1. Treat-all sequences, treat-all elements

Example: Compute the maximum of each sequence, and the sum of all maximums.

Each sequence ends in zero and has at least one element.

```
12  10  8  7  5  0
1   22  0
4   0
3   -4  1  0
```

```
int main() {
    int sum = 0;
    int x;
    while (cin >> x) {
       int m = x;
       while (x != 0) {
         if (x > m) m = x;
         cin >> x
       }
       cout << m << endl;
       sum = sum + m;
    }
    cout << sum << endl;
}
```

2. Search sequence, treat-all elements

Example: Check if any of the sequences sums over 50.

Each sequence ends in zero.

```
12  10  -7  5  0
1   22  0
4   0
3   -4  1  0
```

```
int main() {
   bool found = false;
   int x;
   while (cin >> x and not found) {
      int s = 0;
      while (x != 0) {
         s = s + x;
         cin >> x;
      }
      found = (s > 50);
   }
   if (found) cout << “yes” << endl;
   else cout << “no” << endl;
}
```

3. Search sequence, search element

Example: Locate the first sequence that contains a number ending in 3.

Each sequence ends in zero.

```
12  10  7  5  0
1   22  0
4   0
3   4  1  0
```

```
int main() {
   bool found = false;
   int x;
   int p = 0;
   while (cin >> x and not found) {
      bool end3 = false;
      while (x != 0 and not end3) {
         end3 = (x%10 == 3);
         cin >> x;
      }
      found = end3;
      p = p + 1;
   }
   if (found) cout << p << endl;
   else cout << "none" << endl;
}
```

4. Treat-all sequences, search element

Example: Count how many sequences contain a multiple of 10.

Each sequence ends in zero.

```
12  10  -7  5  0
1   22  0
4   0
3   -4  1  0
```

```
int main() {
   int n = 0;
   int x;
   while (cin >> x) {
      bool mult = false;
      while (x != 0) {
         if (x%10 == 0)
            mult = true;
         cin >> x;
      }
      if (mult) n = n + 1;
   }
   cout << n << endl;
}
```

Acciones: cómo llamarlas

Observación 1

Una acción no retorna explícitamente ningún valor, por tanto, sólo se ejecutará, no hay valor a utilizar directamente como las funciones.

```
int main() {
    int a = 5;
    int b = 10;
    int z = intercambio(a, b);          // INCORRECTO!
    cout << intercambio(a, b) << endl;  // INCORRECTO!
}
```

Observación 2

Para los parámetros de salida y entrada/salida, la llamada sólo puede ser hecha con variables.

```
int main() {
    int a = 5;
    int b = 10;
    intercambio(a+1, b); // INCORRECTO! a+1 es una expresión
    intercambio(5, b);   // INCORRECTO! 5 es una expresión
}
```

Paso de parámetros

Los parámetros de una función/acción se pueden pasar de dos formas:

Paso por valor

- Es el tipo de paso cuando el parámetro es de entrada.
- Recuerda que en el momento de la llamada, se le da valor a través de una expresión (que de forma particular, puede ser el valor de una variable).
- La sintaxis es:

```
tipo_de_datos nombre_var
```

Paso por referencia

- Es el tipo de paso cuando el parámetro es de salida o entrada/salida.
- Recuerda que en el momento de la llamada, se establece un alias entre la variable con la que se llama y el nombre del parámetro en la definición.
- La sintaxis es:

```
tipo_de_datos& nombre_var
```

Subprograms: parameter passing

- Use *call-by-value* to pass parameters that must not be modified by the subprogram.

- Use *call-by-reference* when the changes made by the subprogram must affect the variable to which the parameter is bound.

- In some cases, call-by-reference is used to avoid copies of large objects, even though the parameter is not modified.

Observación 1

El & importa!! Y el nombre de los parámetros NO!!

```
    // x, y: parámetros de entrada!
    void intercambio(int x, int y) {
        int aux = x;
        x = y;
        y = aux;
    }

→   int main() {
        int a = 5;
        int b = 10;
        intercambio(a, b);
        cout << a << " " << b << endl;
    }
```

```
    // x, y: parámetros de entrada!
    void intercambio(int x, int y) {
        int aux = x;
        x = y;
        y = aux;
    }

→   int main() {
        int x = 5;
        int y = 10;
        intercambio(x, y);
        cout << x << " " << y << endl;
    }
```

Observación 2

Si es un parámetro de entrada/salida:

- En la definición de la función/acción:
  - La precondición nos dirá qué valores son válidos.
  - Dentro del código se usa (consulta) su valor antes de asignarle uno nuevo.
- En la llamada a la función/acción:
  - La variable con la que se hace la llamada ha de tener un valor válido.

```
// Pre: x entero (vale A), y entero (vale B)
// Post: x vale B, y vale A
void intercambio(int &x, int &y) {
    int aux = x;
    x = y;
    y = aux;
}

int main() {
    int x = 5;
    int y = 10;
    intercambio(x, y);
    cout << x << " " << y << endl;  // Escribirá 10 5
}
```

Observación 3

Si es un parámetro SÓLO de salida:

- En la definición de la función/acción:
  - SÓLO aparece en la postcondición.
  - Dentro del código se le asigna un valor antes de consultarlo.
- En la llamada a la función/acción:
  - La variable con la que se hace la llamada NO tendrá un valor válido.

```
// Pre: n >= 0
// Post: descomposición horaria en horas (h), minutos (m),
//           segundos (s) de n
void descompon(int n, int& h, int& m, int& s) {
    // Como h, m, s son sólo de salida, en este punto NO
    // tendrán valor válido. La acción se ocupa de dárselo
    h = n/3600;
    m = (n%3600)/60;
    s = n%60;
}
```


Observación 3

Si es un parámetro SÓLO de salida:

- En la definición de la función/acción:
  - SÓLO aparece en la postcondición.
  - Dentro del código se le asigna un valor antes de consultarlo.
- En la llamada a la función/acción:
  - La variable con la que se hace la llamada NO tendrá un valor válido.


```
// Pre: Dado una secuencia de naturales
// Post: Escribir su descomposición horaria
int main() {
    int n;
    while (cin >> n) {
        int hora, min, sec;    // No tienen valor válido
        descompon(n, hora, min, sec);
        cout << hora << " " << min << " " << sec << endl;
    }
}
```

GUIA DE PROGRAMACION RECURSIVA

Proponemos una estrategia formada por varias etapas. Se establecen reglas generales (que, por ser generales, están sujetas a excepciones)

Desarrollamos las etapas en paralelo con un ejemplo de uso:

Ejemplo >> Se pide una función recursiva para juntar dos números en uno.

joint(351, 23) ---> 35123

joint(34, 987) ---> 34987

```
//pre: m >= 0 y n >= 0
//post: retorna el numero cuyas primeras cifras son las de m y las ultimas las de n
int joint(int m, int n)
```

Etapas:

Etapa 0: (peliminar) El esqueleto de una funcion (o accion) recusiva es una estructura alternativa:

```
if (...) .......
else if (...) ...
.................
else if (...)
ese ...
```

los casos sin llamada recursiva son los "casos directos". El resto de casos son los "casos recursivos"

Etapa 1: Los casos directos de obtienen respondiendo a "para que valores de los parametros es facil calcular el resultado?".

joint(m,n) es facil de calcular cuando n solo tiene un digito (n < 10). En este caso el resultado es

10*m + n

Por ejemplo, joint(89, 7) es 10*89 + 7 (= 897)

```
if (n < 10) ....//caso directo
else      .....//caso recursivo
```

Etapa 3: resolvemos el caso recursivo en tres pasos

3.a) Proponemos una llamada recursiva. El argumento con el que se hace la llamada

ha de acercarse a uno de los casos directos*

En nuestro ejemplo, una propuesta de llamada recursiva puede ser **joint(m, n/10).**

La propuesta de llamada joint(m, 10*n) no tiene sentido porque no se acerca al caso base

3.b) Punto **magico**: suponemos que la llamada recursiva que proponemos funciona correctamente. Documentamos el resultado que obtenemos de la llamada en un comentario. (La **magia** equivale a la frase "por hipotesis de induccion" en una demostracion por induccion)

En nuestro ejemplo:

```
if (n < 10) return 10*m + n;
else {
   int z = joint(m, n/10)
   //z es el numero cuyas primeras cifras son las de m y sus ultimas son las de n/10
   ....
```

3.c) Completamos el caso recursivo con el codigo necesario para que se cumpla la postconcicion. Si no se consigue, volvemos al paso 3.a)

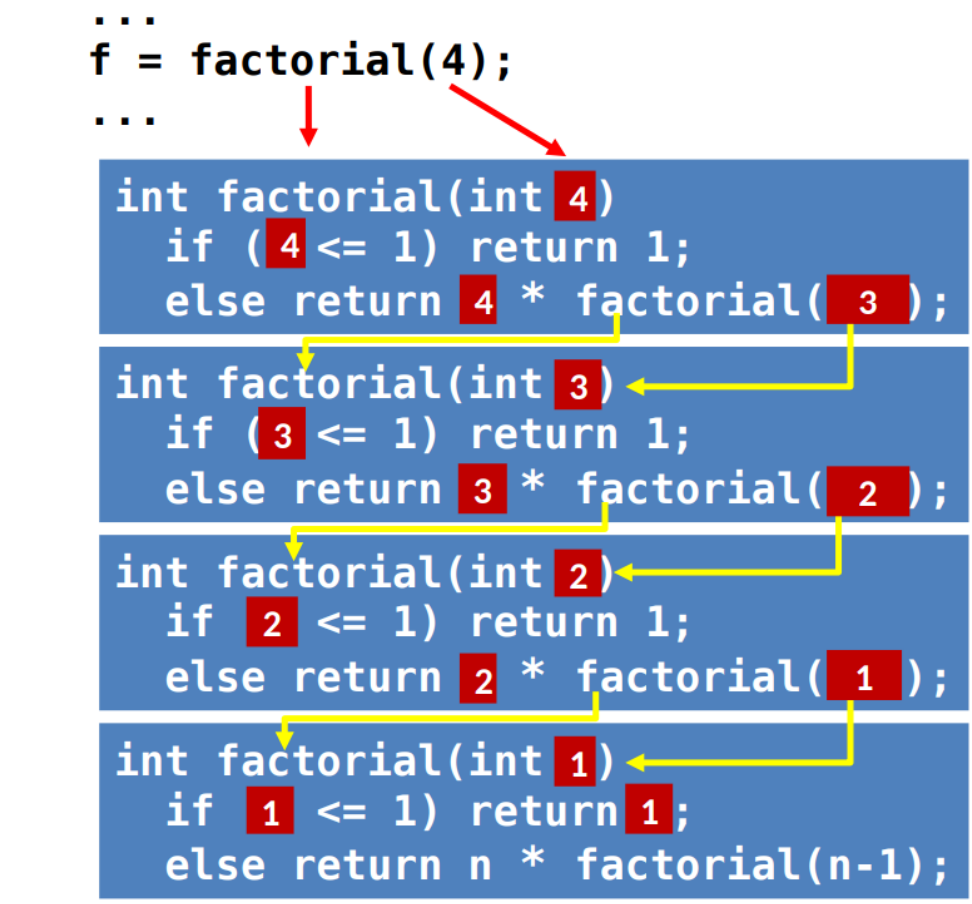
En nuestro ejemplo, necesitamos completar el caso recursivo

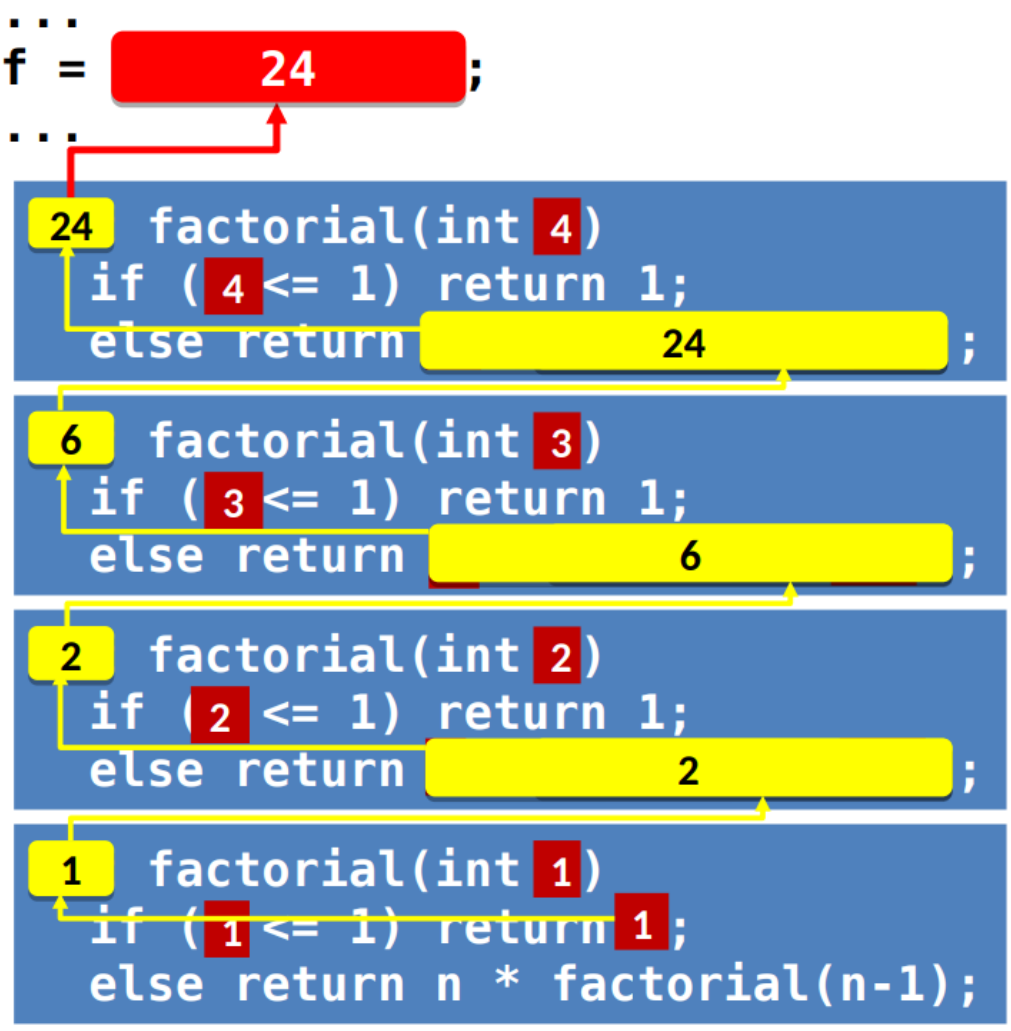
```
//z es el numero cuyas primeras cifras son las de m y sus ultimas son las de n/10
....
....
  //post: retorna el numero cuyas primeras cifras son las de m y las ultimas las de n
return 10*z + n%10
```

Codigo completo:

```
int joint(int m, int n) {
   if (n < 10) return 10*m + n;
   else {
      int z = joint(m, n/10);
      //z es el numero cuyas primeras cifras son las de m y sus ultimas son las de n/10
      return 10*z + n%10;
   }
```

Recursion: behind the scenes





Vectors

- A vector is a data structure that groups values of the same type under the same name.
- Declaration: vector<type> name(n);

- A vector contains *n* elements of the same type (*n* can be any expression).
- name[i] refers to the *i*-th element of the vector (*i* can also be any expression)
- Note: use **#include<vector>** in the program

Declaración de un vector

Sintaxis:

```
#include <vector>          // Biblioteca necesaria

vector<T> nombre_var(S, I);
```

- T: tipos de datos
- S: tamaño del vector (por defecto es 0)
- I: valor inicial de los elementos del vector (si no se indica será inválido)

Los parámetros S, I son opcionales:

- Si sólo indicas uno, será S.
- Si indicas dos, serán S, I.

Ejemplos:

```
int main() {
    vector <int> v;          // vector de enteros llamado v que tiene 0 elementos
    vector <int> w(2);       // vector de enteros de 2 elementos, cada elemento tiene valor
                             // inválido
    vector <int> z(2, 4);    // vector de enteros de 2 elementos, cada elemento con valor 4
}
```

Acceso: siempre por índice

```
nombre_var[E]
```

- E: expresión entera
- Se accede a la posición E del vector nombre_var (sirve tanto para consultar como para actualizar)

```
int main() {
    vector <int> v(3, 1);
    int x = v[0];       // consulto valor posición 0 de v
    v[0] = 10;          // actualizo su valor
    cout << x << " " << v[0] << endl;
}
```

Alerta!

Al ejecutar el programa, si se accede a una posición no válida de un vector, el programa se para y te da el error:

```
container with out-of-bounds index
```

Es un error de ejecución.

Asignación (copia) de un vector a otro:

```
int main() {
    // Opción 1: copia de un vector que se ha modificado a lo
    //           largo del código
    vector<int> v(3, 1);
    (...)
    vector <int> w = v;

    // Opción 2: copia de un vector que se crea sin nombre
    vector<int> z;
    z = vector<int> (20, 5);

    // Opción 3: mala idea
    vector <int> vect(20, 5);
    vector <int> copia(10, 8);
    copia = vect;
}
```

La copia de vectores es una operación costosa.

Saber número de elementos:

```
int n = nombre_var.size();
```

- La función .size() retorna el número de elementos del vector sobre el que se llame.

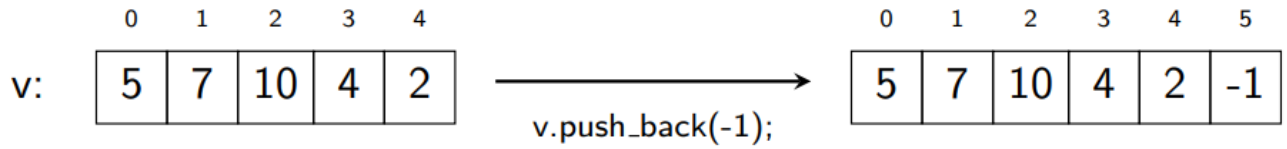
Lectura/escritura de vectores:

```
vector <T> v;
cin >> v;     // ERROR de compilación: cin sólo sobre
              //       tipos de datos simples
cout << v;    // ERROR de compilación: cout sólo sobre
              //       tipos de datos simples
```

Añadir un nuevo elemento:

```
nombre_vector.push_back(valor_T);
```

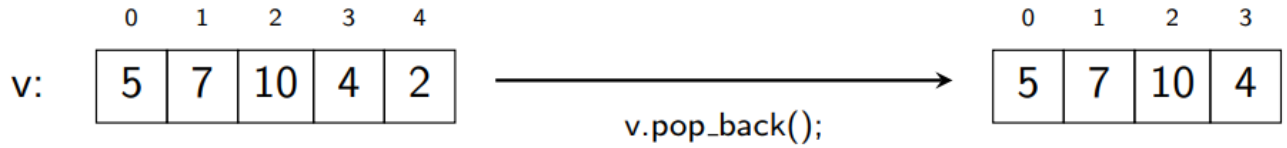
- Al vector *nombre_vector* se le añade un nuevo último elemento con valor *valor_T* y tipo de datos T (el mismo que el del vector).



Borrar último elemento del vector:

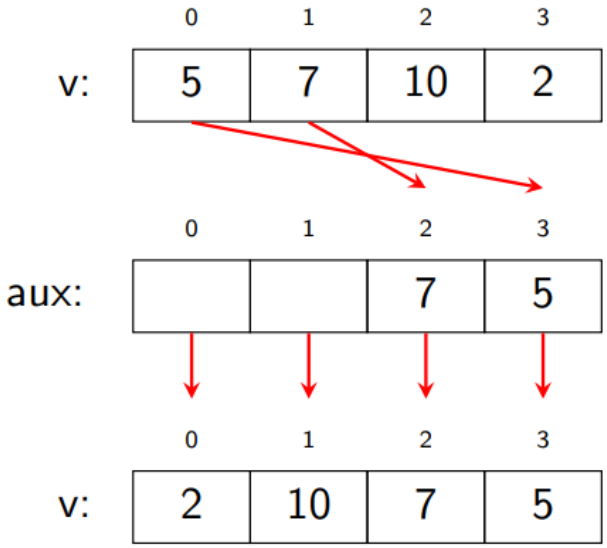
```
nombre_vector.pop_back();
```

- Al vector *nombre_vector* se le quita su último elemento.



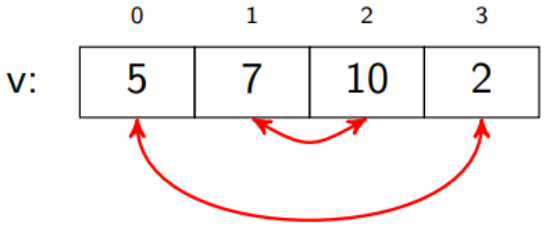
Paso de parámetros: paso por referencia

Algoritmo 1:




```
void invertir(vector<int>& v) {
    int n = v.size();
    vector<int> aux(n);
    for (int i = 0; i < n; ++i) aux[n - 1 - i] = v[i];
    for (int i = 0; i < n; ++i) v[i] = aux[i];
}
```

Algoritmo 2:




```
void invertir(vector<int>& v) {
    int n = v.size();
    for (int i = 0; i < n/2; ++i) {
        int aux = v[i];
        v[i] = v[n - 1 - i];
        v[n - 1 - i] = aux;
    }
}
```

Paso de parámetros: paso por valor

Implementación:

```
// Pre: v es un vector de enteros válido
// Post: retorna la media de sus elementos
double average(const vector<int>& v) {
    int suma = 0;
    int n = v.size();
    for (int i = 0; i < n; ++i) suma = suma + v[i];
    return suma/double(n);      // Recuerda que / es división
                                // real o entera dependiendo
                                // de los operandos
}
```

- Esquemas
  - Recorrido
    - de izquierda a derecha
    - de derecha a izquierda
  - Búsqueda
    - vector no ordenado ⇒ lineal
      - izq-dcha
      - dcha-izq
    - vector ordenado ⇒ dicotómica

Esquemas algorítmicos: Recorrido

Implementación:

```
// Pre: v es un vector no vacío de enteros válido
// Post: retorna su valor mínimo
int minimo_vector(const vector<int>& v) {
    int min = v[0];
    for (int i = 1; i < v.size(); ++i) {
        if (min > v[i]) min = v[i];
    }
    return min;
}
```

Esquemas algorítmicos: Búsqueda lineal

El vector no está ordenado $\Longrightarrow$ búsqueda lineal:

- Empiezo desde un extremo del vector (primer o último elemento).
- Avanzo al siguiente elemento en orden según índice (hacia izquierda o hacia derecha).
- Hasta que encuentro lo que busco o llego al otro extremo del vector.

```
// Pre: ——
// Post: retorna la primera posición de x en v si existe, -1 en caso contrario
int primera_pos_valor(const vector<int>& v, int x) {
    for (int i = 0; i < v.size(); ++i) {
        if (v[i] == x) return i;
    }
    return -1;
}
```

```
// Pre: ——
// Post: retorna la última posición de x en v si existe, -1 en caso contrario
int ultima_pos_valor(const vector<int>& v, int x) {
    int n = v.size();
    for (int i = n - 1; i >= 0; --i) {
        if (v[i] == x) return i;
    }
    return -1;
}
```

Esquemas algorítmicos: Búsqueda dicotómica

El vector está ordenado $\Longrightarrow$ búsqueda dicotómica:

- Miro el valor que hay a la mitad del vector
- Si ese valor es mayor que el buscado, los elementos a su derecha todavía lo son más, así es que el que busco no estará en esas posiciones.
- Si ese valor es menor que el buscado, los elementos a su izquierda todavía lo son más, así es que el que busco no estará en esas posiciones.
- Repetir este razonamiento hasta que encuentro lo que busco o descarto todos los elementos.

Esquemas algorítmicos: Búsqueda dicotómica

```
int busqueda_binaria(const vector<int>& v, int x) {                    // ITERATIVA
    int izq = 0;
    int dcha = v.size() - 1;
    while (izq <= dcha) {
        int m = (dcha + izq)/2;
        if (v[m] < x) izq = m + 1;
        else if (x < v[m]) dcha = m - 1;
        else return m;
    }
    return -1;
}
```

```
int busqueda_binaria(const vector<int>& v, int x, int izq, int dcha) {    // RECURSIVA
    if (izq > dcha) return -1;
    int m = (dcha + izq)/2;
    if (v[m] < x) return busqueda_binaria(v, x, m + 1, dcha);
    else if (x < v[m]) return busqueda_binaria(v, x, izq, m - 1);
    else return m;
}

int main() {
    int n, x;
    cin >> x >> n;
    vector<int> v(n);
    ...
    int pos = busqueda_binaria(v, x, 0, v.size() - 1);
    ...
}
```

String

El tipo de datos string (que hasta ahora lo hemos tratado como un tipo de datos simple), es en realidad un vector de char $\Longrightarrow$ Todo lo explicado para vectores sirve para los strings.

Por ejemplo, el string:

"clase"

lo podemos tratar como el vector de chars:

0	1	2	3	4
'c'	'l'	'a'	's'	'e'

Eso quiere decir que podemos:

- saber su número de carácteres
- acceder a una posición para consultar o modificar su valor

Cuidado con la declaración:

```
string s(10);          // ERROR de compilación      /* Recordad que:
string s(10, 'a');     // ok                           #include <string>
string s = "un string cualquiera";  // ok            */
```

String: Ejemplo 1

Implementación:

```
// Pre: —
// Post: retorna true si c es una vocal
bool es_vocal(char c) {
    if (c == 'a' or c == 'e' or c == 'i' or c == 'o' or c == 'u')
        return true;
    if (c == 'A' or c == 'E' or c == 'I' or c == 'O' or c == 'U')
        return true;
    return false;
}

// Pre: —
// Post: retorna el número de vocales del string s
int num_vocales(const string& s) {
    int n_voc = 0;
    for (int i = 0; i < s.size(); ++i) {
        if (es_vocal(s[i])) ++n_voc;
    }
    return n_voc;
}
```

String: Ejemplo 2

Implementación:

```
// Pre: s es una palabra
// Post: retorna true si s es palíndroma,
//       false en caso contrario
bool palindromo(const string& s) {
    int n = s.size();
    for (int i = 0; i < n/2; ++i) {
        if (s[i] != s[n - 1 - i]) return false;
    }
    return true;
}
```

String: Ejemplo 3

Implementación:

```
// Pre: 0 <= i < s.size()
// Post: retorna true si subs es un substring de s desde s[i]
bool is_substring(const string& subs, const string& s, int i) {
    int j = 0;
    while (i < s.size() and j < subs.size() and s[i] == subs[j]) {
        ++i;
        ++j;
    }
    return j == subs.size();
}

// Pre: —
// Post: retorna la posición de s a partir de la cual subs es un substring,
//       -1 en caso de que subs no sea substring de s.
int substring(const string& subs, const string& s) {
    for (int i = 0; i < s.size(); ++i) {
        if (is_substring(subs, s, i) return i;
    }
    return -1;
}
```

Fíjate que la condición del for de substring podría ser:

$$i < s.size() - subs.size()$$

Typedef

Sintaxis:

```
typedef tipo_de_datos_existente nuevo_nombre;
```

Semántica:

- *nuevo_nombre* es sinónimo de *tipo_de_datos_existente*.

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

typedef vector<int> Polinomio;

int evaluar_polinomio(const Polinomio& p, int x) {
    ...
}

int main() {
    int x, n;
    cin >> x >> n;
    Polinomio p(n);
    for (int i = 0; i < n; ++i) cin >> p[i];
    cout << evaluar_polinomio(p, x) << endl;
}
```

Ordenación (biblioteca algorithm)

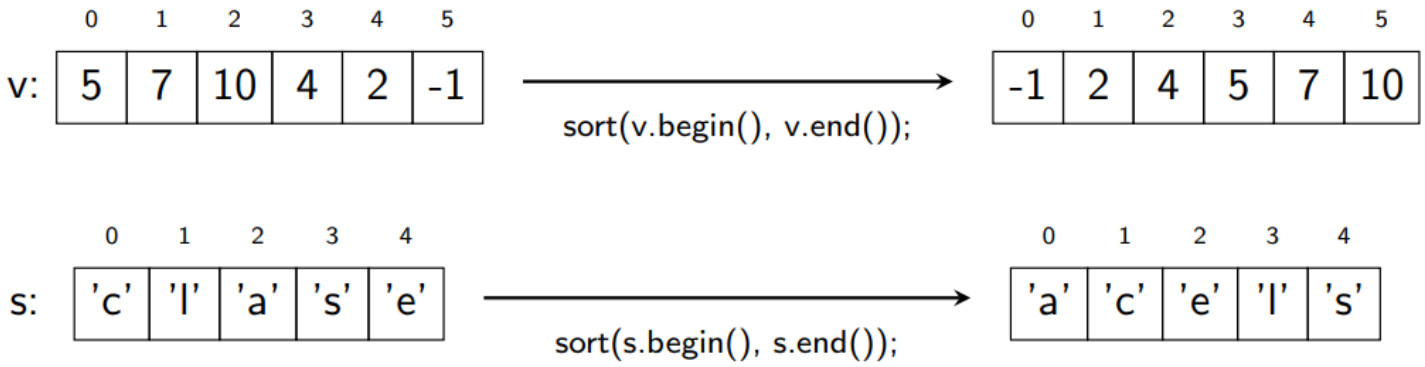
Sintaxis:

```
#include <algorithm>          // Biblioteca necesaria

sort(nombre_vector.begin(), nombre_vector.end());
```

Semántica: Se ordena todo el vector *nombre_vector*, en orden ascendente según el operador $<$ definido para el tipo de datos de ese vector.

Ejemplo:



Ordenación (biblioteca algorithm)

Sintaxis:

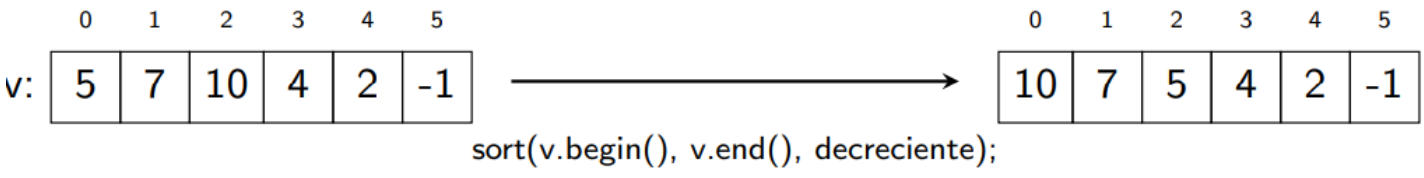
```
// Post: retorna true si a tiene que ir antes que b en el vector,
//       false en caso contrario
bool nom_func_bool(const T& a, const T& b) {...}

sort(nom_v.begin(), nom_v.end(), nom_func_bool);
```

Semántica:
Se ordena todo el vector *nombre_v*, en orden ascendente según la función booleana *nom_func_bool*. T es el tipo de datos del vector. El orden es estricto (es decir, *nom_func_bool(a, a)*$\rightarrow$ false).

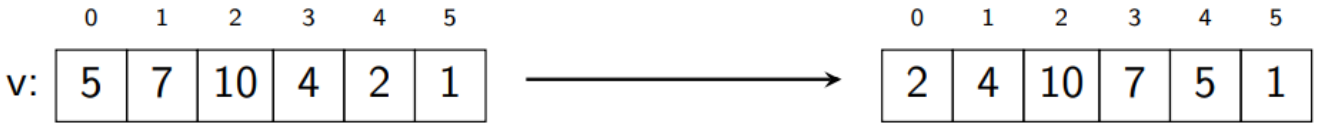
Ejemplo:

```
// quiero que a vaya antes que b en el vector cuando a > b
bool decreciente(int a, int b) {
    if (a > b) return true;
    else return false;
}
```



Escribir un procedimiento tal que dado un vector de enteros, lo transforme de tal manera que sus elementos pares aparezcan antes que los elementos impares. Además, los elementos pares estarán ordenados de forma ascendente, mientras que los impares de forma descendente.

Ejemplo:




```
bool orden(int a, int b) {
    bool par_a = a%2 == 0;
    bool par_b = b%2 == 0;

    if (par_a != par_b) return par_a;      // tienen diferente paridad
    if (par_a) return a < b;               // los dos son pares
    return a > b;                          // los dos son impares
}

void separar(vector <int>& v) {
    sort(v.begin(), v.end(), orden);
}
```