

Asignaciones y expresiones: programación en c++

Tema 6

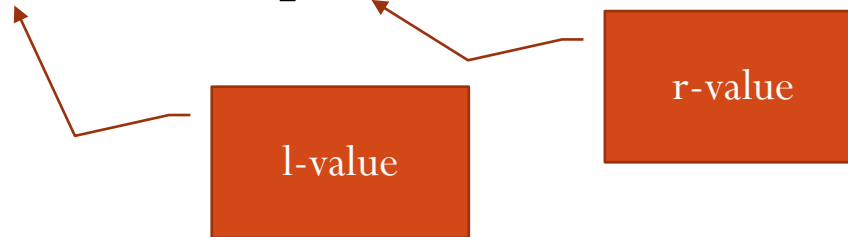


José Ramón Paramá Gabía

El operador asignación

- En C++ se usa el signo igual (=):

`variable=expresión;`



- En C++ es posible también:

`variable1=variable2=...=variableN=expresión;`

Ej.

`i=j=k=3;`

El operador asignación

- Cuando se mezclan datos de diferente tipo en una sentencia de asignación, se producen conversiones de tipo, ya que el valor del lado derecho de la asignación es convertido al tipo del lado izquierdo.
- Esto puede producir pérdida de información. Ej, asignar un *float* a un *int*, asignar un *double* a un *float*, etc.

Operadores aritméticos

Operador	Significado
+	Suma
-	Resta y menos unario
*	Multiplicación
/	División
%	Módulo
++	Incremento
--	Decremento

Mayor precedencia	++, --
	- (menos unario)
	*, /, %
Menor precedencia	+, - (resta)

(C. Martín, A. Urquía, M. A. Rubio. *Lenguajes de programación*, Ed. Uned, 2011)

- Cuando se aplica la división (/) a dos operando enteros el resultado es un cociente entero. Ej: $5/2=2$
- El operador módulo (%) es el resto de la división. Ej: $5\%2=1$

Operadores relacionales y lógicos

Tabla 6.3: Operadores relacionales de C++.

Operador	Significado
==	Igualdad
!=	Desigualdad
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que

Tabla 6.4: Operadores lógicos de C++.

Operador	Significado
!	NOT
&&	AND
	OR

Tabla 6.5: Precedencia de los operadores relacionales y lógicos en C++.

Mayor precedencia	!
	<, <=, >, >=
	==, !=
	&&
Menor precedencia	

Operadores << >>

- Estos operadores están sobrecargados.
- Una de sus funciones es la *desplazamiento a la derecha* (>>) y a la izquierda (<<) de palabras de bits.
- Se aplican tipos básicos de enteros o tipos que tienen equivalencia con los números enteros (como *char* o tipos enumerados).
- La sintaxis es:

`valor >> n°_bits`

`110>>1=11`

`6>>1=3`

`valor<<n°_bits`

`110<<2=11000`

`6<<2=24`

Operadores << >>

- En la cabecera `iomanip` se han sobrecargado estos operadores de manera que permiten realizar operaciones sobre flujos:
 - `>>` *obtener de* el operador admite un operando a izquierdo tipo `std::istream` y un operando derecho de cualquiera de los tipos básicos.
 - `<<` *poner en* el operador admite un operando izquierdo tipo `std::ostream` y un operando derecho cualquiera de los tipos básicos.
- En otras cabeceras estándar, se han sobrecargado estos operadores para que admitan otros tipos (además de los básicos) por ej. la cabecera `string`, que permite el tipo de datos `std::string`

Operando con valores numéricos

- Además de los operadores aritméticos:

Operador	Significado
+	Suma
-	Resta y menos unario
*	Multiplicación
/	División
%	Módulo
++	Incremento
--	Decremento

(C. Martín, A. Urquía, M. A. Rubio. *Lenguajes de programación*, Ed. Uned, 2011)

- La librería estándar de C++ contiene la definición de funciones matemáticas útiles para operar con datos numéricos.
- Para poder usar todas estas funciones, es necesario incluir la cabecera `cmath`.
- Las funciones matemáticas están sobrecargadas para argumentos del tipo *float*, *double* y *long double*

Operando con valores numéricos

- Algunas de las más frecuentes. la unidad de los ángulos es radian.

Tabla 6.6: Algunas funciones declaradas en la cabecera estándar *cmath*.

Función	Significado
<code>acos(x)</code>	Arco coseno de x
<code>asin(x)</code>	Arco seno de x
<code>atan(x)</code>	Arco tangente de x
<code>atan2(y,x)</code>	Arco tangente de y/x.
<code>ceil(x)</code>	Menor entero no menor que x.
<code>cos(x)</code>	Coseno de x
<code>cosh(x)</code>	Coseno hiperbólico de x
<code>exp(x)</code>	e elevado a x (e^x)
<code>fabs(x)</code>	Valor absoluto de x
<code>floor(x)</code>	Mayor entero no mayor que x
<code>fmod(x,y)</code>	Resto de x/y
<code>log(x)</code>	Logaritmo natural (base e) de x
<code>log10(x)</code>	Logaritmo en base 10 de x
<code>pow(x,y)</code>	x elevado a y (x^y)
<code>sin(x)</code>	Seno de x
<code>sinh(x)</code>	Seno hiperbólico de x
<code>sqrt(x)</code>	Raíz cuadrada de x
<code>tan(x)</code>	Tangente de x
<code>tanh(x)</code>	Tangente hiperbólica de x

Entrada por teclado

```
// Fichero: entradaDesdeTeclado.cpp
#include <iostream>
#include <string>

int main()
{
    // Pregunta el nombre
    std::cout << "Por favor, escribe tu nombre: ";

    // Lee el nombre que el usuario escribe en el teclado
    std::string nombre;
    std::cin >> nombre;

    // Escribe un saludo
    std::cout << "Hola " << nombre << "!" << std::endl;
    return 0;
}
```

parama@ubuntu: /sitio/h

Archivo Editar Ver Terminal Solapas Ayuda

parama@ubuntu:/sitio/home/docencia/uned/LP\$./teclado

Por favor, escribe tu nombre: █

parama@ubuntu: /sitio/h

Archivo Editar Ver Terminal Solapas Ayuda

parama@ubuntu:/sitio/home/docencia/uned/LP\$./teclado

Por favor, escribe tu nombre: jose █

parama@ubuntu: /sitio,

Archivo Editar Ver Terminal Solapas Ayuda

parama@ubuntu:/sitio/home/docencia/uned/LP\$./teclado

Por favor, escribe tu nombre: jose

Hola jose!

Entrada por teclado

- Si el flujo de entrada está vacío el programa queda a la espera de que se introduzca algo.
- Los caracteres que se tecleen se introducen en el flujo `std::cin` hasta que el usuario pulse *return*.
- Entonces comienza el volcado en la variable:
 - Se desechan los caracteres en blanco, saltos de línea y tabuladores hasta que se encuentra un carácter.
 - Se vuelcan los caracteres hasta que se encuentra un espacio en blanco, salto de línea o tabulador.
 - Al realizar el volcado se comprueba que la variable destino del volcado sea compatible con cadenas de caracteres.

Entrada por teclado

- Los caracteres del flujo no volcados continúan en el flujo dispuestos a ser volcados en otra variable.

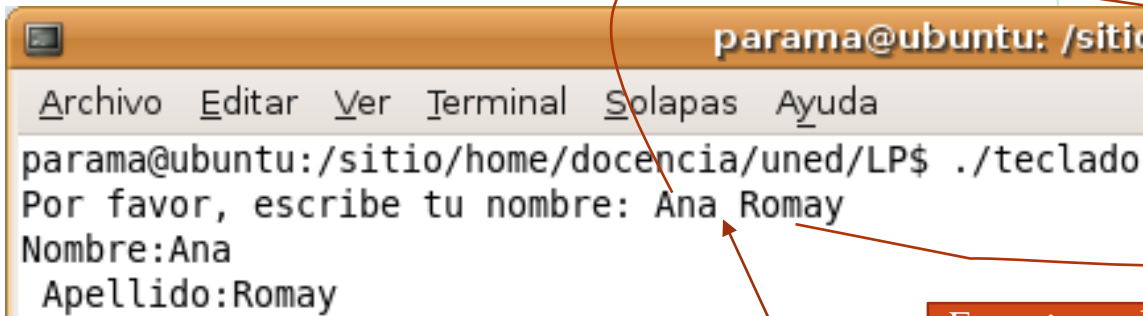
```
int main()
{
    // Pregunta el nombre
    std::cout << "Por favor, escribe tu nombre: ";

    // Lee el nombre que el usuario escribe en el teclado
    std::string nombre;
    std::cin >> nombre;

    std::string apellido;
    std::cin >> apellido;

    // Escribe un saludo
    std::cout << "Nombre:" << nombre << "\n Apellido:" << apellido << std::endl;
    return 0;
}
```

Cuando llega aquí el flujo no está vacío (no espera) y vuelca su contenido en *apellido*



```
parama@ubuntu: /sitio
Archivo Editar Ver Terminal Solapas Ayuda
parama@ubuntu:/sitio/home/docencia/uned/LP$ ./teclado
Por favor, escribe tu nombre: Ana Romay
Nombre:Ana
Apellido:Romay
```

Espacio en blanco corta volcado pero *Romay* queda en el flujo de entrada

Operando con strings

Tabla 6.7: Operadores sobre *strings*.

Operador	Significado
=	Asignación
+	Concatenación
+=	Concatenación y asignación
==	Igualdad
!=	Desigualdad
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
[]	Acceso a subcadenas
<<	Salida
>>	Entrada

(C. Martín, A. Urquía, M. A. Rubio. *Lenguajes de programación*, Ed. Uned, 2011)

Operando con strings

- Concatenación
 - El operador + se ha sobrecargado para concatenar operandos tipo string, donde al menos uno debe ser una variable.

```
// Fichero: holaMundoVars2.cpp
#include <iostream>
#include <string>

int main()
{
    std::string palabra = "mundo";
    std::string exclamacion = "!";
    std::string frase = "Hola " + palabra + exclamacion;
    std::cout << frase << std::endl;
    return 0;
}
```

Asociativo a la izquierda igual que poner("Hola+palabra)+exclamacion

Operando con strings

- El operador $+$ no se puede utilizar para concatenar dos literales *string*.
- Para concatenarlos basta con escribir uno a continuación del otro.
- Por ejemplo: `std::string m= "El valor " "no " "es válido";`

Operando con strings

- Comparación y acceso
 - La comparación de strings se hace carácter a carácter empezando por el primero en orden alfabético.
 - El operador [] permite acceder, especificando el índice, a los caracteres que componen el string.


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

```
int main()
```

```
{
```

```
// Declaraciones, en algunos casos con inicialización
```

```
std::string s1 = "uno";
std::string s2 = "dos\n";
std::string salto = "\n";
std::string s3 = s1 + "(copia s3)" + salto;
std::string s4, s5;
```

```
s4 = s1; // Asignación de un string
s4 += "(copia s4)"; // Concatenación y asignación
```

```
std::cout << "Introduzca s5: ";
std::cin >> s5; // Entrada por teclado
```

```
// Escritura en consola, ilustrando concatenación
```

```
std::cout << "s1: " + s1 + salto +
"s2: " + s2 +
"s3: " + s3 +
"s4: " + s4 + salto +
"s5: " + s5 << std::endl;
```

```
// Comparaciones mayor que, menor que, igual que, diferente de
```

```
std::cout << "s1 mayor que s2?: " << (s1>s2) << salto;
std::cout << "s1 menor que s3?: " << (s1<s3) << salto;
std::cout << "s1 igual que s4?: " << (s1==s4) << salto;
std::cout << "s1 no igual a s4?: " << (s1!=s4) << salto;
```

```
// Comparaciones de igualdad con resultado true
```

```
std::string sla = "uno", slb;
slb = s1;
std::cout << "s1 igual que sla?: " << (s1==sla) << salto;
std::cout << "s1 igual que slb?: " << (s1==slb) << salto;
std::cout << "s1 igual que \"uno\"?: " << (s1=="uno") << salto;
```

```
// Acceso especificando subíndice
```

```
std::cout << "s1 deletreado: " <<
s1[0] << ", " << s1[1] << ", " << s1[2] << std::endl;
```

```
// Comparaciones de igualdad especificando subíndice
```

```
std::cout << "Segundo caracter de s1 es \'n\'?: " << (s1[1]=='n') << salto;
std::cout << "Segundo caracter de s1 es \'o\'?: " << (s1[1]=='o') << salto;
```

```
return 0;
```

```
}
```

```
Introduzca s5: FIN
```

```
s1: uno
```

```
s2: dos
```

```
s3: uno(copia s3)
```

```
s4: uno(copia s4)
```

```
s5: FIN
```

```
s1 mayor que s2?: 1
```

```
s1 menor que s3?: 1
```

```
s1 igual que s4?: 0
```

```
s1 no igual a s4?: 1
```

```
s1 igual que sla?: 1
```

```
s1 igual que slb?: 1
```

```
s1 igual que "uno"?: 1
```

```
s1 deletreado: u, n, o
```

```
Segundo caracter de s1 es 'n'?: 1
```

```
Segundo caracter de s1 es 'o'?: 0
```

Operando con strings

- En la cabecera *string* se encuentran, además de operadores, *funciones miembro* que permiten analizar y manipular las variables de ese tipo.
- En general se suelen invocar de este modo:
`var_string.funcion(lista_valores_parametros)`

Operando con strings

- Funciones miembro:
 - *size*: Devuelve el número de caracteres. Ej. `s.size()`
 - *assign*: asigna a la variable a la que se le aplica bien un *string*, o bien parte de él. Es equivalente a `=` cuando se añade un string completo
 - *append*: añade un *string*, o parte de él, a la variable que se aplica. Es equivalente a `+` cuando se añade un string completo.
 - *insert*: interta en la variable sobre la que se aplica, totalidad o parte de un string.
 - *replace*: reemplaza en la variable sobre la que se aplica, totalidad o parte de un string.
 - *erase*: elimina caracteres de la variable sobre la que se aplica.



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

```
int main()
{
    std::string s1 = "ABCDE_FGHIJ_KLMNO_PQRST_UVWYZ";
    std::string s2 = "123456789";

    std::cout << "Strings iniciales\n" <<
        "s1: " + s1 + "\ns2: " + s2 << std::endl;

    std::cout << "s1 tiene " << s1.size() << " caracteres,\t" <<
        "s1[0] = " << s1[0] <<
        ", s1[" << s1.size()-1 << "] = " << s1[s1.size()-1] << std::endl;

    std::cout << "s2 tiene " << s2.size() << " caracteres,\t" <<
        "s2[0] = " << s2[0] <<
        ", s2[" << s2.size()-1 << "] = " << s2[s2.size()-1] << std::endl;

    // Asignar
    // Se asignan a s3 5 caracteres de s1, comenzando por s1[6]
    std::string s3;
    s3.assign(s1,6,5);
    std::cout << "\nSe define s3 a partir de s1\n";
    std::cout << "s3: " << s3 << std::endl;

    // Insertar
    // s3 se inserta en s2, comenzando en s2[3]
    std::cout << "\nInsertar s3 en s2\n";
    s2.insert(3,s3); //
    std::cout << "s2: " << s2 << std::endl;

    // Borrar
    // Se borran de s2 s3.size() caracteres, comenzando por s2[3]
    std::cout << "\nBorrar de s2 los caracteres insertados\n";
    s2.erase(3,s3.size());
    std::cout << "s2: " << s2 << std::endl;

    // Reemplazar
    // 5 caracteres de s1, comenzando por s1[12], son reemplazados por s2
    std::cout << "\nReemplaza 5 caracteres en s1 por todo s2\n";
    s1.replace(12,5,s2);
    std::cout << "s1: " << s1 << std::endl;

    return 0;
}
```

Strings iniciales

s1: ABCDE_FGHIJ_KLMNO_PQRST_UVWYZ

s2: 123456789

s1 tiene 29 caracteres, s1[0] = A, s1[28] = Z

s2 tiene 9 caracteres, s2[0] = 1, s2[8] = 9

Se define s3 a partir de s1

s3: FGHIJ

Insertar s3 en s2

s2: 123FGHIJ456789

Borrar de s2 los caracteres insertados

s2: 123456789

Reemplaza 5 caracteres en s1 por todo s2

s1: ABCDE_FGHIJ_123456789_PQRST_UVWYZ

Operando con punteros

- El operador *direccion-de* (&) que aplicado a una variable nos devuelve su dirección.
- El operador unario *indirección* (*) que aplicado a una dirección nos devuelve el valor de la variable allí almacenada.

Programa	Salida
<pre>#include <iostream> int main(){ int i=45; int *p; p=&i; std::cout<<"Valor variable i: "<<i<<std::endl; std::cout<<"Valor dirección i: "<<&i<<std::endl; std::cout<<"Valor variable p: "<<p<<std::endl; std::cout<<"Valor variable apuntada por p: "<<*p<<std::endl; return 0; }</pre>	<pre>Valor variable i: 45 Valor dirección i: 0xbfcff26c Valor variable p: 0xbfcff26c Valor variable apuntada por p: 45</pre>

Operando con punteros

- El uso de un puntero no se limita a leer el dato almacenado en la variable apuntada. También se puede modificar el valor de la variable apuntada.

```
#include <iostream>

int main()
{
    int i = 30;    // i: variable entera
    int *p;        // p: puntero a un entero
    p = &i;        // p apunta a i
    std::cout << "Valor de p: " << p << std::endl;
    std::cout << "Valor apuntado por p: " << *p << std::endl;
    std::cout << "Valor de i: " << i << std::endl;
    *p = 25;       // El valor de i pasa a ser 25
    std::cout << "Valor apuntado por p: " << *p << std::endl;
    std::cout << "Valor de i: " << i << std::endl;
    return 0;
}
```

```
Valor de p: 0x28ff24
Valor apuntado por p: 30
Valor de i: 30
Valor apuntado por p: 25
Valor de i: 25
```

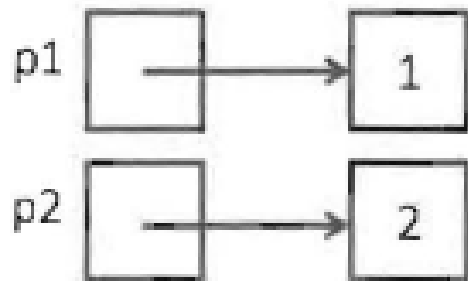
Operando con punteros

- Se puede asignar una variable puntero a otra variable puntero. El efecto es que la dirección de memoria que almacena el puntero de la derecha se copia en el puntero de la izquierda.
- En otras palabras, el puntero de la izquierda pasa a apuntar a la misma variable que apunta el puntero de la derecha.
- Si $p1$ y $p2$ son punteros, entonces $p1 = p2$ hace lo explicado.
- Pero $*p1 = *p2$ hace que la variable que apunta a $p1$ tome el valor de la variable apuntada por $p2$.

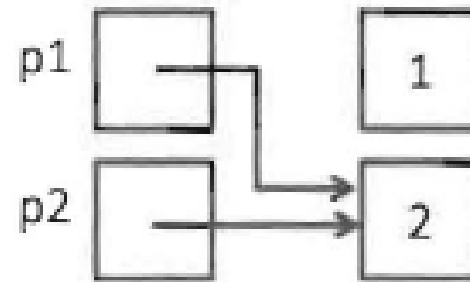
Operando con punteros

`p1 = p2;`

Antes:

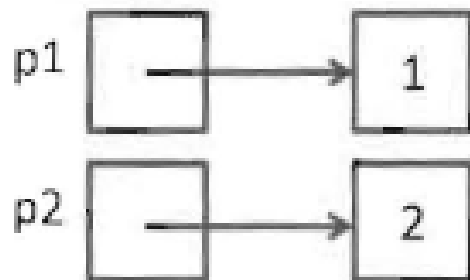


Después:

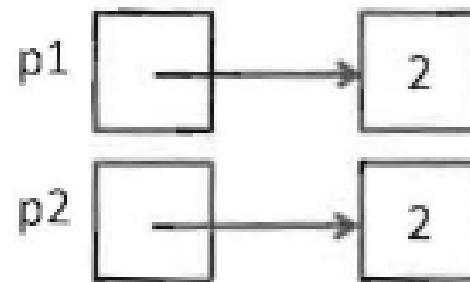


`*p1 = *p2;`

Antes:



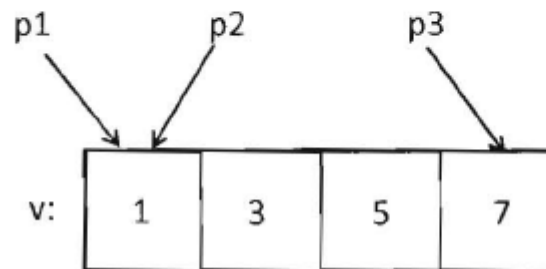
Después:



Relación entre punteros y arrays

- El nombre del array representa la dirección de memoria del primer elemento del array.
- `int v[4]={1, 3, 5, 7}`
- `v` representa la dirección del primer componente del array. Es decir, `v` es idéntico a `&v[0]`.

```
int v[4] = {1, 3, 5, 7};  
int *p1 = v;      // puntero al elemento inicial (conversion implicita)  
int *p2 = &v[0]; // puntero al elemento inicial  
int *p3 = &v[3]; // puntero al último elemento
```



Relación entre punteros y arrays

- Si p es un puntero que apunta a la componente $v[i]$ de un array: p es igual a $\&v[i]$.
- Si n es un entero $p+n$ es un puntero que apunta a $v[i+n]$. Es decir, $p+n$ es idéntico a $\&v[i+n]$, y consecuentemente $*(p+n)$ es idéntico a $v[i+n]$.
- $p+1$ apunta al siguiente componente de v , y $p-1$ al anterior. Es decir, $p+1$ es igual a $\&v[i+1]$ y $p-1$ es igual a $\&v[i-1]$.

Relación entre punteros y arrays

```
#include <iostream>

int main()
{
    double *v;
    int numComponentes = 3;
    v = new double [2*numComponentes];

    // Acceso especificando el índice
    v[0] = 1.5; v[1] = 2.5; v[2] = 3.2;
    v[3] = 5.6; v[4] = 7.9; v[5] = 11.3;
    std::cout << "v = ("
        << v[0] << ", " << v[1] << ", " << v[2] << ", "
        << v[3] << ", " << v[4] << ", " << v[5] << ")"
        << std::endl;

    // Acceso mediante aritmética de punteros
    *v = 0;    *(v+1) = 0.1; *(v+2) = 0.2;
    *(v+3) = 0.3; *(v+4) = 0.4; *(v+5) = 0.5;
    std::cout << "v = ("
        << *v << ", " << *(v+1) << ", " << *(v+2) << ", "
        << *(v+3) << ", " << *(v+4) << ", " << *(v+5) << ")"
        << std::endl;

    delete [] v;
    return 0;
}
```

```
v = (1.5, 2.5, 3.2, 5.6, 7.9, 11.3)
v = (0, 0.1, 0.2, 0.3, 0.4, 0.5)
```

Operando con vectores

- La cabecera *vector* además del tipo de datos *std::vector* incluye la definición de operadores y funciones para trabajar con vectores.
- Acceso a los componentes
 - La indexación de los componentes de un vector comienza en 0, igual que los arrays.
 - Existen varios modos de acceder a los componentes.
 - *Acceso mediante el operador []*: igual que con arrays, si *v* es un vector accedemos al *i*-ésimo elemento con *v[i]*. Como en los arrays no se comprueba que el valor *i* esté dentro del rango de valores posibles para el índice del vector.

Operando con vectores

```
// Fichero: vectorAccesol.cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<double> v(4,0);    // v = (0,0,0,0)
    // Asignación de valor a los componentes
    v[0] = 1;
    v[1] = 3*v[0];
    v[2] = 5*v[1]+v[0];
    v[3] = 7*v[2]+2*v[1]+v[0];
    // Lectura del valor de los componentes
    std::cout << "v = ( " << v[0] << ", " <<
        v[1] << ", " << v[2] << ", " << v[3] << " )" << std::endl;
    return 0;
}
```

v=(1 ,3, 16, 119)

Operando con vectores

- *Acceso mediante la función miembro `at()`.* Se accede al componente i -ésimo de `v` con `v.at(i)`. Con este método, el programa comprueba antes de intentar acceder al elemento que el valor del índice esté dentro del rango de valores posibles para el índice. Esto consume tiempo.

Si el valor está fuera del rango, la ejecución del programa lanza una excepción del tipo `std::out_of_range`. El siguiente fragmento de código muestra un ejemplo de cómo puede capturarse la excepción:

```
try{  
    //accesos usando at()  
}catch (std::out_of_range){  
    //acciones para este caso  
}
```

Si no se captura, la excepción aborta la ejecución del programa.

- *Acceso mediante iterador.* Un iterador es una abstracción de puntero

Operando con vectores

```
// Fichero: vectorAcceso2.cpp
#include <iostream>
#include <stdexcept>
#include <vector>

int main()
{
    std::vector<double> v(4,0);    // v = (0,0,0,0)
    try {
        // Asignación de valor a los componentes
        v.at(0) = 1;
        v.at(1) = 3*v.at(0);
        v.at(2) = 5*v.at(1) + v.at(0);
        v.at(3) = 7*v.at(2) + 2*v.at(1) + v.at(0);
        // Lectura del valor de los componentes
        std::cout << "v = ( " << v.at(0) << ", " <<
            v.at(1) << ", " << v.at(2) << ", " << v.at(3) << " )" << std::endl;
        // Intento de acceso fuera de rango. Lanza la excepción
        std::cout << "Esta línea no se imprime " << v.at(4) << std::endl;
    } catch (std::out_of_range) {
        std::cerr << "Error: vector v fuera de rango" << std::endl;
    }
    return 0;
}
```

Con try

```
v = ( 1, 3, 16, 119 )
```

```
Error: vector v fuera de rango
```

```
Debería seguir por aquí
```

Sin try

```
v = ( 1, 3, 16, 119 )
```

```
terminate called after throwing an instance of 'std::out_of_range'
```

```
what(): vector::_M_range_check
```

```
Cancelado
```

Operando con vectores

- La declaración de un iterador para un vector del tipo `std::vector<tipo>` es de la forma:

```
std::vector <tipo>::iterator nombre_iterador;
```

Por ejemplo

```
std::vector <double>::iterator iter;
```

- La clase `std::vector` tiene funciones miembro que devuelven iteradores:
 - `v.begin()` apunta al primer elemento (al que tiene índice 0)
 - `v.end()` apunta a una posición de memoria más allá del último componente del vector. Si a `v.end()` se le resta uno, se obtiene un iterador que apunta al último componente del vector.

Operando con vectores

- Los operadores $++$ y $--$ pueden usarse para incrementar y decrementar un iterador.
- Sumando o restando un valor entero al iterador, se desplaza ese número de componentes.
- Aplicando el operador $*$ sobre el iterador, se accede al componente del vector

Operando con vectores

```
// Fichero: vectorAcceso3.cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<double> v(4,0); // v = (0,0,0,0)
    // Declaración de un iterador para un vector de double
    std::vector<double>::iterator iter;
    // Se hace apuntar el iterador al primer componente del vector
    iter = v.begin();
    // Asignación de valor a los componentes
    *iter = 1;
    *(iter+1) = 3*(*iter);
    *(iter+2) = 5*(*iter+1) + (*iter);
    *(iter+3) = 7*(*iter+2) + 2*(*iter+1) + (*iter);
    // Lectura del valor de los componentes
    std::cout << "v = ( " << *iter << ", " <<
        *(iter+1) << ", " << *(iter+2) << ", " <<
        *(iter+3) << " )" << std::endl;
    return 0;
}
```

Operando con estructuras

- Para acceder a los miembros de una estructura se utiliza el *operador punto*:

`nombre_variable.nombre_miembro`

- Tanto si la variable es un array, como si el miembro es un array, se puede acceder al componente especificando el correspondiente índice.

```
struct Estrella {  
    char    nombre[30];  
    char    tipo;  
    double  distancia;  
    int     brillo;  
    double  coordenadas[3];  
};  
Estrella e[2];
```

- Mediante `e[1].coordenadas[2]` se accede a la variable tipo `double` que es el tercer elemento del array de doubles `coordenadas`.