

Cuaderno de problemas
Estructuras de Datos y Algoritmos.

Resumen C++

Prof. Isabel Pita

4 de septiembre de 2019

Índice

1. Tipos simples.	3
1.1. Tipos y declaración de variables.	3
1.2. El tipo <code>char</code>	3
1.3. Uso del modificador de acceso <code>const</code>	3
2. Renombramiento de tipos.	4
3. El tipo <code>struct</code>.	4
3.1. Para que se utiliza.	4
3.2. Declaración e inicialización de variables tipo <code>struct</code>	4
3.3. Acceso a los campos de una variable tipo <code>struct</code>	5
4. Sobrecarga de operadores.	5
4.1. Operadores de entrada/salida	5
4.2. Operadores relacionales: menor, mayor, igual, menor o igual, mayor o igual.	6
4.3. Operadores aritméticos: suma, resta, multiplicación, división:	6
4.4. Los operadores incremento y decremento:	6
4.5. El operador paréntesis	6
5. Paso de parámetros.	6
5.1. Formas de paso de parámetros.	6
5.2. Valor por defecto de los parámetros de entrada a una función.	7
6. El tipo <code>std::string</code>	7
7. Vectores.	7
7.1. Cómo declarar una variable de tipo <code>vector</code>	8
7.2. Cómo dar valor a las componentes de un vector.	8
7.3. Funciones para manejar vectores que se utilizarán este curso.	9
8. Matrices.	9
8.1. Cómo declarar una variable de tipo <code>tmatriz</code>	9
8.2. Cómo dar valor a las componentes de una matriz.	9
9. Funciones genéricas.	10
10. Funciones de la librería STL.	10
10.1. Ordenar los elementos de un vector, <code>sort</code>	10
10.2. Intercambio del valor de dos variables, <code>swap</code>	11
10.3. Máximo y mínimo de dos valores, <code>max</code> y <code>min</code>	11
10.4. Generar las permutaciones de los elementos de un vector, <code>next_permutation</code>	11

1. Tipos simples.

1.1. Tipos y declaración de variables.

- C++ proporciona 5 tipos simples: `int`, `float`, `double`, `bool`, `char`.
- Existen tres modificadores que se aplican al tipo `int`: `unsigned`, `short`, `long`.
- Ejemplos de declaración de variables:

```
int n; int n = 3;
int k1, k1; int n = 3, m = 5;
unsigned k; short n; long h;
unsigned int k; short int n; long int h;
unsigned long n; unsigned long long k = 0;
float x = 0.5; double z;
bool q = true; char c = 'r';
```

- El rango de valores que puede almacenar cada tipo se puede consultar en la documentación de la librería `climits`. (www.cplusplus.com - climits)

1.2. El tipo `char`.

- Los posibles valores del tipo son los caracteres definidos en el código ASCII. A cada carácter le corresponde un valor numérico que es su posición en este código.

Para transformar un carácter en su valor numérico o un valor numérico en su carácter se utiliza un casting:

```
char a = 'a'; int n = 97;
std::cout << int(a); std::cout << (int)a;
std::cout << char(n); std::cout << (char)n;
```

Para calcular la posición de una letra en el alfabeto o para calcular el valor entero de un carácter numérico:

```
char letra = 'g';
std::cout << letra - 'a'; // Escribe 6 que es la posición de la letra g en el alfabeto
char caracter = '4';
int n = caracter - '0'; // n toma el valor 4.
```

Otras funciones de la librería `cctype`, útiles sobre los caracteres son (consultar la información en la página cplusplus.com):

- `toupper`. Transforma una letra minúscula en mayúscula.
- `tolower`. Transforma una letra mayúscula en minúscula.
- `islower`. Devuelve cierto si un carácter es una letra minúscula.
- `isupper`. Devuelve cierto si un carácter es una letra mayúscula.
- `isalpha`. Devuelve cierto si un carácter es alfabético.
- `isalnum`. Devuelve cierto si un carácter es alfanumérico.
- `isdigit`. Devuelve cierto si un carácter es numérico.

1.3. Uso del modificador de acceso `const`.

1. Declarar constantes del programa. Valores que no se modifican durante toda la ejecución.

```
const int MAX_PRECIO = 500;
```

- a) Hay que darles valor al declararlas, ya que no se les puede asignar ningún valor posteriormente.
- b) Utilizaremos identificadores en letras mayúsculas para diferenciarlas de las variables del programa.

- c) Pueden declararse globales a todo el programa.
2. Declarar parámetros por referencia como constantes para evitar que se modifiquen.

```
void funcion (std::string const& s);
```

2. Renombramiento de tipos.

Se utiliza cuando el tipo definido es muy largo o cuando se quiere dar significado al tipo. Esto ocurre especialmente con los tipos estructurados.

Para renombrar un tipo utilizaremos la instrucción:

```
using lli = long long int;  
using edad = int;
```

donde:

- lli es el nuevo nombre dado al tipo.
- long long int es el tipo que se quiere renombrar.

3. El tipo struct.

3.1. Para que se utiliza.

Se utiliza para agrupar valores que representen conceptos diferentes:

```
struct tDatos {  
    float precio;  
    int cantidad;  
    char identificador;  
};
```

3.2. Declaración e inicialización de variables tipo struct.

Podemos dar un valor inicial a los *campos* cuando definimos el tipo.

- Para ello podemos definir métodos constructores:

```
struct tDatos {  
    float precio;  
    int cantidad;  
    char identificador;  
    tDatos() { precio = 0.0; cantidad = 0; identificador = 'z';}  
    tDatos(float pr, int ct, char id) { precio = pr; cantidad = ct; identificador = id;}  
};
```

- Las constructoras se llaman igual que el **struct** que se está definiendo;
- Podemos definir varias constructoras siempre que se diferencien en el número o tipo de sus parámetros;
- Si se define una constructora con parámetros debe definirse también una sin parámetros con los valores por defecto.

Para declarar una variable:

```
tDatos datos; // Utiliza la constructora sin parámetros  
tDatos datos(3.5, 4, 'f'); // Utiliza la constructora con tres parámetros
```

- O podemos asignar el valor directamente al campo. En este caso todas las variables que se declaren del tipo estarán inicializadas al valor indicado en la definición del tipo. Si no se da valor a algún *campo*, éste queda sin inicializar al declarar la variable.

```
struct tDatos {
    float precio = 0.0;
    int cantidad = 0;
    char identificador = 'z';
};
```

Para declarar una variable:

```
tDatos datos;
```

- También se pueden inicializar los *campos* al declarar una variable.

```
tDatos datos = {1.0, 100, 'a'};
```

- o se pueden inicializar *campo a campo*, aunque esto puede resultar *tedioso*.

```
tDatos datos;
datos.precio = 1.0;
datos.cantidad = 100;
datos.identificador = 'a';
```

3.3. Acceso a los campos de una variable tipo struct.

- Para acceder a cada campo de forma individual se utiliza el operador *punto*:

```
if (datos.precio < MAX_PRECIO) std::cout << "Rebajas\n";
```

4. Sobrecarga de operadores.

Los operadores se sobrecargan para poder utilizarlos con tipos definidos por el usuario. Los operadores que sobrecargaremos este curso son:

4.1. Operadores de entrada/salida

```
:
struct tPunto {
    int x,y;
    tPunto() {x = 0; y = 0;}
    tPunto(int px, int py) {x = px; y = py;}
};

std::istream& operator>> (std::istream& in, tPunto& p) {
    in >> p.x >> p.y;
    return in;
}

std::ostream& operator<< (std::ostream& out, tPunto const& p) {
    out << p.x << ' ' << p.y << '\n';
    return out;
}

int main() {
    tPunto p;
    std::cin >> p;
    std::cout << p;
}
```

4.2. Operadores relacionales: menor, mayor, igual, menor o igual, mayor o igual.

```
bool operator< (tPunto const& p1, tPunto const& p2) {
    if (p1.x < p2.x) return true;
    else if (p1.y < p2.y) return true;
    else return false;
}
```

4.3. Operadores aritméticos: suma, resta, multiplicación, división:

```
tPunto operator+ (tPunto const& p1, tPunto const& p2) {
    return tPunto(p1.x + p2.x, p1.y+p2.y);
}
```

4.4. Los operadores incremento y decremento:

Los operadores de incremento y decremento pueden utilizarse en forma prefija o en forma posfija. Cada uno tiene un comportamiento diferente. La forma prefija primero incrementa el valor de la variable y a continuación utiliza el valor incrementado en la expresión en que aparece el operador. La forma posfija utiliza el valor sin incrementar en la expresión y lo incrementa posteriormente. Por ello, hay que implementar dos funciones diferentes. Como ambas reciben el mismo nombre, `operator++`, la forma de diferenciarlas es mediante los parámetros. Se añade un parámetro ficticio de tipo entero a las implementaciones de los operadores posfijos. De esta forma el compilador hace la llamada a la función adecuada. A continuación se muestra la implementación del operador incremento.

```
// incremento prefijo
tPunto operator++(tPunto & punto) {
    ++punto.x; ++punto.y;
    return punto;
}
// incremento posfijo
tPunto operator++(tPunto & punto, int) {
    tPunto aux = punto;
    ++punto.x; ++punto.y;
    return aux;
}
```

4.5. El operador paréntesis

Permite definir funciones que pueden pasarse como parámetros a otras funciones.

```
struct comparaPuntos {
    bool operator() (tPunto const& p1, tPunto const& p2) {
        if (p1.x < p2.x) return true;
        else if (p1.y < p2.y) return true;
        else return false;
    }
}

int main() {
    std::vector<tPunto> v;
    ....
    std::sort(v.begin(), v.end(), comparaPuntos());
}
```

5. Paso de parámetros.

5.1. Formas de paso de parámetros.

- *Por valor* que es la forma por defecto que se utiliza cuando no se indica nada. En este caso, se realiza una copia del valor pasado como argumento y se utiliza dicha copia en la función. *Deben evitarse las copias de los tipos estructurados, especialmente de objetos, vectores y cadenas de caracteres, porque pueden resultar muy costosos en tiempo.*
- *Por referencia*, se indica con un `&` después del tipo del parámetro. En este caso, la memoria del parámetro y del argumento coinciden y por lo tanto todos los cambios que se realicen en el parámetro quedarán reflejados en la variable utilizada como argumento.
- *Constante por referencia*, se indica con `const&` después del tipo del parámetro. En este caso, la memoria del parámetro y del argumento coinciden, pero no puede modificarse el valor del parámetro en la función debido a que se declara el parámetro `const`. *Se utiliza cuando el parámetro es un vector, una cadena de caracteres o un objeto, para evitar realizar la copia que resulta costosa.*

5.2. Valor por defecto de los parámetros de entrada a una función.

```
int f (float x, int k = 0, bool cierto = true) { ... }
```

Las llamadas a la función admiten tres formas, con uno, dos o tres argumentos;

```
int main() {
    float y = 5.3; int n = 1; bool ok = false;
    std::cout << f(y) << '\n'; // Se ejecuta la funcion con los parametros k=0 y cierto=true
    std::cout << f(y,n) << '\n'; // Se ejecuta la funcion con el parametro cierto = true
    std::cout << f(y,n,ok) << '\n'; // Se ejecuta la funcion con los valores de entrada
}
```

6. El tipo `std::string`

- La librería STL proporciona la clase `std::string` que nos permite guardar cadenas de caracteres.
- Las cadenas de caracteres constantes, denominadas *literales* se representan entre comillas dobles. Por ejemplo "Hola". Se debe diferenciar entre una cadena de caracteres con un único carácter ("a") del carácter ('a'). Esto es importante, porque no podemos comparar una cadena de caracteres con un carácter.

```
std::string cadena; std::cin >> cadena;
if (cadena = ".") .....
```

Esta comparación es correcta, pero no lo será si comparamos con el carácter '.'.

- Para leer una cadena de caracteres (este curso la entrada de datos se realizará redireccionando un fichero de entrada a la consola):
 - Si la cadena no tiene blancos, tabuladores o saltos de línea, puede utilizarse el operador `std::cin >> aux;` donde el primer parámetro es la consola, y el segundo una cadena de caracteres. Este operador lee todos los caracteres hasta encontrar un blanco, tabulador o salto de línea. El blanco, tabulador o salto de línea no se consume de la entrada.
 - Si la cadena tiene blancos, tabuladores o saltos de línea debemos utilizar la instrucción: `getline(std::cin, aux);`, donde el primer parámetro de la función es la consola, y el segundo parámetro es la cadena de caracteres en que se guarda el valor leído. Con esta instrucción se leen todos los caracteres hasta el final de línea incluido. Existe la opción de leer un número determinado de caracteres, o hasta que aparezca un determinado carácter, ver la documentación: www.cplusplus.com - getline.

7. Vectores.

Existen diversas formas de declarar vectores en C++. Este curso emplearemos la clase `std::vector` definida en la librería STL. Esta clase nos proporciona muchas facilidades para el manejo de los vectores. Permite definir el tamaño del vector en tiempo de ejecución (como ocurre con los vectores dinámicos de C) y aumenta automáticamente el tamaño del vector cuando se inserta un elemento y el vector se encuentra lleno. Además proporciona muchas funciones para el manejo del vector (www.cplusplus.com - vector)

7.1. Cómo declarar una variable de tipo vector.

Hay dos formas básicas de declarar una variable de tipo `vector`.

1. Si conocemos el número de elementos del vector:

```
std::vector<tipo> v(numElems);
```

donde *tipo* es el tipo de las componentes del vector y *numElems* el número de elementos del vector. Si los elementos son de un tipo simple quedan inicializados al valor por defecto del tipo.

Si no queremos inicializar los valores del vector al valor por defecto del tipo, usaremos la declaración:

```
std::vector<tipo> v(numElems,valorInicial);
```

donde *valorInicial* es el valor al que se inicializarán las componentes del vector.

2. Si no conocemos el número de elementos del vector, podemos crear un vector vacío.

```
std::vector<tipo> v;
```

En este caso, el vector está vacío y por lo tanto no podemos dar valor a sus componentes, porque no las tiene. Tendremos que añadir las componentes con la operación `push_back` como se verá en el apartado siguiente.

7.2. Cómo dar valor a las componentes de un vector.

1. Si el vector se ha declarado con un número de elementos, daremos valor a las componentes leyéndolo de consola o con el operador de asignación.

Si el vector se va a recorrer entero empezando por la componente cero podemos utilizar el siguiente bucle `for`:

```
for (int& n : v) std::cin >> n;
```

donde *n* es una referencia de tipo entero a cada componente del vector, empezando por la primera, y *v* es un vector. Observad que se utiliza una referencia para poder dar valor a las componentes del vector. Si el bucle fuese para escribir los valores del vector, no se utilizaría la referencia.

Si el vector no se va a recorrer entero, o si se desea recorrerlo en orden inverso utilizaremos el siguiente bucle:

```
for (int i = v.size(); i ≥ 0; --i) std::cin >> v[i];
```

Obviamente también se puede utilizar este bucle para leer todas las componentes del vector desde el cero.

2. Si el vector se ha declarado vacío, tenemos que utilizar la función `push_back` de la clase `vector`. Esta función añade un elemento al final del vector. Si el vector está lleno, amplía su capacidad antes de añadir el elemento. Si el vector se declaró vacío es porque no se conocía su número de elementos, entonces, para poder tratar la entrada normalmente se utilizará un valor *centinela* para indicar el final de los datos. Utilizaremos el siguiente bucle:

```
int aux;
std::cin >> aux;
while (aux != centinela) {
    v.push_back(aux);
    std::cin >> aux;
}
```


7.3. Funciones para manejar vectores que se utilizarán este curso.

Dado un vector `v`, y una variable `aux`, del tipo al que esté declarado el vector:

- `v.push_back(aux)`; añade el valor `aux` al final del vector. La operación tiene coste amortizado constante, pero en el caso peor (el vector está lleno) tiene coste lineal en el número de elementos del vector, ya que debe realizar una copia.
- `v.clear()`; elimina todos los elementos del vector y los destruye, dejando el tamaño a cero. La operación tiene coste constante cuando los elementos del vector son de un tipo simple que no necesita ser destruido. Lineal respecto al número de elementos del vector si estos deben ser destruidos.
- `v.resize(n)`; modifica el tamaño del vector dejando las primeras `n` componentes. Si el nuevo tamaño es menor que el que tenía el vector, el coste es constante si no es necesario destruir los elementos y lineal en el número de elementos eliminados si estos deben ser destruidos. Si el nuevo tamaño es mayor que el que tenía el vector el coste es lineal en el número de elementos del vector, ya que debe realizarse una copia para permitir los nuevos elementos. *Esta función la utilizaremos únicamente cuando queramos eliminar los elementos mas a la derecha del vector.*
- Las funciones `insert` y `erase` deben **utilizarse con mucho cuidado, porque su coste es lineal en el número de elementos del vector.**

8. Matrices.

8.1. Cómo declarar una variable de tipo `tmatrix`.

Para declarar una matriz de dos dimensiones declararemos un vector cuyas componentes son a su vez vectores. Declaramos la matriz genérica e instanciamos el tipo de los elementos al declarar la variable.

- Utilizaremos un renombramiento del tipo

```
template <class T>
using tmatrix = std::vector<std::vector<T>>;
```

- Para declarar una matriz vacía:

```
tmatrix<int> matriz;
```

- Para declarar una matriz de dimensiones $dim1 \times dim2$:

```
int numElem1, numElem2; std::cin >> numElem1 >> numElem2;
tmatrix<int> matriz(numElem1, std::vector<int>(numElem2));
```

- Para declarar la matriz de dimensiones $dim1 \times dim2$, con valor inicial -1:

```
int numElem1, numElem2; std::cin >> numElem1 >> numElem2;
tmatrix<int> matriz(numElem1, std::vector<int>(numElem2, -1));
```

8.2. Cómo dar valor a las componentes de una matriz.

- Si se reservó memoria al declarar la matriz dándole las dimensiones en la declaración:

```
for (int i = 0; i < numElem1; ++i) {
    for (int j = 0; j < numElem2; ++j)
        std::cin >> matriz[i][j];
}
```

o bien

```
for (std::vector<int> & v : matriz) {
    for (int &n : v) {
        std::cin >> n;
    }
}
```

- Si la matriz es vacía, pero conocemos sus dimensiones, debemos dar valor a las componentes de cada fila antes de añadir la fila a la matriz

```
for (int i = 0; i < numElem1; ++i) {
    // Se leen los valores de una fila
    std::vector<int> v;
    for (int j = 0; j < numElem2; ++j) {
        int aux; std::cin >> aux;
        v.push_back(aux);
    }
    // Se anade la fila a la matriz
    matriz.push_back(v);
}
```

- En este curso, no se utilizarán matrices que no se conozca su dimensión.

9. Funciones genéricas.

Para declarar una función genérica en C++ se utiliza una plantilla: `template`.

```
template <typename T>
bool Search (std::vector<T> const& v, T const& x) {
    size_t i = 0;
    while (i < v.size() && v[i] != x)
        ++i;
    return i < v.size();
}
```

- La función puede ejecutarse con vectores de cualquier tipo de datos, siempre que el tipo tenga definido el operador `!=`.
- La llamada a la función es igual que la llamada a una función no genérica.

```
std::vector<int> v = {5,3,8};
int x = 6;
bool ok = Search(v,x);
```

- El tipo genérico se instancia al tipo del vector. Este debe coincidir con el tipo del segundo parámetro, en otro caso el compilador reporta un error.
- Puede utilizarse indistintamente las palabras reservadas `typename` y `class`.

10. Funciones de la librería STL.

La librería `algorithm` de la STL define una serie de algoritmos que resultarán útiles durante el curso:

10.1. Ordenar los elementos de un vector, `sort`.

Para ordenar los elementos de un vector podemos utilizar la función: `sort` de la librería `algorithm` de la STL.

El primer parámetro es un iterador al primer elemento del vector que se quiere ordenar, el segundo parámetro es un iterador al elemento posterior al último que se quiere ordenar, el tercer parámetro es el operador de comparación que se emplea. Cuando no se utiliza tercer parámetro se ordena con el operador menor. Si el tipo de los elementos a ordenar no tiene definido por defecto un operador menor es necesario definirlo como se explicó en el apartado de *Sobrecarga de operadores*. Si se quiere ordenar con otro criterio, se puede utilizar el operador mayor, indicándolo en el tercer parámetro con `greater` o definiendo un operador función mediante un struct. El operador debe ser siempre estricto.

```

std::vector<int> v;
std::sort(v.begin(), v.end()); // ordena de menor a mayor
std::sort(v.begin(), v.end(), std::greater<int>()) // ordena de mayor a menor

struct paresPrimero {
    bool operator() (int n1, int n2) {
        if (n1%2 == 0) return (n2%2 == 0) ? n1 < n2: true;
        else return (n2%2 == 0) ? false : n1 < n2;
    }
};
// La siguiente instrucción
// ordena primero los pares de menor a mayor y luego los impares
std::sort(v.begin(), v.end(), paresPrimero());

```

En lugar de una clase se permite utilizar una función booleana.

```

bool ordena(int n1, int n2) {
    if (n1%2 == 0){
        if (n2%2 == 0) return n1 < n2;
        else return true;
    }
    else { // n1 impar, n2 par
        if (n2%2 == 0) return false;
        else return n1 < n2;
    }
}
// La siguiente instrucción
// ordena primero los pares de menor a mayor y luego los impares
std::sort(v.begin(), v.end(), ordena);

```

10.2. Intercambio del valor de dos variables, swap.

Para intercambiar el valor de dos variables puedes utilizar la función: `swap` de la librería `utility` de la STL.

```

int a = 3, b = 5;
std::swap(a,b);
std::vector<int> v = {5,4,3};
std::swap(v[0],v[1]);

```

10.3. Máximo y mínimo de dos valores, max y min.

Para calcular el máximo de dos valores podemos utilizar la función `max`, y para calcular el mínimo la función `min`. Ambas se encuentran en la librería `algorithm` de la STL.

```

int a = 3, b = 5;
int c = std::max(a,b);
int d = std::min(a,b);

```

10.4. Generar las permutaciones de los elementos de un vector, next_permutation.

Para generar todas las permutaciones de los elementos de un vector puedes utilizar la función: `next_permutation` de la librería `algorithm` de la STL.

Esta función la utilizaremos para generar casos de prueba de nuestros programas. Nos permite obtener todas las permutaciones de los elementos de un vector. Las permutaciones se generan en orden lexicográfico, por lo tanto para generar todas las posibles se debe comenzar el bucle con el vector ordenado de menor a mayor.

```

std::vector<int> v = {5,4,3};
do {
    for (int n : v) std::cout << n << ' '; // Escribe la siguiente permutación

```

```
        std::cout << '\n';  
    } while (std::next_permutation(v.begin(), v.end()));
```