implicit parameter object determines**
which implementation of the virtual function is invoked)

INHERITANCE (not for evaluation in 2021/2022)

- **Inheritance** is a mechanism for enhancing existing working classes.
- If a new class needs to be implemented and a class representing a more general concept is already available, then the new class can inherit from the existing class.
- The existing, more general class is called the **base class** or **parent class**.
- The more specialized class that inherits from the base class is called the **derived class** or **child class**.
- A **derived class**
 - automatically has all the member variables and functions of the base class
 - can have additional member variables and/or member functions
- Some examples:
 - example 1:
 - in a store, books, magazines and movies are publications having some properties (data members) in common:
 - an ID, a title
 - all these types of publications have some unique properties
 - **Book**, **Magazine** and **Movie** can be declared as classes derived from a base class: **Publication**
 - example 2:
 - a Human is a Primate
 - a Primate is a Mammal
 - a Mammal is an Animal
 - **Mammal** can be derived from **Animal**
 - **Primate** can be derived from **Mammal**
 - **Human** can be derived from **Primate**
 - example 3:
 - FEUP teachers are FEUP people
 - FEUP students are FEUP people

- **Class (/type) hierarchy:**



- People at FEUP have some characteristics/behaviors in common (?):
 - characteristics (class attributes): ID, name, address, ...
 - behaviours (class methods): show record, change address, ...
- Students have some things in special:
 - characteristics: course ID, year, classes taken, ...
 - behaviours : change course, add class taken, ...
- Teachers have some things in special:
 - characteristics: rank (assistant, professor, ...), classes taught, ...
 - behaviours : promote, add a class taught, ...

BASE CLASS: Feup_Person

```
#include <string>

class FeupPerson
{
public:
    FeupPerson(int id, std::string name, std::string address);
    void changeAddress(std::string newAddress);
    void showRecord();
protected: //accessible inside the class and by all of its subclasses
    int id;
    std::string name;
    std::string address;
};
```

"Protected" qualifier

- protected members of a class appear to be private outside the class, but are accessible by derived classes
- Using protected members of a class is a convenience to facilitate writing the code of derived classes.
- Protected members are not necessary
 - derived classes can use the public methods of their ancestor classes to access private members
- Many programmers consider it bad style to use protected member variables because
 - the designer of the base class has no control over the authors of derived classes

DERIVED CLASS: Student

```
#include <vector>
#include <string>
#include "FeupPerson.h"
#include "Class.h"

class Student : public FeupPerson {
public:
    Student(int id, std::string name, std::string address, std::string course, int year);
    void addClassTaken(Class* newClass);
    void changeCourse(int newCourse);
    void showRecord();
private:
    std::string course;
    int year;
    std::vector<Class*> classesTaken; // NOTE the type of array elements
};
```

- **public**, the **base class access specifier**, is needed.
If it were omitted **student** would inherit **privately**.

- If the **base class access specifier** were **private**,
public and **protected** members of the base class
would become **private members of the derived class**.

This means that:

- they are still accessible by members of the derived class
- but cannot be accessed by parts of your program
that are not members of either the base or derived class.
- If the **base class access specifier** were **protected**,
public and **protected** members of the base class
would become **protected members of the derived class**.

Modes of inheritance (summary)

When the component of the base class is declared as:	When the base class is inherited as:	The resulting access in the derived class is:
public	public	public
protected	public	protected
private	public	none
public	protected	protected
protected	protected	protected
private	protected	none
public	private	private
protected	private	private
private	private	none

- **Note :**
The private members in the base class cannot be directly accessed in the derived class,
while protected members can be directly accessed.

Constructors of the base and the derived classes

```
// in FeupPerson.cpp
FeupPerson::FeupPerson(int id, std::string name, std::string address)
{
    this->id = id;
    this->name = name;
    this->address = address;
}

//-----
// in Student.cpp
Student::Student(int id, std::string name, std::string address, std::string course, int year) : FeupPerson(id, name, address) //call to the base constructor
{
    this->course = course;
    this->year = year;
}
```

- This is the only way of calling a base class constructor with parameters (otherwise the default constructor will be automatically called).
- The **attributes** in the base class that are **protected** could be accessed in the constructor of the derived class.
- *More about calling the base class constructors in the next pages.*

Constructing an object of the derived classe

```
Student mieic = Student(12345, "John Silva", "St. John Street", "MIEIC", 3);
```

- ID = 12345
- name = "John Silva"
- person address = "St. John Street"
- course name = "MIEIC"
- classes taken = none yet
- year = 3

Redefining a method in the child class

```
class FeupPerson {  
public:  
    FeupPerson(int id, std::string name, std::string address);  
    void showRecord();  
    void changeAddress(std::string newAddress);  
protected:  
    int id;  
    std::string name;  
    std::string address;  
};  
  
class Student : public FeupPerson {  
public:  
    Student(int id, std::string name, std::string address, std::string  
course, int year);  
    void showRecord(); // redefine the method to display course & classes  
    void addClassTaken(Class* newClass);  
    void changeCourse(std::string newCourse);  
private:  
    std::string course;  
    int year;  
    std::vector<Class*> classesTaken;  
};  
  
//-----  
  
void FeupPerson::showRecord() { // definition in FeupPerson.cpp  
    std::cout << "-----\n";  
    std::cout << "Name: " << name << " ID: " << id << " Address: " << address  
<< "\n";  
    std::cout << "-----\n"; }  
  
void Student::showRecord(){ // definition in Student.cpp  
    std::cout << "-----\n";  
    std::cout << "Name: " << name << " ID: " << id << " Address: " << address  
<< "\n";  
    std::cout << "Course: " << course << "\n";  
    std::vector<Class*>::iterator it;  
    std::cout << "Classes taken:\n";  
    for (it = classesTaken.begin(); it != classesTaken.end(); it++)  
    {  
        Class* c = *it; // Class is a not very good name for a class ☺, but ...  
        std::cout << c->getName() << "\n";  
    }  
    std::cout << "-----\n"; }  
//-----
```

Usage examples

```
FeupPerson peter = FeupPerson(987, "Peter Lee", "St. Peter Street")
//OR FeupPerson peter(987, "Peter Lee", "St. Peter Street")

Student bio123 = Student(123, "John Silva", "St. John Street", "MIB", 3);
peter.showRecord();

Class* c1 = new Class("EDA");
bio123.addClassTaken(c1);

bio123.showRecord();
```

Building a derived class

- A **derived class** (child / descendant class) inherits all the members of the **parent class** (ancestor class)
- The **parent class** contains all the **code common to the child classes**
- The derived class **can add member variables** and **functions**
- The derived class **can re-declare and re-define member functions** of the parent class that will have a different definition in the derived class
- Definitions are not given for inherited functions that are not to be changed

Private members of the parent class

- A member variable (or function) that is private in the parent class is **not accessible to the child class**
- The **parent class member functions must be used to access** the private members of the parent

Derived Class Constructors

- A **base class constructor is not inherited** in a derived class
- The base class constructor **can be invoked** by the constructor of the derived class
- The **constructor of a derived class begins by invoking the constructor of the base class** in the **initialization section**:

```
Student::Student(int id, std::string name, std::string address, std::string
course, int year) : FeupPerson(id, name, address) // call to the base constructor
{
    this->course = course;
    this->year = year;
}
```

- These are the key points about constructors for derived classes:
 - The base-class object is constructed first.
 - The derived-class constructor should pass base-class information to a base-class constructor via a member initializer list.
 - The derived-class constructor should initialize the data members that were added to the derived class.
- If a derived class constructor does not invoke a base class constructor explicitly, the base class default constructor will be used
*(NOTE: if the constructor was overloaded
don't forget to implement the default constructor)*
- If class B is derived from class A and class C is derived from class B, when a object of class C is created
 - The base class A's constructor is the first invoked
 - Class B's constructor is invoked next
 - C's constructor completes execution
- Destructors are invoked in reverse order: C → B → A

Using objects of the ancestor and the descendent classes

- An object of a class type can be used wherever any of its ancestors can be used
- An ancestor cannot be used wherever one of its descendants can be used

```
FeupPerson p;
Student s;
...
p = s; // possible BUT SOME DATA IS SLICED AWAY - SLICING PROBLEM
s = p; // NOT POSSIBLE -> COMPILER ERROR
```

The slicing problem

- It is possible in C++ to avoid the slicing problem
- Using pointers to dynamic variables
 we can assign objects of a derived class to variables of a base class without loosing members of the derived class object

Function redefinition vs. overloading

- A function redefined in a derived class has the same number and type of parameters
(the function signature is the same)
 - the derived class has only one function with the same name as the base class
- An overloaded function has a different number and/or type of parameters than the base class

Function signature

- is the **name of the function** with the **sequence of types in the parameter list** not including the **const keyword** and the **ampersand (&)**
- C++ allows functions to be overloaded on the basis of const-ness of parameters only if the const parameter is a reference or a pointer.
 - Example of invalid overloading:
 - `void f(const int i)`
 - `void f(int i)`
 - Example of valid overloading:
 - `void f(char *s)`
 - `void f(const char *s)`

```
//inherit_02.sln  
//JAS - 2012/12/07
```

```
-----  
// FeupPerson.h  
#ifndef FEUP_PERSON_H  
#define FEUP_PERSON_H  
  
#include <string>  
  
class FeupPerson {  
public:  
    FeupPerson();  
    FeupPerson(int id, std::string name, std::string address);  
    void showRecord();  
    void changeAddress(std::string newAddress);  
protected:  
    int id;  
    std::string name;  
    std::string address;  
};  
#endif
```

```
-----  
//  
//  
//FeupPerson.cpp
```

```
#include <iostream>  
#include "FeupPerson.h"  
  
FeupPerson::FeupPerson()  
{  
    this->id = 0;  
    this->name = "";  
    this->address = "";  
}  
  
FeupPerson::FeupPerson(int id, std::string name, std::string address)  
{  
    this->id = id;  
    this->name = name;  
    this->address = address;  
}  
  
void FeupPerson::showRecord() {  
    std::cout << "-----\n";  
    std::cout << "Name : " << name << std::endl  
          << "ID : " << id << std::endl  
          << "Address : " << address << std::endl;  
    std::cout << "-----\n";  
}  
  
void FeupPerson::changeAddress(std::string newAddress)  
{  
    this->address = newAddress;  
}
```

```

//-----
// Student.h
#ifndef STUDENT_H
#define STUDENT_H

#include <iostream>
#include <vector>
#include <string>
#include "FeupPerson.h"
#include "Class.h"

class Student : public FeupPerson {
public:
    Student() { };
    Student(int id, std::string name, std::string address, std::string course, int year);
    void showRecord();
    void addClassTaken(Class* newClass);
    void changeCourse(std::string newCourse);
private:
    std::string course;
    int year; // 1 = freshman, 2 = sophomore, etc.
    std::vector<Class*> classesTaken;
};

#endif
//-----
//Student.cpp
#include "Student.h"

Student::Student(int id, std::string name, std::string address,
                 std::string course, int year) : FeupPerson(id, name, address)
{
    this->course = course;
    this->year = year;
}

void Student::addClassTaken(Class* newClass)
{
    classesTaken.push_back(newClass);
}

void Student::showRecord()
{
    std::cout << "-----\n";
    std::cout << "Name : " << name << std::endl
        << "ID : " << id << std::endl
        << "Address : " << address << std::endl;
    std::cout << "Course : " << course << std::endl;

    std::vector<Class*>::iterator it;

    std::cout << "Classes taken:\n";
    for (it = classesTaken.begin(); it != classesTaken.end(); it++){
        Class* c = *it;
        std::cout << c->getName() << std::endl;
    }
    std::cout << "-----\n";
}

```

```

//-
// Class.h
#ifndef CLASS_H
#define CLASS_H

#include <string>

class Class { // what a class name!!!
public:
    Class(std::string name);
    std::string getName();
private:
    std::string name;
};

#endif // CLASS_H

//-
//-
// Class.cpp
#include "Class.h"

Class::Class(std::string name)
{
    this->name = name;
}

std::string Class::getName()
{
    return name;
}

//-
//-
// main.cpp
#include <iostream>
#include <string>
#include "Student.h"

int main(){
    FeupPerson dei987 = FeupPerson(987, "Peter Lee", "St. Peter Street");
    Student bio123 = Student(123, "John Silva", "St. John Street", "MIB", 3);

    Class *c1 = new Class("EDA");
    Class *c2 = new Class("EDE");
    Class *c3 = new Class("EDI");

    dei987.showRecord();

    bio123.addClassTaken(c1);
    bio123.showRecord();

    bio123.addClassTaken(c2);
    bio123.addClassTaken(c3);
    bio123.showRecord();
}
//-

```

- Replace the previous main() function with the following one:

```
//-----
// main.cpp
#include <iostream>
#include <string>
#include "Student.h"

int main(){
    Class* c1 = new Class("EDA");
    Class* c2 = new Class("EDE");
    Class* c3 = new Class("EDI");

    Student *mieic234 = new Student(234, "John Souza", "St. John Street",
    "MIEIC", 2);
    mieic234->addClassTaken(c1);
    mieic234->addClassTaken(c3);
    mieic234->showRecord();

    FeupPerson *mieic345 = new Student(345, "Liz Tanner", "St. Liz Street",
    "MIEIC", 2);
    mieic345->showRecord(); // NOTE THE RESULT !!!
    //mieic345->addClassTaken(c1); //UNCOMMENT AND INTERPRET THE RESULT
}

//-----
```

- **NOTE** that:
 - a base class pointer can be used to point to a derived class object
 - ...but ... what happens when **showRecord()** method is invoked through the base class pointer that points to the derived class object ... ?!

POLYMORPHISM (virtual functions)

- Ability of type A to appear as and be used like another type B
 - ex: a Student object can be used like a FeupPerson object
- Suppose one would like to build a vector of Teachers / Students.

```
int main()
{
    std::vector<FeupPerson> p(3);

    p[0] = Teacher(987, "Pedro Santos", "Rua do Pedro", "Assistente",
"MIIB");
    p[1] = Student(123, "Nuno Silva", "Rua do Nuno", "MIIB", 3);
    p[2] = Student(234, "Ana Sousa", "Rua da Ana", "MIEIC", 2);

    for (unsigned int i=0; i<p.size(); i++)
        p[i].showRecord();
    ...
}
```

- Although the assignments in yellow are possible,
some of the fields of Teacher/Student shall be lost (slicing problem)
- This problem is very typical of code that
needs to manipulate objects from a mixture of data types.
 - Derived-class objects are usually bigger than base-class objects
and objects of different derived-classes have different sizes
 - A vector of objects cannot deal with this variation in sizes
 - But a vector of pointers to objects can ... (see next example)
if showRecord() is declared virtual, in FeupPerson class
- In the following, the code for class Teacher is shown

```

// Teacher.h
#ifndef TEACHER_H
#define TEACHER_H
#include <iostream>
#include <vector>
#include <string>
#include "FeupPerson.h"
#include "Class.h"

class Teacher : public FeupPerson {
public:
    Teacher() {};
    Teacher(int id, std::string name, std::string address, std::string rank,
            std::string course);
    void showRecord();
    void addClassTaught(Class* newClass);
    void changeCourse(std::string newCourse);

    void showCourse() {std::cout << course << std::endl;};

private:
    std::string rank;
    std::string course;
    std::vector<Class*> classesTaught;
};

#endif
=====

//Teacher.cpp
#include "Teacher.h"

Teacher::Teacher(int id, std::string name, std::string address,
                 std::string rank, std::string course) : FeupPerson(id,
name, address) {
    this->rank = rank;
    this->course = course;
}

void Teacher::addClassTaught(Class* newClass) {
    classesTaught.push_back(newClass);
}

void Teacher::showRecord() {
    std::cout << "-----\n";
    std::cout << "Name : " << name << std::endl
        << "ID : " << id << std::endl
        << "Address : " << address << std::endl
        << "Rank : " << rank << std::endl
        << "Course : " << course << std::endl;

    std::vector<Class*>::iterator it;

    std::cout << "Classes taught:\n";
    for (it = classesTaught.begin(); it != classesTaught.end(); it++) {
        Class* c = *it;
        std::cout << c->getName() << std::endl;
    }
    std::cout << "-----\n";
}

```

```

int main()
{
    std::vector<FeupPerson *> p(3);

    p[0] = new Teacher(987, "Pedro Santos", "Rua do Pedro", "Assistente",
"MIIB");
    p[1] = new Student(123, "Nuno Silva", "Rua do Nuno", "MIIB", 3);
    p[2] = new Student(234, "Ana Sousa", "Rua da Ana", "MIEIC", 2);

    for (unsigned int i=0; i<p.size(); i++)
        p[i]->showRecord();
    ...
}

```

- Note that
 - the assignments of the above code
assign a derived-class pointer of type `Teacher*` or `Student*`
to a base-class pointer of type `FeupPerson*`
 - this is **legal**
 - the reverse (from a base-class pointer to a derived-class one) is an **error**
- But when one runs the above code, the output is something like:

```

-----
Name      : Pedro Santos
ID       : 987
Address   : Rua do Pedro
-----
-----
Name      : Nuno Silva
ID       : 123
Address   : Rua do Nuno
-----
-----
Name      : Ana Sousa
ID       : 234
Address   : Rua da Ana
-----
```

- The compiler generated code only to call the `FeupPerson's showRecord()` method ...
... because `p[i]` is of type `FeupPerson*`

- However it is possible to **alert the compiler** that the function call must be preceded by the **appropriate function selection**.
- This selection must be **done at run-time**.
- To tell the compiler that a particular call needs to be bound dynamically the **function** must be **tagged as **virtual****:

```
class FeupPerson {
public:
    FeupPerson(); // NOTE: CONSTRUCTORS CAN'T BE MADE VIRTUAL
    FeupPerson(int id, std::string name, std::string address);
    //virtual ~FeupPerson(); // SEE NOTE ON NEXT PAGES
    virtual void showRecord();
    void changeAddress(std::string newAddress);
protected:
    int id;
    std::string name;
    std::string address;
};
```

- The output is:

```
-----  
Name : Pedro Santos  
ID : 987  
Address : Rua do Pedro  
Rank : Assistente  
Course : MIB  
Classes taught:  
-----
```

```
-----  
Name : Nuno Silva  
ID : 123  
Address : Rua do Nuno  
Course : MIB  
Classes taken:  
-----
```

```
-----  
Name : Ana Sousa  
ID : 234  
Address : Rua da Ana  
Course : MIEIC  
Classes taken:  
-----
```

- Such a **selection/call** combination is called **dynamic binding (or late binding)** in contrast to the traditional call which always invokes the same function being called **static binding**.
- The **virtual** keyword must be used **in the base class**.
- When a function is declared **virtual**, all functions with the **same name and parameter types** in derived classes are then **automatically virtual**.
 - However it is considered **good taste** to supply the **virtual** keyword for the derived-classes as well

- Whenever a virtual function is called,
the compiler determines the type of the implicit parameter
in the particular call **at run time**.
 - Ex: `p[i]->showRecord()`
always calls the function belonging to the actual type of the object
to which `p[i]` points,
either
`Teacher::showRecord()`
 - or
`Student::showRecord()`
- Only member functions can be virtual.
- You should **use virtual functions**
only when you need the flexibility of dynamic binding at run time.
- The
`vector<FeupPerson *> p(3)`
collects a mixture of both kinds of FEUP persons.
- Such a **collection** is called **polymorphic**.
- **However ...**
if you try to call a **method that is not implemented in the base class**
- as `showRecord()` is -
for example:
 - `p[0]->addClassTaught(c1);`

you'll get a **compiler error**:

 - `error C2039: 'addClassTaught' : is not a member of 'FeupPerson'`

- This problem can be solved using a dynamic_cast to downcast the pointer

```

int main()
{
    Class* c1 = new Class("EDA");
    Class* c2 = new Class("EDU");

    std::vector<FeupPerson *> p(3);

    p[0] = new Teacher(987, "Pedro Santos", "Rua do Pedro", "Assistente",
    "MIB");
    p[1] = new Student(123, "Nuno Silva", "Rua do Nuno", "MIB", 3);
    p[2] = new Student(234, "Ana Sousa", "Rua da Ana", "MIEIC", 2);

    std::cout << "BEFORE_DYNAMIC_CAST:\n";
    for (unsigned int i=0; i<p.size(); i++)
        p[i]->showRecord();

    for (unsigned int i=0; i<p.size(); i++)
    {
        Teacher *t = dynamic_cast<Teacher *> (p[i]);
        if (t != NULL)
            t->addClassTaught(c1);
        else
        {
            Student *s = dynamic_cast<Student *> (p[i]);
            if (s != NULL)
                s->addClassTaken(c2);
        }
    }

    std::cout << "AFTER_DYNAMIC_CAST:\n";
    for (unsigned int i=0; i<p.size(); i++)
        p[i]->showRecord();
}

```

BEFORE_DYNAMIC_CAST:

```

Name      : Pedro Santos
ID       : 987
Address   : Rua do Pedro
Rank     : Assistente
Course   : MIB
Classes taught:
-----
```

```

Name      : Nuno Silva
ID       : 123
Address   : Rua do Nuno
Course   : MIB
Classes taken:
-----
```

```

Name      : Ana Sousa
ID       : 234
Address   : Rua da Ana
Course   : MIEIC
Classes taken:
-----
```

AFTER_DYNAMIC_CAST: (and call to addClassTaught() and addClassTaken())

AFTER DYNAMIC_CAST: (and call to addClassTaught() and addClassTaken())

```
Name      : Pedro Santos
ID       : 987
Address   : Rua do Pedro
Rank     : Assistente
Course   : MIB
Classes taught:
EDA
```

```
Name      : Nuno Silva
ID       : 123
Address   : Rua do Nuno
Course   : MIB
Classes taken:
EDU
```

```
Name      : Ana Sousa
ID       : 234
Address   : Rua da Ana
Course   : MIEIC
Classes taken:
EDU
```

Dynamic casts

- **dynamic_cast** can be used only with pointers and references to objects.
Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.
- **NOTE:** the operand of a runtime **dynamic_cast** must be a polymorphic class type.
For that, **at least one of the methods of the class must be virtual**.
- **dynamic_cast** is always **successful** when we **cast a class to one of its base classes**
- **When dynamic_cast cannot cast** a pointer because it is not a complete object of the required class , it returns a NULL pointer to indicate the failure.

Named casts

- General form for named cast:
 - *cast-name<type> (expression)*
- where **cast-name** may be
 - **static_cast**
 - **dynamic_cast**
 - **const_cast**
 - **reinterpret_cast**
- **type** is the target type of the conversion
- **expression** is the value to be cast
- **Note:** use casts with caution.

Static cast

- Any type of conversion that the compiler performs implicitly can be explicitly requested using a `static_cast`.
 - `double d = 97.0;`
`int i = static_cast<int>(d);`
 - Compilers often generate a warning for assignment of a larger arithmetic type to a smaller type. The cast informs both the reader and the compiler that we are aware of and not concerned with the potential loss of precision.
- A `static_cast` can also be used to perform conversions that the compiler will not generate automatically:
 - `double q = static_cast<double>(3) / 4;`
OR
 - `double d;`
`void *p = &d; // OK: a void pointer is compatible with any other pointer`
`// pointer p can be used as argument of some special functions`
`// that need to receive a void * as argument`
`// ex: pthread_create() to be studied in the Operating Systems course.`
`// That function will convert the void * back to a double *, using:`
`double *pd = static_cast<double*>(p);`

Const cast

- `const_cast` can be used to remove or add `const` to a variable.
- Only for “advanced” uses... (see: http://en.cppreference.com/w/cpp/language/const_cast)
- Note: even though `const_cast` may remove constness (or volatility) from any pointer or reference, using the resulting pointer or reference to write to an object that was declared `const` (or to access an object that was declared volatile) invokes undefined behavior.

Reinterpret cast

- `reinterpret_cast` generally performs a low-level reinterpretation of the bit pattern of its operands.
- Must be used with caution. The result is machine dependent. Its use requires completely understanding the types involved as well as the details of how the compiler implements the cast.
- Example
 - `int *pi = new(int);`
`*pi = 65 + 66 * 256 + 67 * 256 * 256;`
`char *pc = reinterpret_cast<char *>(pi);`
`cout << pc << endl; // which is the result?`

Dynamic casts

- See previous section.

Old-style casts

- Prior to the introduction of named cast operators, an explicit cast was performed using one of two forms:
 - `(type) expression; // C-language-style cast notation`
 - `type (expression); // function-style cast notation`
- Depending on the types involved, an old-style cast has the same behavior as a `const_cast`, a `static_cast`, or a `reinterpret_cast`.
 - `int i; double d;`
`i = i + (int) d; // same as static_cast`
 - `const char *s = "ABCDE";`
`func((char *) s); // same as const_cast; casts away const`
`// BUT func() CANNOT MODIFY s ...!`
 - `int *pi; // SEE PREVIOUS EXAMPLE`
`char *pc = (char *) pi; // same as reinterpret_cast`
- **Notes:**
 - Although the old-style cast notation is supported by the Standard C++, its use is recommended only when writing code to be compiled under the C language.
 - Two main reasons are that `C++ casting operators` are intended to make the `casting operations` more explicit (see above) and safer:
 - Example:

```
char ch = 65;           // 1 byte
int *p1 = (int*)&ch;    // 4 bytes

*p1 = 112;              // may result in run-time error
                        // or in destruction of other variables

but

int *p2 = static_cast<int*>(&ch); // results in compile-time error
```

Virtual details

- To define a function differently in a derived class and to make it virtual
 - Add keyword **virtual** to the function declaration in the **base class**
 - "virtual" is not needed for the function declaration in the derived class, but is often included
 - **"virtual" is not added to the function definition**
- Virtual functions require considerable overhead so excessive use **reduces program efficiency**
(see section 13.7 from Deitel, 7th ed, for an explanation how it works)
- Making a function **virtual** tells the compiler that you don't know how the function is implemented and **to wait until the function is used** in a program, **then get the implementation** from the object.
 - This is called **late binding**

Overriding vs. redefinition

- Virtual functions whose definitions are changed in a derived class are said to be **overridden** (*in Portuguese, "substituídas"/"ignoradas"/"canceladas"*)
 - Since C++11, in order to tell the compiler that the prototype of the function in the derived class is the same as in the base class, the specifier **override** was introduced

```
class Base {
public:
    // programmer wants to override this in the derived class ...
    virtual void func()
    {
        // ...
    }
};

class Derived : public Base {
public:
    // ...but, did a mistake by putting an argument "int a"
    void func(int a) override // the compiler signals an error:
                            // method with override specifier 'override'
                            // did not override any base class methods
    {
        // ...
    }
};
```

- The specifier **final** used in a function prevents overriding or further overriding.
- A class can also be declared **final** meaning that it cannot be used as a base class:
 - `class MyClass final { ... } ; // this class can't be used as a base class`
- Non-virtual functions whose definitions are changed in a derived class are **redefined**

Virtual Destructors

- **Destructors should be made virtual**
(In C++, constructors cannot be made virtual –
to create an object, you must know its exact type)
- Consider

```
Base *pBase = new Derived;  
...  
delete pBase;
```

- If the destructor in Base is virtual,
the destructor for Derived is invoked
as pBase points to a Derived object,
returning Derived members to the freestore
- The Derived destructor in turn calls the Base destructor
- If the Base destructor is not virtual, only the Base destructor is invoked
This leaves Derived members, not part of Base, in memory.
- **NOTE:**
when you have a definition "Derived d;" the destructor of Base will always be called.

```
// WHEN ARE THE CONSTRUCTORS & DESTRUCTORS OF BASE & DERIVED CLASSES CALLED?  
//  
// ILLUSTRATING THE NEED FOR A VIRTUAL DESTRUCTOR IN A BASE CLASS  
// JAS  
#include <iostream>  
#include <cstring>  
using namespace std;  
  
//-----  
class Base  
{  
public:  
    Base();  
    virtual ~Base();  
    //~Base(); // TRY THIS ALTERNATIVE & ANALYSE THE RESULTS  
private:  
    int dummyB; // just to have an attribute  
    //other attributes  
};  
//-----  
class Derived: public Base  
{  
public:  
    Derived();  
    ~Derived();  
private:  
    int *dummyD; // will point to dynamically allocated memory  
    // other attributes  
};
```

```

//-----  

Base::Base()  

{  

    cout << "Base constructor called\n";  

    // TO DO: initialize attributes  

}  

//-----  

Base::~Base()  

{  

    cout << "Base destructor called\n";  

    // TO DO  

}  

//-----  

Derived::Derived()  

{  

    cout << "Derived constructor called\n";  

    // TO DO: initialize attributes, dynamically allocating memory  

}  

//-----  

Derived::~Derived()  

{  

    cout << "Derived destructor called\n";  

    // TO DO: free the dinamically allocated memory  

}  

//-----  

//-----  

int main()  

{  

    cout << "\n Base b1; ----- \n";  

    Base b1;  

    cout << "\n Derived d1; ----- \n";  

    Derived d1;  

    cout << "\n Base *b2 = new Base(); ----- \n";  

    Base *b2 = new Base();  

    cout << "\n Derived *d2 = new Derived(); ----- \n";  

    Derived *d2 = new Derived();  

    cout << "\n Base *d3 = new Derived(); ----- \n";  

    Base *d3 = new Derived();  

    cout << "\n delete b2; ----- \n";  

    delete b2;  

    cout << "\n delete d2; ----- \n";  

    delete d2;  

    cout << "\n delete d3; ----- \n";  

    delete d3;  

    cout << "\n END OF main() ----- \n";  

    return 0;
}

```

// ANOTHER EXAMPLE OF A POLYMORPHIC VECTOR

(C:\Users\jsilva\Documents\AULAS 2010-2011__PROG\8-Programas\cClocks2)

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <ctime>
#include <cassert>

using namespace std;

//=====
/*
   A class that describes a time of day
   (between 00:00:00 and 23:59:59)
*/
class Time
{
public:

    Time(int hour, int min, int sec); //Constructs a time of day
    Time(); //Constructs a Time object that is set to the time at which the
constructor executes
    int get_hours() const; //Gets the hours of this time
    int get_minutes() const; //Gets the minutes of this time
    int get_seconds() const; //Gets the seconds of this time

private:
    int time_in_secs;
};

//=====

class Clock
{
public:
    Clock(bool use_military);
    virtual string get_location() const; // TRY TO REMOVE virtual qualifier
    virtual int get_hours() const; // TRY TO REMOVE virtual qualifier
    virtual int get_minutes() const; // TRY TO REMOVE virtual qualifier
    bool is_military() const;
private: // COULD BE DECLARED protected
    // (see comment in TravelClock::get_hours() )
    bool military; // means 24 hour notation -> 00:00 ... 23:59
};

//=====

class Travelclock : public clock
{
public:
    Travelclock(bool mil, string loc, int diff);
    string get_location() const;
    int get_hours() const;
private:
    string location;
    int time_difference;
};

//=====
```

```

//=====
Time::Time(int hour, int min, int sec)
{
    assert(0 <= hour);
    assert(hour < 24);
    assert(0 <= min);
    assert(min < 60);
    assert(0 <= sec);
    assert(sec < 60);

    time_in_secs = 60L * 60 * hour + 60 * min + sec;
}
//-----
Time::Time()
{
    time_t now = time(0);
    struct tm t;
    localtime_s(&t, &now);
    time_in_secs = 60L * 60 * t.tm_hour + 60 * t.tm_min + t.tm_sec;
}
//-----
int Time::get_hours() const
{
    return time_in_secs / (60 * 60);
}
//-----
int Time::get_minutes() const
{
    return (time_in_secs / 60) % 60;
}
//-----
int Time::get_seconds() const
{
    return time_in_secs % 60;
}

//=====

Clock::Clock(bool use_military)
{
    military = use_military;
}
//-----
string Clock::get_location() const
{
    return "Local";
}
//-----
int Clock::get_hours() const
{
    Time now;
    int hours = now.get_hours();
    if (military) return hours;
    if (hours == 0)
        return 12;
    else if (hours > 12)
        return hours - 12;
    else
        return hours;
}

```

```

int Clock::get_minutes() const
{
    Time now;
    return now.get_minutes();
}
//-----
bool Clock::is_military() const
{
    return military;
}

//=====

TravelClock::TravelClock(bool mil, string loc, int diff)
: Clock(mil)
{
    location = loc;
    time_difference = diff;
    while (time_difference < 0)
        time_difference = time_difference + 24;
}
//-----
string TravelClock::get_location() const
{
    return location;
}
//-----
int TravelClock::get_hours() const
{
    int h = Clock::get_hours(); //NOTE THIS
    if (is_military())
        return (h + time_difference) % 24;
    else
    {
        h = (h + time_difference) % 12;
        if (h == 0) return 12;
        else return h;
    }
}
//=====

int main()
{
    vector<Clock *> clocks(3);
    clocks[0] = new Clock(true);
    clocks[1] = new TravelClock(true, "Madrid", 1);
    clocks[2] = new TravelClock(false, "Azores", -1);

    for (size_t i = 0; i < clocks.size(); i++)
    {
        cout << clocks[i]->get_location() << " time is "
            << clocks[i]->get_hours() << ":"
            << setw(2) << setfill('0')
            << clocks[i]->get_minutes()
            << setfill(' ') << "\n";
    }
    return 0;
}

```

Pure virtual functions or abstract functions

```
// Example adapted from
// http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming#cite_note-tcp1-1

#include <iostream>
#include <string>

using namespace std;

class Animal
{
public:
    Animal(const string& name) : name(name) {}
    virtual string talk() = 0; //pure virtual function (or abstract function)
    const string name; //public attribute !!! HOW TO MAKE IT PRIVATE?
};

class Cat : public Animal
{
public:
    Cat(const string& name) : Animal(name) {}
    virtual string talk() { return "Meow!"; }
};

class Dog : public Animal
{
public:
    Dog(const string& name) : Animal(name) {}
    virtual string talk() { return "Arf! Arf!"; }
};

int main()
{
    Animal* animals[] = //NOTE the initialization
    {
        new Cat("Mr. Jinks"),
        new Cat("Garfield"),
        new Dog("Milou")
    };

    for(int i = 0; i < 3; i++)
    {
        cout << animals[i]->name << ":" << animals[i]->talk() << endl;
        delete animals[i];
    }
    return 0;
}
```

TO DO:

- in class Cat
comment `virtual string talk() { return "Meow!"; }`
and interpret result
- then replace in class Animal
`virtual string talk() = 0;`
with
`virtual string talk() {return "...!";}`
and interpret result
- Remove all `virtual` keywords and interpret results

Pure virtual functions (or abstract functions)

- Pure virtual function is a virtual function that has no body at all !
 - indicated by a prototype that has no implementation
 - this is indicated by the **pure specifier, = 0**, following the function prototype;
- A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.
- When we add a pure virtual function to our class, we are effectively saying, “**it is up to the derived classes to implement this function**”.
- Using a pure virtual function has **two main consequences**:
 - First, any class with **one or more pure virtual functions** becomes an **abstract base class**, which means that it **can not be instantiated!**
 - Second, **any derived class must define a body for this function.**

Interface classes

- An **interface class** is a class that has **no members variables**, and where **all of the functions are pure virtual!**
- In other words, the class is purely a definition, and has **no actual implementation**.
- Interfaces are useful when you want to **define the functionality that derived classes must implement**, but leave the details of how the derived class implements that functionality entirely up to the derived class.

```
class Document {  
public:  
    // Requirements for derived classes;  
    // they must implement these functions.  
    virtual string identify() = 0;  
    virtual string whereIs() = 0;  
};
```

- An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined.

- Although interface implementations can evolve, interfaces themselves cannot be changed once published.
 - Changes to a published interface may break existing code.
 - If you **think of an interface as a contract**, it is clear that both sides of the contract have a role to play.
 - The **publisher** of an interface agrees never to change that interface,
 - and the **implementer** agrees to implement the interface exactly as it was designed

=====

OBJECT ORIENTED LANGUAGES (remembering ...)

=====

To support the principles of object-oriented programming (OOP), all OOP languages have **three traits in common**:

Encapsulation :

- the mechanism that **binds** together **code** and the **data** it manipulates and keeps both **safe from outside interference and misuse**.
 - code and data may be combined in such a way that a self-contained "**black-box**" is created

Inheritance :

- the process by which **one object can acquire the properties of another object**
- this is important because it supports the **concept of classification**
 - most knowledge is made manageable by hierarchical classifications
 - ex: a Car is a Vehicle
 - a Truck is a Vehicle
 - this relationship is often described as the *is-a* relationship not to be confused with the *has-a* relationship
 - ex: a Vehicle has a Tire (effectively it should have more...)

Polymorphism:

- In programming languages, polymorphism means that **some code or operations or objects behave differently in different contexts**.
- Helps reduce the complexity by allowing the **same interface** to be used **to access a general class of actions**
- In C++, both **compile-time (static)** and **run-time (dynamic)** polymorphism are supported
 - Forms of **compile-time** polymorphism in C++
 - overloading
 - templates
 - Forms of **run-time** polymorphism in C++
 - inheritance + virtual functions

- **Polymorphism (run-time)**
describes a set of objects of different classes with similar behavior.
- **Inheritance** is used to express the commonality between the classes, and ...
... **virtual functions** enable variations in behavior.

Virtual functions

in contrast to all other function call that are **statically bound**

(the compiler determines which function is called
by looking only at the type of the implicit parameter variable)

virtual functions are bound dynamically

(when a virtual function is called,
the **actual type of the implicit parameter object determines**
which implementation of the virtual function is invoked)

===== EXCEPTION HANDLING (not for evaluation in 2021/2022) =====

Things sometimes go wrong ... 😞

- User input errors
- Device errors
 - disk I/O
- Physical limitations
 - memory is exhausted
- Software component failures
 - function performs incorrectly

Some approaches already seen for handling exceptional conditions

- The `fail` predicate used by the stream I/O library
- The `assert` macro
- `exit` the program

Alternative (historical) ways of handling exceptional conditions

- Assume errors will not occur 😞
- Print an error message
- Special return values
 - ```
Stack s;
bool ok = s.push(10);
```
  - but, in the following case ... how to return the special value ?  
`int i = s.pop();`
  - ... in this way ...?  
`bool ok = s.pop(i); //bool Stack::pop (int & value) ???`  
`int i = s.pop(ok); //int Stack::pop (bool & status) ???`
  - alternative (the user must take care ...)  
`if (s.size() > 0)
 {i = s.pop();};`
  - Could also use a `struct` or a `pair` to "join" the boolean and the integer
  - Or return an `std::optional<T>` (since C++17) – out of the scope of this course
- External flags
  - ```
int n = atoi(num_CString);
if (errno == ERANGE) // => #include <cerrno>
    ....;
```
- In debug mode, use `assert` to halt execution
- Error Handlers

- Some errors can be detected and resolved at the point they occur
- Others are non-local; i.e., must be resolved at a “higher-level”

Exception Handling Mechanism

- Later addition to C++
- In C++, exception handling proceeds by:
 - Some library software or your code signals that something unusual has happened
 - This is called **throwing an exception**
 - At some other place in your program you place the code that deals with the exceptional case
 - This is called **handling the exception**
- Exception handling is meant to be used **sparingly**

try-throw-catch mechanism

- An error is signaled by throwing an exception
- **Any type can be thrown**
- If not handled (caught) locally, function exits
- **Does not return to calling point**
- Unwinds call stack, looking for an appropriate handler.

Throwing an Exception

- Example:

```
double futurevalue(double initialAmount, double tax, int numYears)
{
    if (tax < 0 || numYears < 0)
    {
        logic_error description("illegal futurevalue() parameter(s)");
        throw description;
        // ALTERNATIVE:
        // throw logic_error("illegal future_value parameter");
    }
    return initialAmount * pow(1 + tax / 100, numYears);
}
```

- Purpose:
 - Abandon this function and throw a value to an exception handler.

Catching Exceptions

- Supply a handler with the try statement:

```
try
{
    // code that could cause a problem
}
catch (type_name & e) //NOTE: can be received by reference (pref.) or by value
{
    // handler (executed if a problem occurs)
}
```

- The **try block** encloses code that you want to "try" but that could cause a problem
- If an error is thrown in the try clause, execution goes to the **catch clause**
- The type of the catch-block parameter identifies the kind of value the catch-block can catch
- If no appropriate handler is found, the next outer try block is examined

```
#include <iostream>
#include <cmath>
#include <stdexcept>

using namespace std;

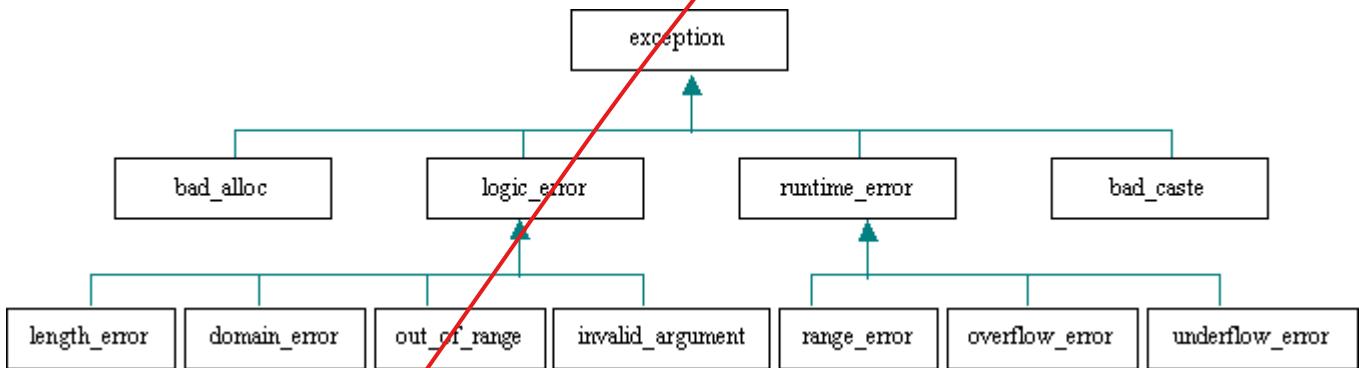
double futureValue(double initialAmount, double tax, int numYears)
{
    if (initialAmount < 0 || tax < 0 || numYears < 0)
    {
        //logic_error description("illegal futurevalue parameter");
        //throw description;
        throw logic_error("illegal futurevalue() parameter(s)");
    }
    return initialAmount * pow(1 + tax / 100, numYears);
}

int main()
{
    double value, amount, tax;
    int years;
    cout << "amount ? "; cin >> amount;
    cout << "tax ? "; cin >> tax;
    cout << "numYears ? "; cin >> years;
    try
    {
        value = Futurevalue(amount, tax, years);
        cout << "future value = " << value << endl;
    }
    catch (logic_error & e)
    {
        cerr << "Processing error: " << e.what() << "\n";
    }
}
```

Catching Exceptions – Syntax

```
try
{
    statements
}
catch (type_name_1 variable_name_1)
{
    statements
}
catch (type_name_2 variable_name_2)
{
    statements
}
...
catch (type_name_N variable_name_N)
{
    statements
}
catch (...) // DEFAULT CATCH BLOCK - handle exceptions not covered
{
    statements
}
```

Standard exception hierarchy (*partial*), in `<stdexcept>`



- All of the exceptions thrown by the C++ Standard Library are objects of classes in this hierarchy.
- Each class in the hierarchy supports a **what()** method that returns a `char*` string describing the exception. You can use this string in an error message.

Values Thrown and Caught

- Can throw (and catch) any type, including integer
- Suggestion: throw a type that derives directly or indirectly from `std::exception`
- Implicit conversion (e.g., `int` to `double`) not performed on thrown values

Users can define their own exceptions

```
// objects of this class can carry the kind of information
// you want thrown to the catch-block
class MyApplicationError
{
public:
    MyApplicationError(const string& r);
    string& what() const; // returns C++-string, instead of C-string (see previous page)
private:
    string reason;
};

MyApplicationError::MyApplicationError(const string& r) : reason(r) {}

string& MyApplicationError::what() const
{
    return reason;
}

...
try
{
    ...
    throw MyApplicationError("illegal value");
}
catch (MyApplicationError& e)
{
    cerr << "Caught exception " << e.what() << "\n";
}
```

Inheriting from standard exceptions

```
class FutureValueError : public logic_error
{
public:
    FutureValueError(string reason);
};

FutureValueError::FutureValueError(string reason) : logic_error(reason) {}

...
...
...

try
{
    code
}
catch (FutureValueError& e) // only catches FutureValueError
{
    handler1
}

catch (logic_error& e) // catches all other logic_error
{
    handler2
}

catch (bad_alloc& e)
{
    handler3
}
```

Nested try-catch blocks

- Although a try-block followed by its catch-block can be nested inside another try-block
 - It is almost always better to place the nested try-block and its catch-block inside a function definition, then invoke the function in the outer try-block
- An error thrown but not caught in the inner try-catch-blocks is thrown to the outer try-block where it might be caught
- If no appropriate handler is found, the next outer try block is examined

```
#include <iostream>
#include <cmath>
#include <stdexcept>

using namespace std;

double f2(double x)
{
    if (x<0) throw invalid_argument("invalid argument in f2() call");
    else return sqrt(x); //invalid_argument is a standard exception class (see prev.pages)
}

double f1(double x)
{
    return 1/f2(x); // the error in f2() is not caught here
}

int main()
{
    try
    {
        cout << f1(-2) << endl;
    }
    catch (invalid_argument& e)
    {
        cerr << "error in f1() call: " << e.what() << "\n";
    }
}
```

When to Throw An Exception

- Throwing exceptions is generally reserved for those cases when handling the exceptional case depends on how and where the function was invoked
- In these cases it is usually better to let the programmer calling the function handle the exception

Rethrowing Exceptions

- The code **within a catch-block** can throw an exception
- This feature can be used **to pass the same or a different exception** up the chain of exception handling blocks
- use **throw with no arguments** to rethrow error

Exceptions and Constructors / Destructors

- Constructors and destructors do not return a value
- **Throwing an exception is a clean way to indicate failure in a constructor**
- **BE CAREFUL:** If a constructor fails, the object is not created
 - The destructor isn't called
 - Subtle source of **leaks (can be avoided - see next example)**

```
DataArray::DataArray(int size)
{
    data = new int[size];
    try
    {
        init(); // call to auxiliary initialization function that can throw an error ...
    }
    catch (...) // Catch any exception that init() throws
    {
        delete[] data;
        data = NULL;
        throw; // Rethrow exception
    }
}
```

- **Don't throw exceptions from within a destructor**
 - destructors are invoked as part of the process of stack unwinding during the recovery from an exception,
 - if a destructor throws an exception this would yield 2 exceptions the one currently being handled by stack unwinding and the one thrown by the destructor ... (it is unclear which one would take priority; **the program is halted**)
- **Note:** **static values** are initialized before main is entered.
There is no try block to catch exceptions

Exceptions – additional notes

Diverse of exceptions

- Throwing an exception allows you to transfer flow of control to almost any place in your program
- Such un-restricted flow of control is sometimes considered poor programming style as it makes programs difficult to understand

Exceptions and performance

(source: Errors and Exception Handling (Modern C++) 2016)

<https://msdn.microsoft.com/en-us/library/hh279678.aspx>

- The exception mechanism has a very minimal performance cost if no exception is thrown. If an exception is thrown, the cost of the stack traversal and unwinding is roughly comparable to the cost of a function call.
- Additional data structures are required to track the call stack after a try block is entered, and additional instructions are required to unwind the stack if an exception is thrown. However, in most scenarios, the cost in performance and memory footprint is not significant.

try-throw-catch review

- The try-block includes a throw-statement
- If an exception is thrown, the try-block ends and the catch-block is executed
- If no exception is thrown, then after the try-block is completed, execution continues with the code following the catch-block(s)
- Catch blocks are examined top to bottom
- catch(...) is used to catch any exception; should be last in list
- Executes the first handler that matches, then stops processing that exception
- C++ unwinds call stack in search of a try block to handle the exception; the exception handler could lie one or more function calls up the stack of execution.
- As the control jumps up in the stack, in a process called stack unwinding, all code remaining in each function past the current point of execution is skipped.
- Local objects and variables in each function that is unwound are destroyed as if the code finished the function normally.
- However, in stack unwinding, pointer variables are not freed, and other cleanup is not performed.
- Before each function is terminated, destructors are called on all stack variables
- An uncaught exception ends your program.

```

// EXCEPTION HANDLING
// A LAST EXAMPLE

#include <iostream>
#include <stdexcept>

using namespace std;

int main()
{
    int numSlices, numGlasses;
    double ratio;

    try
    {
        cout << "Enter number of \"pizza\" slices:\n";
        cin >> numSlices;
        cout << "Enter number of glasses of \"orange juice\":\n";
        cin >> numGlasses;

        if (numGlasses <= 0)
            throw numSlices;

        ratio = static_cast<double>(numGlasses) / numSlices;
        cout << numSlices << " slices.\n"
            << numGlasses << " glasses of juice.\n"
            << "You have " << ratio
            << " glasses of juice for each slice.\n";
    }
    catch (int n)
    {
        cerr << n << " slices, and NO JUICE!!!\n"
            << "Go buy some JUICE.\n\n";
    }

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

```

END