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.

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.*

Namespaces

- Are a means of organizing typenames, variables and functions
  so as to avoid name conflicts.

- 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, ...

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.
- 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`

```
// VARIABLES CAN BE DECLARED ANYWHERE, IN THE SAME CODE BLOCK,
// BEFORE THEY ARE USED
```

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`

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`

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

- 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

- Are language constructs that allow a programmer to control the flow of execution through the program.
- There are 3 main categories:
  - **sequence** → Sequência das instruções
  - **selection** → If…else…, switch… case…
  - **repetition** → For, while, do…while

- **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!**

- **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***

```
#include <iostream>
#include <iomanip>  //needed for stream manipulators: fixed, setprecison

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
}
```

```
/*
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;
}
```

==========

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);
```

- **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 ommited.

- **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 innner 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 ...

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 easist 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`

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

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 <ios> *AND* **#include <limits>**
- Note: **BE CAREFUL!!!**
  a call to **cin.ignore()** when the buffer is empty will stop the execution of the program until something is entered !!!

```
/*
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;
}
```

Example of cin.clear() e cin.ignore():

```
// 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);
```

```
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
```

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 invoque **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
```

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


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

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

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

```
/*
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;
}
```

- **Function call syntax**
  - `function_name(parameter_1, parameter_2, … , parameter_n)`
- **Predefined functions**
  - C++ comes with libraries of predefined functions
  - Example: **sqrt** and **pow** functions
    - `square_side = sqrt(square_area);`
    - `cube_volume = pow(cube_edge, 3.0);`
  - **square_area**, **cube_edge** and **3.0** are the arguments of calls
  - The arguments can be variables, constants or expressions
  - A library must be "included" in the program to make the predefined functions available
    - To include the math library containing **sqrt()** and **pow()**:
      `#include <cmath>`
- **Programmer defined functions**
  - Two components of a function definition:
    - **Function declaration** (or **function prototype**)
      - Shows how the function is called
      - Must appear in the code before the function can be called
      - Syntax:

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

      - Example:

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

      - Tells the **return type** (**double**)
        - the return type can be **int**, **char**, **bool**, … or **void** (*see later*)
      - Tells the name of the function (**total_price**)
      - Tells how many arguments are needed (2)
      - Tells the types of the arguments (**int** and **double**)
      - Tells the **formal parameter names** (**num_items** and **item_price**)
      - Formal **parameters** are like placeholders for the actual **arguments** used when the function is called
      - Formal parameter names can be any valid identifier

- **Function definition**
  - Provides the same information as the function declaration
  - Describes how the function does its task
  - Can appear before or after the function is called
  - Syntax:

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

  - Example:

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

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

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

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

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

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

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

- **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

```
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)
```

```
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
}
```

ATTENTION!!

```
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;

}
```

- **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;
  }
}
```

- **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.

==========

Função como parâmetro:

double func(double);

double integrateTR(double f(double), double a, double b, int n);

Na main: integrateTR(func,a,b,n);

==========

ARRAYS

==========

- **Introduction to arrays**
  - An array is used to process a collection of data of the same type

- **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
```

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

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

  - Initialization after declaration

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

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

```
int a1[3] = {10,20,30};
int a2[3];
a2 = a1; // COMPILATION ERROR ...
cout << a1 << endl; //NOT AN ERROR! BUT ... WHAT DOES IT SHOW?
```

  - Each element must be read/written/processed at a time, using a loop (see previous example)

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

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

...

int studentScore[NUMBER_OF_STUDENTS];
...

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

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

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

...
```

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

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

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

  - Initialization of a multidimensional array when it is declared:

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

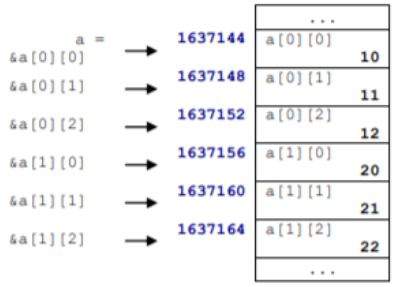
  - Indexing a multidimensional array:
    - **`score[0][0]`** – score of the 1st student in the 1st question
    - **`score[0][1]`** – score of the 1st student in the 2nd question
    - ...
    - **`score[1][2]`** – score of the 2nd student in the 3rd question
    - **NOTE**: the "off-by-one offset" in the index ...
  - **NOTE:**
    When a multidimensional array is used as a formal function parameter, the size of the first dimension is not given, but the remaining dimension sizes must be given in square brackets.
  - Since the first dimension size is not given, you usually need an additional parameter of type **`int`** that gives the size of this first dimension.

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

```
// 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 ommited
{
  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
```



==========

- **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*)

```
vector<double> salaries(5); //vector with 5 elements of type double
```

```
// constructing vectors
#include <iostream>
#include <vector>

int main ()
{
  // constructors used in the same order as described above:
  std::vector<int> first;                                // empty vector of ints
  std::vector<int> second (4,100);                       // four ints with value 100
  std::vector<int> third (second.begin(),second.end());  // iterating through second
  std::vector<int> fourth (third);                       // a copy of third

  // the iterator constructor can also be used to construct from arrays:
  int myints[] = {16,2,77,29};
  std::vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

  std::cout << "The contents of fifth are:";
  for (std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

- **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));
```

- **Accessing elements**

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

```
int main ()
{
  std::vector<int> foo (3,0);
  std::vector<int> bar (5,0);

  bar = foo;
  foo = std::vector<int>();

  std::cout << "Size of foo: " << int(foo.size()) << '\n';
  std::cout << "Size of bar: " << int(bar.size()) << '\n';
  return 0;
}
```

Output:

```
Size of foo: 0
Size of bar: 3
```

- **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

==========

```
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
```

Vector.size()

```
std::vector<int> myints;
std::cout << "0. size: " << myints.size() << '\n';
```

```
0. size: 0
```

Vector.max_size() -> Returns the maximum number of elements that the vector can hold.

Vector.resize(n) -> Resizes the container so that it contains n elements

```
for (int i=1;i<10;i++) myvector.push_back(i);

myvector.resize(5);
myvector.resize(8,100);
myvector.resize(12);
```

Output:

```
myvector contains: 1 2 3 4 5 100 100 100 0 0 0 0
```

Vector.capacity() -> Returns the size of the storage space currently allocated for the vector, expressed in terms of elements.

Vector.empty() -> Returns whether the vector is empty (i.e. whether its size is 0), true if empty, false otherwise

Vector.shrink_to_fit() -> Requests the container to reduce its capacity to fit its size.

Vector.front() -> Returns a reference to the first element in the vector.

Vector.back() -> Returns a reference to the last element in the vector.

Vector.data() -> Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

Vector.assign() -> Assigns new contents to the vector, replacing its current contents, and modifying its size accordingly.

```
first.assign (7,100);             // 7 ints with a value of 100

std::vector<int>::iterator it;
it=first.begin()+1;

second.assign (it,first.end()-1); // the 5 central values of first

int myints[] = {1776,7,4};
third.assign (myints,myints+3);   // assigning from array.

std::cout << "Size of first: " << int (first.size()) << '\n';
std::cout << "Size of second: " << int (second.size()) << '\n';
std::cout << "Size of third: " << int (third.size()) << '\n';
```

Output:

```
Size of first: 7
Size of second: 5
Size of third: 3
```

Vector.push_back(element) -> Adds a new element at the end of the vector, after its current last element.

Vector.pop_back() -> Removes the last element in the vector, effectively reducing the container size by one.

Vector.swap() -> Exchanges the content of the container by the content of x, which is another vector object of the same type. Sizes may differ. A = {1,2}; B = {2,3,4}; A.swap(B); → A= {2,3,4}; B={1,2};

Vector.clear() -> Removes all elements from the vector (which are destroyed), leaving the container with a size of 0.

Vector.insert() -> The vector is extended by inserting new elements before the element at the specified position, effectively increasing the container size by the number of elements inserted.

```
std::vector<int> myvector (3,100);
std::vector<int>::iterator it;

it = myvector.begin();
it = myvector.insert ( it , 200 );

myvector.insert (it,2,300);

// "it" no longer valid, get a new one:
it = myvector.begin();

std::vector<int> anothervector (2,400);
myvector.insert (it+2,anothervector.begin(),anothervector.end());

int myarray [] = { 501,502,503 };
myvector.insert (myvector.begin(), myarray, myarray+3);

std::cout << "myvector contains:";
for (it=myvector.begin(); it<myvector.end(); it++)
  std::cout << ' ' << *it;
std::cout << '\n';
```

Output:

```
myvector contains: 501 502 503 300 300 400 400 200 100 100 100
```

Find (using vector) -> Returns an iterator to the first element in the range [first,last) that compares equal to val. If no such element is found, the function returns last.

```
// find example
#include <iostream>     // std::cout
#include <algorithm>    // std::find
#include <vector>       // std::vector

int main () {
  // using std::find with array and pointer:
  int myints[] = { 10, 20, 30, 40 };
  int * p;

  p = std::find (myints, myints+4, 30);
  if (p != myints+4)
    std::cout << "Element found in myints: " << *p << '\n';
  else
    std::cout << "Element not found in myints\n";

  // using std::find with vector and iterator:
  std::vector<int> myvector (myints,myints+4);
  std::vector<int>::iterator it;

  it = find (myvector.begin(), myvector.end(), 30);
  if (it != myvector.end())
    std::cout << "Element found in myvector: " << *it << '\n';
  else
    std::cout << "Element not found in myvector\n";

  return 0;
}
```

Output:

```
Element found in myints: 30
Element found in myvector: 30
```

count() -> Returns the number of elements in the range [first,last) that compare equal to val.

```
// count algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::count
#include <vector>       // std::vector

int main () {
  // counting elements in array:
  int myints[] = {10,20,30,30,20,10,10,20};   // 8 elements
  int mycount = std::count (myints, myints+8, 10);
  std::cout << "10 appears " << mycount << " times.\n";

  // counting elements in container:
  std::vector<int> myvector (myints, myints+8);
  mycount = std::count (myvector.begin(), myvector.end(), 20);
  std::cout << "20 appears " << mycount  << " times.\n";

  return 0;
}
```

Output:

```
10 appears 3 times.
20 appears 3 times.
```

Vector.erase(n):

```
// erasing from vector
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;

  // set some values (from 1 to 10)
  for (int i=1; i<=10; i++) myvector.push_back(i);

  // erase the 6th element
  myvector.erase (myvector.begin()+5);

  // erase the first 3 elements:
  myvector.erase (myvector.begin(),myvector.begin()+3);

  std::cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size(); ++i)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';

  return 0;
}
```

sort() -> Sorts the elements in the range [first,last) into ascending order.

```
bool myfunction (int i,int j) { return (i<j); }

struct myclass {
  bool operator() (int i,int j) { return (i<j);}
} myobject;

int main () {
  int myints[] = {32,71,12,45,26,80,53,33};
  std::vector<int> myvector (myints, myints+8);               // 32 71 12 45 26 80 53 33

  // using default comparison (operator <):
  std::sort (myvector.begin(), myvector.begin()+4);           //(12 32 45 71)26 80 53 33

  // using function as comp
  std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)

  // using object as comp
  std::sort (myvector.begin(), myvector.end(), myobject);     //(12 26 32 33 45 53 71 80)

  // print out content:
  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
myvector contains: 12 26 32 33 45 53 71 80
```

set_intersection() -> Constructs a sorted range beginning in the location pointed by result with the set intersection of the two sorted ranges [first1,last1) and [first2,last2).

```
#include <iterator>
std::vector<int> s3;
set_intersection(s1.begin(),s1.end(),s2.begin(),s2.end(), std::back_inserter(s3));
```

set_union()-> União entre dois vetores ordenados.

std::vector<int> v;
std::set_union(x.begin(), x.end(), y.begin(), y.end(), std::back_inserter(v));

unique() -> Removes all but the first element from every consecutive group of equivalent elements in the range [first,last). (Para remover todos os duplicaods, ordenar primeiro)

```
// unique algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::unique, std::distance
#include <vector>       // std::vector

bool myfunction (int i, int j) {
  return (i==j);
}

int main () {
  int myints[] = {10,20,20,20,30,30,20,20,10};           // 10 20 20 20 30 30 20 20 10
  std::vector<int> myvector (myints,myints+9);

  // using default comparison:
  std::vector<int>::iterator it;
  it = std::unique (myvector.begin(), myvector.end());   // 10 20 30 20 10 ?  ?  ?  ?
                                                         //             ^

  myvector.resize( std::distance(myvector.begin(),it) ); // 10 20 30 20 10

  // using predicate comparison:
  std::unique (myvector.begin(), myvector.end(), myfunction);   // (no changes)

  // print out content:
  std::cout << "myvector contains:";
  for (it=myvector.begin(); it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
myvector contains: 10 20 30 20 10
```

==========

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**.

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 sintactically 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, if 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 adress 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 adresses
  - 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

- **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.
- C++ supports 2 dynamic allocations systems:
  - the one defined by C: using functions **malloc()** and **free()**
  - the one defined by C++: using operators **new** and **delete**

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;`

C dynamic memory allocation

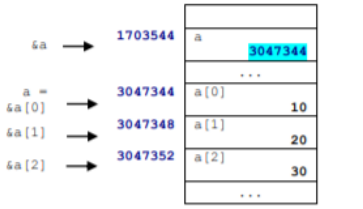
- The core of C's allocation system consists of the functions: **malloc()** and **free()**
  - => **#include <cstdlib>**
  - **void *malloc(size_t number_of_bytes);**
    - **number_of_bytes** is the number of bytes of memory you wish to allocate
    - the return value is a **void** pointer
      - in 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 sucessful 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

```
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;

      // dinamically 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 dinamically 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
```



Em C: (Alocar 2D)

```
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);
}
```

REALLOC->

```
void* realloc (void* ptr, size_t size);
```

```
more_numbers = (int*) realloc (numbers, count * sizeof(int));
```

Em C++:

```
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;
}
```

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

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.

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

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.";
```

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.

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

Safe versions of string manipulation functions

- 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)

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;
```

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;
```

Constructor:

```
// string constructor
#include <iostream>
#include <string>

int main ()
{
  std::string s0 ("Initial string");

  // constructors used in the same order as described above:
  std::string s1;
  std::string s2 (s0);
  std::string s3 (s0, 8, 3);
  std::string s4 ("A character sequence");
  std::string s5 ("Another character sequence", 12);
  std::string s6a (10, 'x');
  std::string s6b (10, 42);      // 42 is the ASCII code for '*'
  std::string s7 (s0.begin(), s0.begin()+7);
```

Output:

```
s1:
s2: Initial string
s3: str
s4: A character sequence
s5: Another char
s6a: xxxxxxxxxx
s6b: **********
s7: Initial
```

String.size() -> Returns the length of the string, in terms of bytes.

String.length() -> Returns the length of the string, in terms of bytes.

String.max_size() -> Returns the maximum length the string can reach.

String.capacity() -> Returns the size of the storage space currently allocated for the string, expressed in terms of bytes.

```
std::string str ("Test string");
std::cout << "size: " << str.size() << "\n";
std::cout << "length: " << str.length() << "\n";
std::cout << "capacity: " << str.capacity() << "\n";
std::cout << "max_size: " << str.max_size() << "\n";
```

A possible output for this program could be:

```
size: 11
length: 11
capacity: 15
max_size: 429496729
```

String.resize(n) ->Resizes the string to a length of n characters.

```
void resize (size_t n);
void resize (size_t n, char c);
```

```
std::string str ("I like to code in C");
std::cout << str << '\n';

unsigned sz = str.size();

str.resize (sz+2,'+');
std::cout << str << '\n';

str.resize (14);
std::cout << str << '\n';
```

Output:

```
I like to code in C
I like to code in C++
I like to code
```

String.resize(n) -> Requests that the string capacity be adapted to a planned change in size to a length of up to n characters.

```
std::string str;

std::ifstream file ("test.txt",std::ios::in|std::ios::ate);
if (file) {
  std::ifstream::streampos filesize = file.tellg();
  str.reserve(filesize);

  file.seekg(0);
  while (!file.eof())
  {
    str += file.get();
  }
  std::cout << str;
}
```

String.clear() -> Erases the contents of the string, which becomes an empty string (with a length of 0 characters).

```
char c;
std::string str;
std::cout << "Please type some lines of text. Enter a dot (.) to finish:\n";
do {
  c = std::cin.get();
  str += c;
  if (c=='\n')
  {
     std::cout << str;
     str.clear();
  }
} while (c!='.');
```

String.empty() -> Returns whether the string is empty (i.e. whether its length is 0).

```
std::string content;
std::string line;
std::cout << "Please introduce a text. Enter an empty line to finish:\n";
do {
  getline(std::cin,line);
  content += line + '\n';
} while (!line.empty());
std::cout << "The text you introduced was:\n" << content;
```

String.shrink_to_fit() -> Requests the string to reduce its capacity to fit its size.

```
std::string str (100,'x');
std::cout << "1. capacity of str: " << str.capacity() << '\n';

str.resize(10);
std::cout << "2. capacity of str: " << str.capacity() << '\n';

str.shrink_to_fit();
std::cout << "3. capacity of str: " << str.capacity() << '\n';
```

Possible output:

```
1. capacity of str: 100
2. capacity of str: 100
3. capacity of str: 10
```

String.back() -> Returns a reference to the last character of the string.

```
std::string str ("hello world.");
str.back() = '!';
std::cout << str << '\n';
```

Output:

```
hello world!
```

String.front() -> Returns a reference to the first character of the string.

```
std::string str ("test string");
str.front() = 'T';
std::cout << str << '\n';
```

Output:

```
Test string
```

String.append(n) -> Extends the string by appending additional characters at the end of its current value.

```
int main ()
{
  std::string str;
  std::string str2="Writing ";
  std::string str3="print 10 and then 5 more";

  // used in the same order as described above:
  str.append(str2);                       // "Writing "
  str.append(str3,6,3);                   // "10 "
  str.append("dots are cool",5);          // "dots "
  str.append("here: ");                   // "here: "
  str.append(10u,'.');                    // ".........."
  str.append(str3.begin()+8,str3.end());  // " and then 5 more"
  str.append<int>(5,0x2E);                // "....."

  std::cout << str << '\n';
  return 0;
}
```

Output:

```
Writing 10 dots here: .......... and then 5 more.....
```

String.push_back(n) -> Appends character c to the end of the string, increasing its length by one. `void push_back (char c);`

String.assign(n) -> Assigns a new value to the string, replacing its current contents.

```
std::string str;
std::string base="The quick brown fox jumps over a lazy dog.";

// used in the same order as described above:

str.assign(base);
std::cout << str << '\n';

str.assign(base,10,9);
std::cout << str << '\n';         // "brown fox"

str.assign("pangrams are cool",7);
std::cout << str << '\n';         // "pangram"

str.assign("c-string");
std::cout << str << '\n';         // "c-string"

str.assign(10,'*');
std::cout << str << '\n';         // "**********"

str.assign<int>(10,0x2D);
std::cout << str << '\n';         // "----------"

str.assign(base.begin()+16,base.end()-12);
std::cout << str << '\n';         // "fox jumps over"
```

Output:

```
The quick brown fox jumps over a lazy dog.
brown fox
pangram
c-string
**********
----------
fox jumps over
```

String.insert(n) -> Inserts additional characters into the string right before the character indicated by pos (or p).

```
std::string str="to be question";
std::string str2="the ";
std::string str3="or not to be";
std::string::iterator it;

// used in the same order as described above:
str.insert(6,str2);                 // to be (the )question
str.insert(6,str3,3,4);             // to be (not )the question
str.insert(10,"that is cool",8);    // to be not (that is )the question
str.insert(10,"to be ");            // to be not (to be )that is the question
str.insert(15,1,':');               // to be not to be(:) that is the question
it = str.insert(str.begin()+5,','); // to be(,) not to be: that is the question
str.insert (str.end(),3,'.');       // to be, not to be: that is the question(...)
str.insert (it+2,str3.begin(),str3.begin()+3); // (or )

std::cout << str << '\n';
```

Output:

```
to be, or not to be: that is the question...
```

String.erase(n) -> Erases part of the string, reducing its length.

```
int main ()
{
  std::string str ("This is an example sentence.");
  std::cout << str << '\n';
                                       // "This is an example sentence."
  str.erase (10,8);                    //            ^^^^^^^^
  std::cout << str << '\n';
                                       // "This is an sentence."
  str.erase (str.begin()+9);           //           ^
  std::cout << str << '\n';
                                       // "This is a sentence."
  str.erase (str.begin()+5, str.end()-9);  //       ^^^^^
  std::cout << str << '\n';
                                       // "This sentence."
  return 0;
}
```

Output:

```
This is an example sentence.
This is an sentence.
This is a sentence.
This sentence.
```

String.replace(n) -> Replaces the portion of the string that begins at character pos and spans len characters (or the part of the string in the range between [i1,i2)) by new contents.

```
std::string base="this is a test string.";
std::string str2="n example";
std::string str3="sample phrase";
std::string str4="useful.";

// replace signatures used in the same order as described above:

// Using positions:                 0123456789*123456789*12345
std::string str=base;           // "this is a test string."
str.replace(9,5,str2);          // "this is an example string." (1)
str.replace(19,6,str3,7,6);     // "this is an example phrase." (2)
str.replace(8,10,"just a");     // "this is just a phrase."     (3)
str.replace(8,6,"a shorty",7);  // "this is a short phrase."    (4)
str.replace(22,1,3,'!');        // "this is a short phrase!!!"  (5)

// Using iterators:                                     0123456789*123456789*
str.replace(str.begin(),str.end()-3,str3);               // "sample phrase!!!"      (1)
str.replace(str.begin(),str.begin()+6,"replace");        // "replace phrase!!!"     (3)
str.replace(str.begin()+8,str.begin()+14,"is coolness",7);  // "replace is cool!!!"    (4)
str.replace(str.begin()+12,str.end()-4,4,'o');           // "replace is cooool!!!"  (5)
str.replace(str.begin()+11,str.end(),str4.begin(),str4.end());// "replace is useful."    (6)
std::cout << str << '\n';
```

Output:

```
replace is useful.
```

String.swap(n) -> Exchanges the content of the container by the content of str, which is another string object. Lengths may differ. `void swap (string& str);`

String.pop_back() -> Erases the last character of the string, effectively reducing its length by one.

String.c_str() -> Returns a pointer to an array that contains a null-terminated sequence of characters (i.e., a C-string) representing the current value of the string object.
This array includes the same sequence of characters that make up the value of the string object plus an additional terminating null-character ('\0') at the end.

String.copy(n) -> Copies a substring of the current value of the string object into the array pointed by s. This substring contains the len characters that start at position pos.

```
char buffer[20];
std::string str ("Test string...");
std::size_t length = str.copy(buffer,6,5);
buffer[length]='\0';
std::cout << "buffer contains: " << buffer << '\n';
```

Output:

```
buffer contains: string
```

String.find(n) -> Searches the string for the first occurrence of the sequence specified by its arguments.

```
int main ()
{
  std::string str ("There are two needles in this haystack with needles.");
  std::string str2 ("needle");

  // different member versions of find in the same order as above:
  std::size_t found = str.find(str2);
  if (found!=std::string::npos)
    std::cout << "first 'needle' found at: " << found << '\n';

  found=str.find("needles are small",found+1,6);
  if (found!=std::string::npos)
    std::cout << "second 'needle' found at: " << found << '\n';

  found=str.find("haystack");
  if (found!=std::string::npos)
    std::cout << "'haystack' also found at: " << found << '\n';

  found=str.find('.');
  if (found!=std::string::npos)
    std::cout << "Period found at: " << found << '\n';

  // let's replace the first needle:
  str.replace(str.find(str2),str2.length(),"preposition");
  std::cout << str << '\n';

  return 0;
}
```

```
string (1)    size_t find (const string& str, size_t pos = 0) const;
c-string (2)  size_t find (const char* s, size_t pos = 0) const;
buffer (3)    size_t find (const char* s, size_t pos, size_t n) const;
character (4) size_t find (char c, size_t pos = 0) const;
```

Notice how parameter *pos* is used to search for a second instance of the same search string. Output:

```
first 'needle' found at: 14
second 'needle' found at: 44
'haystack' also found at: 30
Period found at: 51
There are two prepositions in this haystack with needles.
```

String.rfind(n) -> Searches the string for the last occurrence of the sequence specified by its arguments.

```
// string::rfind
#include <iostream>
#include <string>
#include <cstddef>

int main ()
{
  std::string str ("The sixth sick sheik's sixth sheep's sick.");
  std::string key ("sixth");

  std::size_t found = str.rfind(key);
  if (found!=std::string::npos)
    str.replace (found,key.length(),"seventh");

  std::cout << str << '\n';

  return 0;
}
```

```
string (1)    size_t rfind (const string& str, size_t pos = npos) const;
c-string (2)  size_t rfind (const char* s, size_t pos = npos) const;
buffer (3)    size_t rfind (const char* s, size_t pos, size_t n) const;
character (4) size_t rfind (char c, size_t pos = npos) const;
```

```
The sixth sick sheik's seventh sheep's sick.
```

String.find_first_of(n) -> Searches the string for the first character that matches any of the characters specified in its arguments.

```
// string::find_first_of
#include <iostream>       // std::cout
#include <string>         // std::string
#include <cstddef>        // std::size_t

int main ()
{
  std::string str ("Please, replace the vowels in this sentence by asterisks.");
  std::size_t found = str.find_first_of("aeiou");
  while (found!=std::string::npos)
  {
    str[found]='*';
    found=str.find_first_of("aeiou",found+1);
  }

  std::cout << str << '\n';

  return 0;
}
```

```
string (1)    size_t find_first_of (const string& str, size_t pos = 0) const;
c-string (2)  size_t find_first_of (const char* s, size_t pos = 0) const;
buffer (3)    size_t find_first_of (const char* s, size_t pos, size_t n) const;
character (4) size_t find_first_of (char c, size_t pos = 0) const;
```

```
Pl**s*, r*pl*c* th* v*w*ls *n th*s s*nt*nc* by *st*r*sks.
```

String.find_first_not_of(n) -> Searches the string for the first character that does not match any of the characters specified in its arguments.

```
// string::find_first_not_of
#include <iostream>       // std::cout
#include <string>         // std::string
#include <cstddef>        // std::size_t

int main ()
{
  std::string str ("look for non-alphabetic characters...");

  std::size_t found = str.find_first_not_of("abcdefghijklmnopqrstuvwxyz ");

  if (found!=std::string::npos)
  {
    std::cout << "The first non-alphabetic character is " << str[found];
    std::cout << " at position " << found << '\n';
  }

  return 0;
}
```

```
string (1)    size_t find_first_not_of (const string& str, size_t pos = 0) const;
c-string (2)  size_t find_first_not_of (const char* s, size_t pos = 0) const;
buffer (3)    size_t find_first_not_of (const char* s, size_t pos, size_t n) const;
character (4) size_t find_first_not_of (char c, size_t pos = 0) const;
```

```
The first non-alphabetic character is - at position 12
```

String.find_last_of(n) -> Searches the string for the last character that matches any of the characters specified in its arguments.

```
// string::find_last_of
#include <iostream>       // std::cout
#include <string>         // std::string
#include <cstddef>         // std::size_t

void SplitFilename (const std::string& str)
{
  std::cout << "Splitting: " << str << '\n';
  std::size_t found = str.find_last_of("/\\");
  std::cout << " path: " << str.substr(0,found) << '\n';
  std::cout << " file: " << str.substr(found+1) << '\n';
}

int main ()
{
  std::string str1 ("/usr/bin/man");
  std::string str2 ("c:\\windows\\winhelp.exe");

  SplitFilename (str1);
  SplitFilename (str2);

  return 0;
}
```

```
string (1)    size_t find_last_of (const string& str, size_t pos = npos) const;
c-string (2)  size_t find_last_of (const char* s, size_t pos = npos) const;
buffer (3)    size_t find_last_of (const char* s, size_t pos, size_t n) const;
character (4) size_t find_last_of (char c, size_t pos = npos) const;
```

```
Splitting: /usr/bin/man
 path: /usr/bin
 file: man
Splitting: c:\windows\winhelp.exe
 path: c:\windows
 file: winhelp.exe
```

String.find_last_not_of(n) -> Searches the string for the last character that does not match any of the characters specified in its arguments.

```
// string::find_last_not_of
#include <iostream>       // std::cout
#include <string>         // std::string
#include <cstddef>        // std::size_t

int main ()
{
  std::string str ("Please, erase trailing white-spaces   \n");
  std::string whitespaces (" \t\f\v\n\r");

  std::size_t found = str.find_last_not_of(whitespaces);
  if (found!=std::string::npos)
    str.erase(found+1);
  else
    str.clear();            // str is all whitespace

  std::cout << '[' << str << "]\n";

  return 0;
}
```

```
string (1)    size_t find_last_not_of (const string& str, size_t pos = npos) const;
c-string (2)  size_t find_last_not_of (const char* s, size_t pos = npos) const;
buffer (3)    size_t find_last_not_of (const char* s, size_t pos, size_t n) const;
character (4) size_t find_last_not_of (char c, size_t pos = npos) const;
```

```
[Please, erase trailing white-spaces]
```

String.substr(n) -> Returns a newly constructed string object with its value initialized to a copy of a substring of this object.

```
// string::substr
#include <iostream>
#include <string>

int main ()
{
  std::string str="We think in generalities, but we live in details.";
                                       // (quoting Alfred N. Whitehead)

  std::string str2 = str.substr (3,5);     // "think"

  std::size_t pos = str.find("live");      // position of "live" in str

  std::string str3 = str.substr (pos);     // get from "live" to the end

  std::cout << str2 << ' ' << str3 << '\n';

  return 0;
}
```

```
string substr (size_t pos = 0, size_t len = npos) const;
```

Output:

```
think live in details.
```

Swap(String) -> Exchanges the values of string objects x and y, such that after the call to this function, the value of x is the one which was on y before the call, and the value of y is that of x.

```
// swap strings
#include <iostream>
#include <string>

main ()
{
  std::string buyer ("money");
  std::string seller ("goods");

  std::cout << "Before the swap, buyer has " << buyer;
  std::cout << " and seller has " << seller << '\n';

  swap (buyer,seller);

  std::cout << " After the swap, buyer has " << buyer;
  std::cout << " and seller has " << seller << '\n';

  return 0;
}
```

```
void swap (string& x, string& y);
```

Output:

```
Before the swap, buyer has money and seller has goods
 After the swap, buyer has goods and seller has money
```

==========

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;
}
```

==========

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
```

Definir uma struct e inicializar um pointer do tipo node

```
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;
```

- **typedef**
  - The keyword **typedef** provides a mechanism for creating synonims (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
```

==========

UNION & ENUM types

==========

- **UNIONS**
  - A **structure** is a user-defined data type available that allows combining data items of different kinds.
  - A **union** is a special data type that allows storing different data types in the same memory location.
  - To define a union, you must use the union statement in the same way as you did while defining a structure:

```
union Numbers
{
  int  x;
  double d;
};
```

  - The memory alocated is shared by all the individual elements of the union.
  - The size of the union is the size of the largest element.
  - Possible **usages** of union's are (*see next examples*):
    - to save memory by using the same memory region for storing different objects at different times;
      - but ...
        there must be some way to known which object is effectively stored in the union (*see next example*)
    - to do aliasing (access the same data using different names)

==========

- **ENUMERATED TYPES**
  - An **enumerated type (enumeration)** is a user-defined data type which can be assigned some limited values.
  - These values can assigned automatically or be defined by the programmer at the time of declaring the enumerated type (*see next examples*).
  - We declare an enumrated type using the following syntax:
    - **enum** *enumeratedTypeName* {*value1*, *value2*, …..*valueN*};
    - **enum** is a C/C++ keyword
  - We can create enumerated type variables as same as the normal variables.
    - *enumeratedTypeName variableName* = value;

==========

STREAMS / FILES

==========

I/O Streams

- **I/O** refers to program **I**nput and **O**utput
- I/O is done via stream objects
- A **stream** is a flow of data
- Input stream: data flows into the program
  - Input can be from
    - the keyboard
    - a file
- Output stream: data flows out of the program
  - Output can be to
    - the screen
    - a file
- Input and Output stream: data flows either into or out of the program
  - only possible with files

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

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`**
- Ouput-file streams of are 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 **`"oufile.txt"`** that is in directory **`"C:\Mieic\Prog\programs"`**
    - note the double backslash in the string argument
      - necessary in Windows systems where the directories of the path are separated by **`'\'`**
  - Alternatively:
    - **`ifstream inStream("infile.txt");`**
    - calls the constructor of **`ifstream`** class that automatically tries to open the file

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;
}
```

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

setw(n) -> Sets the field width to be used on output operations.

```
std::cout << std::setw(10);
std::cout << 77 << std::endl;
```

Output:

```
        77
```

setfill(n) ->

```
std::cout << std::setfill ('x') << std::setw (10);
std::cout << 77 << std::endl;
```

Output:

```
xxxxxxxx77
```

fixed -> Sets the floatfield format flag for the str stream to fixed.

```
double a = 3.1415926534;
double b = 2006.0;
double c = 1.0e-10;

std::cout.precision(5);

std::cout << "default:\n";
std::cout << a << '\n' << b << '\n' << c << '\n';

std::cout << '\n';

std::cout << "fixed:\n" << std::fixed;
std::cout << a << '\n' << b << '\n' << c << '\n';
```

Possible output:

```
default:
3.1416
2006
1e-010

fixed:
3.14159
2006.00000
0.00000
```

Setprecision(n) -> Sets the decimal precision to be used to format floating-point values on output operations.

```
int main () {
  double f =3.14159;
  std::cout << std::setprecision(5) << f << '\n';
  std::cout << std::setprecision(9) << f << '\n';
  std::cout << std::fixed;
  std::cout << std::setprecision(5) << f << '\n';
  std::cout << std::setprecision(9) << f << '\n';
  return 0;
}
```

Output:

```
3.1416
3.14159
3.14159
3.141590000
```

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

Random access

- The C++ I/O system manages 2 pointers associated with a file (*NOTE: not to be confused with memory pointers, presented in previous section*):
  - the get pointer, which specifies where in the file the next input operation will occur
  - the put pointer, which specifies where in the file the next output operation will occur
- You can perform random access (in a nonsequential fashion) by using the **seekg()** and **seekp()** functions.
- **tellg()** and **tellp()** can be used to obtain the current position of the pointers.
- Note: you can't call **seekp**/**tellp** on an instance of **ifstream** and you can't call **seekg**/**tellg** on an instance of **ofstream**. However, you can use both on an instance of **fstream**.

- 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 &**

```
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;  //NOTE: get() READS SPACE AND NEWLINE CHARACTERS
    if (next == '\n')
    {
      n++;
      fout << n << ' ';
    }
    fin.get(next);
  }

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

  return 0;
}
```

INPUT/OUTPUT – BINARY FILES

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

#include <iostream>
#include <fstream>

using namespace std;


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

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

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

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

  f.close();
```

STRINGSTREAMS

String Streams

- We saw how a stream can be connected to a file.
- A stream can also be connected to a string.
- With stringstreams you can perform input/output from/to a string.
- This allows you to convert numbers
  (or any type with the << and >> stream operators overloaded) to and from strings.
- To use stringstreams =>
  - **#include <sstream>**
- The **istringstream** class reads characters from a string
- The **ostringstream** class writes characters to a string.

Stringstream uses

- A very common use of string streams is:
  - to accept input one line at a time and then to analyze it further.
    - by using stringstreams you can avoid mixing **cin >>** ... and **getline()**
    - *see examples in the following pages*
  - to use standard output manipulators to create a formatted string

istringstream

- Using an **istringstream**,
  you can read numbers that are stored in a string by using the >> operator:

```
string input = "March 25, 2014";
istringstream instr(input); //initializes 'instr' with 'input'
string month, comma;
int day, year;
instr >> month >> day >> comma >> year;
```

- Note that this input statement yields **day** and **year** as integers.
  Had we taken the string apart with **substr**, we would have obtained only strings.
- Converting strings that contain digits to their integer values is such a common operation
  that it is useful to write a helper function for that purpose:

```
int string_to_int(string s)
{
   istringstream instr;
   instr.str(s); // ALTERNATIVE way to initialize 'instr' with 's'
                 // to the initialization mode used above
   int n;
   instr >> n;
   return n;
}
```

ostringstream

- By writing to a string stream, you can convert numbers to strings.
- By using the **<<** operator, the number is converted into a sequence of characters.

```
ostringstream 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 <-> 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 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);
```

Example:

```
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;
```

==========

CLASSES

==========

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.

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 memberFunctId( 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

```
#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
```

```
// 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)
}
//---------------------------------------------------------------------------
```

```
// f1.Fraction(1,2); // can't invoke constructor on existing object
f1 = Fraction(1,2); // ... but can do this :-) EXPLICIT CONSTRUCTOR CALL
```

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);`

```
// 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);
}
```

Another example:

```
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;
};
```

Static ->

Esta definição seria colocada no espaço global do programa

Neste contexto, o qualificativo **static** significa que só existe uma instância de numBooks para todos os objetos da classe Book

```
IdentNum Book::numBooks = 0; //static variable 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
```

Example 2:

```
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;
};
```

```
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;
}

// 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);
```

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.

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

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

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

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.

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"

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.

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

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

```
// #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()
}

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

==========

TEMPLATES – GENERIC PROGRAMMING

==========

FUNCTION TEMPLATES / GENERIC FUNCTIONS

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;
}
```

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++;
}
```

```cpp
// 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;
}
// ---------------------------------------------------------------------
```

```cpp
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";
  }

  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;
}
```

Para definir uma variável com 2 tipos:

```cpp
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;
}
```

E desta forma pode-se fazer:

```cpp
Queue<string,5> qs;

Queue<double> qd;  // NOTE: MAXSIZE defaults to 10
```

Classe de com dois objetos de tipos diferentes com template:

```
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;
}
//----------------------------------------
```

```
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
```

STL – STANDARD TEMPLATE LIBRARY

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).*
  - ***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.*

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
```

Category	Description
input	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.
output	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.
forward	Combines the capabilities of input and output iterators and retains their position in the container (as state information).
bidirectional	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.
random access	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.

Container	Type of iterator supported
Sequence containers (first class)	
`vector`	random access
`deque`	random access
`list`	bidirectional
Associative containers (first class)	
`set`	bidirectional
`multiset`	bidirectional
`map`	bidirectional
`multimap`	bidirectional

- **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

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

Constant iterators

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

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

Iterator operation	Description
Bidirectional iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.
Random-access iterators	
`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`.
`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`.

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());
```

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 (=> *container*.erase() on the next program instruction), replace
- Other Algorithms
  - ex: count, count_if, accumulate

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*).

Count(n) -> Returns the number of elements in the range [first, last) that compare equal to val.

```
int main () {
  // counting elements in array:
  int myints[] = {10,20,30,30,20,10,10,20};   // 8 elements
  int mycount = std::count (myints, myints+8, 10);
  std::cout << "10 appears " << mycount << " times.\n";

  // counting elements in container:
  std::vector<int> myvector (myints, myints+8);
  mycount = std::count (myvector.begin(), myvector.end(), 20);
  std::cout << "20 appears " << mycount  << " times.\n";

  return 0;
}
```

Output:

```
10 appears 3 times.
20 appears 3 times.
```

Equal(n) -> Compares the elements in the range [first1,last1) with those in the range beginning at first2, and returns true if all of the elements in both ranges match.

```
bool mypredicate (int i, int j) {
  return (i==j);
}

int main () {
  int myints[] = {20,40,60,80,100};               //   myints: 20 40 60 80 100
  std::vector<int>myvector (myints,myints+5);     // myvector: 20 40 60 80 100

  // using default comparison:
  if ( std::equal (myvector.begin(), myvector.end(), myints) )
    std::cout << "The contents of both sequences are equal.\n";
  else
    std::cout << "The contents of both sequences differ.\n";

  myvector[3]=81;                                 // myvector: 20 40 60 81 100

  // using predicate comparison:
  if ( std::equal (myvector.begin(), myvector.end(), myints, mypredicate) )
    std::cout << "The contents of both sequences are equal.\n";
  else
    std::cout << "The contents of both sequences differ.\n";

  return 0;
}
```

Output:

```
The contents of both sequences are equal.
The contents of both sequence differ.
```

Remove(n) -> Transforms the range [first,last) into a range with all the elements that compare equal to val removed, and returns an iterator to the new end of that range.

```
// remove algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::remove

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};      // 10 20 30 30 20 10 10 20

  // bounds of range:
  int* pbegin = myints;                          // ^
  int* pend = myints+sizeof(myints)/sizeof(int); // ^                       ^

  pend = std::remove (pbegin, pend, 20);         // 10 30 30 10 10 ?  ?  ?
                                                 // ^              ^
  std::cout << "range contains:";
  for (int* p=pbegin; p!=pend; ++p)
    std::cout << ' ' << *p;
  std::cout << '\n';

  return 0;
}
```

Output:

```
range contains: 10 30 30 10 10
```

Copy(n) -> Copies the elements in the range [first,last) into the range beginning at result.

```
// copy algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::copy
#include <vector>       // std::vector

int main () {
  int myints[]={10,20,30,40,50,60,70};
  std::vector<int> myvector (7);

  std::copy ( myints, myints+7, myvector.begin() );

  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it = myvector.begin(); it!=myvector.end(); ++it)
    std::cout << ' ' << *it;

  std::cout << '\n';

  return 0;
}
```

Output:

```
myvector contains: 10 20 30 40 50 60 70
```

Random_shuffle(n) -> Rearranges the elements in the range [first,last) randomly.

```
// random_shuffle example
#include <iostream>     // std::cout
#include <algorithm>    // std::random_shuffle
#include <vector>       // std::vector
#include <ctime>        // std::time
#include <cstdlib>      // std::rand, std::srand

// random generator function:
int myrandom (int i) { return std::rand()%i;}

int main () {
  std::srand ( unsigned ( std::time(0) ) );
  std::vector<int> myvector;

  // set some values:
  for (int i=1; i<10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9

  // using built-in random generator:
  std::random_shuffle ( myvector.begin(), myvector.end() );

  // using myrandom:
  std::random_shuffle ( myvector.begin(), myvector.end(), myrandom);

  // print out content:
  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
    std::cout << ' ' << *it;

  std::cout << '\n';

  return 0;
}
```

Possible output:

```
myvector contains: 3 4 1 6 8 9 2 7 5
```

Sort(n) -> Sorts the elements in the range [first,last) into ascending order.

```
bool myfunction (int i,int j) { return (i<j); }

struct myclass {
  bool operator() (int i,int j) { return (i<j);}
} myobject;

int main () {
  int myints[] = {32,71,12,45,26,80,53,33};
  std::vector<int> myvector (myints, myints+8);               // 32 71 12 45 26 80 53 33

  // using default comparison (operator <):
  std::sort (myvector.begin(), myvector.begin()+4);           //(12 32 45 71)26 80 53 33

  // using function as comp
  std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)

  // using object as comp
  std::sort (myvector.begin(), myvector.end(), myobject);     //(12 26 32 33 45 53 71 80)

  // print out content:
  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
myvector contains: 12 26 32 33 45 53 71 80
```

Merge(n) -> Combines the elements in the sorted ranges [first1,last1) and [first2,last2), into a new range beginning at result with all its elements sorted.

```
int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);

  std::sort (first,first+5);
  std::sort (second,second+5);
  std::merge (first,first+5,second,second+5,v.begin());

  std::cout << "The resulting vector contains:";
  for (std::vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
The resulting vector contains: 5 10 10 15 20 20 25 30 40 50
```

Accumulate(n) -> Returns the result of accumulating all the values in the range [first,last) to init.

Output:

```
using default accumulate: 160
using functional's minus: 40
using custom function: 220
using custom object: 280
```

```
int myfunction (int x, int y) {return x+2*y;}
struct myclass {
       int operator()(int x, int y) {return x+3*y;}
} myobject;

int main () {
  int init = 100;
  int numbers[] = {10,20,30};

  std::cout << "using default accumulate: ";
  std::cout << std::accumulate(numbers,numbers+3,init);
  std::cout << '\n';

  std::cout << "using functional's minus: ";
  std::cout << std::accumulate (numbers, numbers+3, init, std::minus<int>());
  std::cout << '\n';

  std::cout << "using custom function: ";
  std::cout << std::accumulate (numbers, numbers+3, init, myfunction);
  std::cout << '\n';

  std::cout << "using custom object: ";
  std::cout << std::accumulate (numbers, numbers+3, init, myobject);
  std::cout << '\n';

  return 0;
}
```

Ao fazer esta instrução, evitamos ter sempre de escrever vector<int>::conts_iterator :

```
typedef vector<int>::const_iterator vecIntIterator;
```

Assim:

```
for (vecIntIterator vPtr = v.begin(); vPtr != v.end(); vPtr++)
      cout << *vPtr << " ";
cout << "}" << endl;
```

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).

Reverse iterators:

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

LIST/SET:

Constructor:

```
// constructing lists
#include <iostream>
#include <list>

int main ()
{
  // constructors used in the same order as described above:
  std::list<int> first;                                // empty list of ints
  std::list<int> second (4,100);                       // four ints with value 100
  std::list<int> third (second.begin(),second.end());  // iterating through second
  std::list<int> fourth (third);                       // a copy of third

  // the iterator constructor can also be used to construct from arrays:
  int myints[] = {16,2,77,29};
  std::list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

  std::cout << "The contents of fifth are: ";
  for (std::list<int>::iterator it = fifth.begin(); it != fifth.end(); it++)
    std::cout << *it << ' ';

  std::cout << '\n';

  return 0;
}
```

Output:

```
The contents of fifth are: 16 2 77 29
```

(LIST/SET)Operator= ->

```
// assignment operator with lists
#include <iostream>
#include <list>

int main ()
{
  std::list<int> first (3);      // list of 3 zero-initialized ints
  std::list<int> second (5);     // list of 5 zero-initialized ints

  second = first;
  first = std::list<int>();

  std::cout << "Size of first: " << int (first.size()) << '\n';
  std::cout << "Size of second: " << int (second.size()) << '\n';
  return 0;
}
```

```
Size of first: 0
Size of second: 3
```

(List/Set/Map).empty() -> Returns whether the list container is empty (i.e. whether its size is 0).

```
int main ()
{
  std::list<int> mylist;
  int sum (0);

  for (int i=1;i<=10;++i) mylist.push_back(i);

  while (!mylist.empty())
  {
     sum += mylist.front();
     mylist.pop_front();
  }

  std::cout << "total: " << sum << '\n';

  return 0;
}
```

Output:

```
total: 55
```

(List/Set/Size).size() -> Returns the number of elements in the list container.

```
// list::size
#include <iostream>
#include <list>

int main ()
{
  std::list<int> myints;
  std::cout << "0. size: " << myints.size() << '\n';

  for (int i=0; i<10; i++) myints.push_back(i);
  std::cout << "1. size: " << myints.size() << '\n';

  myints.insert (myints.begin(),10,100);
  std::cout << "2. size: " << myints.size() << '\n';

  myints.pop_back();
  std::cout << "3. size: " << myints.size() << '\n';

  return 0;
}
```

Output:

```
0. size: 0
1. size: 10
2. size: 20
3. size: 19
```

(List/Set/Map).max_size() -> Returns the maximum number of elements that the list container can hold.

```
// list::max_size
#include <iostream>
#include <list>

int main ()
{
  unsigned int i;
  std::list<int> mylist;

  std::cout << "Enter number of elements: ";
  std::cin >> i;

  if (i<mylist.max_size()) mylist.resize(i);
  else std::cout << "That size exceeds the limit.\n";

  return 0;
}
```

List.front() -> Returns a reference to the first element in the list container.

```
// list::front
#include <iostream>
#include <list>

int main ()
{
  std::list<int> mylist;

  mylist.push_back(77);
  mylist.push_back(22);

  // now front equals 77, and back 22

  mylist.front() -= mylist.back();

  std::cout << "mylist.front() is now " << mylist.front() << '\n';

  return 0;
}
```

Output:

```
mylist.front() is now 55
```

List.back() -> Returns a reference to the last element in the list container.

```
// list::back
#include <iostream>
#include <list>

int main ()
{
  std::list<int> mylist;

  mylist.push_back(10);

  while (mylist.back() != 0)
  {
    mylist.push_back ( mylist.back() -1 );
  }

  std::cout << "mylist contains:";
  for (std::list<int>::iterator it=mylist.begin(); it!=mylist.end() ; ++it)
    std::cout << ' ' << *it;

  std::cout << '\n';

  return 0;
}
```

Output:

```
mylist contains: 10 9 8 7 6 5 4 3 2 1 0
```

List.emplace_front(n) -> Inserts a new element at the beginning of the list, right before its current first element. This new element is constructed in place using args as the arguments for its construction.

```
std::list< std::pair<int,char> > mylist;

mylist.emplace_front(10,'a');
mylist.emplace_front(20,'b');
mylist.emplace_front(30,'c');

std::cout << "mylist contains:";
for (auto& x: mylist)
  std::cout << " (" << x.first << "," << x.second << ")";

std::cout << std::endl;
```

Output:

```
mylist contains: (30,c) (20,b) (10,a)
```

List.assign(n) -> Assigns new contents to the list container, replacing its current contents, and modifying its size accordingly.

```
// list::assign
#include <iostream>
#include <list>

int main ()
{
  std::list<int> first;
  std::list<int> second;

  first.assign (7,100);                      // 7 ints with value 100

  second.assign (first.begin(),first.end()); // a copy of first

  int myints[]={1776,7,4};
  first.assign (myints,myints+3);            // assigning from array

  std::cout << "Size of first: " << int (first.size()) << '\n';
  std::cout << "Size of second: " << int (second.size()) << '\n';
  return 0;
}
```

Output:

```
Size of first: 3
Size of second: 7
```

List.push_front(n) -> Inserts a new element at the beginning of the list, right before its current first element. The content of val is copied (or moved) to the inserted element.

```
// list::push_front
#include <iostream>
#include <list>

int main ()
{
  std::list<int> mylist (2,100);         // two ints with a value of 100
  mylist.push_front (200);
  mylist.push_front (300);

  std::cout << "mylist contains:";
  for (std::list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;

  std::cout << '\n';
  return 0;
}
```

Output:

```
300 200 100 100
```

List.pop_front() -> Removes the first element in the list container, effectively reducing its size by one.

```
// list::pop_front
#include <iostream>
#include <list>

int main ()
{
  std::list<int> mylist;
  mylist.push_back (100);
  mylist.push_back (200);
  mylist.push_back (300);

  std::cout << "Popping out the elements in mylist:";
  while (!mylist.empty())
  {
    std::cout << ' ' << mylist.front();
    mylist.pop_front();
  }

  std::cout << "\nFinal size of mylist is " << mylist.size() << '\n';

  return 0;
}
```

Output:

```
Popping out the elements in mylist: 100 200 300
Final size of mylist is 0
```

List.emplace_back(n) -> Inserts a new element at the end of the list, right after its current last element. This new element is constructed in place using args as the arguments for its construction.

```
// list::emplace_back
#include <iostream>
#include <list>

int main ()
{
  std::list< std::pair<int,char> > mylist;

  mylist.emplace_back(10,'a');
  mylist.emplace_back(20,'b');
  mylist.emplace_back(30,'c');

  std::cout << "mylist contains:";
  for (auto& x: mylist)
    std::cout << " (" << x.first << "," << x.second << ")";

  std::cout << std::endl;
  return 0;
}
```

Output:

```
mylist contains: (10,a) (20,b) (30,c)
```

List.push_back(n) -> Adds a new element at the end of the list container, after its current last element. The content of val is copied (or moved) to the new element.

```
std::cout << "Please enter some integers (enter 0 to end):\n";

do {
  std::cin >> myint;
  mylist.push_back (myint);
} while (myint);

std::cout << "mylist stores " << mylist.size() << " numbers.\n";
```

List.pop_back() -> Removes the last element in the list container, effectively reducing the container size by one.

```
// list::pop_back
#include <iostream>
#include <list>

int main ()
{
  std::list<int> mylist;
  int sum (0);
  mylist.push_back (100);
  mylist.push_back (200);
  mylist.push_back (300);

  while (!mylist.empty())
  {
    sum+=mylist.back();
    mylist.pop_back();
  }

  std::cout << "The elements of mylist summed " << sum << '\n';

  return 0;
}
```

```
The elements of mylist summed 600
```

List.emplace(n) -> The container is extended by inserting a new element at position. This new element is constructed in place using args as the arguments for its construction.

```
// list::emplace
#include <iostream>
#include <list>

int main ()
{
  std::list< std::pair<int,char> > mylist;

  mylist.emplace ( mylist.begin(), 100, 'x' );
  mylist.emplace ( mylist.begin(), 200, 'y' );

  std::cout << "mylist contains:";
  for (auto& x: mylist)
    std::cout << " (" << x.first << "," << x.second << ")";

  std::cout << '\n';
  return 0;
}
```

Output:

```
mylist contains: (200,y) (100,x)
```

List.insert(n) -> The container is extended by inserting new elements before the element at the specified position.

```
// inserting into a list
#include <iostream>
#include <list>
#include <vector>

int main ()
{
  std::list<int> mylist;
  std::list<int>::iterator it;

  // set some initial values:
  for (int i=1; i<=5; ++i) mylist.push_back(i); // 1 2 3 4 5

  it = mylist.begin();
  ++it;       // it points now to number 2           ^

  mylist.insert (it,10);                        // 1 10 2 3 4 5

  // "it" still points to number 2                      ^
  mylist.insert (it,2,20);                      // 1 10 20 20 2 3 4 5

  --it;       // it points now to the second 20            ^

  std::vector<int> myvector (2,30);
  mylist.insert (it,myvector.begin(),myvector.end());
                                                // 1 10 20 30 30 20 2 3 4 5
                                                //               ^
  std::cout << "mylist contains:";
  for (it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
mylist contains: 1 10 20 30 30 20 2 3 4 5
```

List.remove(n) -> Removes from the container all the elements that compare equal to val. This calls the destructor of these objects and reduces the container size by the number of elements removed.

```
int myints[]= {17,89,7,14};
std::list<int> mylist (myints,myints+4);

mylist.remove(89);

std::cout << "mylist contains:";
for (std::list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
  std::cout << ' ' << *it;
std::cout << '\n';
```

Output:

```
mylist contains: 17 7 14
```

(List/Set).erase(n) -> Removes from the list container either a single element (position) or a range of elements ([first,last)).

```
// erasing from list
#include <iostream>
#include <list>

int main ()
{
  std::list<int> mylist;
  std::list<int>::iterator it1,it2;

  // set some values:
  for (int i=1; i<10; ++i) mylist.push_back(i*10);

                           // 10 20 30 40 50 60 70 80 90
  it1 = it2 = mylist.begin(); // ^^
  advance (it2,6);         // ^                 ^
  ++it1;                   //    ^              ^

  it1 = mylist.erase (it1);   // 10 30 40 50 60 70 80 90
                              //    ^           ^

  it2 = mylist.erase (it2);   // 10 30 40 50 60 80 90
                              //    ^           ^

  ++it1;                      //       ^        ^
  --it2;                      //       ^     ^

  mylist.erase (it1,it2);     // 10 30 60 80 90
                              //        ^

  std::cout << "mylist contains:";
  for (it1=mylist.begin(); it1!=mylist.end(); ++it1)
    std::cout << ' ' << *it1;
  std::cout << '\n';

  return 0;
}
```

```
// erasing from set
#include <iostream>
#include <set>

int main ()
{
  std::set<int> myset;
  std::set<int>::iterator it;

  // insert some values:
  for (int i=1; i<10; i++) myset.insert(i*10);  // 10 20 30 40 50 60 70 80 90

  it = myset.begin();
  ++it;                                         // "it" points now to 20

  myset.erase (it);

  myset.erase (40);

  it = myset.find (60);
  myset.erase (it, myset.end());

  std::cout << "myset contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
mylist contains: 10 30 60 80 90
```

Output:

```
myset contains: 10 30 50
```

(List/Set/Map).swap(n) -> Exchanges the content of the container by the content of x, which is another list of the same type. Sizes may differ.

```
// swap lists
#include <iostream>
#include <list>

int main ()
{
  std::list<int> first (3,100);   // three ints with a value of 100
  std::list<int> second (5,200);  // five ints with a value of 200

  first.swap(second);

  std::cout << "first contains:";
  for (std::list<int>::iterator it=first.begin(); it!=first.end(); it++)
    std::cout << ' ' << *it;
  std::cout << '\n';

  std::cout << "second contains:";
  for (std::list<int>::iterator it=second.begin(); it!=second.end(); it++)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
first contains: 200 200 200 200 200
second contains: 100 100 100
```

List.resize() -> Resizes the container so that it contains n elements.

```
// resizing list
#include <iostream>
#include <list>

int main ()
{
  std::list<int> mylist;

  // set some initial content:
  for (int i=1; i<10; ++i) mylist.push_back(i);

  mylist.resize(5);
  mylist.resize(8,100);
  mylist.resize(12);

  std::cout << "mylist contains:";
  for (std::list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;

  std::cout << '\n';

  return 0;
}
```

```
mylist contains: 1 2 3 4 5 100 100 100 0 0 0 0
```

(List/Set/Map).clear() -> Removes all elements from the list container (which are destroyed), and leaving the container with a size of 0.

```
std::list<int> mylist;
std::list<int>::iterator it;

mylist.push_back (100);
mylist.push_back (200);
mylist.push_back (300);

std::cout << "mylist contains:";
for (it=mylist.begin(); it!=mylist.end(); ++it)
  std::cout << ' ' << *it;
std::cout << '\n';
```

```
mylist.clear();
mylist.push_back (1101);
mylist.push_back (2202);

std::cout << "mylist contains:";
for (it=mylist.begin(); it!=mylist.end(); ++it)
  std::cout << ' ' << *it;
std::cout << '\n';
```

Output:

```
mylist contains: 100 200 300
mylist contains: 1101 2202
```

List.splice(n) -> Transfers elements from x into the container, inserting them at position.

```
// splicing lists
#include <iostream>
#include <list>

int main ()
{
  std::list<int> mylist1, mylist2;
  std::list<int>::iterator it;

  // set some initial values:
  for (int i=1; i<=4; ++i)
     mylist1.push_back(i);      // mylist1: 1 2 3 4

  for (int i=1; i<=3; ++i)
     mylist2.push_back(i*10);   // mylist2: 10 20 30

  it = mylist1.begin();
  ++it;                         // points to 2

  mylist1.splice (it, mylist2); // mylist1: 1 10 20 30 2 3 4
                                // mylist2 (empty)
                                // "it" still points to 2 (the 5th element)

  mylist2.splice (mylist2.begin(),mylist1, it);
                                // mylist1: 1 10 20 30 3 4
                                // mylist2: 2
                                // "it" is now invalid.
```

```
  it = mylist1.begin();
  std::advance(it,3);           // "it" points now to 30

  mylist1.splice ( mylist1.begin(), mylist1, it, mylist1.end());
                                // mylist1: 30 3 4 1 10 20

  std::cout << "mylist1 contains:";
  for (it=mylist1.begin(); it!=mylist1.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  std::cout << "mylist2 contains:";
  for (it=mylist2.begin(); it!=mylist2.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
mylist1 contains: 30 3 4 1 10 20
mylist2 contains: 2
```

List.remove_if(n) -> Removes from the container all the elements for which Predicate pred returns true. This calls the destructor of these objects and reduces the container size by the number of elements removed.

```
// list::remove_if
#include <iostream>
#include <list>

// a predicate implemented as a function:
bool single_digit (const int& value) { return (value<10); }

// a predicate implemented as a class:
struct is_odd {
  bool operator() (const int& value) { return (value%2)==1; }
};

int main ()
{
  int myints[]= {15,36,7,17,20,39,4,1};
  std::list<int> mylist (myints,myints+8);   // 15 36 7 17 20 39 4 1

  mylist.remove_if (single_digit);           // 15 36 17 20 39

  mylist.remove_if (is_odd());               // 36 20

  std::cout << "mylist contains:";
  for (std::list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
mylist contains: 36 20
```

List.unique(n) -> The version with no parameters (1), removes all but the first element from every consecutive group of equal elements in the container.

```
// list::unique
#include <iostream>
#include <cmath>
#include <list>

// a binary predicate implemented as a function:
bool same_integral_part (double first, double second)
{ return ( int(first)==int(second) ); }

// a binary predicate implemented as a class:
struct is_near {
  bool operator() (double first, double second)
  { return (fabs(first-second)<5.0); }
};

int main ()
{
  double mydoubles[]={ 12.15,  2.72, 73.0,  12.77,  3.14,
                       12.77, 73.35, 72.25, 15.3,  72.25 };
  std::list<double> mylist (mydoubles,mydoubles+10);

  mylist.sort();             //  2.72,  3.14, 12.15, 12.77, 12.77,
                             // 15.3,  72.25, 72.25, 73.0,  73.35
```

```
  mylist.unique();           //  2.72,  3.14, 12.15, 12.77
                             // 15.3,  72.25, 73.0,  73.35

  mylist.unique (same_integral_part);  //  2.72,  3.14, 12.15
                                       // 15.3,  72.25, 73.0

  mylist.unique (is_near());           //  2.72, 12.15, 72.25

  std::cout << "mylist contains:";
  for (std::list<double>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
mylist contains: 2.72 12.15 72.25
```

List.merge(n) -> Merges x into the list by transferring all of its elements at their respective ordered positions into the container (both containers shall already be ordered).

```
// list::merge
#include <iostream>
#include <list>

// compare only integral part:
bool mycomparison (double first, double second)
{ return ( int(first)<int(second) ); }

int main ()
{
  std::list<double> first, second;

  first.push_back (3.1);
  first.push_back (2.2);
  first.push_back (2.9);

  second.push_back (3.7);
  second.push_back (7.1);
  second.push_back (1.4);

  first.sort();
  second.sort();

  first.merge(second);

  // (second is now empty)

  second.push_back (2.1);

  first.merge(second,mycomparison);

  std::cout << "first contains:";
  for (std::list<double>::iterator it=first.begin(); it!=first.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
first contains: 1.4 2.2 2.9 2.1 3.1 3.7 7.1
```

List.sort() -> Sorts the elements in the list, altering their position within the container.

```
// list::sort
#include <iostream>
#include <list>
#include <string>
#include <cctype>

// comparison, not case sensitive.
bool compare_nocase (const std::string& first, const std::string& second)
{
  unsigned int i=0;
  while ( (i<first.length()) && (i<second.length()) )
  {
    if (tolower(first[i])<tolower(second[i])) return true;
    else if (tolower(first[i])>tolower(second[i])) return false;
    ++i;
  }
  return ( first.length() < second.length() );
}

int main ()
{
  std::list<std::string> mylist;
  std::list<std::string>::iterator it;
  mylist.push_back ("one");
  mylist.push_back ("two");
  mylist.push_back ("Three");

  mylist.sort();

  std::cout << "mylist contains:";
  for (it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  mylist.sort(compare_nocase);

  std::cout << "mylist contains:";
  for (it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
mylist contains: Three one two
mylist contains: one Three two
```

List.reverse() -> Reverses the order of the elements in the list container.

```
// reversing list
#include <iostream>
#include <list>

int main ()
{
  std::list<int> mylist;

  for (int i=1; i<10; ++i) mylist.push_back(i);

  mylist.reverse();

  std::cout << "mylist contains:";
  for (std::list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;

  std::cout << '\n';

  return 0;
}
```

Output:

```
mylist contains: 9 8 7 6 5 4 3 2 1
```

SET

Constructor() :

```
// constructing sets
#include <iostream>
#include <set>

bool fncomp (int lhs, int rhs) {return lhs<rhs;}

struct classcomp {
  bool operator() (const int& lhs, const int& rhs) const
  {return lhs<rhs;}
};

int main ()
{
  std::set<int> first;                           // empty set of ints

  int myints[]= {10,20,30,40,50};
  std::set<int> second (myints,myints+5);        // range

  std::set<int> third (second);                  // a copy of second

  std::set<int> fourth (second.begin(), second.end());  // iterator ctor.

  std::set<int,classcomp> fifth;                 // class as Compare

  bool(*fn_pt)(int,int) = fncomp;
  std::set<int,bool(*)(int,int)> sixth (fn_pt);  // function pointer as Compare

  return 0;
}
```

Set.insert(n) -> Extends the container by inserting new elements, effectively increasing the container size by the number of elements inserted.

```cpp
// set::insert (C++98)
#include <iostream>
#include <set>

int main ()
{
  std::set<int> myset;
  std::set<int>::iterator it;
  std::pair<std::set<int>::iterator,bool> ret;

  // set some initial values:
  for (int i=1; i<=5; ++i) myset.insert(i*10);    // set: 10 20 30 40 50

  ret = myset.insert(20);               // no new element inserted

  if (ret.second==false) it=ret.first;  // "it" now points to element 20

  myset.insert (it,25);                 // max efficiency inserting
  myset.insert (it,24);                 // max efficiency inserting
  myset.insert (it,26);                 // no max efficiency inserting

  int myints[]= {5,10,15};              // 10 already in set, not inserted
  myset.insert (myints,myints+3);

  std::cout << "myset contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
myset contains: 5 10 15 20 24 25 26 30 40 50
```

Set.emplace(n) -> Inserts a new element in the set, if unique. This new element is constructed in place using args as the arguments for its construction.

```cpp
// set::emplace
#include <iostream>
#include <set>
#include <string>

int main ()
{
  std::set<std::string> myset;

  myset.emplace("foo");
  myset.emplace("bar");
  auto ret = myset.emplace("foo");

  if (!ret.second) std::cout << "foo already exists in myset\n";

  return 0;
}
```

Output:

```
foo already exists in myset
```

Set.count(n) -> Searches the container for elements equivalent to val and returns the number of matches.

```cpp
// set::count
#include <iostream>
#include <set>

int main ()
{
  std::set<int> myset;

  // set some initial values:
  for (int i=1; i<5; ++i) myset.insert(i*3);    // set: 3 6 9 12

  for (int i=0; i<10; ++i)
  {
    std::cout << i;
    if (myset.count(i)!=0)
      std::cout << " is an element of myset.\n";
    else
      std::cout << " is not an element of myset.\n";
  }

  return 0;
}
```

Output:

```
0 is not an element of myset.
1 is not an element of myset.
2 is not an element of myset.
3 is an element of myset.
4 is not an element of myset.
5 is not an element of myset.
6 is an element of myset.
7 is not an element of myset.
8 is not an element of myset.
9 is an element of myset.
```

Set.lower_bound(n) -> Returns an iterator pointing to the first element in the container which is not considered to go before val (i.e., either it is equivalent or goes after).

Set.upper_bound(n) -> Returns an iterator pointing to the first element in the container which is considered to go after val.

Notice that `lower_bound(30)` returns an iterator to `30`, whereas `upper_bound(60)` returns an iterator to `70`.

```
myset contains: 10 20 70 80 90
```

```cpp
// set::lower_bound/upper_bound
#include <iostream>
#include <set>

int main ()
{
  std::set<int> myset;
  std::set<int>::iterator itlow,itup;

  for (int i=1; i<10; i++) myset.insert(i*10); // 10 20 30 40 50 60 70 80 90

  itlow=myset.lower_bound (30);                //       ^
  itup=myset.upper_bound (60);                 //                   ^

  myset.erase(itlow,itup);                     // 10 20 70 80 90

  std::cout << "myset contains:";
  for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Set.find(n) -> Searches the container for an element equivalent to val and returns an iterator to it if found, otherwise it returns an iterator to set::end.

```
// set::find
#include <iostream>
#include <set>

int main ()
{
  std::set<int> myset;
  std::set<int>::iterator it;

  // set some initial values:
  for (int i=1; i<=5; i++) myset.insert(i*10);    // set: 10 20 30 40 50

  it=myset.find(20);
  myset.erase (it);
  myset.erase (myset.find(40));

  std::cout << "myset contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output:

```
myset contains: 10 30 50
```

MAP

Constuctor:

```
// constructing maps
#include <iostream>
#include <map>

bool fncomp (char lhs, char rhs) {return lhs<rhs;}

struct classcomp {
  bool operator() (const char& lhs, const char& rhs) const
  {return lhs<rhs;}
};

int main ()
{
  std::map<char,int> first;

  first['a']=10;
  first['b']=30;
  first['c']=50;
  first['d']=70;

  std::map<char,int> second (first.begin(),first.end());

  std::map<char,int> third (second);

  std::map<char,int,classcomp> fourth;                 // class as Compare

  bool(*fn_pt)(char,char) = fncomp;
  std::map<char,int,bool(*)(char,char)> fifth (fn_pt); // function pointer as Compare

  return 0;
}
```

Operator=

```
// assignment operator with maps
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> first;
  std::map<char,int> second;

  first['x']=8;
  first['y']=16;
  first['z']=32;

  second=first;                // second now contains 3 ints
  first=std::map<char,int>();  // and first is now empty

  std::cout << "Size of first: " << first.size() << '\n';
  std::cout << "Size of second: " << second.size() << '\n';
  return 0;
}
```

Output:

```
Size of first: 0
Size of second: 3
```

Operator[] -> If k matches the key of an element in the container, the function returns a reference to its mapped value.

If k does not match the key of any element in the container, the function inserts a new element with that key and returns a reference to its mapped value. Notice that this always increases the container size by one, even if no mapped value is assigned to the element (the element is constructed using its default constructor).

```
// accessing mapped values
#include <iostream>
#include <map>
#include <string>

int main ()
{
  std::map<char,std::string> mymap;

  mymap['a']="an element";
  mymap['b']="another element";
  mymap['c']=mymap['b'];

  std::cout << "mymap['a'] is " << mymap['a'] << '\n';
  std::cout << "mymap['b'] is " << mymap['b'] << '\n';
  std::cout << "mymap['c'] is " << mymap['c'] << '\n';
  std::cout << "mymap['d'] is " << mymap['d'] << '\n';

  std::cout << "mymap now contains " << mymap.size() << " elements.\n";

  return 0;
}
```

Notice how the last access (to element 'd') inserts a new element in the map with that key and initialized to its default value (an empty string) even though it is accessed only to retrieve its value. Member function map::find does not produce this effect.

Output:

```
mymap['a'] is an element
mymap['b'] is another element
mymap['c'] is another element
mymap['d'] is
mymap now contains 4 elements.
```

Map.at(n) -> Returns a reference to the mapped value of the element identified with key k.
If k does not match the key of any element in the container, the function throws an out_of_range exception.

```cpp
// map::at
#include <iostream>
#include <string>
#include <map>

int main ()
{
  std::map<std::string,int> mymap = {
                 { "alpha", 0 },
                 { "beta", 0 },
                 { "gamma", 0 } };

  mymap.at("alpha") = 10;
  mymap.at("beta") = 20;
  mymap.at("gamma") = 30;

  for (auto& x: mymap) {
    std::cout << x.first << ": " << x.second << '\n';
  }

  return 0;
}
```

Possible output:

```
alpha: 10
beta: 20
gamma: 30
```

Map.insert(n) -> Extends the container by inserting new elements, effectively increasing the container size by the number of elements inserted.

```cpp
// map::insert (C++98)
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  // first insert function version (single parameter):
  mymap.insert ( std::pair<char,int>('a',100) );
  mymap.insert ( std::pair<char,int>('z',200) );

  std::pair<std::map<char,int>::iterator,bool> ret;
  ret = mymap.insert ( std::pair<char,int>('z',500) );
  if (ret.second==false) {
    std::cout << "element 'z' already existed";
    std::cout << " with a value of " << ret.first->second << '\n';
  }

  // second insert function version (with hint position):
  std::map<char,int>::iterator it = mymap.begin();
  mymap.insert (it, std::pair<char,int>('b',300));  // max efficiency inserting
  mymap.insert (it, std::pair<char,int>('c',400));  // no max efficiency inserting

  // third insert function version (range insertion):
  std::map<char,int> anothermap;
  anothermap.insert(mymap.begin(),mymap.find('c'));

  // showing contents:
  std::cout << "mymap contains:\n";
  for (it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';

  std::cout << "anothermap contains:\n";
  for (it=anothermap.begin(); it!=anothermap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';

  return 0;
}
```

Output:

```
element 'z' already existed with a value of 200
mymap contains:
a => 100
b => 300
c => 400
z => 200
anothermap contains:
a => 100
b => 300
```

Map.erase(n) -> Removes from the map container either a single element or a range of elements ([first,last)).

```cpp
// erasing from map
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator it;

  // insert some values:
  mymap['a']=10;
  mymap['b']=20;
  mymap['c']=30;
  mymap['d']=40;
  mymap['e']=50;
  mymap['f']=60;

  it=mymap.find('b');
  mymap.erase (it);                   // erasing by iterator

  mymap.erase ('c');                  // erasing by key

  it=mymap.find ('e');
  mymap.erase ( it, mymap.end() );    // erasing by range

  // show content:
  for (it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';

  return 0;
}
```

Output:

```
a => 10
d => 40
```

Map.emplace(n) -> Inserts a new element in the map if its key is unique. This new element is constructed in place using args as the arguments for the construction of a value_type (which is an object of a pair type).

```cpp
// map::emplace
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  mymap.emplace('x',100);
  mymap.emplace('y',200);
  mymap.emplace('z',100);

  std::cout << "mymap contains:";
  for (auto& x: mymap)
    std::cout << " [" << x.first << ':' << x.second << ']';
  std::cout << '\n';

  return 0;
}
```

Output:

```
mymap contains: [x:100] [y:200] [z:100]
```

Map.find(n) -> Searches the container for an element with a key equivalent to k and returns an iterator to it if found, otherwise it returns an iterator to map::end.

```
// map::find
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator it;

  mymap['a']=50;
  mymap['b']=100;
  mymap['c']=150;
  mymap['d']=200;

  it = mymap.find('b');
  if (it != mymap.end())
    mymap.erase (it);

  // print content:
  std::cout << "elements in mymap:" << '\n';
  std::cout << "a => " << mymap.find('a')->second << '\n';
  std::cout << "c => " << mymap.find('c')->second << '\n';
  std::cout << "d => " << mymap.find('d')->second << '\n';

  return 0;
}
```

Output:

```
elements in mymap:
a => 50
c => 150
d => 200
```

Map.count(n) -> Searches the container for elements with a key equivalent to k and returns the number of matches.

```
// map::count
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;
  char c;

  mymap ['a']=101;
  mymap ['c']=202;
  mymap ['f']=303;

  for (c='a'; c<'h'; c++)
  {
    std::cout << c;
    if (mymap.count(c)>0)
      std::cout << " is an element of mymap.\n";
    else
      std::cout << " is not an element of mymap.\n";
  }

  return 0;
}
```

Output:

```
a is an element of mymap.
b is not an element of mymap.
c is an element of mymap.
d is not an element of mymap.
e is not an element of mymap.
f is an element of mymap.
g is not an element of mymap.
```

Map.lower_bound(n) -> Returns an iterator pointing to the first element in the container whose key is not considered to go before k (i.e., either it is equivalent or goes after).

Map.upper_bound(n) -> Returns an iterator pointing to the first element in the container whose key is considered to go after k.

```
// map::lower_bound/upper_bound
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator itlow,itup;

  mymap['a']=20;
  mymap['b']=40;
  mymap['c']=60;
  mymap['d']=80;
  mymap['e']=100;

  itlow=mymap.lower_bound ('b');  // itlow points to b
  itup=mymap.upper_bound ('d');   // itup points to e (not d!)

  mymap.erase(itlow,itup);        // erases [itlow,itup)

  // print content:
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';

  return 0;
}
```

```
a => 20
e => 100
```

```
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
```

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

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

No caso de já existir um contacto com o mesmo número, o bool do pair fica falso (ao dar insert num map, se já existir essa key, não ocorre o insert):

```
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
    //
}
```

MULTIMAP permite a existência de elementos com a mesma key;

```
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;
};
```

```
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;
}
```

Outra forma de usar iterators:

```
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;
}
```

Outro exemplo:

```
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?

    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;
}
```

```
Write a text; end with <ENTER> followed by <CTRL-Z>
O meu nome e Tiago o

^Z

Word list:
O: 1
Tiago: 1
e: 1
meu: 1
nome: 1
o: 1

Word to search ? amanha
word_count[amanha] = 0

Word list:
O: 1
Tiago: 1
amanha: 0
e: 1
meu: 1
nome: 1
o: 1
```

```
//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";
```

```cpp
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;
}
```

```
v1 - fill(v1.begin(), v1.end(),1):
  1   1   1   1   1   1   1   1   1   1
v1 - fill_n (v1.begin()+4, 3, 2):
  1   1   1   1   2   2   2   1   1   1
v2 - generate(...,myRand):
  5   9   6   3   9   8   6   1   8   2
```

```cpp
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);
```

```
v1 - fill(v1.begin(), v1.end(),1):
  1   1   1   1   1   1   1   1   1   1
v1 - fill_n (v1.begin()+4, 3, 2):
  1   1   1   1   2   2   2   1   1   1
v2 - generate(...,myRand):
  7  10   3   3   5   8   5   8   4   7
v4: copy(v2.begin(),v2.begin()+5,v4.begin()+2):
  0   0   7  10   3   3   5   0   0   0
reverse(v2.begin(),v2.end()):
  7   4   8   5   8   5   3   3  10   7
merge(v1...,v2...,v3.begin()):
  1   1   1   1   1   1   1   2   2   2   3   3   4   5   5   7   7   8   8  10
```

SINCE C++11

ALTERNATIVE TO THE USE OF calcSquare():

```cpp
transform ( v2.begin(),v2.end(),v3.begin(),calcSquare );
```

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

```cpp
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.

- 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.

```
=========================================================================
FRIEND FUNCTIONS
=========================================================================
```

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

```cpp
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
            );
}
```

```cpp
#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;
};
```

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
{
   ...
}
```

OPERATOR OVERLOADING

```
// 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
```

```
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
```

```
/*
FRACTION.CPP
OPERATOR OVERLOADING examples
Adapted from Big C++ book
*/
#include "fraction.h"
#include <string>
#include <sstream>
#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) ...

... 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 ...

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 assigment 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 atribute 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 sintactically 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

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;
```

```
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;
}
```

// RETURNING POINTERS AND REFERENCES - TWO SIMPLE EXAMPLES

```
// WHAT DOES THIS PROGRAM DO ?

#include <iostream>
#include <cstddef>

using namespace std;

int * f(int vec[], size_t vec_size, int value)
{
      for (size_t i =0; i<vec_size; i++)
            if (vec[i]==value)
                  return &vec[i];
      return NULL;
}

int main()
{
      int a[3] = {1,2,3};

      int * px = f(a,sizeof(a)/sizeof(int), 2);
      // int * px = f(a,sizeof(a)/sizeof(int),5);  // TRY THIS

      if (px != NULL)
      {
            cout << *px << endl << endl;
            *px = 10;
      }

      for (int i=0; i<3; i++)
            cout << a[i] << endl;

}
```

```
#include <iostream>
#include <cstddef>

using namespace std;

int & f(int a[], size_t i)
{
      return a[i];  // NOTE: null references are prohibited;
                    // compare w/previous example
                    // were NULL pointer is returned in some cases
}

int main()
{
  int a[3] = {1,2,3};

  int & x = f(a,1);

  cout << "x = " << x << endl << endl;

  x = 20;           // NOTE: equivalent to f(a,1) = 10;
  // f(a,2) = 30;  // a function used on left side of an assigment…?!!!

  for (int i=0; i<3; i++)
    cout << "a[" << i << "] = " << a[i] << endl;
}
```

```
// 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;
}
```

```
// ... thus enabling the use of cascaded 'set_operations':
d.setDay(10).setMonth(5).setYear(2016);
d.show();
```

More exemples:

```
ostream& operator<<(ostream& out, const Position &p)
{
  out << "(" << p.getX() << "," << p.getY() << ")";
  return out;
}
```

```
Position& get();  // NOTE: not const

Position& Position::get()
{
  return *this;
}
```

```
Position* setX(int x);
Position* setY(int y);

Position* Position::setX(int x)
{
  this->x = x;
  return this;
}

Position* Position::setY(int y)
{
  this->y = y;
  return this;
}
```

```
p1Ptr->setX(30)->setY(40);
```

Implementar operators em classes criadas para poder utilizá-las em sets (por exemplo):

```
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;
};
```

```
bool operator<(const Person& left, const Person& right)
{
      return left.name < right.name; // OR left.age < right.age; you decide
}
```

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!=**

No caso de não ter operador:

- an identical compiling error would occur if, for example, you wanted to declare a map whose key is a Person.

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;
```

OVERLOADING THE () FUNCTION CALL OPERATOR / FUNCTION OBJECTS

- 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=** implementattion,
  for our own **String** class)

```
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 previsous 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;

      return 0;
}
```

Algumas formas de usar Functions objetcs:

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

...

int main() {
...
      vector<int> v(10);

      generate(v.begin(),v.end(),myRand);
      displayVec("generate(...,myRand)",v);
...
```

- 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 ...

Possíveis soluções:
Tornar variáveis globais;
Criar uma classe;

```
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);
}
```

Após criar a classe, na main:

- Now, each time generate() calls its function parameter, it uses the call operator from object 'r'.

```
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);
```

```
// 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();
}
```

O que permite:

```
int main()
{
      set<Person,SortPersonByName> s;
      Person p;
      string name;
      unsigned age;
```

```
template < class T,                          // set::key_type/value_type
           class Compare = less<T>,          // set::key_compare/value_compare
           class Alloc = allocator<T>        // set::allocator_type
         > class set;
```

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 ...

... 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 ir 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

```
//-----------------------------------------------------------------------
// 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 ? ...
}
```

```
// 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;
}
```

```
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); // invoque String simple constructor

      // destroy temporary string
      delete[] tmpCStr; // C-string

      return tmpStr;
}
```

```
#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
```

```
//-----------------------------------------------------------------
// 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)
//-----------------------------------------------------------------
```

```
// DESTRUCTOR - in this case, it is fundamental to a 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 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;
}
```

- **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 dinamically 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 **assigment 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 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 refered to using just **::**
- The global namespace does not require the using directive

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";
      }
}
```

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:

```
namespace ns1
{
   fun1();
   my_function();
}
```

```
namespace ns2
{
   fun2();
   my_function();
}
```

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 !!!

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;
```

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( );
      }
 //...
}
```

- 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

```
using ns1::my_function;
using ns2::my_function;
```

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;

- 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.

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
    {
      ...
    };
}
```

Example:

```
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();
    }
}
```

```
ns1 creator: Mary
ns2 creator: John

hello from Mary!
hello from John!
```

==========

OBJECT ORIENTED LANGUAGES

==========

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)*

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)

- **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.

==========

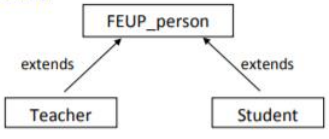
INHERITANCE

==========

- **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

- **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 ommited **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 acessible 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.

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;

};
```

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

void FeupPerson::showRecord() {  // definition in FeupPerson.cpp
  std::cout << "----------------------------\n";
  std::cout << "Name: " << name << " ID: " << id << " Address: " << address
<< "\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

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;
}
```

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 descendents 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

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

}

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

```
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 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";
}

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

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

- 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 explicity, 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

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)`

```
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
}
```

class "FeupPerson" has no member "addClassTaken"

```
----------------------------
Name : John Souza
ID : 234
Address : St. John Street
Course : MIEIC
Classes taken:
EDA
EDI
----------------------------
----------------------------
Name : Liz Tanner
ID : 345
Address : St. Liz Street
----------------------------
```

POLYMORPHISM (virtual functions)

- 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",
"MIB");
  p[1] = Student(123, "Nuno Silva", "Rua do Nuno", "MIB", 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 assigments in yellow are possible,
  some of the fields of Teacher/Student shall be lost (slicing problem)

```
int main()
{
  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);

  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**

- However it is possible to **alert the compiler** that the function call must be preceeded 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;
};
```

- 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

- 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 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

- **But** when one runs the above code, the ouput 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[1]** is of type **FeupPerson***

- The outupt 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:
-----------------------------
```

- 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[1]** points,
    either
    `Teacher::showRecord()`
  - or
    `Student::showRecord()`

- Only member functions can be virtual.
- 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())
------------------------------
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 polimorphic 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?
    ```

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 as the compiler that the prototype of the function in the derived class is the same as in the base classe, the specifier overrride 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 overrriding or further overrriding.
- 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++, construtors 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.*

Pure virtual functions or abstract functions

```
// Example adapted from
// http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming#cite_note-tcpl-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")
        };

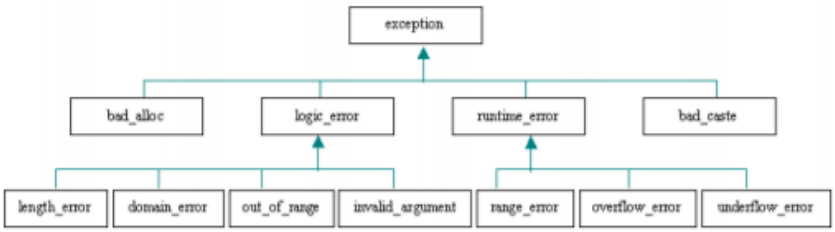
        for(int i = 0; i < 3; i++)
        {
                cout << animals[i]->name << ": " << animals[i]->talk() << endl;
                delete animals[i];
        }
        return 0;
}
```

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 ...)

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.

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 descibed 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...)

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

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
- External flags
  - ```
    int n = atoi(num_CString);
    if (errno == ERANGE) // => #include <cerrno>
      ..... ;
    ```
- In debug mode, use **assert** to halt execution

- 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*

- 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 "**

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.

```
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";
      }

}
```

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 – 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
}
```

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
```

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.

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";
}
```

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 – additional notes

- 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 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

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.

```
// ## Preprocessor

// Comment to end of line
/* Multi-line comment */
#include  <stdio.h>         // Insert standard header file
#include "myfile.h"         // Insert file in current directory
#define X some text         // Replace X with some text
#define F(a,b) a+b          // Replace F(1,2) with 1+2
#define X \
 some text                  // Multiline definition
#undef X                    // Remove definition
#if defined(X)              // Conditional compilation (#ifdef X)
#else                       // Optional (#ifndef X or #if !defined(X))
#endif                      // Required after #if, #ifdef


// ## Literals

255, 0377, 0xff             // Integers (decimal, octal, hex)
2147483647L, 0x7fffffffl    // Long (32-bit) integers
123.0, 1.23e2               // double (real) numbers
'a', '\141', '\x61'         // Character (literal, octal, hex)
'\n', '\\', '\'', '\"'      // Newline, backslash, single quote, double quote
"string\n"                  // Array of characters ending with newline and \0
"hello" "world"             // Concatenated strings
true, false                 // bool constants 1 and 0
nullptr                     // Pointer type with the address of 0

// ## Declarations

int x;                      // Declare x to be an integer (value undefined)
int x=255;                  // Declare and initialize x to 255
short s; long l;            // Usually 16 or 32 bit integer (int may be either)
char c='a';                 // Usually 8 bit character
unsigned char u=255;
signed char s=-1;           // char might be either
unsigned long x =
      0xffffffffL;          // short, int, long are signed
float f; double d;          // Single or double precision real (never unsigned)
```

```
bool b=true;               // true or false, may also use int (1 or 0)
int a, b, c;              // Multiple declarations
int a[10];                // Array of 10 ints (a[0] through a[9])
int a[]={0,1,2};           // Initialized array (or a[3]={0,1,2}; )
int a[2][2]={{1,2},{4,5}};  // Array of array of ints
char s[]="hello";          // String (6 elements including '\0')
std::string s = "Hello"     // Creates string object with value "Hello"
std::string s = R"(Hello
World)";                  // Creates string object with value "Hello\nWorld"
int* p;                   // p is a pointer to (address of) int
char* s="hello";           // s points to unnamed array containing "hello"
void* p=nullptr;           // Address of untyped memory (nullptr is 0)
int& r=x;                 // r is a reference to (alias of) int x
enum weekend {SAT,SUN};     // weekend is a type with values SAT and SUN
enum weekend day;          // day is a variable of type weekend
enum weekend{SAT=0,SUN=1};  // Explicit representation as int
enum {SAT,SUN} day;         // Anonymous enum
enum class Color {Red,Blue};// Color is a strict type with values Red and Blue
Color x = Color::Red;       // Assign Color x to red
typedef String char*;       // String s; means char* s;
const int c=3;            // Constants must be initialized, cannot assign to
const int* p=a;            // Contents of p (elements of a) are constant
int* const p=a;            // p (but not contents) are constant
const int* const p=a;       // Both p and its contents are constant
const int& cr=x;           // cr cannot be assigned to change x
int8_t,uint8_t,int16_t,
uint16_t,int32_t,uint32_t,
int64_t,uint64_t           // Fixed length standard types
auto it = m.begin();        // Declares it to the result of m.begin()
auto const param = config["param"];
// Declares it to the const result
auto& s = singleton::instance();
// Declares it to a reference of the result


// ## STORAGE Classes

int x;                    // Auto (memory exists only while in scope)
static int x;              // Global lifetime even if local scope
extern int x;              // Information only, declared elsewhere


// ## Statements


x=y;                      // Every expression is a statement
int x;                    // Declarations are statements
;                         // Empty statement
{                         // A block is a single statement
   int x;                 // Scope of x is from declaration to end of block
}
if (x) a;                 // If x is true (not 0), evaluate a
else if (y) b;             // If not x and y (optional, may be repeated)
else c;                   // If not x and not y (optional)

while (x) a;               // Repeat 0 or more times while x is true

for (x; y; z) a;           // Equivalent to: x; while(y) {a; z;}

for (x : y) a;             // Range-based for loop e.g.
// for (auto& x in someList) x.y();

do a; while (x);           // Equivalent to: a; while(x) a;

switch (x) {               // x must be int
case X1: a;               // If x == X1 (must be a const), jump here
case X2: b;               // Else if x == X2, jump here
default: c;               // Else jump here (optional)
}
break;                    // Jump out of while, do, or for loop, or switch
continue;                  // Jump to bottom of while, do, or for loop
return x;                  // Return x from function to caller
try { a; }
catch (T t) { b; }         // If a throws a T, then jump here
catch (...) { c; }         // If a throws something else, jump here
```

```
// ## Functions

int f(int x, int y);        // f is a function taking 2 ints and returning int
void f();                   // f is a procedure taking no arguments
void f(int a=0);            // f() is equivalent to f(0)
f();                        // Default return type is int
inline f();                 // Optimize for speed
f() { statements; }         // Function definition (must be global)
T operator+(T x, T y);      // a+b (if type T) calls operator+(a, b)
T operator-(T x);           // -a calls function operator-(a)
T operator++(int);          // postfix ++ or -- (parameter ignored)
extern "C" {void f();}      // f() was compiled in C

// Function parameters and return values may be of any type. A function must either be declared or defined before
// it is used. It may be declared first and defined later. Every program consists of a set of a set of global variable
// declarations and a set of function definitions (possibly in separate files), one of which must be:


int main()  { statements... }     // or
int main(int argc, char* argv[]) { statements... }


// `argv` is an array of `argc` strings from the command line.
// By convention, `main` returns status `0` if successful, `1` or higher for errors.
//
// Functions with different parameters may have the same name (overloading). Operators except `::` `.` `.*` `?:`
may be overloaded.
// Precedence order is not affected. New operators may not be created.

// ## Expressions

// Operators are grouped by precedence, highest first. Unary operators and assignment evaluate right to left. All
// others are left to right. Precedence does not affect order of evaluation, which is undefined. There are no run time
// checks for arrays out of bounds, invalid pointers, etc.

T::X                    // Name X defined in class T
N::X                    // Name X defined in namespace N
::X                     // Global name X
t.x                     // Member x of struct or class t
p-> x                   // Member x of struct or class pointed to by p
a[i]                    // i'th element of array a
f(x,y)                  // Call to function f with arguments x and y
T(x,y)                  // Object of class T initialized with x and y
x++                     // Add 1 to x, evaluates to original x (postfix)
x--                     // Subtract 1 from x, evaluates to original x
typeid(x)               // Type of x
typeid(T)               // Equals typeid(x) if x is a T
dynamic_cast< T>(x)       // Converts x to a T, checked at run time.
static_cast< T>(x)        // Converts x to a T, not checked
reinterpret_cast< T>(x)   // Interpret bits of x as a T
const_cast< T>(x)         // Converts x to same type T but not const

sizeof x                // Number of bytes used to represent object x
sizeof(T)               // Number of bytes to represent type T
++x                     // Add 1 to x, evaluates to new value (prefix)
--x                     // Subtract 1 from x, evaluates to new value
~x                      // Bitwise complement of x
!x                      // true if x is 0, else false (1 or 0 in C)
-x                      // Unary minus
+x                      // Unary plus (default)
&x                      // Address of x
*p                      // Contents of address p (*&x equals x)
new T                   // Address of newly allocated T object
new T(x, y)             // Address of a T initialized with x, y
new T[x]                // Address of allocated n-element array of T
delete p                // Destroy and free object at address p
delete[] p              // Destroy and free array of objects at p
(T) x                   // Convert x to T (obsolete, use .._cast<T>(x))

x * y                   // Multiply
x / y                   // Divide (integers round toward 0)
x % y                   // Modulo (result has sign of x)

x + y                   // Add, or \&x[y]
x - y                   // Subtract, or number of elements from *x to *y
x << y                  // x shifted y bits to left (x * pow(2, y))
```

```
x >> y                  // x shifted y bits to right (x / pow(2, y))

x < y                   // Less than
x <= y                  // Less than or equal to
x > y                   // Greater than
x >= y                  // Greater than or equal to

x & y                   // Bitwise and (3 & 6 is 2)
x ^ y                   // Bitwise exclusive or (3 ^ 6 is 5)
x | y                   // Bitwise or (3 | 6 is 7)
x && y                  // x and then y (evaluates y only if x (not 0))
x || y                  // x or else y (evaluates y only if x is false (0))
x = y                   // Assign y to x, returns new value of x
x += y                  // x = x + y, also -= *= /= <<= >>= &= |= ^=
x ? y : z               // y if x is true (nonzero), else z
throw x                 // Throw exception, aborts if not caught
x , y                   // evaluates x and y, returns y (seldom used)

// ## Classes

class T {               // A new type
private:                // Section accessible only to T's member functions
protected:              // Also accessible to classes derived from T
public:                 // Accessible to all
   int x;               // Member data
   void f();            // Member function
   void g() {return;}   // Inline member function
   void h() const;      // Does not modify any data members
   int operator+(int y);   // t+y means t.operator+(y)
   int operator-();     // -t means t.operator-()
   T(): x(1) {}         // Constructor with initialization list
   T(const T& t): x(t.x) {}// Copy constructor
   T& operator=(const T& t)
   {x=t.x; return *this; } // Assignment operator
   ~T();                // Destructor (automatic cleanup routine)
   explicit T(int a);   // Allow t=T(3) but not t=3
   T(float x): T((int)x) {}// Delegate constructor to T(int)
   operator int() const
   {return x;}          // Allows int(t)
   friend void i();     // Global function i() has private access
   friend class U;      // Members of class U have private access
   static int y;        // Data shared by all T objects
   static void l();     // Shared code.  May access y but not x
   class Z {};          // Nested class T::Z
   typedef int V;       // T::V means int
};
void T::f() {           // Code for member function f of class T
   this->x = x;}        // this is address of self (means x=x;)
int T::y = 2;           // Initialization of static member (required)
T::l();                 // Call to static member
T t;                    // Create object t implicit call constructor
t.f();                  // Call method f on object t

struct T {              // Equivalent to: class T { public:
   virtual void i();    // May be overridden at run time by derived class
   virtual void g()=0; };  // Must be overridden (pure virtual)
class U: public T {     // Derived class U inherits all members of base T
public:
   void g(int) override; };  // Override method g
class V: private T {};  // Inherited members of T become private
class W: public T, public U {};
// Multiple inheritance
class X: public virtual T {};
// Classes derived from X have base T directly

// All classes have a default copy constructor, assignment operator, and destructor, which perform the
// corresponding operations on each data member and each base class as shown above. There is also a default no-argument
// constructor (required to create arrays) if the class has no constructors. Constructors, assignment, and
// destructors do not inherit.

// ## Templates

template <class T> T f(T t);// Overload f for all types
template <class T> class X {// Class with type parameter T
   X(T t); };           // A constructor
```

```
template <class T> X<T>::X(T t) {}
// Definition of constructor
X<int> x(3);              // An object of type "X of int"
template <class T, class U=T, int n=0>
// Template with default parameters

// ## Namespaces

namespace N {class T {};}  // Hide name T
N::T t;                   // Use name T in namespace N
using namespace N;         // Make T visible without N::

// ## `memory` (dynamic memory management)

#include <memory>          // Include memory (std namespace)
shared_ptr<int> x;         // Empty shared_ptr to a integer on heap. Uses reference counting for cleaning up objects.
x = make_shared<int>(12);  // Allocate value 12 on heap
shared_ptr<int> y = x;     // Copy shared_ptr, implicit changes reference count to 2.
cout << *y;               // Dereference y to print '12'
if (y.get() == x.get()) {  // Raw pointers (here x == y)
  cout << "Same";
}
y.reset();                // Eliminate one owner of object
if (y.get() != x.get()) {
  cout << "Different";
}
if (y == nullptr) {        // Can compare against nullptr (here returns true)
  cout << "Empty";
}
y = make_shared<int>(15);  // Assign new value
cout << *y;               // Dereference x to print '15'
cout << *x;               // Dereference x to print '12'
weak_ptr<int> w;          // Create empty weak pointer
w = y;                    // w has weak reference to y.
if (shared_ptr<int> s = w.lock()) { // Has to be copied into a shared_ptr before usage
  cout << *s;
}
unique_ptr<int> z;         // Create empty unique pointers
unique_ptr<int> q;
z = make_unique<int>(16);  // Allocate int (16) on heap. Only one reference allowed.
q = move(z);               // Move reference from z to q.
if (z == nullptr){
  cout << "Z null";
}
cout << *q;
shared_ptr<B> r;
r = dynamic_pointer_cast<B>(t); // Converts t to a shared_ptr<B>


// ## `math.h`, `cmath` (floating point math)

#include <cmath>           // Include cmath (std namespace)
sin(x); cos(x); tan(x);    // Trig functions, x (double) is in radians
asin(x); acos(x); atan(x);  // Inverses
atan2(y, x);              // atan(y/x)
sinh(x); cosh(x); tanh(x);  // Hyperbolic sin, cos, tan functions
exp(x); log(x); log10(x);  // e to the x, log base e, log base 10
pow(x, y); sqrt(x);        // x to the y, square root
ceil(x); floor(x);         // Round up or down (as a double)
fabs(x); fmod(x, y);       // Absolute value, x mod y

// ## `assert.h`, `cassert` (Debugging Aid)

#include <cassert>        // Include iostream (std namespace)
       assert(e);          // If e is false, print message and abort
#define NDEBUG            // (before #include <assert.h>), turn off assert

// ## `iostream.h`, `iostream` (Replaces `stdio.h`)

#include <iostream>        // Include iostream (std namespace)
cin >> x >> y;            // Read words x and y (any type) from stdin
cout << "x=" << 3 << endl;  // Write line to stdout
cerr << x << y << flush;   // Write to stderr and flush
c = cin.get();            // c = getchar();
cin.get(c);               // Read char
cin.getline(s, n, '\n');   // Read line into char s[n] to '\n' (default)
```

```
if (cin)                 // Good state (not EOF)?
// To read/write any type T:
istream& operator>>(istream& i, T& x) {i >> ...; x=...; return i;}
ostream& operator<<(ostream& o, const T& x) {return o << ...;}

// ## `fstream.h`, `fstream` (File I/O works like `cin`, `cout` as above)

#include <fstream>          // Include filestream (std namespace)
ifstream f1("filename");    // Open text file for reading
if (f1)                  // Test if open and input available
f1 >> x;                    // Read object from file
f1.get(s);                  // Read char or line
f1.getline(s, n);           // Read line into string s[n]
ofstream f2("filename");    // Open file for writing
if (f2) f2 << x;            // Write to file

// ## `string` (Variable sized character array)

#include <string>         // Include string (std namespace)
string s1, s2="hello";    // Create strings
s1.size(), s2.size();     // Number of characters: 0, 5
s1 += s2 + ' ' + "world"; // Concatenation
s1 == "hello world"       // Comparison, also <, >, !=, etc.
s1[0];                  // 'h'
s1.substr(m, n);          // Substring of size n starting at s1[m]
s1.c_str();              // Convert to const char*
s1 = to_string(12.05);    // Converts number to string
getline(cin, s);          // Read line ending in '\n'

// ## `vector` (Variable sized array/stack with built in memory allocation)

#include <vector>         // Include vector (std namespace)
vector<int> a(10);        // a[0]..a[9] are int (default size is 0)
vector<int> b{1,2,3};        // Create vector with values 1,2,3
a.size();                // Number of elements (10)
a.push_back(3);           // Increase size to 11, a[10]=3
a.back()=4;              // a[10]=4;
a.pop_back();             // Decrease size by 1
a.front();               // a[0];
a[20]=1;                  // Crash: not bounds checked
a.at(20)=1;               // Like a[20] but throws out_of_range()
for (int& p : a)
p=0;                   // C++11: Set all elements of a to 0
for (vector<int>::iterator p=a.begin(); p!=a.end(); ++p)
*p=0;                  // C++03: Set all elements of a to 0
vector<int> b(a.begin(), a.end());  // b is copy of a
vector<T> c(n, x);        // c[0]..c[n-1] init to x
T d[10]; vector<T> e(d, d+10);      // e is initialized from d

// ## `deque` (Array stack queue)

// `deque<T>` is like `vector<T>`, but also supports:

#include <deque>          // Include deque (std namespace)
a.push_front(x);          // Puts x at a[0], shifts elements toward back
a.pop_front();            // Removes a[0], shifts toward front

// ## `utility` (pair)

#include <utility>        // Include utility (std namespace)
       pair<string, int> a("hello", 3);  // A 2-element struct
a.first;                 // "hello"
a.second;                   // 3

// ## `map` (associative array - usually implemented as binary search trees - avg. time complexity: O(log n))

#include <map>            // Include map (std namespace)
map<string, int> a;       // Map from string to int
a["hello"] = 3;           // Add or replace element a["hello"]
for (auto& p:a)
   cout << p.first << p.second;  // Prints hello, 3
a.size();                // 1

// ## `unordered_map` (associative array - usually implemented as hash table - avg. time complexity: O(1))

#include <unordered_map>  // Include map (std namespace)
```

```
unordered_map<string, int> a; // Map from string to int
a["hello"] = 3;          // Add or replace element a["hello"]
for (auto& p:a)
cout << p.first << p.second;  // Prints hello, 3
a.size();               // 1

// ## `set` (store unique elements - usually implemented as binary search trees - avg. time complexity: O(log n))

#include <set>           // Include set (std namespace)
set<int> s;             // Set of integers
s.insert(123);          // Add element to set
if (s.find(123) != s.end()) // Search for an element
s.erase(123);
cout << s.size();       // Number of elements in set

// ## `unordered_set` (store unique elements - usually implemented as a hash set - avg. time complexity: O(1))

#include <unordered_set>  // Include set (std namespace)
unordered_set<int> s;     // Set of integers
s.insert(123);          // Add element to set
if (s.find(123) != s.end()) // Search for an element
s.erase(123);
cout << s.size();       // Number of elements in set

// ## `algorithm` (A collection of 60 algorithms on sequences with iterators)

#include <algorithm>      // Include algorithm (std namespace)
min(x, y); max(x, y);     // Smaller/larger of x, y (any type defining <)
swap(x, y);             // Exchange values of variables x and y
sort(a, a+n);           // Sort array a[0]..a[n-1] by <
sort(a.begin(), a.end()); // Sort vector or deque
reverse(a.begin(), a.end()); // Reverse vector or deque

// ## `chrono` (Time related library)

#include <chrono>         // Include chrono
using namespace std::chrono; // Use namespace
auto from =             // Get current time_point
      high_resolution_clock::now();
// ... do some work
auto to =               // Get current time_point
      high_resolution_clock::now();
using ms =              // Define ms as floating point duration
duration<float, milliseconds::period>;
// Compute duration in milliseconds
cout << duration_cast<ms>(to - from)
.count() << "ms";

// ## `thread` (Multi-threading library)

#include <thread>        // Include thread
unsigned c =
      hardware_concurrency(); // Hardware threads (or 0 for unknown)
auto lambdaFn = [](){     // Lambda function used for thread body
   cout << "Hello multithreading";
};
thread t(lambdaFn);       // Create and run thread with lambda
t.join();               // Wait for t finishes

// --- shared resource example ---
mutex mut;                    // Mutex for synchronization
condition_variable cond;          // Shared condition variable
const char* sharedMes             // Shared resource
      = nullptr;
auto pingPongFn =               // thread body (lambda). Print someone else's message
      [&](const char* mes){
         while (true){
            unique_lock<mutex> lock(mut);// locks the mutex
            do {
               cond.wait(lock, [&](){     // wait for condition to be true (unlocks while waiting which allows other threads
to modify)
                  return sharedMes != mes; // statement for when to continue
               });
            } while (sharedMes == mes);  // prevents spurious wakeup
            cout << sharedMes << endl;
            sharedMes = mes;
```

```
            lock.unlock();              // no need to have lock on notify
            cond.notify_all();          // notify all condition has changed
        }
    };
sharedMes = "ping";
thread t1(pingPongFn, sharedMes);  // start example with 3 concurrent threads
thread t2(pingPongFn, "pong");
thread t3(pingPongFn, "boing");

// ## `future` (thread support library)

#include <future>         // Include future
    function<int(int)> fib =  // Create lambda function
[&](int i){
  if (i <= 1){
    return 1;
  }
  return fib(i-1)
       + fib(i-2);
};
future<int> fut =         // result of async function
    async(launch::async, fib, 4); // start async function in other thread
// do some other work
cout << fut.get();        // get result of async function. Wait if needed.
```