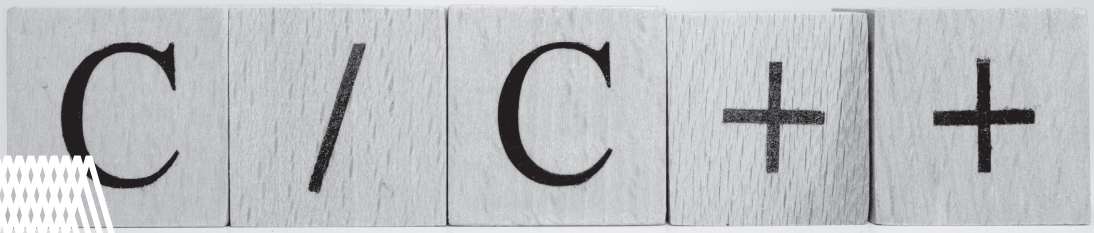
# Introdución á programación

## *en Linguaxe C*

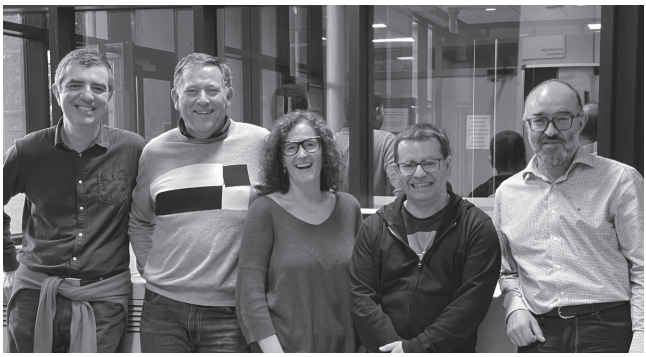



**Manuais**
Serie manuais didácticos

Autores

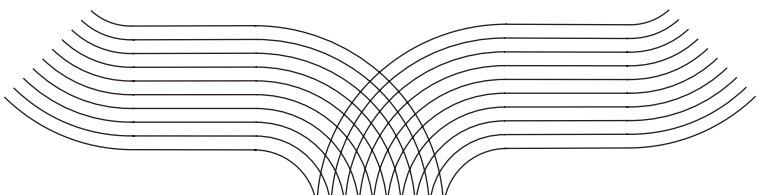
María José Lado Touriño
Leandro Rodríguez Liñares
Pedro Cuesta Morales
Arturo José Méndez Penín
Xosé Antón Vila Sobrino

María José Lado Touriño
Leandro Rodríguez Liñares
Pedro Cuesta Morales
Arturo José Méndez Penín
Xosé Antón Vila Sobrino



María José, Leandro, Pedro, Arturo e Xosé Antón forman parte do Departamento de Informática, Área de Linguaxes e Sistemas Informáticos da Universidade de Vigo. No ámbito da docencia, acumulan máis de 25 anos de experiencia en docencia teórico-práctica en diversas materias de primeiro, segundo e terceiro ciclo, correspondentes a distintas Licenciaturas e Enxeñarías, así coma en diferentes materias de Grao e Posgrao. No eido da investigación, pertencen ao grupo LIA2, da Escola Superior de Enxeñaría Informática (Campus de Ourense), e na actualidade céntranse no estudo de sinais fisiolóxicos, análise de variabilidade da frecuencia cardíaca, e desenvolvemento de ferramentas software neste campo. Todas estas ferramentas están sempre desenvolvidas baixo licenza de software libre e de código aberto, e dispoñibles para calquera persoa interesada no seu emprego, a través da URL https://github.com/milegroup. Entre os resultados máis destacados da súa actividade docente e investigadora pódense destacar a dirección de varias Teses de Doutoramento, diversos libros e capítulos, numerosas publicacións en revistas científicas, indexadas en JCR e SCOPUS, proceedings en congresos, e participación en diversos proxectos e contratos de investigación, redes e agrupacións estratéxicas, financiados todos eles con Fondos FEDER, Ministerio de Ciencia e Innovación, Xunta de Galicia, Universidade de Santiago de Compostela, Universidade de Vigo, así como diversas aportacións da empresa privada.

Servizo de Publicacións
Universidade Vigo



# Manuais

Serie de manuais didácticos

**n.º 092**

**Edición**
Universidade de Vigo
Servizo de Publicacións
Rúa de Leonardo da Vinci, s/n
36310 Vigo

**Deseño da portada**
Julinda Molares Cardoso e Tania Sueiro Graña
Área de Imaxe
Vicerreitoría de Comunicacións e Relacións Institucionais

**Maquetación**
Tórculo Comunicación Gráfica, S. A.

**Fotografía da portada**
Adobe stock

**Impresión**
Tórculo Comunicación Gráfica, S. A.

**ISBN (Libro impreso)**
978-84-1188-034-3

**Depósito legal**
VG 617-2024



Ao ser esta editorial membro da UNE, garántense a difusión e a comercialización das súas publicacións no ámbito nacional e internacional.

Servizo de Publicacións
Universidade de Vigo

HR EXCELLENCE IN RESEARCH

Este volume publícase
co financiamento da

XUNTA
DE GALICIA

# Introdución á programación

## *en Linguaxe C*

Autores

María José Lado Touriño
Leandro Rodríguez Liñares
Pedro Cuesta Morales
Arturo José Méndez Penín
Xosé Antón Vila Sobrino

# Índice

# Listaxe de Programas

# Listaxe de Figuras

# Listaxe de Táboas

# Capítulo 1

# Algoritmos e programas

## 1.1 Elementos dun programa: datos e algoritmos

Unha computadora é unha máquina capaz de capturar información, automatizala e transformala, que pode realizar cálculos e tomar decisións a grandes velocidades. Polo tanto, o **computador** pódese definir como un dispositivo electrónico que acepta uns datos de entrada, efectúa con eles operacións lóxicas (comparacións, copias, seleccións,...) e aritméticas (sumas, produtos,...), e proporciona a información resultante por medio dunha saída. Para realizar as operacións require unhas instrucións (*software*, soporte lóxico) que son executadas por dispositivos (*hardware*, soporte físico).

En Informática, cando falamos dun programa estámonos a referir a *software*, aplicacións e recursos que permiten realizar diferentes tarefas nunha computadora ou dispositivo móbil. Pódese definir un **programa** como un conxunto ordenado de instrucións (**instrución**: conxunto de símbolos que representa unha orde de operación ou tratamento para o ordenador) que se dan á computadora, e nas que se lle indican as operacións ou tarefas que debe realizar. Así, a programación será o proceso de escritura ou codificación dun programa. Un programa en execución é un **proceso**.

Un **algoritmo** é o conxunto de accións executadas para completar unha tarefa, con principio e fin. Pode ser realizado por persoas ou por máquinas. Define un conxunto de pasos ordenados, finitos e delimitados que describen de forma sistemática a execución dunha tarefa. Estes pasos deben estar especificados de xeito moi concreto para que non haxa dúbidas sobre a súa execución e que se acade con éxito.

Os algoritmos poden ser expresados con números, pero tamén con palabras. Correspóndense cunha descrición dun patrón de comportamento, e constan dun conxunto finito de accións. Calquera evento que teña inicio e fin e que involucre unha serie finita de pasos lóxicos pode ser descrito mediante un algoritmo.

***Exemplos***: receita de cociña, instrucións de montaxe dun aparato, ou algoritmo matemático de Euclides para obter o máximo común divisor de dous números enteiros positivos.

Un algoritmo caracterízase por ser:

- *Preciso*: indica orde de realización en cada paso.
- *Definido*: se se segue varias veces, o resultado é idéntico en cada unha delas.
- *Finito*: ten un número determinado de pasos.

Aínda que tanto o algoritmo como o programa son conxuntos finitos de instrucións e teñen principio e fin, a diferenza principal entre os dous conceptos radica no feito de que o algoritmo pode ser representado en linguaxe natural ou código, mentres que os programas unicamente poden ser escritos empregando unha linguaxe de programación. Polo tanto, para executar un algoritmo nunha computadora haberá que representalo en forma de programa. Pódese dicir que un programa é a representación dun algoritmo, e un proceso é a execución do algoritmo.

Os algoritmos e os programas que os representan reciben unha información de entrada, procésana (conxuntos finitos de instrucións que conforma o algoritmo) e devolven como resultado unha saída.

***Exemplo***: instrucións de montaxe dun aparato:

*ENTRADA: pezas e ferramentas necesarias*.

*PROCESO: montaxe*.

*SAÍDA: aparato montado*.

Un algoritmo é así unha secuencia ordenada e finita de instrucións para resolver un problema. Todas as tarefas que executa a computadora baséanse en algoritmos. Deste xeito, un programa informático deséñase con algoritmos, polo que se pode introducir nel unha tarefa e resolvela. A estrutura xeral dun algoritmo é similar á seguinte:

- **Cabeceira:**
  - Nome do algoritmo precedido da palabra ALGORITMO.
- **Corpo**: dous bloques:
  - *Declaracións*: defínense ou decláranse constantes, variables, tipos de datos definidos por nós.
  - *Instrucións*:
    - Inicio/fin: primeira (INICIO) e última (FIN) instrución.
    - Entrada de datos dende o dispositivo estándar : LER(<lista de variables>).
    - Saída de datos no dispositivo estándar: ESCRIBIR(<lista de expresións>).
    - Asignación: <nome de variable> $\leftarrow$ <valor>.
    - Bifurcación: controla que se executen ou non as instrucións, e altera a orde de execución das mesmas.
    - Comentarios: dunha soa liña // ou de varias liñas $\{\}$.

As instrucións comprendidas entre INICIO e FIN constitúen o **bloque de instrucións principal do algoritmo**.

***Exemplo***: algoritmo para sumar dous números introducidos por teclado e amosar o resultado (Programa 1.1).

Programa 1.1: Algoritmo para sumar dous números

```
ALGORITMO SUMAR
INICIO
    LER NUM1
    LER NUM2
    Almacenar en SUMA: suma de NUM1, NUM2
    ESCRIBIR SUMA
FIN
```

A información que se manexa no desenvolvemento dun algoritmo está representada polos **datos**, que non son máis que unha representación simbólica dun atributo. Poden ser de calquera tipo (atributo que se indica á computadora respecto á natureza dos mesmos: numéricos, alfabéticos, símbolos,...). Por eles mesmos non constitúen información, senón que é o procesamento dos mesmos o que a proporciona, e será necesario interpretalos.

Cada dato debe vir definido por:

- *Nome*: permite distinguilo dos restantes elementos.
- *Tamaño*: indica a cantidade de caracteres ou números que se poden utilizar para definir o seu valor.
- *Tipo*: describe se o elemento está constituído por caracteres alfabéticos, numéricos ou símbolos especiais.

En Informática, os datos manéxanse mediante algoritmos, e representan sucesos que se poden interpretar como ordes dun usuario a un sistema de computadora, ou a resposta do propio sistema ás ordes de entrada dadas polo usuario.

## 1.2 Codificación da información en memoria

Como acabamos de ver, unha computadora executa programas (representacións de algoritmos), que están constituídos por un conxunto finito de instrucións, e procesan datos. Para poder executalos, tanto os programas como os datos deben estar almacenados na memoria do ordenador, que unicamente almacenará información binaria (secuencias de 0s e 1s). Polo tanto, todos os datos que se van manexar, independentemente da súa natureza (números, letras, imaxe, son,...), deben ser codificados e representados mediante combinacións de 0s e 1s.

A **codificación** é un proceso de transformación mediante o que se representan de forma unívoca os elementos dun grupo mediante os doutro conxunto, de xeito que cada elemento do primeiro conxunto se corresponde cun elemento distinto do segundo. A computadora emprega un conxunto de oito díxitos binarios para representar un carácter, sexa número ou letra. Cada conxunto de 8 díxitos binarios é un ***byte (B)***. Cada un dos oito díxitos do *byte* é un ***bit (b, Binary Dixit)*** (John Tukey, 1947; Claude Shannon, 1948). Polo tanto, o *bit* é a unidade de información mínima coa que traballa unha computadora.

De xeito similar ao sistema decimal, que está composto por 10 elementos $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, o sistema binario está formado por 2 $\{0, 1\}$, o que significa que soamente ten dous valores. Os *bits* son a base do sistema binario.

Aínda que cun *bit* unicamente poden ser representados dous estados nunha máquina, calquera información pode ser codificada como unha combinación de *bits*, isto é, de 0s e 1s.

***Exemplo***: representación binaria dos díxitos que forman a base do sistema decimal (Táboa 1.1).

Táboa 1.1: Conversión da base decimal ao sistema binario

| **Decimal** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Binario** | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 |

Cando os datos se transforman en *bits* poden ser procesados con gran rapidez polas computadoras, que tamén poden reproducilos infinitas veces de forma exacta, ou transmitilos a grandes velocidades. Tanto o *bit* como o *byte* son unidades moi pequenas. Por iso, recórrese ao emprego de medidas que son múltiplos dos *bytes*. As máis utilizadas decote preséntanse na Táboa 1.2.

Táboa 1.2: Múltiplos dos *bytes* e equivalencias

| **Nome (Símbolo)** | **Valor en *bytes*** |
|---|---|
| 1 Kilobyte (KB) | $2^{10} \approx 10^3$ |
| 1 Megabyte (KB) | $2^{20} \approx 10^6$ |
| 1 Xigabyte (XB) | $2^{30} \approx 10^9$ |
| 1 Terabyte (TB) | $2^{40} \approx 10^{12}$ |
| 1 Petabyte (PB) | $2^{50} \approx 10^{15}$ |
| 1 Exabyte (EB) | $2^{60} \approx 10^{18}$ |

## 1.3 Linguaxes de programación

Para poder executar un algoritmo nunha computadora é necesario escribir un programa para representalo. Deste xeito, haberá que codificar cada unha das instrucións que compoñen o algoritmo e especifican

unha determinada operación que debe realizar a computadora. Esta tarefa pódese realizar por medio das diferentes linguaxes de programación, e implica o coñecemento do conxunto de instrucións e regras do mesmo.

Unha **linguaxe de programación** é o conxunto de símbolos e regras (sintácticas e semánticas) para construír ou redactar programas. É unha linguaxe formal para especificar algoritmos que poidan ser executados nunha computadora ou sistema informático. O algoritmo escrito nunha linguaxe de programación é o **código fonte**.

Coñecer unha linguaxe de programación implica coñecer o conxunto de instrucións coas que traballa. Independentemente da linguaxe de programación elixida, existen catro categorías básicas de instrucións:

- *Entrada/Saída*: permiten transferir datos entre diferentes periféricos de entrada e/ou saída e a memoria central da computadora.
- *Aritmético-lóxicas*: executan operacións aritméticas e lóxicas.
- *Selectivas*: permiten seleccionar diferentes accións dependendo de determinadas condicións.
- *Repetitivas*: repiten un número determinado de veces instrucións ou secuencias de instrucións.

Cando se desexa executar un programa nunha computadora, hai que ter en conta que esta unicamente será capaz de interpretar, e polo tanto executar, aqueles programas que estean escritos en linguaxe máquina, que depende do procesador da computadora. Por iso, pode haber incompatibilidades entre programas escritos en código de diferentes máquinas. Esta linguaxe máquina baséase en secuencias de 0s e 1s, e está moi próxima á computadora, dado que a información está codificada en sistema binario, pero é difícil de comprender e programar para as persoas.

Para facilitar a tarefa de programación xorden as linguaxes de alto nivel, independentes dunha computadora específica.

Daranse a continuación unha serie de conceptos básicos acerca dos diferentes tipos de linguaxes de programación.

## 1.4 Linguaxes máquina e ensamblador

A **linguaxe máquina** é a linguaxe nativa dunha computadora. A súa estrutura está completamente adaptada aos circuítos do procesador, o cal pode interpretala directamente, e non son necesarias as transformacións previas do código que si requiren as linguaxes de alto nivel. O código execútase de xeito moi eficiente.

As instrucións da linguaxe máquina son secuencias binarias que determinan unha operación específica. Cada instrución ten un campo onde se indica o código de operación da mesma, así como as direccións de memoria que interveñen. Son instrucións de máquina ou **código máquina**.

***Exemplo***: programa en linguaxe máquina (Figura 1.1).

```
11001010 00010111 11010101
00111010 01000110 10011101
```

Figura 1.1: Exemplo de programa en linguaxe máquina

A gran vantaxe da linguaxe máquina é que permite cargar un programa en memoria sen necesidade de traducilo, polo que a velocidade de execución é maior. Porén, posúe unha serie de limitacións, como a pouca versatilidade para escribir as instrucións (formato ríxido para a posición dos campos que forman a instrución), dificultade e lentitude na codificación, pouca fiabilidade, imposibilidade de engadir comentarios que fagan o código máis lexible, ou gran dependencia do procesador.

Para poder solucionar as dificultades que supón a programación en linguaxe máquina apareceron as **linguaxes simbólicas**, que prescinden das instrucións binarias e empregan nomes simbólicos para codificar instrucións e direccións de memoria, e as **linguaxes de alto nivel**, máis achegadas á linguaxe humana.

A **linguaxe ensamblador** é unha linguaxe simbólica de baixo nivel, cuxo desenvolvemento comezou arredor de 1950. É máis doada de programar que a linguaxe máquina, e depende tamén do procesador da computadora. Posúe un conxunto de instrucións para representar os códigos de operación denominadas **nemónicos** ou **nemotécnicos**, e que substitúen ás instrucións binarias. Algunhas delas son RES para a resta, ou INC para o incremento.

***Exemplo***: programa en ensamblador (Programa 1.2).

Programa 1.2: Programa en ensamblador que imprime ”Ola mundo” por pantalla

```
.model small
.stack
.data
   saudo   db "Ola mundo", "$"

.code

main  proc              ;Inicia proceso
   mov   ax,seg saudo   ;hmm seg
   mov   ds,ax          ;ds = ax = saudo

   mov   ah,09          ;Function(print string)
   lea   dx,saudo       ;DX = String terminated by "$"
   int   21h            ;Interruptions DOS Functions

;mensaje en pantalla

   mov   ax,4c00h       ;Function (Quit with exit code (EXIT))
   int   21h            ;Interruption DOS Functions

main  endp              ;Termina proceso
end main
```

Un programa en linguaxe ensamblador non se pode executar directamente na computadora, senón que é necesaria a súa tradución á linguaxe da máquina, para que esta o poida executar. O programa orixinal denomínase **código fonte**, mentres que o programa traducido a código máquina é o **programa obxecto**. O **tradutor** de ensamblador encárgase de traducir o código fonte a código máquina (Figura 1.2).

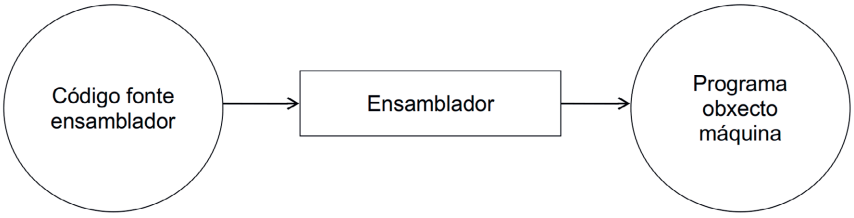



Figura 1.2: Proceso de tradución do código fonte en ensamblador ao código máquina

Ademais da dependencia do procesador da máquina, o ensamblador posúe outros inconvenientes, dado que a programación en linguaxe ensamblador esixe coñecer todas as técnicas de formación e a estrutura física da máquina, e proporciona pouca versatilidade para crear instrucións propias a quen programa.

## 1.5 Linguaxes de alto nivel

As **linguaxes de alto nivel** xorden coa finalidade de facilitar as tarefas de programación, mediante un conxunto de regras e instrucións que sexan más próximas a eles, aínda que non comprensibles directamente pola computadora. Estas linguaxes son independentes da máquina, polo que os programas escritos nestas linguaxes son portables a outras computadoras de forma sinxela.

A aprendizaxe dunha linguaxe de alto nivel é moito máis doada que a da linguaxe máquina ou ensamblador, e proporciona un gran número de instrucións potentes para programar tarefas. Unha única instrución en linguaxe de alto nivel xera varias instrucións en linguaxe máquina.

Centrarémonos neste libro nos fundamentos da programación procedimental en C/C++ (non se verá, por tanto, a parte correspondente á programación orientada a obxectos).

***Exemplo***: código en linguaxe C++ correspondente ao ensamblador do Programa 1.2 (Programa 1.3).

Programa 1.3: Programa en C++ que imprime ”Ola mundo” por pantalla

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

int main()
{
    cout << "Ola mundo" << endl;
    return 0;
}
```

A escritura dun programa nunha linguaxe de alto nivel emprega regras sintácticas parecidas ás das linguaxes humanas. Ademais, a súa modificación é máis sinxela. Porén, tamén posúen desvantaxes: non se aproveitan os recursos internos das computadoras tanto como co ensamblador, maior ocupación de memoria ou necesidade de tradución a linguaxe máquina (aumenta o tempo de posta a punto, así como o tempo de execución).

Resulta practicamente imposible realizar un algoritmo complexo en ensamblador, como consecuencia da simplicidade das súas instrucións, polo que requiriría un programa de tamaño considerable. Pola contra, nas linguaxes de alto nivel ese programa sería máis simple e manexable, e a súa tradución a linguaxe máquina é inmediata, por ser realizada co compilador.

Na actualidade, existen numerosas linguaxes de alto nivel, entre as que se poden citar C/C++, Python, Java, LISP, Fortran ou Matlab/Octave. Existen ademais importantes diferenzas de nivel entre eles; por exemplo, é moito máis complicado facer o mesmo en C/C++ ou Java, que en Python ou Matlab, que son de moito maior nivel. A cambio, estes últimos producen, en xeral, programas executables que aproveitan menos os recursos computacionais do ordenador.

## 1.6 Compilación vs. interpretación

Como xa se indicou, un programa escrito nunha linguaxe de alto nivel non pode ser executado directamente pola computadora. É necesaria unha tradución previa a linguaxe máquina. Para iso emprégase un **tradutor**, que se pode definir como un programa que traduce o programa orixinal (**fonte**) nun programa noutra linguaxe (**obxecto**). O tradutor realiza un cambio de representación dun algoritmo codificado nunha linguaxe a outra linguaxe.

Existen dous tipos de tradutores:

- **Compiladores**: son programas que traducen o programa fonte ao programa obxecto, este último en código máquina. A tradución realízase nunha única operación, denominada **compilación**, na que se

traducen todas as instrucións do programa nun único bloque, de forma conxunta. Como resultado, obtense un **programa executable**, depurado e sen erros, que se pode executar directamente na computadora. Ademais, este programa pódese levar a outra computadora con igual *hardware* e executalo aínda que neste segundo ordenador non estea instalado o compilador. En caso de que se desexe executar de novo, non é necesaria unha segunda compilación, a non ser que se fagan modificacións no código, en cuxo caso si será necesaria. Entre as linguaxes compiladas pódese citar C/C++, Java ou Go.

- **Intérpretes**: son tradutores que parten dun programa fonte, e en lugar de traducir en bloque todo o programa a código máquina, van traducindo e executando unha instrución de cada vez. Selecciónase cada sentenza, tradúcese a linguaxe máquina, detense a tradución, execútase a instrución, selecciónase a seguinte sentenza, e así sucesivamente ata finalizar o programa. A vantaxe destes programas é que se poden ir corrixindo erros a medida que se executa o programa, pero o proceso de tradución é pouco eficiente, porque cada vez que se executa o programa hai que volvelo traducir. Porén, un programa executable feito polo intérprete non pode ser executado se non está instalado o intérprete no ordenador. Como linguaxes interpretadas podemos nomear Python, Ruby e Javascript.

## 1.7 Paradigmas de programación: imperativa, lóxica e funcional

Un **paradigma de programación** é un estilo de desenvolvemento de programas, un modelo para a xeración de código que dá solución a determinados problemas e atinxe ao proceso completo de desenvolvemento de software. A partir do tipo de instrucións que permiten implementar, as linguaxes de programación encádranse nun ou varios paradigmas, dado que xeran códigos que están estruturados de xeito distinto.

Os paradigmas de programación son:

- **Programación imperativa (procedemental)**: é o concepto clásico; no código fonte defínense claramente os pasos que se deben executar e a orde dos mesmos. Séguese o mesmo método que nas linguaxes máquina. As instrucións execútanse unha a unha, de principio a fin, de xeito secuencial, sempre e cando non existan sentenzas de salto. Divídese o programa en partes máis pequenas ou subprogramas (subrutinas, funcións, procedementos) que se invocan entre si. Como exemplo podemos citar C, Pascal, Modula,...
- **Programación orientada a obxectos**: emprégase como abstracción o concepto de clase. Un programa está formado por un conxunto de obxectos, instancias dunha clase. Levan a cabo accións e comunícanse con outros obxectos utilizando mensaxes. Deste xeito, constrúense obxectos compostos de datos e operacións para manipulalos. As linguaxes orientadas a obxectos dispoñen de bibliotecas de obxectos. Ademais, quen programa pode definir obxectos adicionais, así como os seus datos e operacións. Como exemplo están linguaxes como C++, Java, ...
- **Programación declarativa (lóxica)**: neste enfoque, en lugar de buscar un algoritmo que solucione o problema, débese describir, e utilizar o razoamento lóxico (fundamentado na dedución) para responder ás cuestións formuladas. Baséase na representación do coñecemento e lóxica de predicados. Como exemplo están Prolog ou SQL.
- **Programación funcional**: fundamentada no emprego e definición de funcións que se invocan de forma recorrente. Cada programa é un conxunto de funcións, cada unha das cales é unha caixa negra que recibe unha entrada e produce unha saída. Ademais, pódese determinar a interacción entre funcións mediante condicións e recursividade. Exemplos de linguaxes deste tipo son LISP o Haskell.
- **Programación guiada por eventos**: de aparición relativamente recente, nela o fluxo do programa principal, considerado como un lazo indefinido, está determinado por sucesos externos, como pode ser unha acción nosa ou de quen programa. Cando se produce un evento, o control do programa

pasa a executar o procedemento asociado ao evento e sae do lazo indefinido. Pódense citar como linguaxes deste tipo Java ou Visual Basic.

- **Programación concurrente**: permite realizar múltiples tarefas ao mesmo tempo. Existen procesos cooperativos que se executan de forma independente, pero poden compartir información. Cando un programa se executa nunha computadora de xeito concorrente, descomponse en procesos (**febras, fíos de execución**), que se poden executar á vez. Entre as linguaxes que admiten programación concorrente atópanse Java ou Pascal Concorrente.

# Capítulo 2

# Metodoloxía da programación

O proceso de desenvolvemento de *software* adoita ser un proceso laborioso e complexo no que hai que considerar diferentes aspectos á vez. Resulta recomendable seguir uns determinados pasos para acadar o obxectivo final con éxito.

A metodoloxía da programación consiste nunha colección de regras, métodos e principios que permiten o desenvolvemento sistemático de programas para resolver problemas algorítmicos. Inclúe os seguintes pasos:

- Especificación.
- Deseño.
- Codificación e proba.
- Compilación e execución.
- Documentación e mantemento.

Nos seguintes apartados daranse os conceptos fundamentais de cada un destes pasos.

## 2.1 Especificación de algoritmos

Cando se desexa implementar un algoritmo para resolver un problema completo, o algoritmo debe estar coidadosamente especificado e deseñado para cubrir calquera posible evento que poida suceder. Isto significa que, cando se programa un algoritmo, é necesario contemplar de forma rigorosa cada unha das situacións que se poden presentar.

Un problema pode resolverse mediante diferentes algoritmos válidos que poden realizar a mesma tarefa, aínda que utilizando diferentes recursos computacionais (tempo, espazo,...). A pesar das diferenzas, todos eles poden ter a mesma especificación.

Cando se especifica un algoritmo, estase a contestar a pregunta “que fai?”. Non se detallará o modo en que se resolve o problema, senón que se considerará o algoritmo como unha caixa negra, que recibe unhas entradas e produce uns resultados.

A especificación debe ter en conta: 1) quen vai empregar o programa, e detallar as súas obrigas á hora de invocar o algoritmo e o resultado en caso de éxito; 2) que vai programar, que definirá os requirimentos necesarios para a correcta implementación deste algoritmo.

Unha boa especificación debe ser *precisa* (resposta a calquera pregunta sobre o problema), *breve* (máis breve que o código que especifica), e estar escrita con *claridade* (transmisión concisa da idea).

Para poder especificar de forma exacta un algoritmo pódese empregar unha **linguaxe de especificación**, definida como unha linguaxe formal ou semi-formal que permite construír un modelo do algoritmo.

Emprégase xeralmente durante as fases de análise e deseño.

Decotío, estas especificacións non son interpretables ni executables, aínda que existen algunhas contornas de programación que permiten xerar o código correspondente a partir das mesmas.

A partir dunha linguaxe de especificación (coma o **pseudocódigo**, do que falaremos máis adiante), pódese obter o código en diferentes linguaxes de alto nivel, como en C++ ou Python (Figura 2.1).

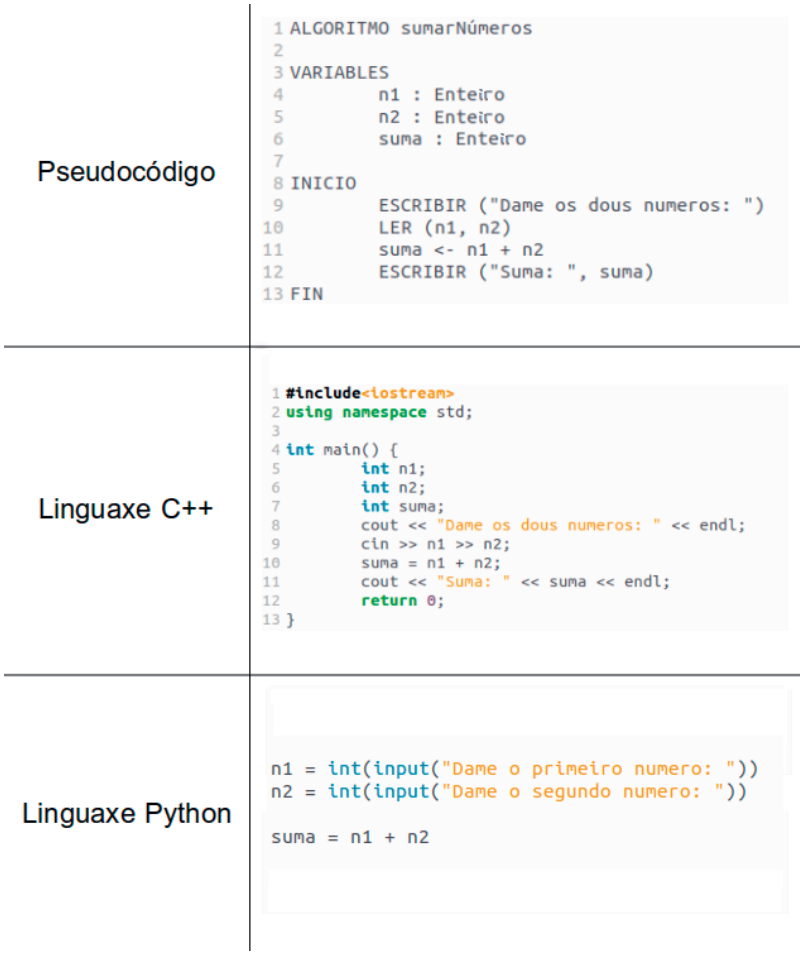



Figura 2.1: Paso de pseudocódigo a C++ e Python

Finalmente, para poder realizar as especificacións técnicas dun algoritmo cómpre identificar primeiro os seus requisitos, isto é, o conxunto de funcións e prestacións que debe incluír para resolver o problema.

## 2.2 Deseño de algoritmos

Unha vez realizada a análise e obtida a especificación do algoritmo, e antes de pasar á súa implementación, é necesario deseñar como se vai construír para resolver o problema proposto. Para iso, contéstase á pregunta “como o fai?”.

No **deseño** determinaranse tanto a estrutura que terá o programa como os datos cos que vai traballar. Ademais, podemos utilizar diferentes modelos de deseño, isto é, diferentes formas de enfrontar o problema e atopar a solución. Existen varias técnicas de deseño de algoritmos, entre as que se poden citar as técnicas de **divide e vencerás** ou a de **volta atrás**.

Para todos eles podemos empregar diferentes ferramentas para representar algoritmos. As dúas máis utilizadas son o **pseudocódigo** e os **diagramas de fluxo**.

### 2.2.1 Pseudocódigo

É unha das ferramentas máis empregadas para o deseño de algoritmos. Pódese definir como unha linguaxe de especificacións de algoritmos, pero non é unha linguaxe de programación real. Consiste nunha serie de pasos detallados que conducen á resolución dun problema determinado.

Mediante pseudocódigo, as instrucións escríbense en palabras similares a calquera idioma (inglés, español,...), o que facilita tanto a escritura como a comprensión dos programas. Permite tamén introducir **comentarios**, é dicir, notas aclaratorias con respecto ás instrucións, pero que no son considerados como tales: a efectos da computadora, son liñas baleiras e no se procesan. Non existen regras estándar de sintaxe, aínda que a meirande parte das persoas que programan emprega regras de escritura similares.

***Exemplo***: pseudocódigo correspondente ao algoritmo que calcula o menor de dous números introducidos por teclado. É necesario realizar o sangrado de liñas (Programa 2.1).

Programa 2.1: Pseudocódigo correspondente ao algoritmo para calcular o menor de dous números

```
ALGORITMO calcularMenor

VARIABLES
    n1 : Enteiro
    n2 : Enteiro

INICIO
    ESCRIBIR ("Introduce dous números: ")
    LER (n1, n2)

    SE n1 = n2 ENTON
        ESCRIBIR ("Os números son iguais")
    SE_NON
        SE n1 < n2 ENTON
            ESCRIBIR ("Menor: ", n1)
        SE_NON
            ESCRIBIR ("Menor: ", n2)
        FIN_SE
    FIN_SE
FIN
```

### 2.2.2 Diagramas de fluxo

Os **diagramas de fluxo** permiten representar dun xeito visual as operacións necesarias para resolver un problema, e a orde en que deben realizarse. Incorporan unha serie de símbolos especiais, incluíndo de forma secuencial os pasos que se deben realizar para solucionar o problema. Facilitan a comprensión de problemas non demasiado complicados, realízanse antes de comezar a implementación do código fonte, e facilitan a comunicación entre persoas que emprean o código e as que o programan. Son moi intuitivos, sinxelos de ler e concisos.

Para crear diagramas de fluxo emprégase un conxunto de símbolos e normas de construción. Haberá que considerar:

- Escritura de arriba a abaixo, e de esquerda a dereita.
- Todos os símbolos deben levar no seu interior información acerca da súa función exacta.
- Unicamente os elementos de decisión entre opcións poderán ter máis dunha saída.
- As liñas non deben cruzarse.

A Figura 2.2 amosa unha listaxe dos símbolos utilizados.

| | |
|---|---|
|  | *Entrada/Saída*: representa lectura de datos dende dispositivo externo (teclado) ou saída cara ao dispositivo externo (pantalla) |
|  | *Terminal*: representa comezo ou fin dun programa |
|  | *Condición*: bifurcación do fluxo de instrucións. Controlada por condicións. Pode haber tamén de *condición múltiple* |
|  | *Proceso:* calquera operación que se leve a cabo cos datos |
|  | *Conector:* conecta un fragmento do diagrama de fluxo con outro na mesma páxina |
|  | *Conector*: conecta un fragmento do diagrama de fluxo con outro en distinta páxina |
|  | *Saída impresa*: indica a presentación de resultados de forma impresa |
|  | *Subrutina*: indica chamada a subrutina ou módulo independente |
|  | *Sentido do fluxo*: indica orde de execución de cada paso do algoritmo |

Figura 2.2: Simboloxía utilizada nos diagramas de fluxo

**Proceso**

Permite realizar operacións cos datos do programa (Figura 2.3).

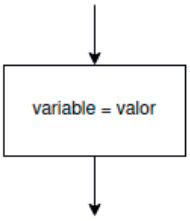



Figura 2.3: Diagrama de fluxo: proceso (asignación)

### Entrada de datos

Permite a entrada dos valores de entrada dende un dispositivo externo (Figura 2.4).

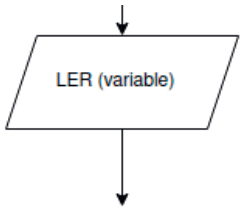



Figura 2.4: Diagrama de fluxo: entrada de datos

### Saída de datos

Permite a saída dos valores procesados do programa de entrada cara un dispositivo externo (Figura 2.5).

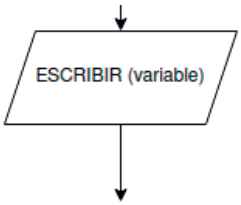



Figura 2.5: Diagrama de fluxo: saída de datos

### Condición

Permite a bifurcación do fluxo en función do cumprimento de determinadas condicións. Pode haber diferentes tipos:

- Condición simple (Figura 2.6).

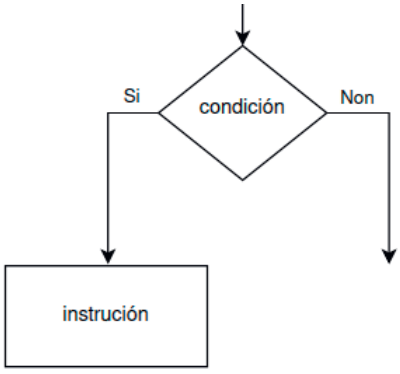



Figura 2.6: Diagrama de fluxo: condición simple

- Condición dobre (Figura 2.7).

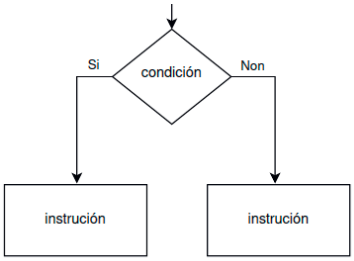



Figura 2.7: Diagrama de fluxo: condición dobre

- Múltiples condicións (Figura 2.8).

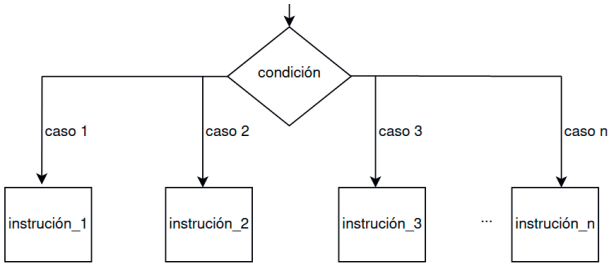



Figura 2.8: Diagrama de fluxo: múltiples condicións

**Repetición**

Empregando liñas de sentido de fluxo e condicións, podemos repetir determinadas operacións dentro do noso programa, isto é, podemos incluír bucles ou lazos. Existen diversas opcións:

- Avaliar a condición ao principio e repetir as operacións sempre que se verifique a devandita condición (Figura 2.9).

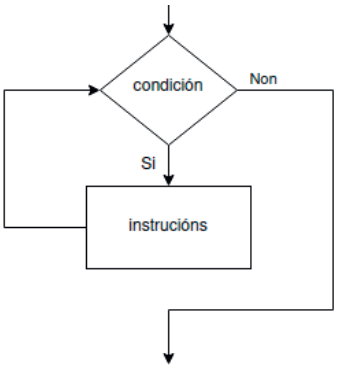



Figura 2.9: Diagrama de fluxo: repetición con avaliación inicial de condición

- Avaliar a condición ao final e repetir as operacións sempre que se verifique a devandita condición (Figura 2.10).

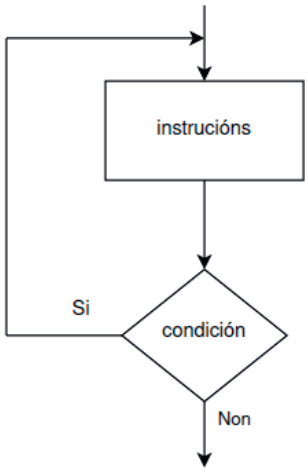



Figura 2.10: Diagrama de fluxo: repetición con avaliación final de condición

- Repetir un número determinado de veces (Figura 2.11).

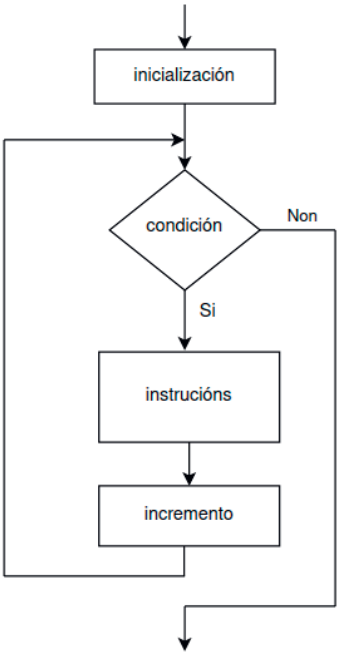



Figura 2.11: Diagrama de fluxo: repetición dun número determinado de veces

Poderemos empregar indistintamente as dúas primeiras, e deixaremos esta terceira opción para cando coñezamos o número de veces que queremos que se repitan as operacións.

***Exemplo***: diagrama de fluxo correspondente ao algoritmo que calcula o menor de dous números introducidos por teclado (Figura 2.12).

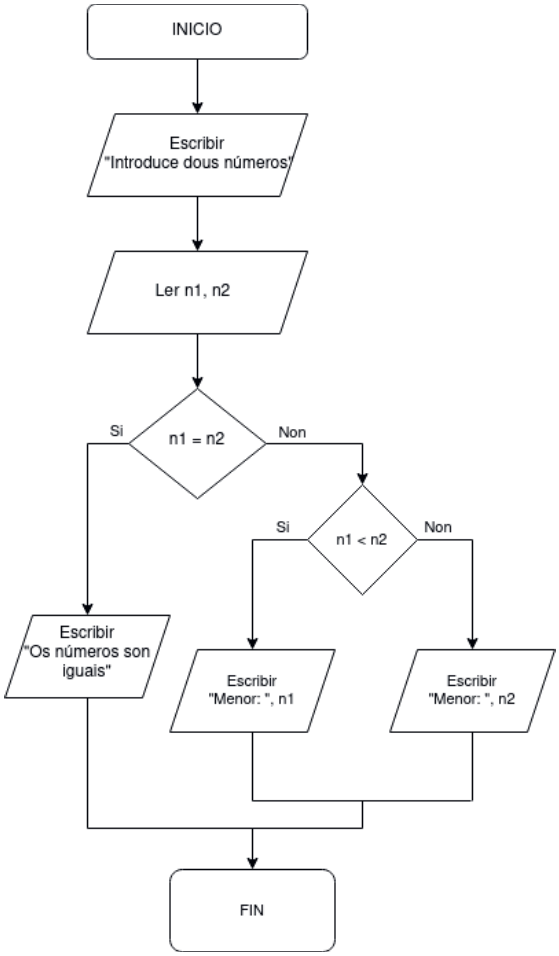



Figura 2.12: Diagrama de fluxo correspondente ao algoritmo para calcular o menor de números

## 2.3 Codificación e probas

A **codificación** consiste na escritura do código correspondente ao algoritmo que se desexa resolver. Hai que ter en conta as consideracións realizadas nas especificacións e no deseño. Pódese realizar en varias etapas. Inicialmente, e a partir do diagrama de fluxo, pódese obter o pseudocódigo correspondente, e escribir o programa nunha linguaxe de programación determinada.

O deseño dun algoritmo é previo á súa codificación. É independente da linguaxe de programación utilizada. Por iso, unha vez obtida a representación dun algoritmo en pseudocódigo, implicará a mesma dificultade o seu paso a unha ou outra linguaxe de programación.

***Exemplo***: código na linguaxe C++ correspondente ao algoritmo que calcula o menor de dous números introducidos por teclado (Programa 2.2).

Programa 2.2: Código en C++ correspondente ao algoritmo para calcular o menor de dous números

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

int main() {
    int n1;
    int n2;
    cout << "Introduce dous numeros" << endl;
    cin >> n1 >> n2;
    if (n1 == n2) {
        cout << "Os numeros son iguais" << endl;
    }
    else {
        if (n1 < n2) {
            cout << "Menor: " << n1 << endl;
        }
        else {
            cout << "Menor: " << n2 << endl;
        }
    }
    return 0;
}
```

Posteriormente, procédese a probar o algoritmo implementado, o que permitirá localizar erros posibles e probables. A **proba** é unha das fases máis importantes no desenvolvemento de calquera algoritmo. Non se considerará que este é correcto ata que non se teñan realizado probas sobre o mesmo, e sexan contempladas todas as posibles situacións e os resultados da execución asociados.

Para poder verificar os erros é necesario dispor dunha ampla variedade de **datos de test** ou **probas**, e ter en conta a entrada tanto de valores normais como de valores extremos que comproben os límites do programa e/ou aspectos especiais.

Coas probas podemos determinar se un programa é incorrecto, en caso de que non se produzan os resultados esperados para un conxunto de probas. Porén, non se poden utilizar para comprobar se o programa é correcto, dado que é practicamente imposible verificar todas as posibles entradas con probas. Para esta tarefa terán que utilizarse técnicas de **verificación formal**.

O éxito na etapa de probas virá determinado por un bo deseño do algoritmo, unha implementación correcta, e un xogo de probas adecuado.

## 2.4 Compilación e execución

Cando o **código fonte** do programa xa está escrito, independentemente da linguaxe de programación de alto nivel elixida, e tal como vimos, é necesario traducilo a unha linguaxe comprensible pola computadora. Este proceso denomínase **compilación**; durante ela execútase o programa cos datos de proba, determinando se o programa contén erros. En caso de que aparezan, será necesario corrixilos antes de volver realizar a compilación. Finalizado o proceso, obtense un **programa obxecto**, que non se pode executar directamente sobre a computadora. A continuación, realízase un **enlazado** do programa obxecto coas funcións que hai nas bibliotecas do compilador, para que poida comunicarse directamente co sistema operativo, tradúcese o programa obxecto a código máquina, e xérase finalmente un **programa executable** (Figura 2.13).

O proceso polo que se detectan e corrixen os erros nun programa denomínase **depuración**.

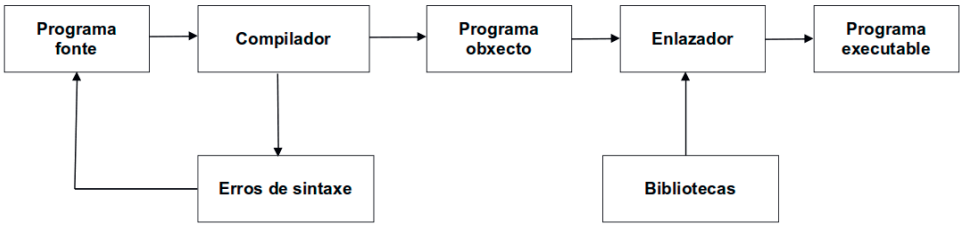



Figura 2.13: Proceso de compilación dun código fonte

## 2.5 Documentación e mantemento

O código correspondente a un programa escríbese unha única vez, aínda que se pode executar un número indefinido de veces. Por iso, é importante elaborar unha boa documentación do programa en todas as súas etapas de desenvolvemento, dende a especificación ata as probas, que permitan unha fácil comprensión e mantemento posterior por calquera persoa. Programas que posúen unha documentación pobre son tamén máis difíciles de depurar e modificar.

A documentación dun programa pode ser:

- *Interna*: incluída dentro do código fonte, xeralmente en forma de comentarios, isto é, liñas de texto ignoradas pola computadora.
- *Externa*: inclúe análise e especificación do algoritmo, diagramas e aspectos de deseño, cuestións importantes de implementación, probas realizadas, e documentación técnica do deseño interno, así como manuais de uso.

Con respecto ao mantemento, pódese definir como a modificación posterior do programa, coa finalidade de corrixir defectos, mellorar o seu rendemento, adaptalo a novas circunstancias,...

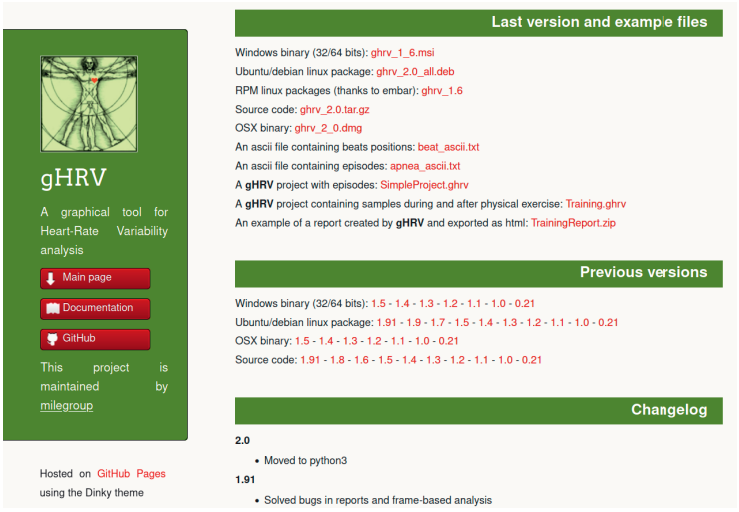



Figura 2.14: Exemplo de programa coas súas diferentes versións por mantemento do mesmo

Cando se realiza o mantemento dun programa, toda a documentación debe ser actualizada. Cada programa adoita denominarse con dous números, da forma x.y (aínda que poden ser 3, da forma x.y.z).

Cambios importantes na implementación do *software* darán lugar a un cambio no primeiro díxito, mentres que pequenas modificacións, como a inclusión de algunhas funcionalidades, farán que varíe unicamente o segundo díxito das mesmas (Figura 2.14).

## 2.6 Exercicios propostos

1. Que pasos son necesarios para construír un algoritmo para calcular a área dun triángulo e amosar o resultado?
2. Que pasos son necesarios para construír un algoritmo que verifique se un número é par?
3. Representar, mediante o correspondente diagrama de fluxo, un algoritmo que calcule a coordenada y dun punto dunha función lineal, dadas a pendente, a intersección co eixe de ordenadas, e a coordenada x do punto. Pódese empregar a seguinte ecuación:

$$y = pendente * x + interseccion$$

4. Realizar o diagrama de fluxo dun algoritmo que calcule a media de 3 notas dun estudante, introducidas por teclado.
5. Deseñar o diagrama de fluxo dun programa que lea por teclado unha temperatura en grados Celsius e a transforme en Fahrenheit e Kelvin, amosando estes valores por pantalla.
6. Facer o diagrama de fluxo necesario para calcular o dobre dun número enteiro n, introducido por teclado, se este é menor que 100.

# Capítulo 3

# Variables e instrucións

## 3.1 Estrutura dun programa

Un **programa** é a representación dun algoritmo mediante unha linguaxe que pode ser executada nunha computadora, un conxunto de instrucións que realizan unha tarefa. Vimos tamén diferentes ferramentas de representación de algoritmos, como son o pseucocódigo e os diagramas de fluxo.

Cando se desexa desenvolver un programa, cómpre analizar con precisión que entradas serán necesarias, as saídas que se producirán tras a súa execución, e os algoritmos requiridos para resolver o problema concreto presentado.

Dentro dun programa pódense diferenciar tres partes:

- **Entrada de datos**: instrucións que toman datos dun dispositivo (teclado, dispositivo de memoria,...). Almacénanse na memoria central para que poidan ser procesados. Son operacións de **lectura**.
- **Proceso ou algoritmo**: instrucións que modifican os datos de entrada e producen como resultado os datos finais. Son, por tanto, os códigos que transforman os datos de entrada en resultados finais.
- **Saída de datos**: instrucións que toman os datos finais da memoria central. Envíanse a un dispositivo (impresora, monitor,...). Son operacións de **escritura**.

Todas as instrucións deben ser escritas na mesma orde na que deben executarse, de forma secuencial. Isto non significa que obrigatoriamente a execución deba ser sempre así. É posible empregar instrucións de bifurcación, decisión ou comparación para interromper a execución secuencial dun programa ou algoritmo. Neste caso, o programa denomínase **non lineal**, mentres que nunha execución completamente secuencial estamos ante un **programa lineal**.

Xa temos visto a estrutura xeral dun algoritmo:

- **Cabeceira**:
  - Nome do algoritmo precedido da palabra ALGORITMO.
- **Corpo**: dous bloques:
  - *Declaracións*: constantes, variables, tipos de datos definidos no programa.
  - *Instrucións*:
    - Inicio/fin.
    - Entrada de datos.
    - Saída de datos.
    - Asignación.
    - Bifurcación.
    - Comentarios.

***Exemplo***: algoritmo en pseudocódigo para calcular o dobre dun número enteiro introducido por teclado (Programa 3.1).

Programa 3.1: Algoritmo para calcular o dobre dun valor enteiro

```
ALGORITMO SUMAR
VARIABLE
    num : Enteiro
INICIO
    ESCRIBIR ("Introduce un número enteiro: ")
    LER (num)
    ESCRIBIR ("Dobre de ", num, ": ", num * 2)
FIN
```

De forma similar, cando traballamos cunha linguaxe de programación específica, os programas teñen unha estrutura xeral determinada.

A estrutura dun programa en C++ será:

- **Cabeceira**:
  - Arquivos de cabeceira e cabeceiras de funcións.
- **Corpo**: dous bloques:
  - *Declaracións*: constantes, variables, tipos de datos definidos no programa.
  - *Instrucións*:
    - Englobadas en funcións. Todos os programas en C++ conteñen como mínimo unha función principal (`main( )`), aínda que xeralmente inclúen outras funcións definidas polo usuario.

Máis especificamente, a estrutura virá dada por:

- **Directivas de preprocesador**: son instrucións que se dan ao compilador antes de compilar o programa. Comezan con # e o seu uso máis habitual en C++ é a súa inclusión en arquivos de cabeceira. As máis habituais son:
  - `#include` (ler arquivo fonte a continuación e inserilo na posición da directiva). Hai un gran número de arquivos de cabeceira estándar. Todos eles teñen extensión .h e inclúen código escrito en linguaxe C++. Algúns exemplos son:

```
#include <iostream>
#include <math>
#include <cstring>
#include <ctime>
```

  - `#define` (defínense datos ou operacións para o programa).
- **Declaracións globais**: funcións definidas polo usuario ou variables que son comúns a todas as funcións do programa.
  - Sitúanse antes da función principal `main( )`.
- **Función principal**: denominada como `main( )`. É o punto de entrada ao programa. A súa estrutura vén dada por:

```
int main()
{
    ...
}
```

  - As sentenzas entre chaves denomínanse bloques.
  - Dúas funcións `main ( )` no programa producen erro.

- **Funcións definidas polo usuario**: un programa en C++ é unha colección de funcións, e todas elas conteñen unha o máis sentenzas destinadas a realizar unha tarefa específica, para a que son creadas. Posúen nome e lista de parámetros opcionais. Devolven un tipo de dato concreto e invócanse polo seu nome e os parámetros opcionais.
  - Pódese asignar calquera nome a unha función, pero é recomendable que describa o seu propósito.
  - Toda función debe ser coñecida no programa antes da súa execución. Se se declara ao principio, falamos de **prototipo de función**.
- **Comentarios**: información que non se compila. En C++ pódense escribir de dous xeitos diferentes:

```
// Comentarios dunha soa liña
/* Comentarios de
    varias liñas...*/
```

De forma xeral, un programa en C++ pódese escribir da seguinte maneira (Programa 3.2).

Programa 3.2: Estrutura xeral dun programa en C++

```
# directivas de preprocesador
declaracions globais de funcions e variables

tipo_retorno main (lista parametros)
{
    declaracions de variables locais;
    sentenzas;      // Comentario
    [return...];
}

tipo_retorno1 funcion_1 (lista parametros)
{
    declaracions de variables locais;
    sentenzas;
    [return...];
}
........

tipo_retornoN funcion_N (lista parametros)
{
    declaracions de variables locais;
    sentenzas;
    [return...];
}
```

***Exemplo***: programa en C++ que calcula o dobre dun número introducido por teclado (Programa 3.3).

Programa 3.3: Programa en C++ para calcular o dobre dun valor enteiro

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

int main()
{
    int num;
    cout << "Introduce un numero enteiro: ";
    cin >> num;
    cout << "Dobre de ", num, ": ", num * 2 << endl;
    return 0;
}
```

## 3.2 Palabras reservadas e identificadores

Unha **palabra reservada** en calquera linguaxe de programación é unha palabra cun significado gramatical especial para esa linguaxe, e non pode ser utilizada como un identificador. Cando se emprega esta palabra de xeito diferente ao uso asignado producirá un erro na execución/compilación do código.

***Exemplo***: `int`, `while`, `if`, `return`.

Un **identificador** é un nome co que se designa un elemento dun programa, como unha función, unha variable ou un procedemento. Fai referencia a un lugar da memoria do programa onde se almacena un dato. Está formado por letras e díxitos, pero non todos os nomes de identificadores así formados son válidos. Para construír un identificador hai que ter en conta as seguintes regras:

- Debe ser significativo.
- No pode coincidir con palabras reservadas da linguaxe.
- Aínda que non se limita a súa lonxitude, nalgúns compiladores de C++ debe ter un máximo de 32 caracteres.
- Comezará por carácter alfabético, seguido de máis caracteres alfanuméricos.
- Pode ser en maiúsculas ou minúsculas.

***Exemplo***: dato1, dato_1, Dato1, Dato_1 (correctos), 1_dato, 1-Dato (incorrectos).

## 3.3 Variables, constantes e tipos de datos simples

Cando se executa un programa existen determinados valores que non deben cambiar durante a execución, mentres que outros van ser modificados durante a mesma.

Unha **variable** é un espazo reservado na memoria da computadora, que pode cambiar de contido durante a execución do programa ou algoritmo. Represéntase mediante un identificador, que debe ser **único**, e está asociada a un tipo de dato determinado. Queda identificada por:

- **Nome**: especifica o lugar da memoria onde se almacena.
- **Tipo**: determina o conxunto de valores que pode tomar, así como as operacións que se poden realizar.

Dependendo da linguaxe de programación ou pseudocódigo, existen diferentes tipos de variables: enteiras, reais, carácter,... Se tentamos asignar a unha variable dun tipo un valor doutro tipo, por exemplo, pódese producir un erro. Tampouco deben asignarse como nomes de variables palabras reservadas da linguaxe.

O contido dunha variable pode ser numérico ou alfanumérico. Se é unha mestura de número e letras, considerarase texto. Ademais, aínda que o valor poida ser modificado no tempo, sempre é único, denominándose ao valor que ten nun momento de execución dado o **valor actual**. Perderase toda a información acerca de calquera outro valor que houbera nun momento anterior. Os nomes elixidos para as variables deben ser representativos do dato que representan, e ter relación con el.

***Exemplo***: variables que representan o nome (texto) e a idade (número) dunha persoa, almacenadas en distintas posicións de memoria (Figura 3.1).

Cando se desenvolve un programa ou se deseña un algoritmo será necesario considerar:

- O número de variables necesarias para realizar as tarefas.
- O tipo de dato que pode almacenar cada unha delas.

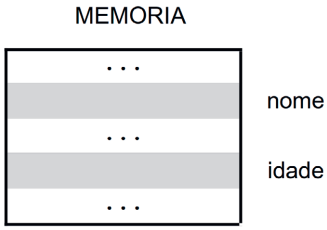



Figura 3.1: Reserva de espazo en memoria para as variables nome e idade

A **declaración** previa de variables, coa conseguinte reserva de espazo en memoria, é obrigatoria na linguaxe de programación C++. Esta declaración farase dentro de cada función, ao principio, ou como variables globais, de ser o caso:

```
tipo1 variable1;
tipo2 variable2;
...........
```

En C++ é posible declarar varias variables do mesmo tipo na mesma liña, separadas por ",".

```
tipo variable1, variable2;
```

***Exemplo***: declaración de variables xénero e idade.

```
char xenero;
int idade;
```

Con respecto ás constantes, ao contrario que as variables, non mudan o seu valor durante a execución do programa ou algoritmo. Polo tanto, pódese definir unha **constante** como un dato que permanece invariable durante a execución do programa.

As constantes poden ser:

- **Literais**: valor de calquera tipo (5, ola).
- **Simbólicas**: cun identificador e valor asignado.

Tamén deben ser declaradas. A declaración farase dentro de cada función, ao principio, ou ben como variable global. É recomendable antepor o modificador `const` para garantir que calquera intento de asignarlle outro valor xerará un erro en tempo de compilación. Tamén se poden declarar empregando a directiva de compilación `#define`:

```
#define CONSTANTE1 valor1

const tipo1 CONSTANTE2 = valor2;
const tipo2 CONSTANTE3 = valor3;
...........
```

En C++ é posible declarar varias constantes do mesmo tipo na mesma liña, separadas por ",".

```
const tipo CONSTANTE1 = valor1, CONSTANTE2 = valor2;
```

***Exemplo***: declaración de constantes xénero, idade e IDADE.

```
# define IDADE 40

const char XENERO = 'F';
const int IDADE_MAXIMA = 50;
```

As variables e as constantes poden almacenar diferentes tipos de datos.

O **tipo de datos** é o conxunto específico de valores dos datos e o conxunto de operacións que actúan sobre eses datos.

Existen dous tipos de datos:

- **Simples** ou básicos, incluídos nas linguaxes de programación.
- **Definidos polo usuario**.

A partir dos tipos de datos simples pódense construír os datos **compostos** ou **agregados** (**estruturados**), definidos como conxuntos de datos simples con relacións definidas entre eles.

Detallaranse a continuación os tipos de datos simples e os definidos polo usuario. Os tipos de datos compostos estudaranse en temas posteriores.

### 3.3.1 Tipos de datos simples

Engloban os datos numéricos, lóxicos, caracteres e cadeas (Táboa 3.1). Veremos a continuación con detalle cada un deles.

Táboa 3.1: Tipos de datos simples

| Tipo de dato | Declaración en C++ |
|---|---|
| **Enteiro** | `int` |
| **Real** | `float (double)` |
| **Lóxico** | `bool` |
| **Carácter** | `char` |
| **Cadea** | `char [tam], string` |

#### Datos numéricos

Son o conxunto de valores numéricos, e poden representarse de dúas formas diferentes:

- **Tipo numérico enteiro**: subconxunto finito de números enteiros, cuxo tamaño depende da linguaxe de codificación e da computadora utilizada. Especificamente, teremos os valores amosados na Táboa 3.2.

Táboa 3.2: Tipos de datos numéricos enteiros

| Tipo de enteiro | Límite inferior | Límite superior |
|---|---|---|
| Enteiro | -32768 | 32767 |
| Enteiro curto | -128 | 127 |
| Enteiro longo | -2147483648 | 2147483647 |

- **Tipo numérico real**: subconxunto de números reais, limitado en tamaño e precisión. Teñen sempre un punto decimal e constan dun número enteiro e unha parte decimal. Poden ser de precisión simple (reais) ou dobre (dobres).

***Exemplo***: datos numéricos en C++:

```
int anos;
float altura;
```

#### Datos lóxicos

Formados polo conxunto de valores VERDADEIRO e FALSO.

Denomínanse tamén *booleanos*, e representan un par de alternativas (si/non, 1/0) para determinadas condicións.

***Exemplo***: datos lóxicos en C++:

```
bool existe;
bool aceso;
```

#### Datos carácter

Neste conxunto atópanse todos os caracteres conocidos pola computadora: unha letra, un número, un símbolo,... Internamente, cada carácter almacénase como un enteiro.

***Exemplo***: datos carácter en C++:

```
char letra;
char grupo;
```

Un carácter está conformado por:

- Os díxitos do 0 ao 9.
- As letras minúsculas e maiúsculas.
- Os caracteres especiais (*,%, −,...).

#### Datos cadea

Son sucesións de caracteres, delimitados por comiñas simples ou dobres, dependendo da linguaxe de programación. Defínese a lonxitude da cadea como o número de caracteres que a forman.

***Exemplo***: datos cadea en C++:

```
char palabra[5];
string frase;
```

### 3.3.2 Tipos de datos definidos no programa

Son tipos de datos definidos por nós, e formados por tipos de datos simples. Entre eles atopamos os enumerados. Permiten definir novos tipos de datos simples, de cardinalidade (n) reducida.

#### Enumerados

Son tipos ordinais (os seus valores pertencen a un conxunto ordenado e finito. Todos teñen predecesor -excepto o primeiro- e sucesor -excepto o último-). Neste tipo de dato non se indica o rango para un tipo existente, senón que se fai unha lista de todos os valores posibles do tipo. Unha **enumeración** é unha lista de valores onde, mentres non se indique un valor particular, o primeiro valor corresponde co valor enteiro 0, o segundo con o 1, e así sucesivamente.

Os enumerados non teñen operadores específicos e melloran a lexibilidade do código. En C++, as constantes represéntanse por identificadores separados por comas e pechados entre chaves. Os valores dun tipo enumerado comezan en 0, a menos que se especifique lo contrario. A sintaxe é a seguinte:

```
enum NomeEnum {IDENTIFICADOR1, IDENTIFICADOR2, .., IDENTIFICADORn};
```

***Exemplo***: enumerado en C++:

```
enum Semaforo {VERMELLO, AMARELO, VERDE};
enum DiasLaborables {LUNS = 1, MARTES, MERCORES, XOVES, VENRES};
```

## 3.4 Instrucións de asignación

O conxunto de instrucións dunha linguaxe depende da propia linguaxe. Porén existen, como xa vimos, unha serie de instrucións independentes, que poden ser empregadas de modo xeral para construír calquera algoritmo en calquera linguaxe. Estas instrucións son: inicio/fin, entrada de datos, saída de datos, asignación e bifurcación.

Veremos a continuación todas estas instrucións, xunto coas expresións e operadores que se poden empregar para escribir un algoritmo.

Unha **instrución de asignación** é aquela que permite asignar un valor determinado ou o resultado da avaliación dunha expresión a unha variable.

Aínda que xa as vimos no Tema 2, recordaremos nesta Sección e nas seguintes a representación de cada instrución no diagrama de fluxo correspondente. Así, a Figura 3.2 amosa a asignación dun valor a unha variable no diagrama de fluxo.

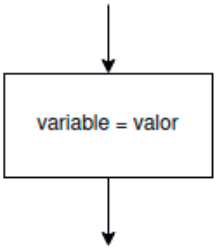



Figura 3.2: Diagrama de fluxo: asignación de valor a unha variable

En C++, unha instrución de asignación para unha variable determinada (previamente declarada) exprésase da seguinte forma:

```
tipo variable;
...........
variable = valor (expresion);
```

O valor `valor` (ou o resultado da expresión) almacénase na posición de memoria reservada para a variable indicada. Pódense asignar tamén os valores no momento no que se declaran as variables (**inicialización**):

```
tipo variable = valor1 (expresion);
```

***Exemplo***: asignación de valores a variables en C++. Pódese ver que os datos tipo carácter asígnanse con comiñas simples `''`. Para as cadeas, teriamos que empregar comiñas dobres `""`.

```
int idade;
float peso, altura;
char xenero;
...........
idade = 35;
peso = 54.5;
altura = 1.61;
xenero = 'F';
```

## 3.5 Expresións aritméticas e lóxicas

A instrución de asignación permite asignar a unha variable un valor ou o resultado da avaliación dunha expresión. Por iso é imprescindible coñecer o significado das expresións en programación.

Defínese unha **expresión** como unha combinación de constantes, variables, operadores,... Toma un valor que se determina considerando os valores das variables e constantes que a forman ao executar as operacións indicadas. Deste modo, pódese dicir que unha expresión está formada por **operadores** e **operandos**.

As expresións clasifícanse en:

- ***Aritméticas***: producen un resultado de tipo numérico.
- ***Relacionais***: producen un resultado de tipo lóxico.
- ***Lóxicas***: producen un resultado de tipo lóxico.
- ***Carácter***: producen un resultado de tipo carácter.

### 3.5.1 Expresións aritméticas

Son similares ás expresións matemáticas, e están formadas por operandos que son constantes e variables numéricas, e por operadores aritméticos. Xeralmente, o resultado da expresión é do mesmo tipo que os operandos.

Non todos os operadores aritméticos existen en todas as linguaxes de programación. A Táboa 3.3 presenta os operadores aritméticos en C++.

Táboa 3.3: Operadores aritméticos en C++

| Operación | Operador |
|---|---|
| Suma | + |
| Resta | - |
| Produto | * |
| División | / |
| Módulo | % |
| Incremento | ++ |
| Decremento | - - |

En C++, existen tamén outros operadores adicionais (Táboa 3.4).

Táboa 3.4: Operadores adicionais en C++

| Operación | Operador |
|---|---|
| a += b: a = a + b | += |
| a -= b: a = a - b | -= |
| a *= b: a = a * b | *= |
| a /= b: a = a / b | /= |
| a %= b: a = a % b | %= |

Cando temos que realizar varias operacións matemáticas nas que a orde na que se executan afecta ao resultado da expresión, é necesario que apliquemos **regras de prioridade** ou **precedencia** sobre as operacións:

1. Avalíanse primeiro as expresións entre parénteses.
2. En caso de haber parénteses aniñados, execútanse primeiro os máis internos.

3. Os operadores teñen a seguinte orde de prioridade:
   (a) Operador ().
   (b) Operadores ++, – –, + e – unarios.
   (c) Operadores *, /, %.
   (d) Operadores +, – .
4. Se hai dous operadores de igual prioridade nunha expresión, esta avalíase de esquerda a dereita (**asociatividade**).

***Exemplo***: avaliación de expresións con operadores con distinta precedencia:

$(9\%2) + 4 = 1 + 4 = 5$

$7 - 9 * 2 + 8/2 = 7 - 18 + 4 = -7$

### 3.5.2 Expresións relacionais e lóxicas

Estas expresións, denominadas tamén ***booleanas***, producen como resultado VERDADEIRO ou FALSO, e fórmanse por combinación de constantes, variables e expresións lóxicas con operadores lóxicos.

Para comparar os valores de dúas expresións empregamos os operadores relacionais (Táboa 3.5).

Táboa 3.5: Operadores relacionais en C++

| Operación | Operador |
|---|---|
| Menor que | < |
| Maior que | > |
| Menor ou igual que | <= |
| Maior ou igual que | >= |
| Igual | == |
| Distinto | != |

Estes operadores poden operar sobre todos os tipos de datos simples. Cando os aplicamos sobre caracteres, a orde vén determinada pola orde en que aparecen no código (xeralmente ASCII) utilizado polo ordenador para representalos.

Se se traballa sobre valores lóxicos, hai que ter en conta que FALSO $<$ VERDADEIRO.

***Exemplo***: avaliación de expresións con operadores relacionais:

“María” == “maria” FALSO

31.9 + 4.5 $<$ 67.2 VERDADEIRO

Táboa 3.6: Operadores lóxicos en C++

| Operación | Operador |
|---|---|
| e | && |
| ou | \|\| |
| non | ! |

Ademais dos operadores relacionais, existe outro tipo de operadores que producen como resultado valores lóxicos, e son os operadores lóxicos (Táboa 3.6).

Os resultados de verdade correspondentes amósanse nas Táboas 3.7, 3.8 e 3.9.

***Exemplo***: avaliación de expresións con operadores lóxicos

!(“María” == “maria”) VERDADEIRO

Táboa 3.7: Táboa de verdade do operador E para dúas expresións a e b (V: verdadeiro, F: Falso)

| a | b | a e b |
|---|---|---|
| V | V | V |
| V | F | F |
| F | V | F |
| F | F | F |

Táboa 3.8: Táboa de verdade do operador Ou para dúas expresións a e b (V: verdadeiro, F: Falso)

| a | b | a ou b |
|---|---|---|
| V | V | V |
| V | F | V |
| F | V | V |
| F | F | F |

Táboa 3.9: Táboa de verdade do operador Non para unha expresión a (V: verdadeiro, F: Falso)

| a | Non a |
|---|---|
| V | F |
| F | V |

(31.9 + 4.5 $<$ 67.2) VERDADEIRO

De forma similar aos operadores aritméticos, hai regras de prioridade para os operadores relacionais e lóxicos. Esta prioridade vén dada da siguiente forma:

1. !
2. $>$, $>=$, $<$, $<=$
3. $==$, $!=$
4. $\&\&$

### 3.5.3 Expresións tipo carácter

Son expresións das que se obtén sempre como resultado un carácter. Non existen operadores de carácter específicos.

***Exemplo***: avaliación de expresións de tipo carácter:

letra = 's' $\rightarrow$ s

## 3.6 Instrucións de Entrada/Saída (E/S)

As **instrucións de entrada** consisten en asignar os valores de entrada necesarios para a execución do programa a variables previamente declaradas, e cuxo espazo xa foi reservado en memoria. Xeralmente, estes datos recóllense desde o teclado, pero poden obterse por medio do rato, dun arquivo,...

A Figura 3.3 amosa a entrada de datos no diagrama de fluxo.

En C++, a sintaxe para as instrucións de entrada vén dada por:

```
tipo variable;
.....
cin >> variable;
```

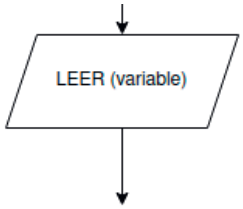



Figura 3.3: Diagrama de fluxo: entrada de datos

Ademais, pódense ler varias variables coa mesma instrución.

```
tipo1 variable1
tipo2 variable2;
..........
cin >> variable1 >> variable2;
```

***Exemplo***: lectura de valores de variables en C++.

```
int idade;
float peso, altura;
char xenero;
..........
cin >> idade;
cin >> peso >> altura;
cin >> xenero;
```

De forma análoga ás instrucións de entrada, existen **instrucións de saída** que permiten escribir os resultados da execución dun programa nunha saída, xeralmente o monitor, aínda que tamén se pode enviar a unha impresora, dirixir a un ficheiro,...

A Figura 3.4 amosa a saída de datos no diagrama de fluxo.

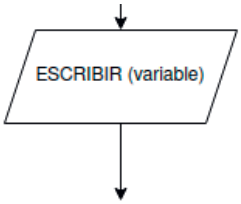



Figura 3.4: Diagrama de fluxo: saída de datos

En C++, a sintaxe para as instrucións de saída vén dada por:

```
tipo variable;
..........
cout << variable;
```

Ademais, pódense escribir varias variables coa mesma instrución.

```
tipo1 variable1
tipo2 variable2;
..........
cout << variable1 << variable2;
```

***Exemplo***: escritura de valores de variables en C++.

```
int idade;
float peso, altura;
```

```
char xenero;
...........
cout << idade;
cout << peso << altura;
cout << xenero;
```

## 3.7 Estruturas de control

Xa estudamos que nun programa as instrucións execútanse sempre de xeito **secuencial**, sempre e cando non se modifique o fluxo de control. Polo tanto, iranse executando a medida que están escritas, e unha detrás doutra.

Se se interrompe o desenvolvemento lineal dun programa e se executa unha instrución de bifurcación, a orde de execución xa non será secuencial, e o programa deixará de ser lineal. Polo tanto, as bifurcacións, denominadas tamén sentenzas de **selección**, **condicionais** ou **alternativas**, poden facer que a execución do programa vaia cara adiante ou cara atrás.

As instrucións de control alternativas permiten seleccionar a seguinte instrución dun programa que será executada, en función dunhas determinadas condicións e de entre varias posibilidades. Para iso poden utilizarse as estruturas de **selección** e de **repetición**.

### 3.7.1 Estruturas de selección

Empréganse para tomar decisións lóxicas, e denomínase tamén **estruturas de decisión**. Permiten controlar a execución do programa en función do valor que tomarán unha ou varias variables, dependendo de determinadas condicións, que serán especificadas en termos de expresións lóxicas. Poden ser simples, dobres, o múltiples.

#### Estrutura condicional simple

Nesta estrutura avalíase unha condición. Se a condición se verifica, isto é, é verdadeira, se executa unha instrución ou conxunto de instrucións. En caso de ser falsa, non se executa nada.

A Figura 3.5 amosa a estrutura condicional simple no diagrama de fluxo.

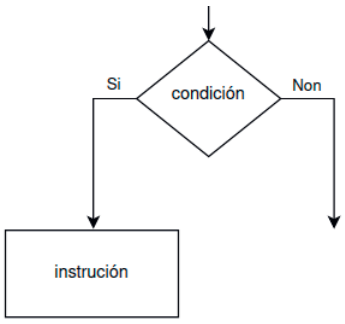



Figura 3.5: Diagrama de fluxo: estrutura condicional simple

En C++, a sintaxe para a estrutura condicional simple vén dada por:

```
if (condicion) {
    instrucion(s);
}
```

Do mesmo xeito que sucede co resto de estruturas repetitivas e selectivas que veremos máis adiante, no caso de levar asociada unha única instrución, as {} son opcionais.

***Exemplo***: estrutura condicional simple en C++.

```
int idade;
..........
if (idade < 18) {
    cout << "Menor de idade" << endl;
}
```

### Estrutura condicional dobre

De forma similar á condicional simple, nesta estrutura avalíase unha condición. Se a condición se verifica, se executa unha instrución ou conxunto de instrucións. En caso de ser falsa, execútase un conxunto diferente de sentenzas.

A Figura 3.6 amosa a estrutura condicional dobre no diagrama de fluxo.

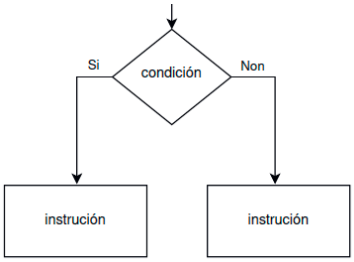



Figura 3.6: Diagrama de fluxo: estrutura condicional dobre

En C++, a sintaxe para a estrutura condicional dobre vén dada por:

```
if (condicion) {
    instrucion(s);
}
else {
    instrucion(s);
}
```

Pódese empregar tamén o **operador condicional**. Este operador ternario en C++ é un xeito compacto de escribir una estrutura condicional `if-else`. A súa sintaxe básica é:

```
condicion ? instrucion1 : instrucion2;
```

Se `condicion` é verdaidera, execútase `instrucion1`; se é falsa, execútase `instrucion2`.

Podemos empregar o operador coma para executar varias instrucións en cada rama do operador ternario.

```
condicion ? (instrucion1.1, instrucion1.2,...) : (instrucion2.1, instrucion2.2,...);
```

***Exemplo***: estrutura condicional dobre en C++.

```
int idade;
...........
if (idade < 18) {
    cout << "Menor de idade" << endl;
}
else {
    cout << "Maior de idade" << endl;
}

// Operador condicional
idade < 18 ? cout << "Menor de idade" << endl : cout << "Maior de idade" << endl;

// Operador condicional con varias instrucións asociadas ao cumprimento da condición
idade < 18 ? (cout << "Menor de idade" << endl, cout << "Quedan " << 18 - idade << "
    anos para ser maior de idade" << endl): (cout << "Maior de idade" << endl, cout <<
    "Tes " << idade - 18 << " anos mais" << endl);
```

### Estrutura condicional múltiple

Cando se programan algoritmos, en moitas ocasións é necesario elixir entre máis de dúas opcións, polo que as estruturas vistas ata o de agora non resolven o problema, e será necesario utilizar estruturas máis complexas, que encadeen unhas condicións con outras. Deste xeito, pódense utilizar estruturas simples ou dobres, ben aniñadas, ben en cascada. Existen varias formas de escritura do algoritmo. A continuación preséntase unha delas:

```
if (condicion1) {
    instrucion(s);
}
else if (condicion2) {
    instrucion(s);
}
else if (condicion3) {
    instrucion(s);
}
...
else {
    instrucion(s);
}
```

Se o número de alternativas que se presentan é elevado, esta estrutura complica o código, empeorando a súa lexibilidade. Por iso, no seu lugar adóitase empregar unha estrutura de elección múltiple.

### Estrutura de selección múltiple

Con esta estrutura podemos seleccionar e avaliar unha expresión, e executar a instrución ou conxunto de instrucións desexadas, elixidas de entre varias opcións posibles.

A Figura 3.7 amosa a estrutura de selección múltiple no diagrama de fluxo. Para cada valor da expresión é posible executar un conxunto de instrucións. Ademais, os valores que toma non teñen que ser nin consecutivos nin únicos. En C++, o valor da expresión debe ser enteiro ou carácter.

En C++, a sintaxe para a estrutura de selección múltiple vén dada por:

```
switch (expresion) {            // expresion debe tomar un valor enteiro (ou carácter)
    case valor1: instrucion(s); // case 'valor1' para carácter
                 [break;]
    case valor2: instrucion(s); // case 'valor2' para carácter
                 [break;]
    ...........
```

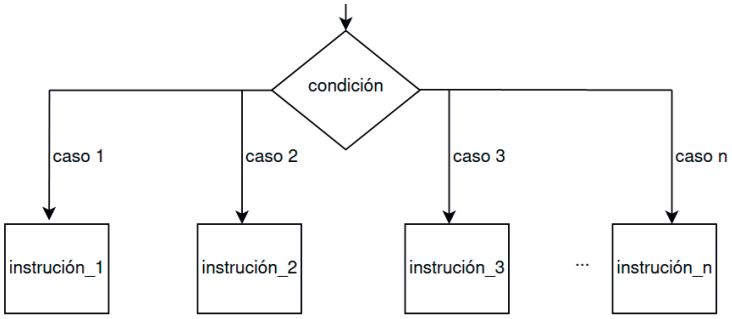



Figura 3.7: Diagrama de fluxo: estrutura de selección múltiple simple

```
    [default] instrucion(s)_default;
}
```

***Exemplo***: estrutura de selección múltiple en C++.

```
int dia;
...........
switch (dia) {
    case 1: cout << "Luns";
            break;
    case 2: cout << "Martes";
            break;
    case 3: cout <<  "Mercores";
            break;
    case 4: cout << "Xoves";
            break;
    case 5: cout << "Venres";
            break;
    default:cout << "Fin de semana";
}
```

### 3.7.2 Estruturas de repetición

En moitas ocasións pode resultar necesario repetir unha instrución ou conxunto delas un determinado número de veces. Cada repetición é unha **iteración**. As estruturas que permiten implementar repeticións coñécense como bucles ou lazos.

A instrucións repetitivas poden ser de tres tipos: `while`, `do/while` e `for`.

#### Estrutura de repetición `while`

Nesta estrutura avalíase unha condición ao principio (bucle controlado pola entrada) e o bucle repítese mentres a condición se verifique. Se a condición non se cumpre, non se executa nada. Así, cando o fluxo dun algoritmo chega a un bucle `while`, podense dar dúas situacións: 1) que a condición se avalíe como falsa, en cuxo caso non se executan as instrucións asociadas, e remata o bucle sen realizar ningunha iteración; e 2) que a condición se avalíe como verdadeira, caso no que se executarán as instrucións do bucle, avaliando novamente con posterioridade a condición. O valor resultante determinará que se produza unha nova iteración ou se remate o bucle. Se a condición se avalía sempre como verdadeira, prodúcese un **bucle infinito**, e o programa non finalizará nunca, o que implica un erro lóxico. Por tanto, para evitar esta situación, terá que haber instrucións dentro do bucle que cambien o valor da condición.

A Figura 3.8 amosa a estrutura de repetición `while` no diagrama de fluxo.

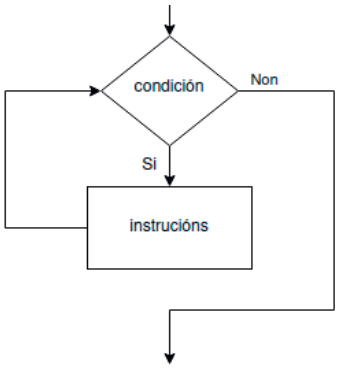



Figura 3.8: Diagrama de fluxo: estrutura de repetición `while`

O número de veces que se executarán as instrucións é descoñecido antes de iniciar o proceso iterativo. Por isto, chámase **iteración indefinida**.

En C++, a sintaxe para a estrutura de repetición `while` vén dada por:

```
while (condicion) {
    instrucion(s);
}
```

***Exemplo***: estrutura de repetición `while` en C++: suma dos números de 0 a 10.

```
int suma = 0;
int numero = 0;
...........
while (numero <= 10) {
    suma += numero;
    numero++;
}
```

### Estrutura de repetición `do/while`

Ao contrario que na estrutura `while`, na estrutura `do/while` a condición avalíase ao final do bucle.

A diferenza principal co bucle `while` é que o bucle `do/while` vai executarse sempre como mínimo unha vez, porque a condición é avaliada ao final, mentres que no bucle `while` pode que non haxa ningunha iteración.

A Figura 3.9 amosa a estrutura de repetición `do/while` no diagrama de fluxo.

En C++, a sintaxe para a estrutura de repetición `do/while` vén dada por:

```
do {
    instrucion(s);
}  while (condicion);
```

***Exemplo***: estrutura de repetición `do/while` en C++: suma dos números de 0 a 10.

```
int suma = 0;
int numero = 0;
...........
do {
    suma += numero;
```

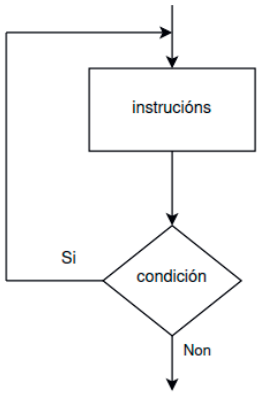



Figura 3.9: Diagrama de fluxo: estrutura de repetición `do/while`

```
    numero++;
} while (numero <= 10);
```

### Estrutura de repetición `for`

Ata o de agora vimos estruturas de repetición nas que a condición de parada do bucle vén dada ao modificar as instrucións do bucle, e non podemos saber por adiantado as veces que se executa. Pero en ocasións sabemos o número de repeticións que desexamos realizar. Neste caso recoméndase usar a estrutura `for`, que se coñece como **iteración definida**.

A Figura 3.10 amosa a estrutura de repetición `for` no diagrama de fluxo.

En C++, a sintaxe para a estrutura de repetición `for` vén dada por:

```
for (valor = valorInicial; condicion; incremento) {
    instrucion(s);
}
```

***Exemplo***: estrutura de repetición `for` en C++: suma dos números de 0 a 10.

```
int suma = 0;
int numero = 0;
...........
for (numero = 0; numero <= 10; numero ++) {
    suma += numero;
}
```

Acabamos de ver que empregamos a estrutura `for` para repetirmos un conxunto de instrucións un número coñecido de veces. A variable que controla o número de veces que se executa o bucle é o **contador**, e toma valores dun modo ordenado. Aínda que no exemplo anterior empregamos por analoxía coas demais a variable `numero`, é habitual (aínda que non obligatorio) denotar os contadores coas letras `i`, `j`, `k`,...

O contador non soamente se emprega nas estruturas `for`, senón que resulta de grande utilidade para coñecer o número de veces que ocorre algo, o que axudará a comezar ou finalizar procesos, ou enviar mensaxes de información, por exemplo.

Á hora de utilizar contadores é necesario ser coidadoso, xa que se a definición é incorrecta, pode provocar bucles infinitos ou número incorrecto de repeticións.

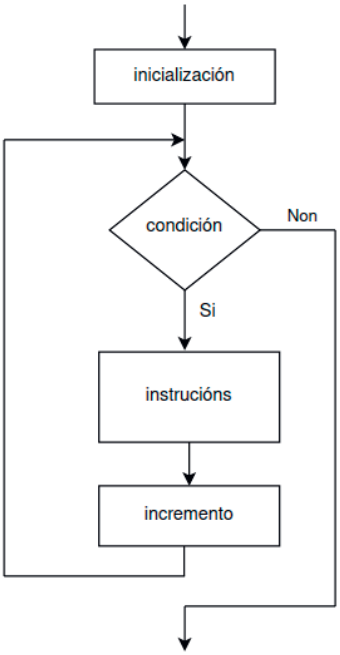



Figura 3.10: Diagrama de fluxo: estrutura de repetición `for`

Para a inicialización do contador é necesario asignarlle un valor inicial, que se verá incrementado ou decrementado, xeralmente dentro do bucle.

Ademais dos contadores, existe outra variable amplamente utilizada en programación. É o **acumulador**, definido como un elemento cuxo contido actual é xenerado a partir do seu valor inmediatamente anterior, e está moi relacionado co contador. De feito, pódese considerar que un contador é un tipo concreto de acumulador. Habitualmente, os valores dun acumulador xéranse por medio de sumas e restas sucesivas, aínda que se poden construir a partir de productos o divisións tamén. Ao igual que os contadores, poden servir para controlar os bucles.

### 3.7.3 Estruturas de salto

Ademais das estruturas de selección e de repetición, existe outro tipo de estruturas de control, denominadas de **salto**, que permiten realizar saltos no fluxo de control dun algoritmo, alterando de forma brusca a secuencia normal da súa execución, e transferindo o control do programa a outro lugar dentro do mesmo.

Todos os programas poden ser escritos sen utilizar instrucións de salto, e o seu emprego na codificación dos algoritmos únicamente é desexable sempre e cando se utilice con precaución e non se perda o control do bucle.

#### Estrutura de salto `break`

Pódese empregar para interromper unha sentenza de iteración, e a súa execución fai que o fluxo de control salte fóra, á sentenza inmediatamente a continuación da instrución de iteración.

Esta instrución adoita empregarse frecuentemente xunto cunha sentenza `if`, actuando como condición

interna do bucle. Tamén é moi habitual o seu emprego en estruturas de selección múltiples (`switch`), despois das instrucións asociadas a cada opción. Ás veces, o emprego de `break` pode facer o programa máis rápido ou doado de entender.

***Exemplo***: instrucións para procesar o bucle (remata), ao atopar o primeiro múltiplo de 5 entre 1 e 100, mediante unha estrutura `break` en C++ (equivalería a traballar coa condición do `break`, xunto coa condición do `while` cun && lóxico).

```
int numero = 1;
...........
do {
    if (numero % 5 == 0) {
        break;
    }
    numero++;
} while (numero <= 100);
```

#### Estrutura de salto `continue`

Emprégase para romper a execución normal dun bucle, pero este non remata: termina a iteración en curso, e transfírese o control do programa á condición de saída do bucle, onde se decidirá se se vai realizar unha nova iteración. Por tanto, esta instrución salta as instrucións que están despois dela na iteración dun bucle.

***Exemplo***: instrucións para procesar o bucle (convértese nun bucle infinito), ao atopar o primeiro múltiplo de 5 entre 1 e 100 , mediante unha estrutura `continue` en C++.

```
int numero = 1;
...........
do {
    if (numero % 5 == 0) {
        continue;
    }
    numero++;
} while (numero <= 100);
```

## 3.8 Exercicios propostos

1. Cales dos siguientes identificadores son válidos e por que?
   (a) INICIO;
   (b) variable1;
   (c) VARIABLE1;
   (d) 1variable;
   (e) variable-1;
   (f) *variable;
   (g) 2020;
   (h) variable 1;
2. Se queremos declarar unha variable de tipo enteiro faremos...
3. Que variable necesitariamos e como almacenariamos a información se desexamos gardar datos referentes a altura e o peso dunha persoa?
4. Cales das seguintes declaracións son incorrectas e por que?
   (a) char letra = a;

(b) int int = 35;
(c) float real = 5.3;
(d) enum Semaforo = “VERMELLO”, “AMARELO”, “VERDE”;
(e) VERDADEIRO;

5. Que tipos de datos son necesarios para almacenar a siguiente información?
   (a) número de materias por curso;
   (b) DNI;
   (c) altura dunha persoa;
   (d) resultado dunha aposta;
   (e) datos de conta bancaria;
   (f) letra ‘A’;
   (g) capital de España;
   (h) conxunto de días da semana;
6. Que se almacena en memoria en cada caso?
   (a) int a = 8;
   (b) int a = 9, b; b = a++;
   (c) int a = 9, b; b = ++a;
   (d) float v1 = 3.9;
   (e) char letra = ‘p’;
7. Que valores se obteñen ao avaliar as seguintes expresións?
   (a) int v1 = 15; v1++;
   (b) int v1 = 15, v2 = 18; v1 / v2;
   (c) int v1 = 15, v2 = 18; v1 % v2;
   (d) float v1 = 3.4, v2 = 5.8; v2 += v1;
   (e) float v1 = 3.4, v2 = 5.8 ; v1 || v2;
   (f) float v1 = 3.4, v2 = 5.8; v1 && v2;
   (g) int v1 = 15, v2 = 18; v1 == v2;
   (h) float v1 = 3.4, v2 = 5.8 ; v1 % v2;
   (i) float v1 = 3.4, v2 = 5.8; v1 = ++v2;
   (j) float v1 = 3.4, v2 = 5.8 ; !(v1 == v2);
   (k) float v1 = 3.4, v2 = 5.8; v1 != v2;
   (l) int v1 = 4, v2 = 5, v3 = 6; v1 * v2 / v3;
   (m) int v1 = 4, v2 = 5, v3 = 6; v1 * v2 % v3;
   (n) int v1 = 4, v2 = 5, v3 = 6; v1 + v2 - 1 != 5 || v3 $>=$ v1-v2 && v1 $>$ 2;
   (o) const int v1 = 4, v1++;
   (p) float v1 = 3.9; !( v1 $<$ 3.5) && !(v1 $>$ 7.2);
8. Escribir un algoritmo que calcule a coordenada y dun punto dunha función lineal, dadas a pendente, a intersección co eixe de ordenadas, e a coordenada x do punto. Representar o algoritmo mediante o correspondente diagrama de fluxo. Pódese empregar a seguinte ecuación:

$$y = pendente * x + interseccion$$

9. Escribir o código correspondente dun algoritmo que calcule a media de 3 notas dun estudante introducidas por teclado.
10. Codificar un programa que lea por teclado unha temperatura en grados Celsius e a transforme en Fahrenheit e Kelvin, amosando estes valores por pantalla.

11. Cal é a saída do seguinte código?

```
int valor = 10;
switch (valor) {
    case 10:
        cout << ++valor;
    case 11 :
    cout << ++valor;
        break;
    case 12 :
        cout << ++valor;
        break;
}
```

12. Cal é a saída do seguinte código?

```
int a = 10, b = 10, c = 2;
if (a == b) {
    a *= c;
}
else if (a != b) {
    a /= c;
}
else {
    a *= c;
}
cout << a;
```

13. Que se imprime por pantalla?

```
int a = 15;
if (++a != 16) {
    cout << "a non vale 16";
}
```

14. Que se imprime por pantalla?

```
int a = 15;
switch (a % 9) {
    case 1: cout << a;
            break;
    case 2: (a %2 != 0) ? cout << "a:" << a : cout << "outro valor";
    default: cout << "default";
}
```

15. Implementar o código necesario para calcular o dobre dun número enteiro n, introducido por teclado, se este é menor que 100.
16. Escribir un algoritmo para ler tres números reais por teclado e que verifique se o terceiro é a suma dos dous primeiros.
17. Escribir o código necesario para visualizar no monitor, en función do valor do tamaño (introducido por teclado) da pizza que desexa comprar, o prezo da mesma: pequena: 12 €, mediana: 15 €, e grande 18 €.
18. Codificar un algoritmo para calcular se un número é múltiplo de 3 e de 5 á vez.
19. Construír un algoritmo para calcular a área dun círculo, un cadrado ou un triángulo, segundo a opción seleccionada desde teclado por medio dun menú. En cada caso o radio, o lado ou a base e a altura introduciranse tamén por medio do teclado.
20. Escribir o código para imprimir a suma de dous enteiros introducidos por teclado, sempre e cando sexan múltiplos, e a resta noutro caso.
21. Que instrución (empregando o operador condicional) debemos usar para imprimir a suma de dúas variables, sempre e cando unha sexa múltiplo da outra, e a resta noutro caso?

22. Cantas veces se repite a mensaxe no seguinte código?

```
int n;
for (int i = 0; i < 4; i++); {
   cout <<"Introduce un numero:" ;
   cin >> n;
}
```

23. Cal é a saída do seguinte código?

```
int a = 1, b = 2, maior;
while ( (a < 4) || (b < 100) ) {
    if (a > b) {
        maior = a;
    }
    else {
        maior = b;
    } else {
        maior = 0;
    }
    a = a + maior;
    b = b * maior;
}
cout << a << b;
```

24. Cal é a saída do seguinte código?

```
int x = 5;
do {
    cout << --x;
} while (x < 10);
```

25. Canto valen n, i, stop e suma ao final da execución?

```
int i, suma = 0, parar = 1, n = 4;
for (i = 1; i <= n && parar != 0; i++) {
    if ( i == n/2 ) {
        parar = 0;
    }
    else {
        suma = suma + i + 2;
    }
}
```

26. Escribir o código dun algoritmo para calcular a suma dos primeiros 50 números naturais.
27. Escribir o algoritmo para calcular o factorial dun número.
28. Soamente con estruturas de repetición, escribir un programa para calcular e visualizar os cadrados dos números pares comprendidos entre o 1 e o 100.

# Capítulo 4

# Programación estruturada

## 4.1 Teorema da programación estruturada

Un dos aspectos máis importantes que hai que ter en conta cando se codifica un programa nunha linguaxe de alto nivel é o control da execución. Xa vimos que, mentres non aparezan estruturas de control nin condicións que deban ser cumpridas, as instrucións vanse executando de forma lineal, de xeito secuencial, unha detrás doutra a medida que aparecen.

Cando se impoñen estruturas de control, por exemplo condicionais, poderanse executar diferentes conxuntos de instrucións, en función do valor dunha determinada expresión, ou tamén repetir diversas sentenzas, se se fai uso das estruturas repetitivas.

Estudamos tamén as (pouco recomendables) estruturas de salto, que permiten transferir o control do fluxo a outras partes do programa, e que deben ser coidadosamente utilizadas. Entre elas, aínda que non se estudou, atópase a instrución ir-a (`goto`), que permite saltar dun punto a outro do programa, e considerada prexudicial, xa que realiza unha transferencia unidireccional de control a outra liña de código.

Aínda que esta instrución era moi usada nas primeiras linguaxes de programación, o seu emprego foise limitando cada vez máis coa aparición dos programas estruturados.

Os programas que usan como estruturas de control unicamente as secuenciais, as condicionais e as repetitivas son **programas estruturados**.

Pódese definir a **programación estruturada** como un paradigma de programación baseado no emprego das tres anteriores estruturas de control, xunto con funcións o subrutinas (serán estudadas no seguinte tema).

A programación estruturada apóiase no **teorema de Böhm-Jacopini**, segundo o cal podemos codificar un algoritmo empregando só tres tipos de estruturas de control.

**Teorema da programación estruturada (Böhm-Jacopini)**:

Toda función computable pode ser implementada nunha linguaxe de programación utilizando unicamente tres tipos de estruturas de programación:

- Estruturas secuenciais.
- Estruturas condicionais.
- Estruturas repetitivas.

Un **programa propio** escríbese usando unicamente as anteriores estruturas de control, e verifica:

- Ten un único punto de entrada e outro de saída.
- Existe sempre, alomenos, un camiño que vai dende o inicio ata o fin do algoritmo, e pode ser seguido.

- Non posúe bucles infinitos.

Para que a programación sexa estruturada, todos os programas teñen que ser propios. O obxectivo do paradigma da programación estruturada é o de desenvolver algoritmos con claridade, doados de escribir, verificar, ler e modificar.

Unha das principais características dun programa estruturado é que se pode ler en secuencia, dende o principio ata o final, e ter unha comprensión total do mesmo. Isto é así porque é moito máis sinxelo entender un código que se escribe de forma sucesiva, que se as instrucións estiveran dispostas nunha orde diferente ao longo do programa. Esta facilidade de lectura é unha consecuencia de utilizar soamente tres estruturas de control, eliminando ademais o desvío de fluxo de control, excepto en circunstancias moi concretas.

Unha das técnicas empregadas para aumentar a comprensión dun programa consiste na **segmentación**, mediante a que se divide o código en porcións máis pequenas de instrucións, arredor de 50 liñas. Deste modo, un programa estruturado está constituído por **segmentos**, formados por unhas poucas instrucións. Cada un deles ten unha entrada e unha saída, sen bucles infinitos e sen instrucións que non se executan nunca. Os segmentos combínanse empregando estruturas de control secuenciais, condicionais e repetitivas. Como resultado desta combinación obtense un programa tamén estruturado.

O segmentos deben comunicarse entre eles de xeito controlado, e deben ser independentes, de forma que un cambio nun deles non afecte a estrutura xeral do programa.

Entre as vantaxes da programación estruturada pódense citar:

- Existe un control da execución. Esta pode ser secuencial, aínda que pode haber partes que se executen ou non en función de determinadas condicións.
- O custo da resolución de cada subproblema será menor que o custo de resolver o problema total como un todo.
- A estrutura do programa é máis clara, e a lexibilidade do código é maior.
- Prodúcese unha redución do esforzo nas probas e depuración.
- Os erros pódense detectar e corrixir de xeito máis doado.
- Existe unha redución dos custos de mantemento.
- Increméntase o rendemento de quen programa, facilitando o seu traballo en paralelo.
- Posibilítase en maior grao a reutilización do código.

Como resumo, pódese dicir que o principio fundamental da programación estruturada radica en que se poida comprender totalmente o programa en todo momento. Para iso, é necesario empregar:

- Estruturas de control limitadas.
- **Recursos abstractos**: descomposición de tarefas complexas, supondo que cada unha das súas partes xa está resolta.
- Deseño descendente do programa.

## 4.2 Deseño descendente

O **deseño descendente** (***top-down***) consiste en deseñar os algoritmos en etapas, dos conceptos xerais aos detalles.

Esta técnica de deseño ten os seguintes obxectivos básicos:

- Simplificar os problemas e os segmentos ou subprogramas de cada descomposición.

- Programar cada unha das partes do programa independentemente do resto, e incluso por diferentes persoas.

- Estruturar o programa final en forma de bloques, de modo que se facilite a súa lectura e mantemento.

Acabamos de ver que unha das vantaxes da programación estruturada é que mellora a lexibilidade do código e simplifica a súa escritura, xa que o divide en segmentos diferentes que despois se comunicarán entre eles. Non é posible coñecer anticipadamente o número e tamaño destes segmentos, e haberá que chegar a un compromiso. En calquera caso, se un bloque de código resulta excesivamente grande ou complicado, sempre poderá ser dividido noutros menores, cada un dos cales aportará a solución dun problema menor específico. Deste xeito, divídese de forma sucesiva o problema xeral noutros máis pequenos, de desenvolvemento máis sinxelo e mellor seguimento.

O deseño descendente é un procedemento de resolución dun problema mediante o que, a través de refinamentos iterativos, partimos do maior nivel de abstracción (máis xeral) e continuamos cara a niveis inferiores (detalles), descompoñendo o problema noutros máis pequenos (Figura 4.1).

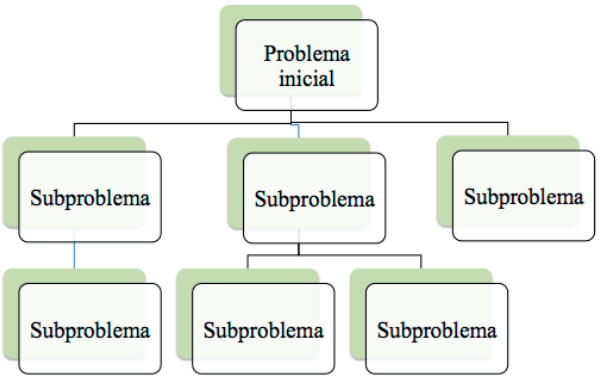



Figura 4.1: Deseño descendente

Vemos así que o deseño descendente ten unha estrutura xerárquica. Nela vese a relación existente entre un segmento e o resto do programa. O problema inicial e os subproblemas no cumio da estrutura xerárquica conterán as funcións de control de máis alto nivel, mentres que os subproblemas en posicións inferiores estarán relacionados con funcionalidades máis detalladas, referentes a unha subtarefa máis específica.

Co deseño descendente pódense observar de forma sinxela as relacións que hai entre as distintas funcións. Isto facilita a comprensión do que debe facer o programa, e garante que a tarefa se realice de forma exacta.

Existe ademais relación entre as diferentes etapas do refinamento, levándose a cabo a través dos fluxos de entrada e saída de información.

***Exemplo***: aplicación do deseño descendente ao cálculo do gasto medio de electricidade anual dunha vivenda (Figura 4.2).

En oposición ao deseño descendente atópase o **deseño ascendente** (***bottom-up***), que identifica os problemas que deben ser codificados a medida que aparecen, para resolver o problema de xeito inmediato, pero non se ten en conta a integración de todos os segmentos. Ademais, neste enfoque non se consideran os obxectivos globais do problema que se debe resolver, polo que en moitas ocasións non se satisfán.

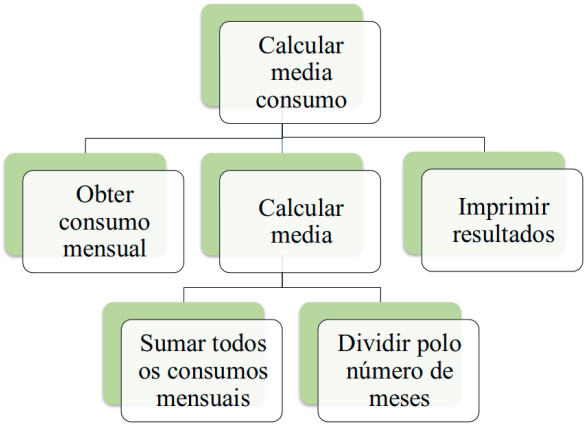



Figura 4.2: Aplicación do deseño descendente ao cálculo do gasto medio eléctrico anual

## 4.3 Exercicios propostos

1. Representar mediante un esquema descendente un algoritmo para sumar dous números.
2. Empregando a metodoloxía descendente, deseñar un algoritmo que almacene os datos de identidade dun alumno e as súas cualificacións obtidas en 3 probas diferentes, e calcule e visualice a media aritméticas delas.
3. Desenvolver o deseño descendente dun algoritmo que, a partir da base e da altura dun triángulo, valores introducidos por teclado, calcule a área deste, e visualice o resultado.
4. Supoñamos que a Dirección da ESEI necesita saber, para cada curso de Grao, o número de estudantes por curso, o estudante que ten unha mellor cualificación media, e a cualificación media do curso. Para iso, cada coordinador de curso proporciona á Dirección o listado de estudantes e as súas cualificacións medias. Deseñar un esquema descendente para un algoritmo que resolva o problema anterior.
5. Unha empresa de dispositivos móbiles posúe varias franquías en todo o país. Necesita coñecer cantos dispositivos vendeu cada franquía no último año, cal foi a media de ventas en todas as franquías, e cal foi a sucursal que máis vendeu. Para iso, as franquías proporcionan como información as ventas mensuais durante o derradeiro ano. Resolver este problema mediante un deseño descendente.
6. Desenvolver un esquema descendente para o problema de solucionar a ecuación cadrática tipo $ax^2 + bx + c = 0$.
7. A partir do radio dunha circunferencia, introducido por teclado, desenvolver un deseño descendente dun algoritmo para calcular o seu perímetro e a súa superficie, e visualizar o resultado.

# Capítulo 5

# Programación modular

A **programación modular** é un paradigma de programación que consiste en dividir un programa en módulos ou subprogramas co fin de facelo máis lexible e manexable. É unha evolución da programación estruturada para solucionar problemas de programación máis grandes e complexos dos que esta pode resolver.

A programación modular amplía o deseño descendente como método de resolución de problemas, e permite:

- División dun programa en segmentos (**módulos**).
- Control da comunicación entre módulos.
- Modificación de partes do programa sen que iso afecte os restantes módulos.

Cada unha das partes independentes ou módulos denomínase **subprograma**. Divídense en **funcións** (**subrutinas**) e **procedementos**. En termos algorítmicos, coñécense como **subalgoritmos**.

Deste modo, coa programación modular temos:

- **Módulo principal**: transfire o control aos distintos módulos ou subalgoritmos.
- **Subalgoritmos ou módulos**: máis pequenos, seguen as regras da programación estruturada e pódense representar con pseudocódigo.

Ademais, no paradigma de programación modular prodúcese **encapsulación**, é dicir, ocultación de detalles das funcións contidas nun módulo.

## 5.1 Funcións e procedementos

Os subprogramas actúan exactamente igual que un programa completo, de xeito que poden recibir datos de entrada, operar con eles, e obter uns resultados. A diferenza entre os dous termos radica no feito de que o programa emprega o subprograma para realizar unha tarefa específica e determinada. Para iso, no momento en que hai que executar o subprograma, o control do fluxo pasa ao devandito subprograma (**invocación** ou **chamada**), retornando ao programa principal, xusto ao punto no que se realizou a invocación, no momento no que esta tarefa finaliza (Figura 5.1).

Á vista da Figura 5.1, pódese observar que, tal e como se dixo, os módulos son independentes, e é posible traballar de xeito simultáneo neles, aínda que, en xeral, non se poden executar en paralelo. Ademais, podemos modificar calquera subprograma sen que isto afecte aos demais. Obsérvase tamén que existen dúas chamadas ou invocacións á función `lerEnteiro()`, pero o algoritmo só está codificado unha vez.

Así, os subprogramas só se escriben unha vez, aínda que se empreguen varias veces durante a exe-

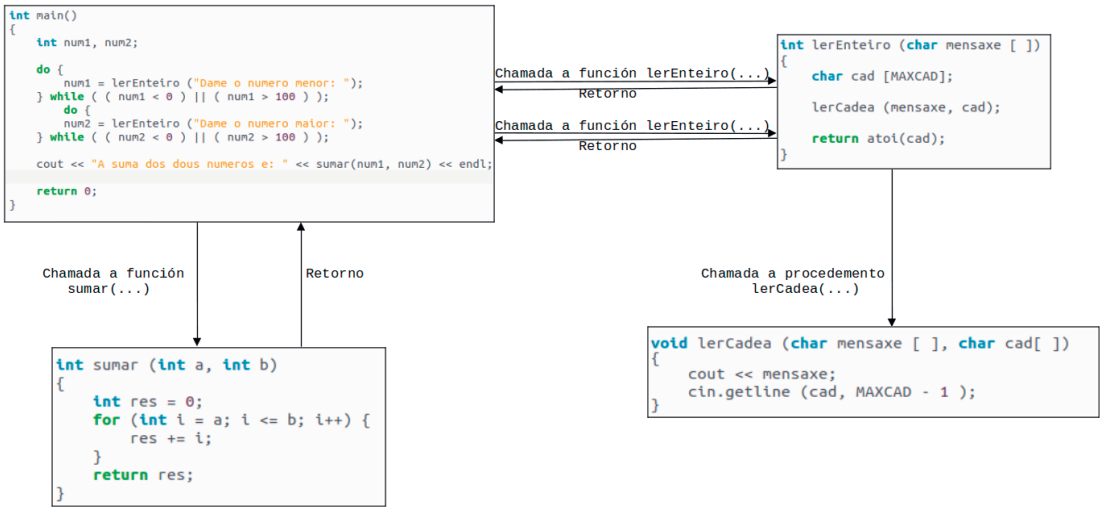



Figura 5.1: Representación gráfica de programa e subprograma

cución do programa, cada vez cuns datos ou resultados distintos. Isto implica unha maior facilidade para reutilización de código, e un menor custo ao resolver varios subproblemas de forma illada con respecto á resolución do problema global, dado que a súa complexidade diminúe considerablemente.

Estudaremos a continuación as funcións e os procedementos en detalle.

## 5.2 Declaración e chamada de funcións

Unha **función** é un conxunto de instrucións que realizan unha tarefa específica, e que se executan dende outra función ou procedemento. Poden invocarse ou chamarse varias veces durante a execución dun programa. Aceptan datos de entrada, procésanos, e producen unha saída. Estes datos de entrada denomínanse **argumentos**.

Todas as linguaxes de programación posúen dous tipos de funcións:

- **Intrínsecas** ou **internas**: funcións estándar, incorporadas á propia linguaxe, que realizan cálculos habituais. Xa están programadas e poden ser usadas por calquera.
- **Funcións definidas no programa**: poden ser escritas por nós para realizar tarefas repetitivas, de modo que se reduce o tamaño dun programa. Deste xeito, determinados segmentos de código fan unha determinada tarefa e, unha vez finalizada, voltan ao punto do programa principal dende o que foron invocados.

Cando se traballa con funcións é necesario saber os datos ou parámetros de entrada cos que van operar, e o tipo de dato ou resultado que produce a execución do código. Temos:

- **Parámetros formais**: nome que reciben os parámetros na propia definición ou declaración da función. Non teñen ningún valor específico e poden ser ningún, un ou varios. Pertencen á función invocada, e reciben o valor ou a dirección de memoria dos parámetros actuais da función que a invoca no momento no que se executa a chamada.
- **Parámetros actuais**: valores reais que toman os parámetros nas chamadas á función. Pertencen ao módulo que realiza a chamada e o seu valor ou dirección de memoria é enviado á función invocada.

Pódese dicir que os parámetros formais son os datos ou resultados como se coñecen dentro da función, mentres que os parámetros actuais son os datos que se envían á función, ou os resultados que se recollen dela, en cada chamada.

A declaración ou definición da función realízase mediante os parámetros formais. A invocación ou chamada dunha función é necesaria para que se execute o código asociado á mesma. Isto faise dende o programa principal ou dende outro subprograma, utilizando o seu nome, seguido dos parámetros actuais ou reais entre parénteses.

A forma dunha función é similar á dun algoritmo, polo que constará das mesmas partes:

- **Cabeceira**: definición da función e parámetros formais necesarios. Os parámetros formais poden ir acompañados de tres posibles modificadores: entrada, saída, ou entrada/saída.
- **Corpo**: declaración de parámetros formais e instrucións. Debe incluír unha instrución para devolver un determinado valor ao algoritmo principal.

As funcións decláranse asociadas a un tipo de valor. Este valor será o que devolverá cada unha delas e é obrigatorio. Para poder devolver este valor, todas as funcións deben ter como última liña a instrución `return`, seguida do valor que se devolve.

En C++, a declaración farase do seguinte xeito:

```
tipo nomeFuncion (parametros)
{
    declaracion variables locais;
    instrucions;
    ............
    return valor;
}
```

A súa invocación dende outro subprograma vén dada por:

```
nomeFuncion (parametros actuais);
```

A instrución `return` finaliza inmediatamente a execución da función, devolvendo o control do fluxo ao punto de programa ou subprograma dende o que foi invocada.

Naquelas linguaxes que inclúan seccións de declaracións, os prototipos das funcións deben colocarse nesta zona. En C++ colocaranse ao principio, antes do código do programa principal.

Ao invocar unha función, a cada parámetro formal asígnaselle o valor actual (real), execútase o corpo de instrucións, e devólvese o valor resultante ao punto de invocación.

***Exemplo***: función para calcular o valor absoluto dun número en C++ (Programa 5.1).

Programa 5.1: Función para calcular o valor absoluto dun número

```
int calcularAbsoluto (int x)
{
    if ( x < 0 ) {
       return -1 * x;
    }
    else {
       return x;
    }
}
```

Todas as funcións devolven, a través da instrución `return`, un valor único ao programa ou subprograma dende o que foron invocadas.

Nalgunhas ocasións é necesario devolver varios valores, polo que as funcións poden resultar de valor limitado e non ser adecuadas. Nestes casos, é necesario dispor doutro subprograma: os procedementos ou subrutinas.

Pódese definir un **procedemento** como un subprograma que realiza unha tarefa específica, e pode ter parámetros ou non. Non devolve ningún resultado. Así, os procedementos son construcións que dan nome a un conxunto de sentenzas e declaracións asociadas, e úsanse para resolver un subproblema dado. Invócanse de forma similar ás funcións, e a declaración tamén é similar, coa diferenza de que carecen de sentenza `return`, e o seu nome non se asocia a ningún dos resultados que obtén.

En C++, a declaración farase indicando como tipo `void` (non atribución dun tipo):

```
void nomeProcedemento (parametros)
{
    declaracion variables locais;
    instrucions;
    ............
}
```

A súa invocación dende outro subprograma vén dada por:

```
nomeProcedemento (parametros actuais);
```

Ao invocar un procedemento, cada parámetro formal substitúese polo valor real, substitúese o corpo da declaración do mesmo pola súa chamada, e execútase o código resultante.

***Exemplo***: procedemento para calcular o valor absoluto dun número en C++ (Programa 5.2).

Programa 5.2: Procedemento para calcular o valor absoluto dun número

```
void calcularAbsoluto (int x)
{
    if ( x < 0 ) {
       cout << -1 * x;
    }
    else {
       cout << x;
    }
}
```

Debemos lembrar sempre que o programa principal é a función `main ()`, que sempre debe estar, aínda que non haxa que chamar a ningunha función.

## 5.3 Paso de parámetros

Os parámetros ou argumentos son variables locais que reciben un valor de entrada antes do comezo da execución do corpo dunha función. O seu ámbito de validez é o propio corpo da función, e serven de nexo de comunicación entre subprogramas, ou entre estes e o programa principal.

Cando se invoca unha función ou procedemento, establécese unha correspondencia entre os parámetros formais e os actuais.Cando se realiza unha chamada a un subprograma, establecendo a correspondencia entre os parámetros actuais e os formais, é importante que se manteña a consistencia posicional de tipos, isto é, que coincidan en cada posición de parámetros os tipos de variables correspondentes a cada un deles. En caso contrario, o compilador detectaría un erro.

Supoñamos unha función `func`, e unha serie de variables locais da mesma, definidas da seguinte forma:

```
tipo func (tipo1 D1, tipo2 D2,... tipon Dn) // D1, D2,..., Dn: parámetros formais
{
    tipo1 v1;
    tipo2 v2;
       ...
```

```
        tipon vn;
           ...
    }
```

No momento no que se invoca a función `func(...)` por medio da correspondente instrución `func (v1, v2,..., vn)`, estaranse tratando `v1, v2,..., vn` como os parámetros actuais ou reais. Idéntica situación teremos ao traballarmos con procedementos.

Hai dous métodos diferentes para establecer a correspondencia de parámetros:

- **Posicional**: emparéllanse os parámetros reais e formais segundo a súa posición na lista. Debe existir o mesmo número de parámetros nas dúas listas. Na definición da función ou procedemento hai que indicar o tipo de parámetro formal. Cando hai un número elevado de parámetros, este método facilita a lexibilidade do código.

  ***Exemplo***: dado un procedemento `void p (int num, float val)`, na chamada ao mesmo dada por `p (5, 3.8)`, os parámetros asignaranse na orde na que se pasan. Así, o primeiro valor, 5, asignarase ao primeiro parámetro, `num` (os dous enteiros), mentres que o segundo valor, 3.8, será asignado ao segundo parámetro `val` (ambos reais).
- **Por nome explícito**: denominada tamén paso de parámetros por nome, indica de forma explícita a correspondencia entre os parámetros actuais e os formais. Non é unha correspondencia usada maioritariamente polas linguaxes de programación, que si empregan case en exclusiva a posicional.

En relación aos tipos de parámetros, pódense clasificar en:

- **Entrada (E)**: transmiten información do programa que invoca ao subprograma.
- **Saída (S)**: devolven resultados.
- **Entrada/Saída (E/S)**: mandan valores ao subprograma o devolven resultados.

Cando se traballa con funcións, estas poden devolver valores ao programa principal tanto como resultado da súa execución, por medio da sentenza `return`, como mediante parámetros. En cambio, un procedemento só pode devolver resultados mediante parámetros.

Existen diferentes métodos para pasar parámetros a un subprograma dende outro subprograma ou dende o programa principal. Os dous máis importantes son o paso por valor e o paso por referencia.

### 5.3.1 Paso por valor

Os parámetros trátanse como variables locais, e os valores iniciais actuais asígnanse copiando os valores dos argumentos correspondentes.

No **paso por valor**, na chamada a unha función ou procedemento créase unha copia dos valores destes argumentos na memoria; será con esta copia de valores coa que se traballe dentro do corpo da función, de xeito que cando esta remate a súa execución, as posicións de memoria nas que se almacenaron estas copias destrúense e pérdese toda información acerca do seu contido. Isto significa que os cambios que se producen nos parámetros ao executar o subprograma non se trasladan aos parámetros orixinais, que serán unicamente parámetros de entrada. A única opción para enviar modificacións ao programa que invoca o subprograma é a través da sentenza `return`.

En C++, a declaración farase do xeito habitual, xa coñecido:

```
tipo nomeFuncion (tipo1 d1, tipo2 d2,...);
void nomeProcedemento (tipo1 d1, tipo2 d2,...);
```

A invocación será igual que a que xa vimos:

```
nomeFuncion (parametros actuais);
nomeProcedemento (parametros actuais);
```

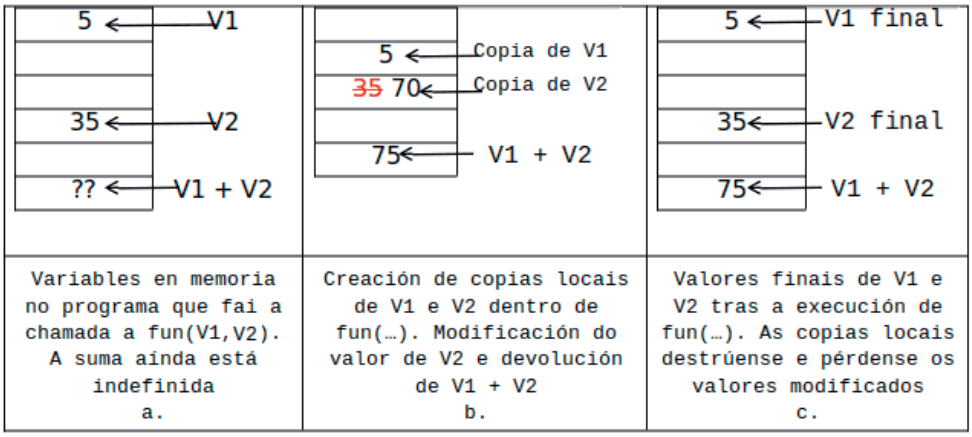



Figura 5.2: Creación de copias de variables no paso por valor

***Exemplo***: a Figura 5.2 amosa a reserva de memoria e o paso por valor ao executar a función invocada por `fun (V1, V2)`. Esta función duplica primeiramente o valor `V2`, e devolve como resultado a suma dos valores `V1` e `V2`:

```
int fun (int V1, int V2)
{
    V2 *= 2;
    return V1 + V2;
}
```

Supoñamos que `V1 = 5`, e `V2 = 35`. A Figura 5.2.a amosa a reserva inicial de memoria, mentres que a Figura 5.2.b presenta a creación de copias de `V1` e `V2` ao realizar o paso por valor, así como o novo valor de `V2` ao executar `fun (V1, V2)` (e a instrución `V2 *= 2;`). Finalmente, a Figura 5.2.c mostra os valores finais de `V1` e `V2`, cando xa rematou a execución de `fun (V1, V2)`. O control do fluxo está outra vez no programa principal (o valor de `V2` non varía, e segue a ser 35, e devólvese o resultado da suma de `V1 + V2`).

Así, no paso por valor os parámetros formais reciben unha copia do valor dos parámetros actuais. Pola súa banda, os parámetros actuais non se ven afectados polas modificacións que se produzan nas súas copias dentro de cada función.

### 5.3.2 Paso por referencia

En ocasións o paso por valor, ademais de consumir máis memoria e tempo de execución, non é suficiente, xa que é necesario que algúns parámetros de entrada se devolvan tamén modificados tras a execución do subprograma, e coa sentenza `return` unicamente é posible devolver un valor.

Nestas situacións débese empregar un método diferente para o paso de argumentos. O **paso por referencia** consiste en pasar como argumento a dirección de memoria do parámetro actual, de xeito que se pode acceder a el tanto dende o programa que invoca o subprograma como dende este último. Este paso de dirección de memoria debe ser transparente a que programa.

Unha **referencia** é un valor que permite acceder indirectamente a un dato determinado dentro dun programa, mediante a súa dirección de memoria. Cando se utilizan referencias, non se pasan copias dos valores orixinais, senón que se crea unha referencia coa mesma dirección en memoria do parámetro actual. Polo tanto, a variable coa que operará o subprograma que recibe a referencia será un sinónimo da variable orixinal, idéntica en dirección de memoria e contido. Por iso, calquera modificación que se faga

no contido desta variable afectará de forma inmediata a variable orixinal.

No paso por referencia os parámetros formais non reciben unha copia dos actuais: pásase directamente a dirección de memoria. Calquera modificación destes parámetros dentro da función ou procedemento modifica o valor dos parámetros actuais. En C++, para indicar o paso dun parámetro por referencia emprégase o operador &.

En C++, a declaración farase do xeito habitual, pero insertando o operador &:

```
tipo nomeFuncion (tipo1 &d1, tipo2 &d2,...);
void nomeProcedemento (tipo1 &d1, tipo2 &d2,...);
```

A invocación será **idéntica** que no caso do paso por valor:

```
nomeFuncion (parametros actuais);
nomeProcedemento (parametros actuais);
```

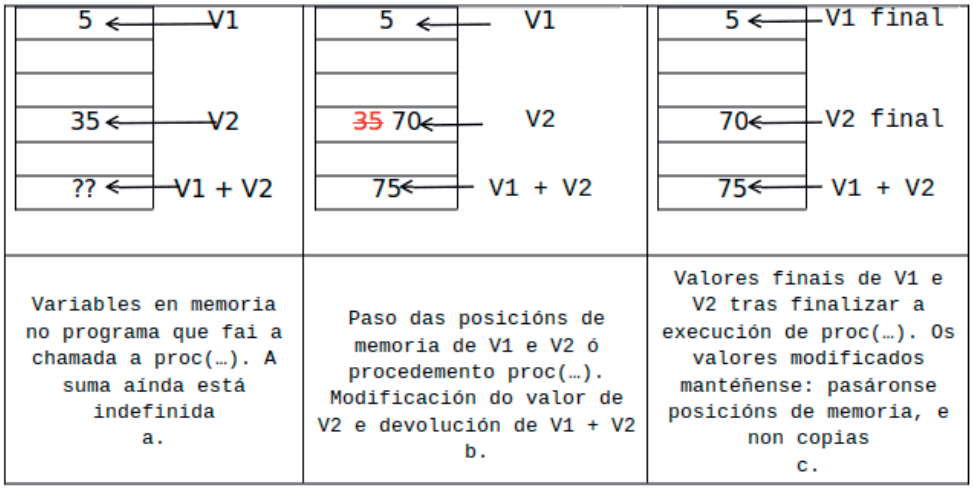



Figura 5.3: Paso de posicións de memoria no paso por referencia

***Exemplo***: a Figura 5.3 amosa o paso por referencia de parámetros no procedemento invocado por `proc (V1, V2, suma)`. Este procedemento duplica primeiramente o valor `V2`, e obtén como resultado a suma dos valores `V1` e `V2`:

```
void proc (int &V1, int &V2, int &suma)
{
    V2 *= 2;
    suma = V1 + V2;
}
```

Supoñamos que `V1 = 5`, e `V2 = 35`. A Figura 5.3.a amosa a reserva inicial de memoria, mentres que a Figura 5.3.b presenta os novos valores de `V2` e `suma` ao executar `proc (V1, V2, suma)` (e as instrucións `V2 *= 2; suma = V1 + V2;`) nas posicións de memoria orixinais. Finalmente, a Figura 5.3.c mostra os os valores finais de `V1`, `V2` e `suma`, cando xa rematou a execución de `proc (V1, V2, suma)`. O control do fluxo está outra vez no programa principal (o valor de `V2` varía, e pasa a ser 70, e en `V1 + V2` almacénase o valor de 75).

***Exemplo***: programa para intercambiar o contido de dúas variables enteiras no paso por valor en C++. Pódese ver que unicamente se intercambian dentro do procedemento.

Como resultado, tras a execución as variables non se modifican no programa principal (Programa 5.3).

Programa 5.3: Intercambio do valor de dúas variables enteiras no paso por valor

```
#include <iostream>
using namespace std;
void intercambiar (int a, int b);
int main ()
{
    int  v1, v2;
    cin >> v1 >> v2;
    intercambiar (v1,v2);
    cout << v1 << v2;
    return 0;
}
void intercambiar (int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Para que se realice o intercambio anterior, unicamente basta con indicar que se fará un paso de variables por referencia. As modificacións nos valores das variables que teñen lugar dentro do subprograma, mantéñense ao voltar ao programa principal (Programa 5.4).

Programa 5.4: Intercambio do valor de dúas variables enteiras no paso por referencia

```
#include <iostream>
using namespace std;
void intercambiar (int &a, int &b);
int main ()
{
    int  v1, v2;
    cin >> v1 >> v2;
    intercambiar (v1,v2);
    cout << v1 << v2;
    return 0;
}
void intercambiar (int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

## 5.4 Variables locais e globais

En temas anteriores definiuse unha variable como unha posición de memoria da computadora na que se almacena un contido dun determinado tipo de dato (número enteiro, número real, carácter,...). Como xa vimos, non todos os tipos de datos (nin por tanto, as variables) ocupan o mesmo espazo, pero todas elas deben ser declaradas previamente ao seu tratamento nun programa, función ou procedemento. A declaración de variables poderá facerse en diferentes partes do código, ben antes do programa principal (función `main ()`), ou ben na cabeceira do mesmo ou dun subprograma. Unha variable declarada na cabeceira dun programa poderá ser utilizada en calquera parte do mesmo, mentres que unha variable declarada dentro dun subprograma unicamente será visible nel. Denominaremos:

- **Variable local**: declarada dentro dun subprograma. Pode ter o mesmo nome que outra variable noutro suprograma ou incluso no algoritmo principal. Cada variable fará referencia a unha posición de memoria diferente.
- **Variable global**: declarada fóra de calquera función, normalmente ao principio do programa.

Defínese o **ámbito** ou **alcance** dunha variable como a parte do programa na que é coñecida. No caso de variables locais, o ámbito limítase ao subprograma no cal foi definida, mentres que, para unha variable global, o ámbito son todas as funcións que compoñen o programa, de forma que calquera delas pode acceder a esa variable para lela ou modificar o seu valor. É dicir, pódese acceder á súa dirección de memoria dende calquera parte do programa.

Cando se codifica un algoritmo, o uso de variables locais permite que os subprogramas sexan independentes. Comunícanse entre si por medio da lista de parámetros correspondente. Estas variables, fóra do ámbito deste subprograma, non terán ningún significado nin serán accesibles dende outras funcións ou procedementos. Ademais, cando remata a execución do subprograma no que foron declaradas, son destruídas.

As variables globais teñen a vantaxe de que permiten que varios subprogramas compartan información sen necesidade de comunicarse a través da lista de parámetros. En ocasións permiten simplificar o programa, sobre todo cando se necesita pasar un número considerable de argumentos a cada función, algúns dos cales pode ser recomendable que sexan variables globais. Porén non é recomendable o uso destas variables, aínda que aparentemente parezan moi útiles, debido a varios motivos:

- Diminúe a lexibilidade do código.
- Poden producirse efectos colaterais, por exemplo cando existe unha alteración non desexada do contido dunha variable global dentro dunha función, ben por invocación, ben por esquecer definir na función unha variable local ou un parámetro formal con ese nome.
- Atenta contra a modularidade, xa que supón compartir espazos de memoria con outras funcións, e diminuír a independencia dos subprogramas.

Programa 5.5: Variables definidas en distintos ámbito

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

int obterAbsoluto (int x);

int main()
{
    int valor, absoluto;
    cin >> valor;
    absoluto = obterAbsoluto (valor);
    cout << valor << " " << absoluto;
    return 0;
}

int obterAbsoluto (int x)
{
     int valor;
     if (x < 0)
         valor = -1 * x;
     else
         valor = x;
     cout << valor;
     return valor;
}
```

O Programa 5.5 amosa un exemplo de variables definidas en diferentes ámbitos no cálculo do valor absoluto dun número mediante funcións en C++. Supondo que se introduce como valor de entrada -4, ao chegar á liña `absoluto = obterAbsoluto (valor);` o control do fluxo pasa á función `obterAbsoluto (...)` , de forma que a instrución `cout<< valor;` produce como resultado 4. Unha vez rematada a execución da función, e devolto o control de fluxo ao programa principal, a liña `cout<< valor << ""<< absoluto;` visualizará por pantalla os valores -4 e 4. Vemos así que valor aparece dúas veces cos valores 4 e -4, dependendo do ámbito da variable á que se fai referencia.

En C++, o operador :: coñécese como o **operador de resolución de ámbito**, e permite identificar e eliminar a ambigüidade dos identificadores utilizados en ámbitos diferentes.

## 5.5 Recursividade

Fálase de **recursividade** cando unha función ou procedemento se invocan a si mesmos.

Cando un subprograma chama a outro, a pía do sistema (estrutura que permite almacenar e recuperar datos, só se pode acceder ao último dato almacenado nun momento determinado) almacena os seus parámetros formais e variables, así como a posición do código á que se debe devolver o control do fluxo unha vez finalice o subprograma. Neste momento, recupéranse da pía os parámetros formais e as variables. Esta mesma situación aparece cando un subprograma se chama a si mesmo, de forma que en cada chamada se almacenan na pía copias dos parámetros formais e das variables do subprograma. Se esta operación se realiza de forma indefinida, a pía desborda e aparecen problemas de memoria. Polo tanto, cando se executan programas recursivos, é importante impor algunha condición para saber cando debe rematar a execución. É habitual empregar contadores ou condicións lóxicas.

Calquera problema que se pode resolver de forma recursiva poderase resolver tamén mediante estruturas repetitivas ou iteracións.

A recursividade preséntase como unha alternativa elegante ás estruturas repetitivas, sempre e cando:

- Exista un criterio, chamado **criterio base** (de parada), polo cal o procedemento ou función non se invoque a si mesmo.
- En cada chamada ao subprograma, este estará máis cerca do criterio base.

En termos xerais:

- Elixirase a recursividade cando así o requira a natureza do problema, de modo que a programación mediante enfoque recursivo sexa máis sinxela que mediante estruturas iterativas.
- Evitarase a recursividade naquelas situacións nas cales o rendemento sexa crítico, ou se consuman elevados recursos en termos de tempo computacional e memoria.
- En caso de igual facilidade para resolver un problema mediante recursividade ou iteracións, empregarase a solución iterativa, posto que se executa máis rapidamente, porque non hai que facer chamadas adicionais a subprogramas, e emprega menos memoria.

***Exemplo***: cálculo do factorial dun número en C++ empregando recursividade (Programa 5.6).

Programa 5.6: Algoritmo recursivo para o cálculo do factorial dun número en C++

```
int factorial (int n)
{
    if (n == 0) {
        return 1; // Caso base
    }
    else {
        return n * factorial (n - 1);
    }
}
```

***Exemplo***: cálculo da potencia empregando recursividade (Programa 5.7). A potencia pódese definir como: $a^n = a^{n/2} * a^{n/2}$ se n é par, e $a^n = a^{(n-1)/2} * a^{(n-1)/2} * a$ se n é impar .

Programa 5.7: Algoritmo recursivo para o cálculo da potencia en C++

```
int calcularPotencia (int b, int n)
{
    int pote;
     if (n <= 0) {      // Caso base
         return 1;
    }
     if (n % 2 == 0) {  // n par
         pote = calcularPotencia (b, n/2);
         return pote * pote;
     }
     else {             // n impar
         pote = calcularPotencia (b, (n - 1)/2);
         return pote * pote * b;
     }
}
```

## 5.6 Bibliotecas

Xa vimos que todas as linguaxes de programación posúen dous tipos de funcións: as definidas propias, definidas por nós, xa estudadas en profundidade, e as intrínsecas ou internas, incorporadas na propia linguaxe.

Unha **biblioteca** é un arquivo que contén o código dun conxunto de funcións e procedementos, independentes entre si. Non posúe estrutura de programa executable, polo que non se pode executar de modo similar a un programa. Será un programa o que, con chamadas aos subprogramas da biblioteca, os poida executar.

Nunha librería non existe o punto de inicio dun programa. Isto significa que se poderá chamar calquera subprograma que conteña con independencia do lugar no que estea definida. Polo tanto, a súa execución non será secuencial. Ao igual que os programas, as bibliotecas poden necesitar, para o seu correcto funcionamento, outras bibliotecas.

As bibliotecas facilitan a reutilización do código, evitando ademais a codificación de determinadas operacións (como as rutinas de entrada ou saída de datos, as aritméticas,...). Estas tarefas de baixo nivel están xa codificadas en practicamente todas as linguaxes de alto nivel, e están dispoñibles para a construcción do programa en bibliotecas estándar. Para usalas, só é necesario coñecer o nome da biblioteca e das funcións e procedementos que se desexan utilizar.

***Exemplo***: algunhas bibliotecas estándar en C++ son `math`, `iostream`, `string`. Inclúense mediante a directiva `#include`.

Ademais destas bibliotecas predefinidas, pódense crear, de modo similar ás funcións propias, bibliotecas. Para iso, cando programamos, créase un arquivo externo ao programa, no cal incluirá o código correspondente aos subprogramas por el definidos. Este código non se incluirá no programa principal, senón que unicamente escribirá unha instrución que fai referencia a cada biblioteca que vaia ser necesaria, e, en cada caso, a invocación á correspondente función. As bibliotecas pódense clasificar:

- **Estáticas**: enlázanse ao programa no momento da compilación (mentres se constrúe o programa obxecto).
- **Dinámicas**: enlázanse en tempo de execución.

## 5.7 Exercicios propostos

1. Dado o seguinte fragmento de código dun programa principal, que prototipo de función cres que será compatible coa chamada realizada?

```
...
int main ()
{
    float f1, f2, f3;
    ...
    f3 = obterValor (f1,  f2);
    cout << "Valor de f3: " << f3 << endl;
    ...
    return 0;
}
```

2. Cal é a saída do seguinte código?

```
...
int calcularProduto (int a, int b);

int main ()
{
    int n1 = 6, n2 = 4;
    n2 = calcularProduto (n1, n2);
    cout << "n1: " << n1 << ", n2: " << n2 << endl;
    return 0;
}

int calcularProduto (int a, int b)
{
    a = 20;
    return a * b;
}
```

3. Que se imprime por pantalla ao executar as seguintes instrucións?

```
...
int fun (int y, int x);

int main ()
{
    int x = 3, y = 7;
    cout << x << ", " << y << ", " << fun (y, x) << ", " << fun (x, y) << endl;
    return 0;
}

int fun (int y, int x)
{
    return x - y;
}
```

4. Que ocorre ao executar o seguinte código?

```
...
int calcular (int a, int b, int s, int r);

int main ()
{
    int n1 = 8, n2 = 2, suma, resta, produto, code;
    code = calcular (n1, n2, suma, resta);
    cout << suma << resta << produto << endl;
    return 0;
}
```

```
int calcular (int a, int b, int s, int r)
{
    int p;
    s = a + b;
    r = a - b;
    p = a * b;

    return 0;
}
```

5. Escribir un subprograma que devolva o valor de $5x^2 + 2x + 1$.
6. Implementar un programa para solicitar por teclado un enteiro positivo menor que 100. Unha vez introducido correctamente, preguntará se se quere introducir outro. O programa parará cando se introduza 'n' ou 'N'. Utilizar unha función para introducir o valor. Converter a función en procedemento.
7. Escribir un algoritmo que conteña unha función que calcule o cubo dun número enteiro, introducido por teclado.
8. Escribir unha función que calcule a potencia dun número enteiro. Tanto a base como o expoñente serán os argumentos.
9. Escribir un programa para calcular a área dun triángulo, mediante dous procedementos: un para introducir os valores de base e altura, e outro para calcular a área.
10. Escribir un algoritmo que solicite por teclado as coordenadas cartesianas dun punto e as transforme a coordenadas polares mediante un subprograma.
11. Codificar un programa que conteña un procedemento para calcular o índice de masa corporal dunha persoa.
12. Escribir un programa que solicite ao usuario dos números enteiros positivos e calcule, empregando unha función, o cociente da división enteira do maior deles entre o menor, mediante sumas e restas.
13. Empregando funcións e/ou procedementos, codificar un programa que calcule a área dun círculo, un cadrado ou un triángulo, segundo sexa a opción seleccionada dende teclado mediante un menú.
14. Implementar un programa que solicite por teclado un número determinado de segundos e calcule, a partir dese valor e mediante un subprograma, as horas e minutos correspondentes.
15. Implementar un algoritmo que solicite por teclado dous número enteiros entre 1 e 150 e, por medio dun menú, permita seleccionar as seguintes opcións, que serán implementadas como subprogramas: a) sumalos, b) restalos, c) multiplicalos, d) dividir o maior entre o menor, e) rematar a execución.
16. Desenvolver un programa coas seguintes funcións: a) introducir un valor enteiro impar n comprendido entre 1 e 49 (débese verificar que é impar entre 1 e 49, en caso de non selo, deberase solicitar un novo valor), b) calcular a suma dos termos impares dende 1 ata n, e c) calcular o produto dos termos impares dende 1 ata n.
17. Que ocorre ao executar o seguinte programa?

```
...
int func (int n);

int main ()
{
    cout << func (20);
    return 0;
}

int func (int n)
{
    return func (n - 1) + func (n);
}
```

18. Dada unha variable enteira n = 8, cal é o resultado de executar as seguintes instrucións?

```
...
int func (int n);

int main ()
{
    cout << func(n);
    return 0;
}

int func (int n)
{
    if ((n == 0) || (n == 1) || ( n == 2)) {
        return 1;
    }
    else {
        return func (n - 2) + n - 1;
    }

}
```

19. Dada unha variable enteira n = 5, que se imprime por pantalla ao executar o seguinte código?

```
...
int func (int n);

int main ()
{
    cout << func (n);
    return 0;
}

int func (int n)
{
    if (n == 1) {
        return 1;
    }
    else {
        return func (n - 1) * n;
    }

}
```

20. Dada unha variable enteira n = 4, que pasa cando se executa o seguinte código?

```
...
int func (int n);

int main ()
{
    cout << func (n);
    return 0;
}

int func (int n)
{
    if (n == 1) {
        return 1;
    }
    else {
        return func (n - 1) + n - 1;
    }

}
```

21. Mediante recursividade, programar un algoritmo que permita visualizar un número enteiro introducido por teclado ao revés.
22. Escribir unha función recursiva para o produto enteiro de dos números.
23. Escribir un algoritmo que inclúa unha función recursiva para calcular a suma dos N primeiros números naturais, con N introducido por teclado no algoritmo principal.
24. Escribir un programa que, mediante recursividade, obteña os valores dos N primeiros termos da serie de Fibonacci.
25. Implementar un algoritmo que inclúa unha función para pasar un número enteiro positivo de decimal a binario, de forma recursiva.

# Capítulo 6

# Depuración e probas

Xa se viu en temas anteriores que o desenvolvemento de software implica sempre unha etapa na que o software implementado debe ser probado, e corrixidos todos os erros. Estudamos así que un programa non se considerará correcto ata que se teña verificado, isto é, ata que se teñan realizado con éxito probas sobre el, contemplando os posibles escenarios, así como resultados da súa execución. Veremos a continuación os erros que poden aparecer e as probas que se deben realizar para detectalos e poder corrixilos.

## 6.1 Erros

O proceso de identificación e corrección de erros coñécese como **depuración**. Denomínase **erro** a calquera situación que produza un funcionamento incorrecto do programa, dende pequenos erros de sintaxe ata bloqueos do sistema. Durante a execución dun programa pódense producir tres tipos diferentes de erros:

- ***De compilación***: erros de sintaxe, fan que a computadora non entenda a instrución. O código non se pode executar, xa que non se xerou o programa obxecto nin, polo tanto, o executable. Habitualmente, son erros fáciles de localizar. Un exemplo sería en linguaxe de programación C++ non escribir o ; ao final dunha sentenza.
- ***De execución***: a computadora entende as instrucións, pero non as pode executar. Detense a execución e infórmase do erro. Detéctanse unicamente mentres se está a executar o programa. Como exemplo, podemos citar a división por 0.
- ***Lóxicos***: débense ao deseño de algoritmo. O programa funciona, pero non produce os resultados esperados. É un tipo de erro difícil de detectar. Poden ser, por exemplo, escribir un ; despois dunha sentenza `if`, ou cambiar nunha fórmula matemática un operador por outro.

Na práctica, resulta imposible programar sen cometer erros, pero existen algúns factores que afectan tanto o número como a envergadura dos mesmos. Entre eles están:

- A experiencia de quen programa.
- O emprego de metodoloxías de desenvolvemento de software adecuadas.
- O deseño de subprogramas independentes, e a verificación final tras a integración de todos eles no programa global.
- A atención de quen programa.

Para localizar erros nun código é importante utilizar as facilidades proporcionadas polas contornas de programación empregadas, incluír no código instrucións que visualicen as liñas executadas e valores obtidos, de ser o caso, empregar puntos de ruptura para facer un seguimento da execución paso a paso,

analizar os valores de cada variable en cada un deles,... Trátase de localizar os puntos nos que se poden producir erros, e seguilos durante todo o programa. Haberá que ir delimitando o erro de forma sucesiva, ata reducilo a unhas cantas instrucións.

Algunhas das estratexias que se poden empregar para depurar erros son:

- Instrucións de escritura de variables (`cout << ...`): pódense empregar naqueles puntos nos que resulte interesante coñecer o valor que toman as variables, para saber se son correctos. Nestes casos, aconséllase indicar cun comentario a función destas sentenzas adicionais de código.
- Estruturas condicionais: é recomendable empregar puntos de ruptura, ou sentenzas de escritura de variables (`cout << ...`) para verificar a decisión tomada polo algoritmo en cada caso.
- Estruturas repetitivas: é importante situar puntos de ruptura ao principio e ao final dos bucles, e analizar os valores das variables nos mesmos.
- Funcións e/ou procedementos: novamente, a inserción de puntos de ruptura e sentenzas de escritura de variables (`cout << ...`) antes e despois da execución destes subprogramas permite verificar o correcto funcionamento dos mesmos.

## 6.2 Probas

As probas permitirán determinar o correcto funcionamento dun programa ou algoritmo. Esta corrección virá determinada polos resultados (esperados ou non) obtidos por un conxunto de probas. Para ter éxito na etapa de probas é importante ter tanto un deseño adecuado do algoritmo, como un conxunto de probas apropiado.

A fase de probas é una parte fundamental no proceso de desenvolvemento de software. Nela débense eliminar todos os erros posibles. Un primeiro paso consiste en probar cada subprograma con datos de entrada válidos, que conducen a una situación coñecida, e ver que os resultados son os esperados. Débense incluír tamén datos inválidos, así como datos aleatorios. Cando todos os subprogramas teñen sido probados, hai que integrar o programa completo e realizar as correspondentes probas de integración. Non é necesario esperar a ter o código completo para comezar coas probas, senón que cada módulo ou subprograma pódese ir probando a medida que se desenvolve o código.

Aínda que non é posible comprobar todas as posibles entradas dun programa con probas, si se poden utilizar para iso técnicas de **verificación formal**. Con esta técnica será necesario construír probas matemáticas que axuden a determinar se os programas fan o que teñen que facer. Deste modo, haberá que aplicar regras formais para verificar se un programa cumpre a súa especificación.

Á hora de elixir o conxunto de probas hai que ter en conta que o obxectivo é atopar erros aínda non detectados, e que un bo caso de proba é aquel que ten una gran probabilidade de encontrar un erro oculto ata o momento. Ademais, una proba con éxito é aquela que atopa un erro que, doutro xeito, non se descubriría.

Para obter un bo caso de proba hai que considerar:

- A probabilidade da proba de atopar un erro debe ser elevada.
- O obxectivo da proba debe ser probar se o programa non fai o que debe, e fai o que non debe.
- Débese eliminar a redundancia, e deseñar diferentes probas para diferentes obxectivos.
- Cada proba debe realizarse por separado.

Xeralmente, os erros de software adoitan repetirse. Ademais, considerando un sistema de software grande, sempre existe un número reducido de módulos ou subprogramas que tenden a producir máis erros que o resto. Polo tanto, haberá que identificar estes módulos e probalos de forma exhaustiva para atopar os principais erros do sistema. A porcentaxe de fallo así detectada será superior á que se detectaría se se

probasen todos os módulos de modo uniforme. Ademais, será necesario que as probas poidan seguirse ata os requisitos iniciais do programa, se planifiquen adecuadamente antes de comezar o proceso de proba, e empecen polos detalles e avancen cara o máis xeral.

Existen varias metodoloxías de probas de software. As máis utilizadas son as probas de caixa branca e as de caixa negra.

### 6.2.1 Probas de caixa branca

Chamadas tamén **estruturais**, coñécense os detalles internos de implementación e os camiños de execución. Deberanse escribir casos de proba para verificar as rutas lóxicas do software, e casos de proba para que se executen sempre alomenos unha vez co resultado verdadeiro e outra co falso os bucles, as estruturas de decisión, e os conxuntos específicos de condicións.

As principais técnicas para crear un caso de proba son:

- **Cobertura de camiños**: escríbense casos de proba para que se executen todos os camiños do programa. Para iso son necesarios varios casos de proba, determinar camiños independentes, verificar o cumprimento de condicións e as condicións imposibles (código que non se executa nunca),... Dado que un programa pode ter un número elevado de camiños, non é posible probalos todos. Por iso, para escribir os casos de proba hai diversos criterios. Entre eles, pódese utilizar o **criterio de ruta básica**, no cal se representa o programa nun **grafo de fluxo** (amosa todos os camiños polos que se pode pasar ao executar un programa), calcúlase a **complexidade ciclomática** (medida da complexidade lóxica dun programa) e determínase o conxunto básico de camiños independentes. Non se entrará no cálculo de complexidades ciclomáticas nin na representación de programas mediante grafos de fluxo.
- **Probas de condicións**: contrástase se se cumpre ou non cada parte dunha condición. Escribiranse casos de proba por cada operando lóxico, verificando se cada expresión se cumpre ou non, isto é, toma os valores verdadeiro e falso.
- **Probas de bucles**: escribiranse casos de proba para verificar que as estruturas repetitivas non son infinitas e funcionan correctamente.
  - *Bucles simples*: para un bucle que se repite m veces, faranse as seguintes probas: pasar por alto o bucle, pasar una soa vez, pasar dúas veces, pasar n $<$ m veces polo bucle, pasar m-1 e m+1 veces.
  - *Bucles aniñados*: faranse probas co bucle máis interno, como se fose simple, mantendo o mínimo número de iteracións no externo. A continuación faranse probas co seguinte bucle máis externo, mantendo valores típicos para o máis interno.

***Exemplo***: casos de proba de caixa branca para estrutura de selección.

```
int a, b;
...
if (a < 0) {
    a *= 2;
}
if (b != 0) {
    b = 5;
}
```

**Casos de proba**

1. COBERTURA DE CAMIÑOS e PROBA DE CONDICIÓNS
   - `a = -5, b = 0`: a verdadeiro, b falso
   - `a = 3, b = 8`: a falso, b verdadeiro

2. PROBA DE BUCLES
    - Non hai.

***Exemplo***: casos de proba de caixa branca con estruturas condicionais e bucles.

```
int x1 = 5, x2 = 3, x3 = 4, x4, numVeces;
...
cin >> numVeces;
for (int i = 1; i < numVeces; i++) {
    if ((x1 > 0) && (x2 > 2) && (x3 > 3)) {
        x4 = x1 * x2 * x3;
        if (x4 > 0) {
            x4 /= 3;
        }
        else {
            x4 += 1;
        }
    }
}
```

**Casos de proba**

1. COBERTURA DE CAMIÑOS e PROBA DE CONDICIÓNS
    - Se `x1 > 0, x2 > 2, x3 > 3`, execútase x4 = x1 * x2 * x3, pero nunca se executa x4 += 1, porque o impide a primeira condición.
    - Se non se cumpre que `x1 > 0, x2 > 2, x3 > 3`, nunca se executaría nada.
2. PROBA DE BUCLES
    - Se `numVeces > 0` executaranse as instrucións.
    - Se `numVeces <= 0` nunca se executarán as instrucións.

### 6.2.2 Probas de caixa negra

Son probas funcionais, que se executan dende o punto de vista de quen emprega o programa, e están centradas na correcta execución do software con respecto á súa temporalidade e exactitude. Non se consideran os detalles internos, senón as funcións específicas.

Dentro das probas de caixa negra atópanse as **probas de integración** (proban a interacción entre módulos), e as probas ***beta*** (consisten en obter una versión preliminar do programa, proporcionala a un grupo moi específico de persoas usuarias- xeralmente expertas-, e ver como se comporta o software). A versión final implementarase incluíndo comentarios e aportacións destas persoas.

Cando se realizan probas de caixa negra hai que ter en conta que o código do programa non está dispoñible.

Á hora de planificar e deseñar as probas dun programa, é recomendable considerar que contén erros mentres non se demostre o contrario, e que non existe ningunha proba que poida probar que non hai erros. Ademais, será necesario comezar as probas antes de finalizar a codificación, e as modificacións deberán realizarse unha a unha.

Cada proba debe verificar se se cumpren as especificacións do programa. Cómpre dispor de datos de proba adecuados. Isto significa que hai que coñecer a saída correcta para cada entrada de proba, e esta debe incluír os datos de entrada que orixinen máis erros.

Existen métodos diferentes para buscar datos de proba que conduzan a erros ao executar o programa:

- **Partición equivalente**: divide os parámetros de entrada nun conxunto de clases de datos, que son denominadas **clases de equivalencia**. Considérase que calquera elemento dunha clase de

equivalencia é representativo do resto do conxunto. A partir das clases de equivalencia obtéñense as clases de equivalencia válidas (xeran valores esperados) e inválidas (xeran valores inesperados). Para crealas será necesario identificar os valores de entrada, así como os seus rangos (valores máximo e mínimo) e conxuntos (valores permitidos), valores lóxicos,...

- **Valores de proba extremos**: non é posible probar todos os valores externos, pero si se poden probar os valores extremos dentro do rango de valores posible para os datos de entrada. É unha técnica complementaria á das clases de equivalencia.

***Exemplo***: clases de equivalencia válidas e inválidas en probas de caixa negra para un contador das horas dun día (Táboa 6.1).

Táboa 6.1: Clases de equivalencia para un contador de horas nun día

| Casos | Valor | Clase válida | Clases inválidas |
|---|---|---|---|
| contador | enteiro | 1: $1 <= valor <= 24$ | 2: $valor < 1$ |
| | | | 3: $valor > 24$ |

***Exemplo***: casos de proba de valores extremos para unha variable que indica os meses do ano (Táboa 6.2).

Táboa 6.2: Casos de proba de valores extremos para variable indicando meses do ano

| Casos | Valores (rango [1-12]) |
|---|---|
| Valor no límite inferior do rango de entrada | 1 |
| Valor un por enriba do límite inferior | 2 |
| Valor válido dentro do rango | 6 |
| Valor un por debaixo do límite superior | 11 |
| Valor no límite superior do rango de entrada | 12 |

Para depurar un programa é necesario seguir una estratexia que permita avaliar, en primeiro lugar, os compoñentes máis simples, e avanzar progresivamente ata probar todo o software como un todo. Establécense os seguintes pasos:

- **Probas unitarias**: próbase cada función ou procedemento. Poden ser de caixa branca e de caixa negra.
- **Probas de integración**: combínanse os subprogramas e próbanse de novo. Adóitanse empregar probas de caixa negra.
- **Probas do sistema**: combínase o programa completo cos compoñentes *hardware* necesarios e verifícase o cumprimento dos requirimentos funcionais. Empréganse probas de caixa negra.
- **Probas de aceptación**: a clientela comproba que o software funciona tal e como debe. Utilízanse probas de caixa negra.

Existen diferentes ferramentas que permiten realizar diversas probas unitarias nos distintas linguaxes de programación. Pódense citar CppUnit (`http://cppunit.sourceforge.net`) para C/C++, JUnit (`http://junit.org`) para Java, ou PHPUnit (`https://phpunit.de/`) para PHP. Outros, como Mocha (`https://mochajs.org/`), ou Jasmine (`https://jasmine.github.io/`) están enfocados ás probas de integración. Para aquelas probas que inclúen verificación de requirimentos funcionais pódense usar, por exemplo, JMeter (`https://jmeter.apache.org/`) ou Cucumber (`https://cucumber.io/`).

## 6.3 Exercicios propostos

1. A partir da función para verificar se un número é primo, deseñar probas de caixa branca para verificar a súa corrección, probando todos os camiños, condicións e bucles para este algoritmo, considerando

unicamente enteiros positivos.

2. Para o exercicio anterior, deseñar casos de proba de caixa negra para os valores límite e para clases equivalentes.
3. Deseñar un conxunto de probas de caixa negra para un algoritmo que verifica contrasinais para acceder a unha determinada aplicación. Para a verificación considéranse as siguientes regras:
   (a) O contrasinal contén máis de 8 caracteres, e menos de 12.
   (b) Emprega maiúsculas e minúsculas, díxitos e *, - ou . ao final.
   (c) Non pode ser unha palabra reservada.
4. Supondo l1, l2 e l3 os lados dun triángulo, deseñar casos de proba de caixa branca e caixa negra para o algoritmo que verifica se o triángulo é equilátero, isósceles ou escaleno (consideramos que os lados son valores enteiros).

# Capítulo 7

# Estruturas e unións

Ata o de agora estivemos a traballar con tipos de datos simples, isto é, numéricos, tipo carácter ou lóxicos. Non obstante, en moitas ocasións é necesario tratar con conxuntos ou series de valores relacionados entre si, e o seu procesamento mediante tipos de datos simples pode resultar complexo. Por iso, a meirande parte das linguaxes de programación inclúen os tipos de **datos compostos** ou **estruturados**.

Estes tipos de datos defínense agrupando tipos de datos simples, ou ben outros tipos previamente definidos no programa. Posúen un identificador común que representa múltiples datos individuais, cada un dos cales pode ser referenciado de forma independente.

Estes tipos de datos caracterízanse por:

- Os tipos de datos que os compoñen.
- As operacións que se poden realizar.

Dependendo da forma de almacenamento en memoria, podemos clasificar os tipos de datos estruturados en:

- **Estruturas estáticas**: o tamaño ocupado en memoria defínese antes da execución do programa. Non se pode modificar durante a súa execución. Entre estes tipos de datos están as estruturas, arrais, arquivos,...
- **Estruturas dinámicas**: non posúen limitacións de tamaño. Pódense construír mediante punteiros (soportados en moitas linguaxes, e que estudaremos máis adiante). Engloban as listas, pilas, colas, árbores,...

Atendendo ao tipo dos datos que o compoñen, unha estrutura de datos composta pode ser:

- **Homoxénea**: todos os valores ou datos que a forman son do mesmo tipo (denominado tipo base). Exemplo: arrais, cadeas.
- **Heteroxénea**: os valores ou datos que a forman non teñen que ser necesariamente do mesmo tipo. Exemplo: estruturas.

Estudaremos a continuación as estruturas.

## 7.1 Estruturas

Denominadas tamén **rexistros**, son estruturas de datos estáticas heteroxéneas. Almacenan diferentes tipos de datos loxicamente relacionados baixo unha mesma variable. Desígnase como **campo** ou **membro** a cada un dos elementos de datos do rexistro ou estrutura. Caracterízanse por:

- Cada campo é dun determinado tipo (simple ou estruturado, incluso pode ser outra estrutura).

- O nome de cada campo é único.
- Cada campo aparece nunha orde determinada no rexistro.
- Para definir o rexistro é necesario especificar tanto o nome como o tipo de cada campo.

Para traballar con datos de tipo estrutura cómpre declaralos primeiro, e despois asignar valores a cada un dos seus membros.

En C++, a declaración farase do seguinte xeito:

```
struct [NomeEstrutura] {
    tipo1 nomeMembro1;
    tipo2 nomeMembro2;
    tipo3 nomeMembro3;
    ..........
    tipoN nomeMembroN;
};

NomeEstrutura nomeVariable; // Declaración de variables
```

Cando se declara unha estrutura, estase a reservar espazo en memoria para cada un dos seus membros.

***Exemplo***: declaración dunha estrutura para definir un libro (por título, autor, prezo e número de páxinas) en C++.

```
struct Libro {
    char titulo[100];
    char autor[100];
    float prezo;
    int numPaxinas;
};

Libro meuLibro;
Libro teuLibro;
```

No exemplo anterior, aínda que non se estudou en profundidade, vemos que aparece o tipo de dato cadea (`char [ ]`). Consideraremos de momento unicamente que está formado por elementos de tipo carácter, e tratarémolo como tal.

É posible tamén traballar con estruturas aniñadas, isto é, con estruturas nas que un ou varios membros poden ser outras estruturas. Neste caso, as estruturas máis internas (membros doutras estruturas) deben ser declaradas antes.

***Exemplo***: declaración de dúas estruturas aniñadas para definir un libro (por título, autor, data de publicación, prezo e número de páxinas) en C++. A Figura 7.1 amosa a representación gráfica do almacenamento en memoria da estrutura `Libro`.

```
struct Data {
    int dia;
    int mes;
    int ano;
};

struct Libro {
    char titulo[100];
    char autor[100];
    Data dataPublicacion;
    float prezo;
    int numPaxinas;
};

Libro meuLibro;
Libro teuLibro;
```

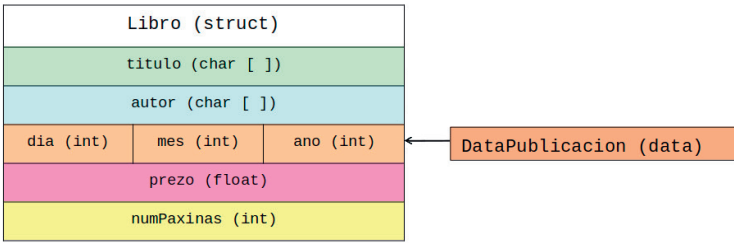



Figura 7.1: Representación da estrutura Libro

## 7.2 Unións

Unha **unión** é un **rexistro variable**, que contén soamente un dos seus membros á vez durante a execución do programa. Isto significa que os membros comparten o mesmo espazo en memoria.

Os membros dunha unión poden ser de calquera tipo, e poden conter dous ou máis tipos de datos diferentes.

Dado que durante a execución só se almacenará un tipo en memoria, o número de *bytes* empregado para almacenar a unión debe ser suficiente para almacenar o campo de maior tamaño.

Mentres que nunha estrutura a modificación dun membro non afecta aos restantes, nunha unión todos os membros ocupan o mesmo espazo, e só pode estar activo un nun momento determinado, polo que a modificación dun deles afectará os demais.

En C++, a declaración farase do seguinte modo:

```
union [NomeUnion] {
    tipo1 nomeMembro1;
    tipo2 nomeMembro2;
    tipo3 nomeMembro3;
    ..........
    tipoN nomeMembroN;
};

NomeUnion nomeVariable; // Declaración de variables
```

***Exemplo***: declaración dunha unión para obter a temperatura en graos Celsius, Fahrenheit e Kelvin en C++.

```
union Temperatura {
    float cel;
    float fah;
    float kel;
};

Temperatura temperaturaHoxe;
```

De forma similar ás estruturas, é posible traballar con unións aniñadas, estruturas con unións aniñadas, unións con estruturas aniñadas,...

## 7.3 Operacións

Cando se traballa con estruturas ou unións poderanse realizar determinadas operacións para o seu procesamento. Estas son operacións de acceso a cada un dos seus membros, operacións de asignación, e

operacións de lectura e escritura. Ademais, haberá moitas outras operacións que poderán facerse en función do tipo de variable e campo a campo: sumar (numéricos, por exemplo), comparar (numéricos ou cadeas),...

### 7.3.1 Acceso aos membros

Para acceder a cada un dos membros dunha estrutura ou unión en C++ emprégase o operador ".". Veremos máis adiante que, en caso de traballar con punteiros a estruturas, o operador empregado será "→".

### 7.3.2 Asignación

A asignación de valores a unha estrutura ou unión de xeito global soamente poderá facerse se se fai a outra estrutura ou unión do mesmo tipo, é dicir, cos mesmos membros.

Suporemos as estruturas **Data** e **Libro**, definidas xa anteriormente, e as variables **meuLibro**, **teuLibro**, onde **meuLibro** toma os seguintes valores (Figura 7.2).

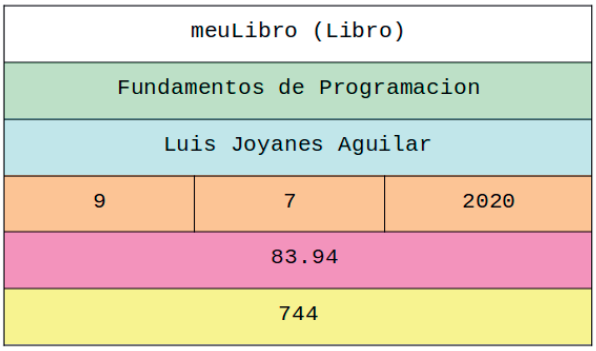



Figura 7.2: Representación da variable meuLibro

A instrución de asignación de xeito global virá dada por:

```
teuLibro = meuLibro;
```

Tras a asignación os valores dos membros de **teuLibro** son iguais aos de **meuLibro**.

Tamén é posible asignar cada membro de xeito individual, como calquera variable. Para iso, hai que acceder primeiro ao membro empregando o operador ".", e asignaráselle posteriormente un valor.

En C++, esta asignación farase do seguinte modo:

```
meuLibro.prezo = 59.8;
teuLibro.dataPublicacion.dia = 26; // Asignación ao membro de rexistro aniñado
```

En C++ é tamén posible asignar de forma global todos os campos dunha estrutura no momento da declaración:

```
Libro teuLibro = {"A praia dos afogados", "Domingo Villar", 20, 2, 2009, 21.85, 521};
```

### 7.3.3 Lectura e escritura

As operacións de lectura e escritura deben realizarse campo a campo. Continuando coas variables **meuLibro**, **teuLibro**, as operacións de lectura e escritura en C++ virán dadas por:

```
cin >> meuLibro.dataPublicacion.mes; // Lectura
cout << meuLibro.titulo;             // Escritura
```

## 7.4 Estruturas como parámetros

Ata o de agora, sempre que traballamos con parámetros de funcións e procedementos empregamos tipos de datos simples. Sen embargo, os parámetros dos subprogramas non se limitan a estes tipos de datos básicos, senón que é posible pasar como argumentos tipos de datos estruturados. Deste xeito, é posible o paso de estruturas como argumentos de función e procedementos. Ademais, poden ser parámetros de E, S ou E/S. No primeiro caso, pasaranse por valor, ao igual que sucede cos tipos de datos básicos, mentres que no caso de ser parámetros de S ou E/S, o paso será por referencia, de modo que accederemos dende o subprograma á dirección de memoria do parámetro real.

Cando se traballa con estruturas, aplícanse as mesmas regras que aos tipos fundamentais: as estruturas pasadas por valor non conservarán os seus cambios ao finalizar a execución da función ou procedemento. As estruturas pasadas por referencia conservarán os cambios feitos unha vez se devolva o control do fluxo ao programa principal, ao rematar a execución do subprograma.

Un aspecto que hai que considerar cando se traballa con estruturas é que as variables deste tipo poden chegar a ocupar unha cantidade considerable de memoria. Por iso, é frecuente utilizar o paso por referencia cando se envían como argumentos de subprogramas, aínda que non se vaian modificar os valores da estrutura.

***Exemplo***: supoñamos declarada unha estrutura `Persoa`, que conterá datos persoais acerca dunha persoa, un procedemento de lectura de datos `introducir (...)`, e outro de saída de datos por pantalla, `visualizar (...)`. Para introducir os datos dende o teclado pódese pasar a estrutura como parámetro por referencia do procedemento `introducir (...)`, e como parámetro por valor do procedemento `visualizar (...)` (Programa 7.1).

Programa 7.1: Estruturas como argumentos de funcións e procedementos

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

struct Persoa {
    char nome[100];
    char DNI[10];
    int idade;
};

void introducir (Persoa &p);
void visualizar (Persoa p);

int main ()
{
    Persoa unhaPersoa;
    introducir (unhaPersoa);
    visualizar (unhaPersoa);
    return 0;
}

void introducir (Persoa &p)
{
    cin >> p.nome;
    cin >> p.DNI;
    cin >> p.idade;
}
```

```
vod visualizar (Persoa p)
{
    cout << p.nome;
    cout << p.DNI;
    cout << p.idade;
}
```

## 7.5 Exercicios propostos

1. Supoñamos unha aplicación que pode identificar as persoas das dúas seguintes formas: 1) DNI (8 díxitos e letra), CIF (letra e 8 díxitos), pero soamente dunha de cada vez. Deseñar a estrutura de datos máis adecuada para elo.
2. Supoñamos unha persoa que posúe tres domicilios diferentes, en función da época do ano. Deseñar un rexistro que permita almacenar os seus datos persoais xunto a información referente ao domicilio actual nun momento determinado.
3. Nunha farmacia desexan informatizar o sistema de información acerca dos seus produtos, de forma que teñan almacenados, para cada un deles, os seguintes datos: nome, tipo de medicamento (analxésico, antipirético, antibiótico,...), código de barras, prezo, existencias e data de caducidade. Escribir a estrutura adecuada.
4. Supoñamos unha estrutura p de tipo Pelicula:

   ```
   struct Pelicula {
       char titulo[50];
       float duracion;
   };
   ```

   Como se realiza o acceso ao seu título?
5. Definir unha estrutura adecuada para almacenar as características dun vehículo: tipo, marca, número de rodas, e prezo. Engadir un campo que almacene o peso do coche, e que poderá almacenarse en quilogramos ou en toneladas (valores excluíntes). Escribir o código correspondente. Implementar un programa que lea os datos dun vehículo dende teclado e despois os almacene nunha variable de tipo vehículo.
6. Supoñamos que nunha clase determinada temos a seguinte información para cada materia: nome, curso, número de estudantes e estatística de cualificacións (máxima nota, mínima nota e media de cualificacións). Implementar un algoritmo que lea por teclado a información da materia de Programación I e a visualice por pantalla.
7. Que imprime o seguinte código?

   ```
   ...
   struct Persoa {
      char nome[50];
      int numOrde;
   };

   void actualizar (Persoa P);

   int main ()
   {
       Persoa Persoa1 = {"Persoa 1", 15};
       actualizar (Persoa1);
       cout << Persoa1.nome << ", " << Persoa1.numOrde << endl;
       return 0;
   ```

```
}

void actualizar (Persoa P)
{
    P.numOrde++;
}
```

8. Cal é a saída por pantalla ao executar as seguintes instrucións?

```
...
struct Data {
    int ano;
    int mes;
    int dia;
};
struct Cliente {
    char nome[50];
    char enderezo[100];
    Data alta;
};

int main ()
{
    Cliente cliente1 = {"Cliente 1", "ESEI- Campus As Lagoas", 2024, 10, 11};
    cout << cliente1;
    return 0;
}
```

9. Que valor se almacena en exemplo1.b ao remate da execución do seguinte código?

```
...
struct Exemplo {
    int a;
    char cadea[10];
    float b;
};

int main ()
{
    Exemplo exemplo1 = {5, "cadea11", 3.4};
    Exemplo exemplo2;
    exemplo1.a = 3;
    exemplo2.b = 5.7;
    exemplo1 = exemplo2;
    return 0;
}
```

10. Como debe ser o prototipo de calcularProduto, para que en prod se almacene o producto de a e b?

```
...
struct Fraccion {
    int num;
    int den;
};

int main ()
{
    Fraccion a = {2,5}, b = {3,6}, prod;
    calcularProduto (a, prod, b);
    return 0;
}
```

11. Cal é o resultado de executar o seguinte código?

```
...
struct Data {
    int ano;
    int mes;
    int dia;
};

void transferirData (Data &d);

int main ( )
{
    Data d = {2000, 12, 31};
    transferirData (d);
    cout << d.ano << ", " << d.mes << ", " << d.dia << endl;

    return 0;
}

void transferirData (Data &d) {
    d.ano = 2019;
    d.mes = 11;
    d.dia = 5;
}
```

12. Cal é a saída das seguintes instrucións?

```
...
struct Persoa {
   char nome[50];
   int numOrde;
};

int main ()
{
    Persoa P1 = {"Persoa 1", 15};
    Persoa P2 = {"Persoa 2", 20};
    (P1 == P2) ? cout << "Iguais" : cout << "Distintas" << endl;
    return 0;
}
```

13. Que se obtén ao executar o seguinte código?

```
...
struct Produto {
    char nome[40];
    int cantidade;
    float prezo;
};

void calcular1 (Produto p);
void calcular2 (Produto &p);

int main ()
{
    Produto p = {"Leituga", 5, 0.75};

    calcular1 (p);
    calcular2 (p);

    cout << p.nome <<", " << p.cantidade << ", " << p.prezo << endl;

    return 0;
}
```

```
void calcular1 (Produto p) {
    p.cantidade++;
    p.prezo /= 2;
}

void calcular2 (Produto &p) {
    p.cantidade++;
    p.prezo /= 2;
}
```

14. Deseñar a estrutura necesaria que permita definir un punto en coordenadas cartesianas. Implementar un algoritmo que, mediante as correspondentes funcións, lea dende teclado dous puntos e calcule a distancia entre eles.
15. Escribir a estrutura que permite almacenar a información referente aos empregados dunha empresa. Cómpre poder gardar información referente a nome, DNI, salario e data do contrato. Escribir un algoritmo que obteña o empregado máis antigo de dous empregados introducidos por teclado. Hai que utilizar funcións para introducir os datos e para calcular a antigüidade.
16. Engadir ao exercicio anterior unha función que calcule o salario medio dos dous empregados introducidos por teclado.
17. Escribir un programa que lea dúas fichas de dous estudantes, codificadas como estruturas, cos datos nome, apelidos, DNI, idade, e ano de apertura de expediente, e decida se os datos idade e ano de apertura de expedientes que aparecen na primeira ficha cadran cos da segunda. Débense implementar funcións para a lectura de datos e para a comprobación.
18. Escribir tres rexistros diferentes para almacenar as variables necesarias para definir un trapecio, un triángulo e un rectángulo. Deseñar tamén un rexistro que permita almacenar unha figura xeométrica xenérica. Escribir un programa que solicite por teclado o tipo de figura que se desexa introducir e, a partir do valor introducido e as variables que definen a figura xeométrica, calcule o seu perímetro e amose o resultado por pantalla. É necesario empregar funcións para todas as operacións.

# Capítulo 8

# Arrais

No tema anterior vimos que, ademais dos tipos de datos simples, existen tamén os tipos de datos estruturados, definidos como agrupacións de datos simples, que poden ser do mesmo tipo ou non, con relacións entre eles. Estudáronse xa os rexistros, formados por membros que poden ser de distinto tipo, e que son estruturas estáticas heteroxéneas.

Neste tema estudaremos os arrais, outro tipo de estrutura de datos estática, pero con características diferentes ás das estruturas e a s unións.

## 8.1 Definición

Defínese un **arrai** como un conxunto finito e ordenado de datos homoxéneos, cada un dos cales é denominado elemento. Ao ser homoxéneo, todos os elementos son do mesmo tipo (carácter, enteiro, real, estrutura,...), e gárdanse en posicións sucesivas de memoria. Ademais, cada un deles pode ser identificado mediante unha orde: primeiro, segundo.... Polo tanto, é posible o acceso individual a cada elemento do arrai.

Dado que son estruturas estáticas, o seu tamaño debe ser especificado en tempo de compilación (adóitase utilizar unha constante simbólica enteira positiva), e non pode variar durante a mesma.

Tanto os rexistros como os arrais son tipos de datos estruturados. Sen embargo, existe unha diferenza fundamental entre os dous. Mentres que un arrai é unha estrutura de datos homoxénea, un rexistros pode tener compoñentes de diferentes tipos de datos.

Ademais, poderanse definir arrais de rexistros, onde cada elemento do arrai é unha estrutura ou unión, e á inversa, rexistros onde os seus compoñentes poden ser arrais.

Dependendo da forma na que se declaran ou constrúen, existen diferentes tipos de arrais: vectores, matrices, e arrais en máis de dúas dimensións.

## 8.2 Vectores

Un **vector** é un arrai nunha dimensión. Pódese acceder de forma directa a cada un dos seus elementos unicamente cun índice. Ademais, cada un destes elementos procésase como unha variable simple, que ocupa unha posición de memoria.

Defínese o **subíndice** ou índice dun elemento como a posición do elemento na ordenación do arrai. Xeneralmente toma valores de 0 a N-1 (N: tamaño do arrai). O primeiro elemento correspóndese co subíndice 0, e o último co subíndice N-1. O **límite inferior** (**superior**) é o valor mínimo (máximo) permitido

do arrai. Finalmente, o **rango** é o número de elementos que conforman o arrai.

### 8.2.1 Almacenamento

Os vectores almacénanse en posicións consecutivas de memoria. Para un elemento calquera `i`, se consideramos que o límite inferior = 0, a posición vén dada por:

$$B + i * S$$

Onde B é a dirección de inicio da memoria, e S é o tamaño (número de *bytes*) en memoria que ocupa cada dato do tipo básico que forma parte do vector.

O elemento `i` dun vector `v` represéntase como `v[i]`, tendo en conta que o primeiro elemento é `v[0]`, e o último `v[N-1]` (N é o tamaño do arrai).

A Figura 8.1 amosa a representación gráfica do almacenamento en memoria dun vector.

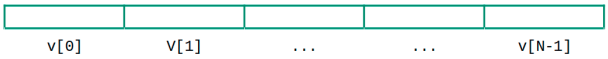



Figura 8.1: Vector en memoria

Cando escribimos un programa non temos que preocuparnos polo almacenamento en memoria dun vector, xa que todas as linguaxes de programación proporcionan métodos para iso.

Todos os vectores indéxanse dende 0. Pola súa banda, os índices pódense expresar mediante unha constante enteira, unha variable enteira ou unha expresión enteira máis complexa.

### 8.2.2 Declaración

De forma similar a calquera outro tipo de dato, os vectores deben ser declarados na zona correspondente dos algoritmos e programas, como as demais variables.

En C++, a declaración farase do seguinte modo:

```
tipo nomeVector [TAM]; // TAM: número de elementos do vector
```

En C++ non se comproba se os índices dos vectores están dentro dos rangos definidos. Noutras linguaxes si se pode realizar esta comprobación, o que pode ralentizar a execución dos programas. En C++, se durante a execución algún índice sae do rango, pode producirse un erro de segmentación ou non, pero en calquera caso o programa non funcionará correctamente, porque corremos o risco de sobrescribir calquera outra variable do programa. Este tipo de erros é o máis común en programación, e o programa non indica onde se produce.

Para acceder a cada un dos elementos do vector, utilízase o **operador de indexación** `[]`, de xeito que para acceder ao elemento `i`, escribirase `v[i]`.

***Exemplo***: declaración de vectores de tipo enteiro e real en C++.

```
float v1[5];
int v2[4];
```

### 8.2.3 Operacións

As operacións con vectores implican o procesamento dos elementos individuais do vector. Estudaremos asignación e inicialización, lectura/escritura, percorrido e actualización. Hai tamén operacións de ordenación e busca, pero non se tratarán nesta materia.

### Asignación e inicialización

Realízanse coas correspondentes instrucións de asignación e inicialización, xa coñecidas. Suporemos declarado un `vector v[TAM]` (o tipo é indiferente).

Para asignar a un elemento `v[i]` un valor emprégase:

```
v[i] = valorAsignado;
```

Para asignar valores a todos os elementos do vector cómpre utilizar unha estrutura repetitiva (**for**, **while**, **do/while**). Como exemplo, farémolo coa estrutura **for**:

```
for (i = 0; i < TAM; i++) {
    v[i] = valorAsignado;
}
```

En C++ é posible tamén inicializar o vector no momento da declaración:

```
tipo v[TAM] = {v1, v2, v3,... vTAM-1};
```

Pódense declarar tamén vectores indeterminados, sempre e cando se inicialicen no momento da súa declaración. O tamaño asignado cadrará co número de elementos asignados.

```
tipo v[] = {v1, v2, v3,..., vTAM-1};
tipo v[] = {v1, v2};
```

***Exemplo***: asignación de elementos de vectores en C++.

```
const int TAM = 10;
int v[TAM];
int v2[5] = {1, 2, 3, 4, 5};
int v3[] = {6, 7,8};
int i;
........
for (i = 0; i < TAM; i++) {
    v[i] = i * 2;
}
```

### Lectura e escritura

Realízanse con as correspondentes instrucións de lectura e escritura, xa coñecidas. Suporemos declarado un `vector v[TAM]` (o tipo é indiferente).

En C++, para ler/escribir o contido dun elemento i emprégase:

```
cin >> v[i];
cout << v[i];
```

Para ler/escribir os valores de todos os elementos do vector cómpre utilizar unha estrutura repetitiva (**for**, **while**, **do/while**). Como exemplo, farémolo con for:

```
for (i = 0; i < TAM; i++) {
    cin >> v[i];
    cout << v[i];
}
```

***Exemplo***: lectura e escritura de elementos de vectores en C++.

```
const int TAM = 10;
int v[TAM];
int i;
........
```

```
for (i = 0; i < TAM; i++) {
    cin >> v[i];
    cout << v[i];
}
```

### Percorrido

Para percorrer un vector empréganse estruturas repetitivas. As variables de control do bucle utilízanse como subíndices do vector. As instrucións de percorrido para cada estrutura iterativa veñen dadas por:

```
const int TAM = tamVector;
tipo v[TAM];
int i;
........

// Estrutura for:
for (i = 0; i < TAM; i++) {
    operar con v[i];
    ........
}

// Estrutura while:
i = 0;
while (i < TAM) {
    operar con v[i];
    ........
i++;
}

// Estrutura do-while:
i = 0;
do {
    operar con v[i];
    ........
    i++;
} while (i < TAM);
```

***Exemplo***: visualización de posicións pares de elementos de vectores en C++.

```
const int TAM = 10;
int v[TAM];
int i;
........
i = 0;
while (i < TAM) {
    if (i % 2 == 0) {
        cout << v[i];
    }
    i++;
}
```

### Actualización

Inclúe as operacións de engadir, inserir e borrar.

#### Engadir

Consiste en engadir un novo elemento ao final do vector. Unicamente será necesario comprobar se hai suficiente memoria para o novo elemento. É dicir, só haberá que verificar se aínda non están definidos todos os elementos do arrai, e inserir o elemento na primeira posición libre do mesmo.

```
const int TAM = tamVector;
tipo v[TAM];
int i;
int numElementos;          // Número elementos definidos en v (< TAM)
........
if (numElementos < TAM) {
    v [numElementos] = valorAsignado;
    numElementos++;
}
```

***Exemplo***: engádese un elemento con valor 35 a un vector de 10 elementos enteiros, con 7 posicións ocupadas, en C++.

```
const int TAM = 10;
int v[TAM];
int i;
int numElementos = 7;
........
if (numElementos < TAM) {
    v[numElementos] = 35;
    numElementos++;
}
```

**Inserir**

Consiste en inserir un novo elemento dentro do vector. Para iso será necesario realizar un desprazamento dalgúns elementos, para inserir o novo na posición indicada (supondo que hai espazo no vector).

```
const int TAM = tamVector;
tipo v[TAM];
int i;
int numElementos;       // Número elementos definidos en v (< TAM)
int pos;                // Posición do elemento a inserir (< numElementos)
........
if (numElementos < TAM) {
    cin >> pos;
    for (i = numElementos - 1; i >= pos; i--) {
        v[i+1] = v[i];  // Desprazamento
    }
    cin >> v[pos];
    numElementos++;
}
```

***Exemplo***: insírese un elemento con valor 8 na posición 5 dun vector de 10 elementos, con 7 posicións ocupadas, en C++.

```
const int TAM = 10;
int v[TAM];
int i;
int numElementos = 7;
int pos;
........
if (numElementos < TAM) {
    cin >> pos;          // 5
    for (i = numElementos - 1; i >= pos; i--) {
        v[i+1] = v[i];
    }
    cin >> v[pos];       // 8
    numElementos++;
}
```

**Borrar**

Consiste en eliminar un elemento dentro do vector. Esta operación poderase realizar sempre e cando haxa elementos definidos no vector. Cómpre realizar un desprazamento en sentido contrario á inserción.

```
const int TAM = tamVector;
tipo v[TAM];
int i;
int numElementos;       // Número elementos definidos en v (<= TAM)
int pos;                // Posición do elemento a borrar
........
if (numElementos > 0) {
    cin >> pos;
    for (i = pos + 1; i < numElementos; i++) {
        v[i-1] = v[i]; // Desprazamento
    }
    numElementos--;
}
```

***Exemplo***: elimínase a posición 5 dun vector de 10 elementos, con 7 posicións ocupadas, en C++.

```
const int TAM = 10;
int v[TAM];
int i;
int numElementos = 7;
int pos;
........
if (numElementos > 0) {
    cin >> pos;       // 5
    for (i = pos + 1; i < numElementos; i++) {
        v[i-1] = v[i];
    }
    numElementos--;
}
```

## 8.3 Matrices

Unha **matriz** é un arrai bidimensional, isto é, un vector de vectores. Á súa vez, cada elemento do vector será un vector. Dado que é un tipo de datos homoxéneo, todos os seus elementos son do mesmo tipo. Ademais, a orde de cada un deles é significativa.

Para identificar cada elemento dunha matriz son necesarios dous índices. Unha matriz posúe dous dimensións (unha por cada subíndice). A primeira delas, representada polo primeiro subíndice, identifica as filas, e a segunda (segundo subíndice) as columnas (Figura 8.2).

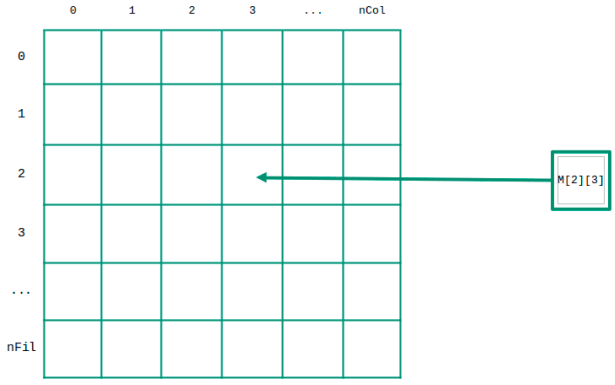



Figura 8.2: Representación dunha matriz

Supondo unha matriz M de NFIL filas (0.. NFIL-1) e NCOL columnas (0..NCOL-1), o seu tamaño (número máximo de elementos) será NFIL * NCOL.

### 8.3.1 Almacenamento

Aínda que a matriz se represente como unha táboa con dúas dimensións, realmente a memoria dunha computadora é lineal. Por iso, as matrices deben están linealizadas para o seu almacenamento en memoria. Isto pode facerse de dúas formas diferentes:

- **Orde de fila maior**: vanse almacenando os elementos de cada unha das filas, sucesivamente (Figura 8.3).

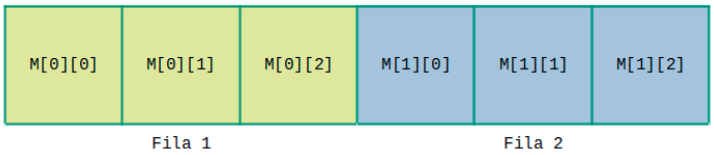



Figura 8.3: Orde de fila maior para unha matriz M[2][3]

- **Orde de columna maior**: vanse almacenando os elementos de cada unha das columnas, sucesivamente (Figura 8.4).

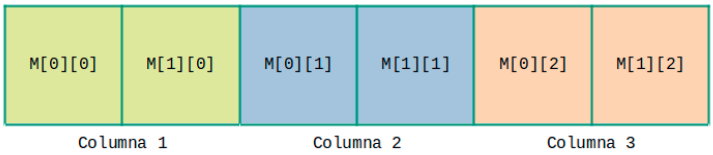



Figura 8.4: Orde de columna maior para unha matriz M[2][3]

O elemento `i` dunha matriz `m` represéntase como `m[i][j]` en C++. O primeiro elemento é `m[0][0]`, e o último `m[NFIL-1][NCOL-1]`, considerando de novo como NFIL o número de filas, e NCOL o de columnas.

Todas as matrices indéxanse dende 0 nas dúas dimensións. Os índices pódense expresar mediante constante enteira, variable enteira o expresión enteira máis complexa.

### 8.3.2 Declaración

De forma similar a calquera outro tipo de dato, as matrices deben ser declaradas na zona correspondente dos algoritmos e programas, como as demais variables.

En C++, a declaración farase do seguinte modo:

```
tipo nomeMatriz [NFIL][NCOL];
```

En C++ non se comproba se os índices das matrices (dimensións) están dentro dos rangos definidos.

Para acceder a cada un dos elementos da matriz, utilízase o operador `[]`, de xeito que para acceder ao elemento na posición `i`, `j`, escribirase `m[i][j]`.

***Exemplo***: declaración de matrices de tipo enteiro e real en C++.

```
int m1[5][5];
int m2[3][4];
```

### 8.3.3 Operacións

As operacións con matrices implican o procesamento ou tratamento dos elementos individuais da matriz, o ben tratamento por filas o por columnas. Estudaremos: asignación e inicialización, lectura/escritura, percorrido, actualización. Hai tamén operacións de ordenación e busca, pero non se tratarán neste libro.

#### Asignación e inicialización

Realízanse coas correspondentes instrucións de asignación e inicialización, xa coñecidas. Suporemos declarada unha matriz `m[NFIL][NCOL]` (o tipo é indiferente).

Para asignar a un elemento `m[i][j]` un valor emprégase:

```
m[i][j] = valorAsignado;
```

Para asignar valores a todos os elementos da matriz cómpre utilizar unha estrutura repetitiva (`for`, `while`, `do/while`). Como exemplo, farémolo coa estrutura `for`:

```
for (i = 0; i < NFIL; i++) {
    for (j = 0; j < NCOL; j++) {
        m[i][j] = valorAsignado;
    }
}
```

En C++ é posible tamén inicializar a matriz no momento da declaración:

`tipo m[NFIL][NCOL]` = $\{v_{00}, v_{01}, v_{02}, \ldots v_{0,NCOL-1}, v_{10}, v_{11}, v_{12} \ldots v_{1,NCOL-1} \ldots v_{NFIL-1,0}, v_{NFIL-1,1}, v_{NFIL-1,2}, \ldots v_{NFIL-1,NCOL-1}\}$;

Cando estamos a traballar con C++, podemos declarar tamén matrices que teñan unha dimensión indeterminada, sempre e cando as devanditas matrices se inicialicen no momento da súa declaración.

Neste caso, o tamaño que terá a matriz deberá cadrar co número de elementos asignados. Farase do seguinte xeito:

`tipo m[][NCOL]` = $\{\{v_{00}, \ldots v_{0,NCOL-1}\}, \ldots \{v_{NFIL-1,0}, \ldots v_{NFIL-1,NCOL-1}\}\}$;

***Exemplo***: asignación de elementos de matrices en C++.

```
const int NFIL = 5;
const int NCOL = 5;
int m[NFIL][NCOL];
int i, j;
........
for (i = 0; i < NFIL; i++) {
    for (j = 0; j < NCOL; j++) {
        m[i][j] = i + j;
    }
}
```

#### Lectura e escritura

Realízanse coas correspondentes instrucións de lectura e escritura, xa coñecidas. Suporemos declarada unha matriz `m[NFIL][NCOL]` (o tipo é indiferente).

En C++, para ler/escribir o contido dun elemento i,j emprégase:

```
cin >> m[i][j];
cout << m[i][j];
```

Para ler/escribir os valores de todos os elementos da matriz cómpre utilizar unha estrutura repetitiva (`for`, `while`, `do/while`). Como exemplo, farémolo con `for`:

```
for (i = 0; i < NFIL; i++) {
    for (j = 0; j < NCOL; j++) {
        cin >> m[i][j];
        cout << m[i][j];
    }
}
```

***Exemplo***: lectura e escritura de elementos de matrices en C++.

```
const int NFIL = 5;
const int NCOL = 5;
int m[NFIL][NCOL];
int i, j;
........
for (i = 0; i < NFIL; i++) {
    for (j = 0; j < NCOL; j++) {
        cin >> m[i]][j];
        cout << m[i][j];
    }
}
```

### Percorrido

Para percorrer unha matriz, empréganse dobres estruturas repetitivas, para percorrer cada fila e cada unha da columnas. As variables de control dos bucles empréganse como subíndices da matriz. Escribiremos a continuación as instrucións de percorrido por filas e por columnas para a instrución for. As sentenzas con `while` e `do/while` serían similares.

```
const int NFIL = numFilas;      // numFilas indica o número de filas da matriz
const int NCOL = numColumnas;   // numColumnas indica o número de columnas da matriz

tipo m[NFIL][NCOL];
int i, j;
........

// Percorrido por filas:
for (i = 0; i < NFIL; i++) {
    for (j = 0; j < NCOL; j++) {
        operar con m[i][j];
        ........
    }
}

// Percorrido por columnas:
for (j = 0; j < NCOL; j++) {
    for (i = 0; i < NFIL; i++) {
        operar con m[i][j];
        ........
    }
}
```

***Exemplo***: visualización de posicións de índices pares de elementos de matrices en C++.

```
const int NFIL = 5;
const int NCOL = 5;
```

```
int m[NFIL][NCOL];
int i, j;
........
for (i = 0; i < NFIL; i++) {
    for (j = 0; j < NCOL; j++) {
        if ((i % 2 == 0) && (j % 2 == 0)) {
            cout << m[i][j];
        }
    }
}
```

### Actualización

Inclúe as operacións de inserir e borrar.

**Inserir**

Consiste en inserir unha fila ou unha columna dentro da matriz. Para iso será necesario realizar un desprazamento dalgunhas filas ou columnas, para inserir a nova na posición indicada (supondo que hai espazo na matriz).

```
// Inserir unha fila
    const int NFIL = numFilas;          // numFilas indica o número de filas da matriz
    const int NCOL = numColumnas;       // numColumnas indica o número de columnas da matriz

    tipo m[NFIL][NCOL];
    int i, j;
    int nFilas;                         // Número de filas definidas en m
    int nColumnas;                      // Número de columnas definidas en m
    int pos;                            // Posición da fila a inserir (< nFilas)

    ........
    if (nFilas < NFIL) {
        cin >> pos;
        for (i = nFilas - 1; i >= pos; i--) { // Percorrido por filas
            for (j = 0; j < nColumnas; j++) {
                m[i+1][j] = m[i][j];    // Desprazamento
            }
        }
        for (j = 0; j < nColumnas; j++) {
             cin >> m[pos][j];
        }
        nFilas++;
}

// Inserir unha columna
    const int NFIL = numFilas;
    const int NCOL = numColumnas;

    tipo m[NFIL][NCOL];
    int i, j;
    int nFilas;                          // Número de filas definidas en m
    int nColumnas;                       // Número de columnas definidas en m
    int pos;                             // Posición da columna a inserir (< nColumnas)

    ........
    if (nColumnas < NCOL) {
        cin >> pos;
        for (j = nColumnas - 1; j >= pos; j--) { // Percorrido por columnas
            for (i = 0; i < nFilas; i++) {
               m[i][j+1] = m[i][j];     // Desprazamento
            }
```

```
        }
        for (i = 0; i < nFilas; i++) {
            cin >> m[i][pos];
        }
        nColumnas++;
}
```

***Exemplo***: insírese unha fila na posición 1 dunha matriz de 5x5, con 3 filas e 5 columnas xa definidas, en C++.

```
    const int NFIL = 5;
    const int NCOL = 5;

    int m[NFIL][NCOL];
    int i, j;
    int nFilas;                     // Número de filas definidas en m
    int nColumnas;                  // Número de columnas definidas en m
    int pos;                        // Posición da fila a inserir (< nFilas)

    ........
    if (nFilas < NFIL) {
        cin >> pos;                 // 1
        for (i = nFilas - 1; i >= pos; i--) {
            for (j = 0; j < nColumnas; j++) {
                m[i+1][j] = m[i][j];
            }
        }
        for (j = 0; j < nColumnas; j++) {
             cin >> m[pos][j];
        }
        nFilas++;
}
```

**Borrar**

Consiste en eliminar unha fila ou columna dentro dunha matriz. Esta operación poderase realizar sempre e cando haxa filas ou columnas definidas na matriz. Cómpre realizar un desprazamento en sentido contrario á inserción.

```
//Borrar unha fila
    const int NFIL = numFilas;          //numFilas indica o número de filas da matriz
    const int NCOL = numColumnas;       //numColumnas indica o número de columnas da matriz

    tipo m[NFIL][NCOL];
    int i, j;
    int nFilas;                         // Número de filas definidas en m
    int nColumnas;                      // Número de columnas definidas en m
    int pos;                            // Posición da fila a eliminar (< nFilas)

    ........
    if (nFilas > 0) {
        cin >> pos;
        for (i = pos + 1; i < nFilas; i++) { // Percorrido por filas
            for (j = 0; j < nColumnas; j++) {
                m[i-1][j] = m[i][j];    // Desprazamento
            }
        }
        nFilas--;
}

//Borrar unha columna
    const int NFIL = numFilas;
    const int NCOL = numColumnas;
```

```
    tipo m[NFIL][NCOL];
    int i, j;
    int nFilas;                          // Número de filas definidas en m
    int nColumnas;                       // Número de columnas definidas en m
    int pos;                             // Posición da columna a eliminar (< nColumnas)

    ........
    if (nColumnas < 0) {
        cin >> pos;
        for (j = pos + 1; j < nColumnas; j++) { // Percorrido por columnas
            for (i = 0; i < nFilas; i++) {
              m[i][j-1] = m[i][j];       //Desprazamento
            }
        }
        nColumnas--;
}
```

***Exemplo***: elimínase a columna na posición 2 dunha matriz de 5x5 de enteiros, con 3 filas e 5 columnas xa definidas, en C++.

```
    const int NFIL = 5;
    const int NCOL = 5;

    int m[NFIL][NCOL];
    int i, j;
    int nFilas;                     // Número de filas definidas en m
    int nColumnas;                  // Número de columnas definidas en m
    int pos;                        // Posición da fila a inserir (< nFilas)

    ........
    if (nColumnas < 0) {
        cin >> pos;                 // 2
        for (j = pos + 1; j < nColumnas; j++) {
            for (i = 0; i < nFilas; i++) {
               m[i][j-1] = m[i][j];
            }
        }
        nColumnas++;
}
```

## 8.4 Arrais multidimensionais

Ata o de agora traballamos con arrais dunha e dúas dimensións. Pero tamén é posible definir arrais en N dimensións. Para poder acceder a cada un dos elementos do arrai serán necesarios N subíndices, un por cada dimensión.

Os arrais multidimensionais tamén deben ser declarados na zona correspondente dos algoritmos e programas, como as demais variables.

En C++ a declaración farase do seguinte modo:

```
tipo nomearrai [dim1][dim2]...[dimN];
```

En C++, cando se declaran e inicializan á vez os arrais indeterminados, é necesario especificar todas as dimensións excepto a primeira.

```
tipo nomearrai [][dim2]...[dimN] = {v1, v2, v3,...}
```

De forma similar aos vectores e ás matrices, para acceder a cada un dos elementos do arrai multidimensional empregamos o operador `[]`, de forma que para acceder ao elemento na posición `i`, `j`, `k`,..., dun arrai multidimensional `a`, escribirase `a[i][j][k]`....

As operacións de asignación e inicialización fanse de xeito similar aos casos xa estudados, mentres que para o percorrido destes arrais comprirá incluír unha estrutura iterativa por cada dimensión. A inserción e borrado pódense realizar tamén por cada unha das dimensións.

***Exemplo***: declaración de arrais multidimensionais enteiros, en C++.

```
int a1[5][5][5][5];
int a2[][2][2] = {1, 2, 3, 4, 1, 2, 3, 4};
```

Finalmente, xa se indicou ao principio do tema que se pode traballar con arrais de rexistros, e que estes poden ter como membro un arrai. Isto permite manter a integridade dos datos e mellorar a organización.

A continuación amosamos un exemplo de cada un destes casos.

***Exemplo***: arrai de estruturas en C++.

```
........
struct Libro {
    char titulo[100];
    char autor[100];
    float prezo;
    int numPaxinas;
};
........
int main ()
{
    Libro setLibros[10];
    cin >> setLibros[0].prezo;
    cout << setLibros[3].titulo;
    return 0;
}
```

***Exemplo***: arrai como membro dunha estrutura en C++.

```
........
struct Estudante {
    char nome[100];
    int curso;
    float notas [4];
};
........
int main ()
{
    Estudante Pepe;
    Estudante Xoana;

    cin >> Pepe.curso;
    cout << Xoana.notas[0];
    return 0;
}
```

## 8.5 Arrais como parámetros

Os arrais son un tipo de dato estruturado que pode ser pasado como argumento de funcións e/ou procedementos. Para iso, é necesario declarar previamente o arrai tal e como se viu, e como parámetro da función. Poderá ser de E, S, o E/S.

Cando se invoca a función, farase simplemente co nome do arrai, sen especificar corchetes nin índices.

Para un vector pasado como argumento (para arrais en dúas ou máis dimensións sería similar) a declaración farase do seguinte modo:

```
...
tipo  func (tipo v[]);

int main ()
{
    tipo retorno;
    tipo ta[TAM];
    retorno = func (ta);
    return 0;
}

tipo func (tipo ta[]) {
    tipo datoRetorno;
    instrucions;
    return datoRetorno;
}
```

Na declaración da función, no caso de traballar con vectores, non será necesario especificar o tamaño, polo que os corchetes aparecerán baleiros. Hai que ter en conta tamén que en C++ un arrai sempre se pasa por referencia (non é necesario poñer o operador &). Polo tanto, accederase dende o subprograma directamente á posición de comezo do arrai, e calquera modificación que se faga sobre este permanecerá ao rematar a execución e devolver o control do fluxo ao programa principal.

Pódese observar a chamada á función desde o programa principal: `func (ta)`. Non se especifica o tamaño do arrai, nin se empregan corchetes. Se, por exemplo, escribimos `ta[i]`, onde i é un enteiro positivo entre 0 e TAM-1, unicamente estamos a pasar como argumento un único elemento do arrai, non o arrai no seus conxunto. A chamada dende un procedemento sería idéntica.

***Exemplo***: paso de arrais como parámetros de funcións en C++ (Programa 8.1).

Programa 8.1: Paso de arrais como parámetros en C++

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

#define TAM 5
float calcularMedia (float datos[]);

int main ()
{
   float datos[TAM] = {2.8, 5.6, 4.3, 9.7, 1.2};
   float media;
   media = calcularMedia (datos);
   return 0;
}

float calcularMedia (float datos[])
{
   float suma = 0.0;
   int i;
   for (i = 0; i < TAM; i++) {
      suma += datos[i];
   }
   return (suma/TAM);
}
```

## 8.6 Exercicios propostos

1. Que imprime o seguinte código?

```
...
int main ()
{
    int s = 0;
    int v[5] = {1, 2, 3, 4, 5};

    for (int i = 5; i > 0; i--) {
        s += v[i];
    }
    cout << s;
    return 0;
}
```

2. Supoñamos `float v[10]`, con `numEl = 7` elementos definidos. Permiten as seguintes instrucións agregar sempre outro elemento nunha posición pos $< 10$?

```
...
if (numEl > 10) {
    for (i = numEl - 1; i >= pos; i--) {
        v[i+1] = v[i];
    }
    cin >> v[pos];
    numEl++;
}
```

3. Que saca por pantalla o seguinte código?

```
...
int main ()
{
    int x[5] = {9, 4, 7 , 6, 4};
    int suma = 0;
    int i;

    for (i = 0 ; i < 5 ; i++) {
        x[i] = x[i] + i;
    }

    for (i = 1 ; i < 4 ; i++) {
        suma += x[i];
    }

    cout << suma << endl;
    return 0;
}
```

4. Cal é o resultado de executar as seguintes instrucións?

```
...
int main()
{
    int v[4], i = 0;

    while (i < 4) {
        v[i] = i + 2;
        i++;
    }

    for (i = 0; i <= 4; i++) {
        cout << v[i] << endl;
    }
    return 0;
}
```

5. Supoñamos `int tam = 5; int v1[] = {1, 2, 3, 4, 5}, v2[5];`. Que fai o seguinte fragmento de código?:

```
...
for (int i = tam - 1; i >= 0; i--) {
    v2[i] = v1[tam - 1 - i];
}
```

6. Cal é a saída das seguintes instrucións?

```
...
int main ()
{
    int v1[5] = {1, 3, 5, 7, 9};
    int v2[5] = {2, 4, 6, 8, 10};

    for (int i = 0; i < 5; i++) {
        cout << v2[v1[i]] << ", ";
    }
    return 0;
}
```

7. Cal é o resultado de executar o seguinte programa?

```
...
int main ()
{
    int k = 1, v[5], suma = 0;

    for (int i = 0 ; i < 5 ; i++ ) {
        suma = i + ++k;
        v[i] = suma + ++k;
    }
    cout << v[3];
    return 0;
}
```

8. Escribir un programa que lea un arrai de 5 números reais e calcule a suma dos elementos negativos.
9. Implementar o código para verificar se un vector de 10 enteiros posúe o 0 entre os seus elementos.
10. Dispoñemos de N temperaturas almacenadas nun arrai. Desexamos calcular a media de todas elas e obter o número de temperaturas maiores ou iguais que a media. Escribir o código correspondente.
11. Que obtemos ao executar o seguinte trozo de código?

```
...
int m[5][5];

for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++) {
        m[i][j] = 5;
        cout << m[i][j] << endl;
    }
}
```

12. Que se imprime por pantalla ao executar as seguintes instrucións?

```
...
int main ()
{
    int m1[3][3] = {{1, 1, 1}, {2, 2, 2}, {3, 3, 3}};
    int m2[3][3];

    for (int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++) {
```

```
            if (m1[i][j] % 2 == 0) {
                m2[i][j] = 0;
            }
            else {
                m2[i][j] = 1;
            }
        }
    }
    cout << m2[0][0] << " " << m2[1][1] << " " << m2[2][2];
    return 0;
}
```

13. Que saca por pantalla o seguinte código?

```
...
int main ()
{
    int x[5][2] = {{9, 8}, {4, 5}, {7, 3} , {6, 2}, {4, 1}};
    int suma = 0;

    for (int i = 0 ; i < 5 ; i++) {
        for (int j = 0 ; j < 2 ; j++) {
            x[i][j] = x[i][j] + i + j;
        }
    }

    for (int i = 0 ; i < 4 ; i++) {
        for (int j = 0 ; j < 1 ; j++) {
            suma += x[i][j];
         }
    }
    cout << suma << endl;
    return 0;
}
```

14. Codificar un algoritmo que calcule o número de valores menores de 1000 dunha matriz de números reais, de 10 filas e 10 columnas.

15. Escribir un programa que lea dende o teclado e visualice por pantalla os elementos dunha matriz de 3 x 3.

16. Escribir as instrucións necesarias para sumar todos os elementos dunha matriz de números reais de 4 filas e 5 columnas.

17. Escribir un programa que pida por teclado os datos dunha matriz de 3 x 3, e conteña un subprograma que amose a trasposta.

18. Que se almacena na variable v tras a execución das seguintes instrucións?

```
int f (int v[][5]);

int main ()
{
    int v, v1[2][5] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    v = f (v1);
    return 0;
}

int f(int v[][5]) {
    ...
}
```

19. Escribir un programa que sume dúas matrices cuxos elementos foron previamente introducidos por teclado. O programa debe conter tres funcións: unha para ler cada elemento de cada matriz por teclado, outra para sumar as dúas matrices, e unha terceira para visualizar o resultado.
20. Implementar un procedemento que tome como entrada unha matriz de números enteiros, de 4 x 4, e devolva a mesma matriz multiplicada por 3.
21. Codificar coa estrutura de datos máis adecuada o nome, apelidos, DNI e código numérico (tres díxitos) dun máximo de 100 clientes. Implementar un programa que, mediante funcións, lea do teclado o número de clientes nun momento determinado, e a información correspondente a cada un deles, e os visualice por pantalla.
22. Supoñamos un arrai de 10 valores enteiros. Implementar subprogramas recursivos para: a) introducir os valores dos elementos por teclado, b) calcular o valor máximo de todos eles, c) buscar un valor determinado no arrai e devolver a súa posición nel, d) sumar todos os elementos do arrai, e e) visualizar o arrai ao revés.

# Capítulo 9

# Ficheiros

## 9.1 Definición

Ata o de agora, estivemos traballando con programas que manexan e almacenan datos na memoria principal da computadora. Pero o almacenamento en variables, estruturas, arrais,..., é temporal, de xeito que se perde o seu valor ben cando saímos do ámbito de influencia do dato, ben cando remata a execución do programa. Isto non é suficiente coa gran maioría das aplicacións; en moitas ocasións cómpre almacenar a información que se manexa de forma permanente noutros dispositivos de memoria para un posterior procesamento. Estas coleccións de datos almacenadas son os **ficheiros**, e os datos neles almacenados son os **datos persistentes**, que permanecen despois da execución do programa.

Un **arquivo**, polo tanto, relaciónase co almacenamento permanente de datos, e permite fraccionar grandes volumes de datos en unidades máis pequenas que poden ser almacenadas na memoria central e procesadas por un programa. Posúe un formato predefinido (xeralmente elixido por quen programa), e o seu contido está accesible para lectura e/ou escritura conforme a unhas pautas preestablecidas, de acordo co seu formato.

De modo xeral, un **arquivo** ou **ficheiro** é un conxunto de datos estruturados nunha colección de entidades elementais ou básicas, denominadas **rexistros**, que son de igual tipo e constan de diferentes entidades de nivel máis baixo, denominadas **campos** (Figura 9.1).


Figura 9.1: Representación dun arquivo

Os rexistros que forman o ficheiro están relacionados entre si, teñen un aspecto común, e organízanse cun propósito específico, de tal forma que poidan ser recuperados con facilidade, actualizados, borrados, ou almacenados de novo no ficheiro.

Finalmente, un grupo de ficheiros relacionados é unha **base de datos**.

***Exemplo***: campos dun rexistro que almacenan información sobre os artigos dun almacén. Para cada artigo almacénase o nome, o número de unidades existentes, o prezo e a data de entrada (Figura 9.2).

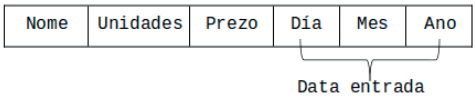



Figura 9.2: Exemplo de campos dun rexistro

Vemos así que un **rexistro** é unha colección de información, un conxunto de elementos loxicamente relacionados. Pode ser de lonxitude fixa ou variable. É un concepto similar ao da estrutura que xa estudamos. Pola súa banda, un **campo** é un ítem de datos elementais, caracterizado polo seu tamaño e tipo de datos, e pode incluso variar en lonxitude. É a unidade mínima de información dun rexistro.

Centrándonos xa en C++, denomínase **ficheiro** a cada dispositivo físico, mentres que o canal de comunicación entre cada dispositivo e o programa é un **fluxo** (***stream***). Cada arquivo visualízase como un fluxo secuencial de *bytes*: na entrada, un programa extrae *bytes* dun fluxo de entrada, e na saída insire *bytes* no fluxo de saída. Todas as operacións de entrada e saída realízanse mediante fluxos, e non temos que preocuparnos do dispositivo físico.

Así, un **arquivo** é un fluxo secuencial de *bytes*, que remata cunha **marca de fin de ficheiro** ou un número de *byte* específico gravado no sistema. Cando un programa procesa un ficheiro, recibe unha indicación ao alcanzar o final.

Existen dous tipos de fluxos: de **entrada** (envía datos dende unha fonte ao programa), ou de **saída** (envía datos dende un programa a un destino).

En C++, os fluxos son un tipo especial de variable, coñecida como **obxecto**. Xa coñecemos dous fluxos básicos de C++, que son `cin` e `cout`, que permiten operacións de entrada e de saída. Para poder empregalos, é necesario incluír algunhas bibliotecas no programa. Para traballar con ficheiros, en C++ usaranse ademais:

```
istream          // Fluxos de entrada estándar

ostream          // Fluxos de saída estándar

iostream         // Fluxos de entrada/saída estándar

ifstream         // Fluxos de entrada de arquivos

ofstream         // Fluxos de saída de arquivos

fstream          // Lluxos de entrada/saída de arquivos
```

O arquivo de cabeceira `fstream` inclúe as definicións das funcións básicas para a entrada e a saída de arquivos. En C++ existen dous tipos de ficheiros:

Abrir

**Libraria.txt**

```
1 As malas mulleres;Marilar Aleixandre;Galaxia;2020
2 Historia de un canalla;Julia Navarro;Plaza & Janes;2016
3 Irene;Pierre Lemaitre;Editions du Masque;2006
4 El imperio final;Brandon Sanderson;Nova;2008
5 Berta Isla;Javier Marias;Alfaguara;2017
6 Pua;Lorenzo Silva;Destino;2003
7 A praia dos afogados;Domingo Villar;Galaxia;2009
8 Un animal chamado neboa;Ledicia Costas;Xerais;2024
9
10
11
```

Figura 9.3: Exemplo de arquivo de texto

- **Ficheiros de texto**: son directamente editables a través dun editor de texto. As operacións de escritura e lectura de ficheiros deste tipo serán similares ás empregadas para escribir en pantalla e ler de teclado (Figura 9.3).

- **Ficheiros binarios**: non son directamente editables a través dun editor de texto. A información gárdase na memoria (permanente) tal e como está na memoria principal. Son similares a un vector, e para acceder aos seus datos son necesarias funcións específicas, diferentes ás dos ficheiros de texto (Figura 9.4).


Figura 9.4: Exemplo de arquivo binario

A diferencia entre os ficheiros de texto e os binarios radica no modo no que o programa que trata co arquivo interpreta a información contida nel. Cando se crea un arquivo dun xeito determinado, haberá que traballar sempre da mesma forma, e non se poderá cambiar o modo de ler ou escribir información.

Unha vez coñecido o concepto de ficheiro, así como o dos fluxos en C++, veremos a continuación os diferentes tipos de acceso que se poden realizar.

## 9.2 Tipos de acceso: secuencial e directo

Os datos almacénanse nun soporte (medio físico). Cando se fala de xestión de arquivos existen dous tipos de soporte: **secuencial**, no cal os rexistros son escritos uns a continuación dos outros, polo que para acceder a un rexistro determinado haberá que pasar antes polos anteriores; e **direccionable**, estruturado de tal modo que se pode localizar directamente unha información, sen necesidade de pasar polos rexistros anteriores. Neste caso, cada rexistro debe ter un campo que o diferencie dos demais.

Segundo o soporte, existen dous tipos de acceso aos ficheiros: **secuencial** e **directo** ou **aleatorio**. Os arquivos de acceso secuencial e directo diferéncianse basicamente nas operacións que se empregan para acceder aos datos.

### 9.2.1 Acceso secuencial

O ficheiro organízase como unha colección de rexistros almacenados sucesivamente sobre o soporte, de tal forma que se accederá a eles de forma consecutiva, na mesma orde de lectura na que foron gravados. Os arquivos así organizados conteñen un rexistro particular (o último) que inclúe unha marca de fin de ficheiro, e non poden abrirse simultaneamente para lectura e escritura.

Os arquivos secuenciais son os máis sinxelos de manexar, xa que requiren menos funcións. Hai que ter en conta que neles o punto de lectura e escritura está sempre determinado. Ademais, é posible engadir datos no final do arquivo, pero haberá que percorrelo enteiro, o cal se traduce nun acceso pouco eficiente.

### 9.2.2 Acceso directo

A orde física dos rexistros non ten que corresponderse necesariamente coa orde na que foron almacenados os datos.

A vantaxe dos ficheiros de acceso directo ou aleatorio é que podemos escribir e ler rexistros en calquera orde e posición, e son moi rápidos ao acceder á información que conteñen. Non obstante, é necesario programar a relación que existe entre o contido dun rexistro e a posición que ocupa. Poden existir ocos libres entre rexistros.

## 9.3 Operacións con ficheiros

Independentemente do tipo de acceso, para a declaración dun ficheiro en C++, e supondo que os datos están almacenados dun xeito determinado, unicamente haberá que definir un fluxo e asocialo ao arquivo, da forma:

```
tipo nomeFluxo;
```

Para traballar cun arquivo inicialmente cómpre que estea aberto. Nesta situación poderán realizarse unha serie de operacións básicas, especificamente as que tratan coa propia estrutura dos arquivos e están predefinidas. Estas operacións permitirán escribir a información no ficheiro, recuperala e poñela en memoria. Poden realizarse as seguintes operacións: crear, abrir, pechar, ler e escribir. Será tamén posible a consulta dun rexistro e a súa modificación, así como a inserción de novos rexistros ou o borrado dos xa existentes.

### 9.3.1 Crear

Consiste en definir un ficheiro mediante un nome de arquivo. Se existise un arquivo anterior co mesmo nome, destruiríase. Esta operación é a primeira que se debe realizar. É necesario coñecer o nome do dispositivo (lugar onde se situará o arquivo unha vez creado), asignar un nome ao arquivo, e coñecer o seu tamaño, a súa organización (secuencial o directa), e tamaño do bloque físico o rexistro.

Durante o proceso de creación poden producirse diversos erros se xa existe no soporte outro ficheiro co mesmo nome, hai un problema de *hardware* que impide o proceso ou a comunicación, non temos memoria, ou existen parámetros de entrada erróneos.

Para crear un arquivo, haberá que facelo a través dun fluxo de saída, mediante `ofstream`.

```
ofstream nomeFluxo ("nomeArquivo");
```

### 9.3.2 Abrir

Ábrese un arquivo creado con anterioridade. Establécese comunicación entre a computadora e o soporte, de modo que os rexistros son accesibles para lectura, escritura, ou as dúas operacións á vez, de permitilo o tipo de acceso.

A operación de apertura de ficheiro usa a información que coñecemos para relacionala coa descrición física e preparar un descritor para acceder á información contida no ficheiro. Esta operación debe incluír como parámetros o nome do dispositivo, a canle de comunicación e o nome do arquivo.

Produciranse erros se non se atopa o dispositivo indicado, se o arquivo xa está en uso para outra operación, ou se existen problemas de *hardware*.

Existen dous modos diferentes de abrir un ficheiro:

1. Definir un fluxo, e despois asocialo a un arquivo mediante a función `open()`.

```
tipo nomeFluxo;                      //Definición do fluxo
nomeFluxo.open ("nomeArquivo", modo); //Asociación a arquivo mediante open()
```

2. Definir o fluxo e abrir directamente o ficheiro.

```
tipo nomeFluxo ("nomeArquivo", modo); //Definición e apertura directa
```

Os modos de apertura do arquivo amósanse na Táboa 9.1. Pola súa banda, o acceso pode corresponder a só lectura, oculto,...

Táboa 9.1: Modos de apertura de arquivos en C++

| Modo | Descrición |
|---|---|
| `ios::in` | Entrada |
| `ios::out` | Saída |
| `ios::binary` | Binario (por defecto ábrese en modo texto) |
| `ios::nocreate` | Erro ao abrir se o arquivo non existe |
| `ios::noreplace` | Erro ao abrir se o arquivo xa existe |
| `ios::trunc` | Destrúe o contido do arquivo se xa existe |
| `ios::ate` | Busca polo final ao abrir o arquivo |
| `ios::app` | Engádese ao final do arquivo (só arquivos abertos en modo saída) |

Por defecto, os ficheiros de entrada (`ifstream`) ábrense unicamente para lectura, e os ficheiros de saída só para escritura. Créase un ficheiro novo ou, no caso de que este xa exista, bórrase e créase de novo.

Independentemente do modo de apertura, é recomendable comprobar que o ficheiro abriu correctamente:

```
if (!nomeFluxo) {
    cout << "Erro de apertura";
}
```

Os arquivos poden abrirse para lectura ou para escritura. No primeiro caso, colócase un punteiro no primeiro rexistro do arquivo, e unicamente se permiten operacións de lectura dos rexistros do arquivo.

Cando se abre para escritura, o punteiro colócase ao final do último rexistro. Para acceso directo, o ficheiro pódese abrir para lectura e escritura simultáneas.

### 9.3.3 Pechar

Córtase a comunicación entre a computadora e o soporte, permitindo deste modo o acceso ao arquivo dende outros programas ou por outras persoas. Esta operación destrúe as estruturas creadas para abrir o ficheiro e actualízao, escribindo toda a información que queda pendente.

Para pechar un ficheiro, só é necesario coñecer o seu nome, e que estea aberto. Empregaremos a instrución:

```
nomeFluxo.close();
```

Ao pechar un arquivo aberto para escritura, a marca de fin de arquivo colócase despois do último rexistro. Para detectar se se chegou a esta marca emprégase a seguinte función, que devolve 0 se non se chegou aínda ao final do ficheiro:

```
while(!nomeFluxo.eof()) {
   ..........     // Accédese ao ficheiro
}
```

### 9.3.4 Ler

Consiste en copiar os rexistros do ficheiro dende o soporte á memoria central.

A lectura dun ficheiro permite recuperar a información gardada nel e poñela en memoria para poder traballar directamente con ela. Para poder realizar esta operación, é necesario coñecer exactamente o modo no que foi gardada a información.

A lectura será diferente nos arquivos de textos e binarios. Máis adiante veremos as instrucións necesarias en cada caso. A instrución más sinxela vén dada por:

```
nomeFluxo >> variable;
```

### 9.3.5 Escribir

Consiste en copiar os rexistros do ficheiro dende a memoria central ao soporte.

Esta operación utilízase para poñer a información que está na memoria do ordenador nun ficheiro, para poder utilizala posteriormente.

Dependendo do tipo de ficheiro no que se desexe gardar a información (texto ou binario), a información escribirase no ficheiro de forma diferente. A instrución más sinxela vén dada por:

```
nomeFluxo << variable;
```

### 9.3.6 Renomear

Permite renomear un ficheiro cun novo nome dado.

Emprégase a intrución

```
rename("nomeInicial", "nomeFinal");
```

***Exemplo***: programa que crea un arquivo de texto para escritura, escribe a información nel, pecha o arquivo, ábreo para lectura e le a información (Programa 9.1).

Programa 9.1: Tratamento de arquivos de texto

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

int main()
{
    int num;
```

```
    ofstream fsaida("enteiros.txt");

    if (!fsaida){
        cout << "Erro apertura ficheiro";
        return 1;
    }

    fsaida << 15 << endl;
    fsaida.close();

    ifstream fentrada("enteiros.txt");
    if (!fentrada){
       cout << "Erro apertura ficheiro ";
       return 1;
    }
    return 0;
}
```

## 9.4 Funcións de tratamento de ficheiros

En C++, ademais das operacións básicas estudadas, dispomos dunha serie de funcións que facilitan o tratamento de ficheiros. Distinguiremos entre os ficheiros de texto e os ficheiros binarios.

### 9.4.1 Ficheiros de texto

Empregaremos este tipo de ficheiros para almacenar información que deba ser procesada con editores de texto. Pódense realizar as mesmas operacións que sobre `cin` e `cout`.

Supoñamos que desexamos ler un número coñecido de valores enteiros dun fluxo `f`. Poderiamos utilizar as xa coñecidas estruturas repetitivas. Por exemplo, poderíanse ler 5 elementos dun ficheiro coas instrucións:

```
for (int i = 0; i < 5; i++) {
    f >> dato; // getline (f, dato) para liña completa
}
```

Porén é moi habitual non coñecer con antelación o número de elementos que contén un determinado ficheiro, polo que haberá que recorrer a outras estratexias para poder ler toda a información que contén, e alcanzar o final do ficheiro. Para iso, pódese empregar a instrución `while`, que devolverá `true` se se le con éxito:

```
while (f >> dato) {
    ..........
}
```

De forma similar á lectura, a escritura realízase con:

```
f << variable;
```

### 9.4.2 Ficheiros binarios

Os ficheiros binarios son aqueles que foron abertos mediante o modo `ios::binary`.

Para ler un carácter dun fluxo binario úsase a función `get()`, cuxo prototipo é:

```
istream &get(char &ch)
```

Esta función introduce o seguinte carácter do fluxo de entrada e gárdao no argumento tipo carácter. Se se chegou ao final do arquivo, devolve NULL.

***Exemplo***: lectura de caracteres dun arquivo asociado a un fluxo `f`, ata chegar ao fin de arquivo.

```
char c;
f.get(c);
while (!f.eof()){
    cout << c;
    f.get(c);
}
```

Para escribir un carácter nun fluxo binario úsase a función `put()`, cuxo prototipo é:

```
ostream &put(char ch)
```

***Exemplo***: escritura de carácter nun arquivo asociado a un fluxo `f`.

```
char c;
f.put(c);
```

Para a lectura e escritura de bloques de datos (número fixo de *bytes*) empréganse as funcións `read()` e `write()`, cuxos prototipos son:

```
istream &read (unsigned char *punteiro, int num)
ostream &write (const unsigned char *punteiro, int num)
```

onde `num` especifica o número de *bytes* que se van ler na función `read()`, ou os que se van escribir na función `write()`. O modificador `unsigned` indica que non se vai almacenar o signo da variable, e o `"*"` fai referencia ao tipo de variable punteiro, no que profundaremos no seguinte Tema. Como adianto, diremos que un **punteiro** é unha variable que almacena no seu interior unha dirección de memoria dunha variable dinámica. Decárase como `tipo *nomePunteiro;`, e debe apuntar sempre ao tipo de dato correcto (enteiro, real, carácter,...).

Se a función `read()` detecta o final do arquivo antes de ler os num *bytes* especificados, simplemente devolve os caracteres que conseguiu ler. Se se desexa coñecer o número de caracteres lidos (na última extracción), pódese utilizar a función `gcount()`, cuxo prototipo é:

```
int gcount()
```

Nos ficheiros binarios pódese controlar a entrada mediante as seguintes funcións:

- `istream &get(char *punteiro, int num, char delimitador='\n')`: le ata num-1 caracteres. Para cando atopa o carácter delimitador, cando se chega ao fin do ficheiro, ou ata que se leron num-1 caracteres. Se se detecta o carácter delimitador, non se elimina do fluxo nin se almacena en punteiro.
- `istream &getline(char *punteiro, int num, char delimitador='\n')`: similar a `get()`, pero se se para por ter atopado o carácter delimitador, este elimínase do fluxo de entrada, aínda que non se engade ao punteiro.
- `getline(istream &, string, char delimitador='\n')`: almacena no `string` (tipo de dato que será estudado máis adiante) os caracteres que extrae do fluxo de entrada indicado ata que se detecte o carácter delimitador.

### 9.4.3 Control de E/S

A continuación, preséntanse diversas funcións que nos van permitir controlar as operacións de E/S:

- `istream &ignore(int n=1, int delim = EOF)`: úsase para extraer e descartar n caracteres ou ata atopar o carácter delimitador. O acrónimo EOF fai referencia a *End-Of-File*, fin de arquivo.

- `int peek()`: emprégase para saber o seguinte carácter dun fluxo de entrada sen extraelo do mesmo. Se é o final do arquivo, esta función devolve EOF.
- `istream &putback(char c)`: úsase para saber cal foi o último carácter lido.
- `ostream &flush()`: emprégase para pasar os datos almacenados na memoria intermedia ao arquivo.

C++ mantén información sobre o éxito ou fracaso das operacións de E/S realizadas. Para iso, cada fluxo ten asociada unha combinación de *bits* que varían segundo o comportamento do mesmo, e polo tanto determinan o seu estado (Táboa 9.2).

Táboa 9.2: Indicadores que determinan o estado do fluxo en C ++

| Indicador | Descrición | Valores |
|---|---|---|
| `ios::goodbit` | Indica se os outros indicadores están ou non a cero | 0 se non hai erro, 1 en caso de que exista algún erro |
| `ios::eofbit` | Indica se se atopa no final do ficheiro | 0 se é fin de arquivo, 1 se non é fin de arquivo |
| `ios::failbit` | Indica se se produciu un erro pero non se perderon caracteres. Normalmente son erros recuperables | 0 se non existe un erro, 1 se existe un erro ou fin de arquivo |
| `ios::badbit` | Indica que se produciu un erro que como consecuencia supón perda de datos. Erros polo xeral non recuperables | 0 se se produciu un erro fatal, 1 se non se produciu un erro fatal |

Para obter información sobre estes indicadores é necesario usar as seguintes funcións (Táboa 9.3).

Táboa 9.3: Funcións que permiten obter información dos indicadores

| Función | Descrición |
|---|---|
| `int rdestate()` | Devolve un enteiro onde se codifica o estado do fluxo |
| `int bad()` | Devolve verdadeiro se *badbit* está a 1 |
| `int eof()` | Binario (devolve verdadeiro se é final de arquivo) |
| `int fail()` | Devolve verdadeiro se *failbit* ou *badbit* están a 1 |
| `int good()` | Devolve verdadeiro se non hai erros, é dicir ningún dos bits está activado |

Se se desexa modificar o estado do fluxo a un novo valor, úsase a función `clear()`, e o seu argumento é o novo estado asignado ao fluxo. O seu prototipo é `void clear(int bandeiras = 0)` (por defecto é `goodbit`).

### 9.4.4 Acceso directo

C++ asocia directamente dous punteiros a cada arquivo, que avanzan automaticamente unha posición cada vez que se realiza a operación de E/S que teñen asociada. Son:

- **Punteiro de entrada** `get`: indica o número de *byte* no arquivo dende o que se vai ler a seguinte operación de entrada.
- **Punteiro de saída** `put`: indica o número do *byte* do arquivo no que se vai posicionar a seguinte operación de saída.

Pódense modificar estes punteiros mediante as seguintes funcións:

- `istream &seekg(streamoff desprazamento, seek\_dir orixe)`: move o punteiro `get` tantos *bytes* como indica o desprazamento, a partir da posición indicada polo argumento `orixe`.

- **ostream &seekp(streamoff desprazamento, seek_dir orixe)**: move o punteiro **put** tantos *bytes* como indica o desprazamento a partir da posición indicada polo argumento **orixe**.

O tipo **streamoff** está definido para conter o maior valor correcto que pode tomar **desprazamento**, e **seek_dir** pode tomar os seguintes valores:

- **ios::beg**: móvese a partir do principio do arquivo.
- **ios::cur**: móvese a partir da posición actual.
- **ios::end**: móvese a partir do final do arquivo.

***Exemplo***:

```
fluxoES.seekg(n, ios::cur); // Avanza o punteiro get n bytes
fluxoES.seekg(0, ios::end); // Coloca o punteiro get ao final do arquivo
```

Para coñecer a posición actual de cada uno deses dous punteiros empréganse as funcións:

- **streampos tellg()**: devolve a posición actual do punteiro de entrada **get**.
- **streampos tellp()**: devolve a posición actual do punteiro de saída **put**.

***Exemplo***: ficheiro que almacena os días da semana. Acceso secuencial e directo (Programa 9.2).

Programa 9.2: Ficheiro binario que almacena os días da semana

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

    int main()
    {
        int i;
        char meses[12][10] = {"xaneiro", "febreiro", "marzo", "abril", "maio", "xunho", "
            xullo", "agosto", "setembro", "outubro", "novembro", "decembro"};
        char mes[10];

        // Almacenar os nomes dos meses nun ficheiro
        ofstream fsaida("meses.dat", ios::out | ios::binary);
        if (!fsaida) {
            cout << "Erro de apertura";
            return 0;
        }
        cout << "Almacenando meses do ano..." << endl;
        for (i = 0; i < 12; i++) {
            fsaida.write(meses[i],10);
        }
        fsaida.close();

        ifstream fentrada("meses.dat", ios::in | ios::binary);
        if (!fentrada) {
            cout << "Erro de apertura";
            return 0;
        }

        //Acceso secuencial
        cout << "Meses do ano:" << endl;
        fentrada.read(mes, 10);
        do {
            cout << mes << endl;
            fentrada.read(mes, 10);
        } while(!fentrada.eof());

        fentrada.clear();
```

```
    // Acceso directo
    cout <<"\nMeses da primavera:" << endl;
    for (i = 2; i <= 5; i++) {
        fentrada.seekg(10*i, ios::beg);
        fentrada.read(mes, 10);
        cout << mes << endl;
    }

    cout << "Introduce un numero de 1 a 12: ";
    cin >> i;
    fentrada.seekg(10*(i-1), ios::beg);
    fentrada.read(mes, 10);
    cout << "Mes: " << i << "-->" << mes << endl;
    fentrada.close();

    return 0;
}
```

## 9.5 Exercicios propostos

1. Supondo que existe un arquivo de texto ”datos.txt”, que contén información relativa a varias persoas (nome, apelido, idade e código), que se visualiza na pantalla ao executar o seguinte código?

```
...
char nome[30];
char apelido[50];
int idade;
int codigo;
ifstream fentrada;

fentrada.open("datos.txt", ios::in);
if (!fentrada) {
    cout <<"Erro de apertura" ;
    return 1;
}

while (!fentrada.eof()) {
    fentrada >> nome >> apelido >> idade >> codigo;
    cout << nome << " " << apelido << ", " << idade << ", " << codigo << endl;
}
fentrada.close();
```

2. Supoñamos dous arquivos de texto ”f1.txt” e ”f2.txt”, que teñen o mesmo número de caracteres. Implementar un programa que lea os dous arquivos, carácter a carácter, e os escriba intercalados nun terceiro arquivo ”f3.txt”. Por exemplo, se o primeiro carácter de ”f1.txt” é ’a’, e o de ”f2.txt” é ’x’, debe escribirse en ”f3.txt” a secuencia ” ax”.
3. Escribir un programa que xere un arquivo de texto chamado ”datos.txt”, que conteña 10 liñas, cada unha delas da seguinte forma: número enteiro + espazo branco + cadea + espazo branco + número enteiro. Os valores de cada liña deberán lerse por teclado.
4. Escribir un programa que lea os datos do arquivo xerado no exercicio anterior e os amose por pantalla, supoñendo que non coñecemos o número de liñas que contén.
5. Supoñamos un arquivo de texto ”infoPersoas.txt”, que contén 50 estruturas de tipo Persoa, onde cada unha delas vén definida polo seu nome, a súa idade, e o seu salario. Escribir un programa que recupere todos os datos existentes no arquivo, e os amose por pantalla. A partir dos datos recuperados, deberase crear o arquivo ”novos.txt”, que conteña únicamente os datos correspondentes ás persoas que teñan menos de 30 anos.

6. Supondo que existe un arquivo binario ”datosPersoas.dat”, que contén información relativa a varias persoas (nome, apelido, idade e código), que se visualiza na pantalla ao executar o seguinte código?

```
...
struct Persoa {
    char nome[30];
    char apelido[50];
    int idade;
    int codigo;
};

ifstream fentrada;

Persoa per;

fentrada.open("datos.dat", ios::in|ios::binary);
if (!fentrada) {
    cout <<"Erro de apertura" ;
    return 1;
}
fentrada.read((char *)&per, sizeof(Persoa));
while (!fentrada.eof()) {
    cout << per.nome << " " << per.apelido << ", " << per.idade << ", " << per
        .codigo << endl;
    fentrada.read((char *)&per, sizeof(Persoa));
}
fentrada.close();
```

7. Escribir un programa binario que lea por teclado unha cadea de 50 caracteres e a garde nun arquivo ”caracteres.dat” escribíndoa carácter a carácter.
8. Supoñamos definida unha estrutura Punto, que leva conta das coordenadas (x, y), dun punto no espazo bidimensional. Supoñamos un arquivo binario ”puntos.dat” que almacena elementos de tipo Punto. Escribir un algoritmo que recupere a información do arquivo e a amose por pantalla.

# Capítulo 10

# Xestión dinámica de memoria

## 10.1 Concepto de punteiro

Xeralmente, os programas almacenan os seus datos na memoria en tres áreas diferentes:

- **Memoria global**: almacénanse as variables globais ou estáticas e as constantes. É dicir, nesta zona de memoria están todos aqueles datos que están presentes dende o comezo do programa ata que remata. Posúe tamaño fixo, coñecido ao comezo da execución do programa.
- **Pía (*stack*)**: as variables aparecen e desaparecen nun momento puntual da execución dun programa. Utilízase xeralmente, como xa vimos, para almacenar variables locais ás funcións. A pía actúa como un espazo de anotación temporal. Cando se invoca unha función, as súas variables temporais están activas só durante a súa execución. Cando a función remata, esas variables xa non existen. Non parece adecuado reservar un espazo na memoria global para estas variables. Por tanto, é unha zona na que se insiren e borran variables de forma continua. O seu tamaño é variable e coñécese só en tempo de execución.
- **Montón (*heap*)**: almacénanse as variables dinámicas, a memoria está dispoñible para a súa reserva e liberación en calquera momento durante a execución dun programa. Ao igual que a pía, ten tamaño variable, non coñecido con antelación á execución.

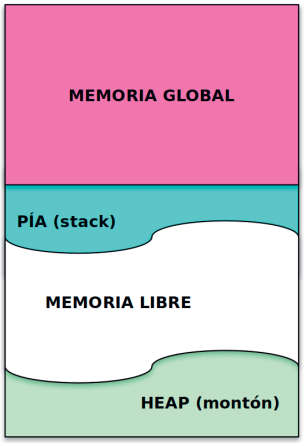



Figura 10.1: Representación das zonas de memoria

O sistema operativo reserva un espazo inicial para o *stack* e para o *heap*, e as dúas rexións crecen/de-

crecen nese espazo máximo. A Figura 10.1 representa de forma gráfica estas tres zonas de memoria.

Ata o de agora, estivemos a traballar con diferentes tipos de datos, que se declaran sempre en tempo de compilación, e cuxo tamaño non pode variar durante toda a execución do programa. Porén, estes non son os únicos datos que se poden manexar, e pode ser necesario, nun momento determinado, necesitar máis memoria durante a execución dun programa. Podemos ter:

- **Datos estáticos**: teñen tamaño e forma constantes durante a execución dun programa, e determínanse en tempo de compilación. Alóxanse na memoria global. Se se coñece o número e tipo de datos globais, pode saberse en calquera momento canta memoria ocupan durante a execución do programa. O problema que presentan é que non se pode dimensionar a estrutura de antemán, polo que se pode dar tanto o desperdicio como a falta de memoria.
- **Datos dinámicos**: teñen tamaño e forma variable durante a execución dun programa. Créanse e destrúense en tempos de execución. Estes datos permanecen activos dende o momento no que se chama o subprograma ao que están asociados, e deixan de ser utilizables ao rematar a execución do subprograma. Alóxanse no *stack* (datos locais) e no *heap* (son obxectos dinámicos, créanse e destrúense a vontade, segundo a necesidade, e en tempo de execución). Posúen a vantaxe de que permiten dimensionar a estrutura de datos de xeito preciso, de modo que a asignación de memoria se vai realizando a medida que se necesita.

Cando se solicita un dato en tempo de execución, resérvase memoria no *heap* para aloxalo, e cando se libera a memoria que ocupaba, pasa a estar dispoñible para ser asignada a outros obxectos.

Como vemos, en ocasións é necesario traballar con estruturas de datos máis complexas, cuxo tamaño cambia de forma dinámica durante a execución do programa, polo que a memoria deberá ser reservada e liberada en tempo de execución.

Como solución, moitas linguaxes de programación incorporan o concepto de punteiro, que se presenta así como a ferramenta básica para a creación de variables dinámicas.

Non se pode entender o concepto de memoria dinámica sen punteiros. Un programa pode manipular unha porción de memoria dinámica porque dispón da dirección da primeira posición na que se aloxa, e esta almacénase nun dato de tipo punteiro.

Un **punteiro** é unha variable que almacena no seu interior unha dirección de memoria dunha variable dinámica. Gráficamente, un punteiro pódese representar da forma seguinte (Figura 10.2).

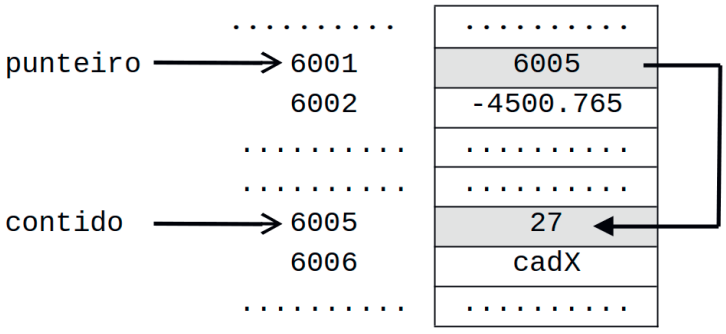



Figura 10.2: Representación gráfica dun punteiro

A Figura 10.2 presenta unha variable de tipo punteiro na dirección de memoria 6001. No interior desta variable aparece como contido o valor 6005, que corresponde a outra dirección de memoria, cuxo interior almacena o contido 27.

Como calquera outra variable en C++, os punteiros deben ser declarados no programa ou subprograma

correspondente.

En C++, a declaración farase do seguinte modo:

```
tipo *nomePunteiro;
```

***Exemplo***: declaración dun punteiro a enteiro e outro a real en C++.

```
int *pe;
float *pr;
```

Un punteiro é un tipo de dato simple que sempre está asociado a outro tipo, e debe apuntar sempre ao tipo de dato correcto. Un punteiro a unha variable enteira sempre se declarará como tipo enteiro, un punteiro a unha variable real será de tipo real,...

De forma similar aos datos, que poden ser constantes e variables, pódense declarar tamén **punteiros constantes**. Son punteiros que non se poden cambiar, pero si é posible modificar os datos aos que apunta. Por exemplo, se se define e inicializa á vez un punteiro constante, apuntándoo a unha dirección determinada de memoria, poderá cambiarse o contido da memoria á que apunta, pero non a dirección.

Ademais dos punteiros constantes, existen os **punteiros a constantes**. Neste caso, os punteiros pódense modificar e apuntar a unha constante diferente, pero os datos apuntados por eles non poden alterarse.

Ao traballar con direccións de memoria e acceder a elas directamente, os punteiros permiten realizar operacións de manera moi eficiente. Deste modo, pódense optimizar en gran medida os algoritmos que requiren acceso frecuente ou repetitivo a grandes cantidades de datos.

O emprego de punteiros presenta unha serie de vantaxes na programación, entre as que podemos citar a creación de código eficiente e rápido, a posibilidade de traballar con direccións de memoria directamente, e a xestión de memoria dinámica durante a execución dun programa.

Como resumo, pódese dicir que un punteiro é unha variable como calquera outra, na que se almacena unha dirección de memoria que apunta a outra variable, na cal están os datos aos que apunta o punteiro. Polo tanto, un punteiro apunta a unha dirección de memoria. É importante tamén ter en conta que a declaración dunha variable de tipo punteiro non quere dicir que necesariamente exista un contido apuntado pola mesma (hai que apuntalo). Por iso, hai que crear e destruír explicitamente o contido apuntado polas variables de tipo punteiro.

## 10.2 Asignación e liberación de memoria

Unha vez temos declarado un punteiro, cómpre asignarlle memoria para poder almacenar con posterioridade un dato do tipo ao que apunta. De forma similar, cando se remata de utilizar esta memoria, deberá ser liberada, eliminando tamén con isto o dato apuntado. Por exemplo, se temos declarada unha variable de tipo punteiro a enteiro, e queremos asignarlle un valor, primeiro teremos que reservar memoria.

A memoria dinámica pódese ir modificando durante a execución do programa, pero é máis difícil de manexar. Ademais, en ocasións pode repercutir no rendemento do programa, xa que ao reservar a memoria dinamicamente hai que realizar varias tarefas (buscar un bloque de memoria libre, almacenar posición e tamaño da memoria asignada,...), o que resulta unha carga adicional para o sistema.

As variables dinámicas poden existir ou non. Isto pode presentar problemas á hora de empregalas no programa, xa que non é posible declaralas e reservar o espazo suficiente en tempo de compilación, como xa se dixo. Actualmente, todas as linguaxes de programación proporcionan rutinas para poder asignar e liberar memoria de forma correcta. Algunhas incluso realizan a tarefa de liberación de memoria de forma automática.

Veremos a continuación as instrucións de reserva e liberación de memoria dinámica en C++. Para isto, suporemos previamente declarado un punteiro p (polo momento, dá igual o tipo).

A reserva farase a través do operador **new**, que asigna memoria dinámica suficiente para conter un valor do tipo especificado.

```
p = new tipoDato; // Reservar memoria
```

Adoita ser habitual declarar e reservar memoria nun só paso, asignando ademais o valor inicial ao contido:

```
p = new tipoDato[(valorInicializacion)]; // Reservar memoria
```

É tamén necesario comprobar se a memoria foi reservada de xeito correcto, de forma que se devolva a dirección desta memoria en caso de que exista suficiente, e o punteiro NULL se non existe.

```
if ((p = new tipo_dato) == NULL) { // Reservar memoria
    cout << "Erro de asignacion de memoria\n";
}
else {
    instrucions;
}
```

Unha vez finalizado o emprego da memoria dinámica, deberá ser liberada mediante o operador delete:

```
delete p; // Liberar memoria
```

***Exemplo***: reserva dinámica de memoria e posterior liberación en C++.

```
int *pe;

if ((pe = new int) == NULL) {
    cout << "Erro de asignacion\n";
}
else {
    operacions con pe;
    delete pe;
}
```

Debemos ter unha instrucción de reserva de memoria e unha de liberación por cada un dos punteiros existentes no programa. Unha vez está reservada a memoria para unha variable de tipo punteiro, pódese almacenar nela un dato do tipo apuntado.

Para iso, o primeiro que faremos será empregar as sentenzas de asignación xa coñecidas. Supoñamos dous punteiros a enteiro **pEnteiro1**, **pEnteiro2**. Realízanse as operacións:

- Reserva de memoria de **pEnteiro1**.
- Asignación do valor 37 a **pEnteiro1**.
- Asignación de **pEnteiro1** a **pEnteiro2**.
- Liberación de **pEnteiro1**.

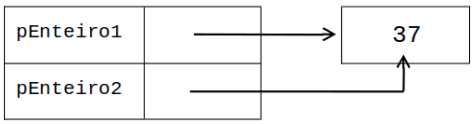



Figura 10.3: Situación final de memoria tras a asignación de punteiros

Graficamente, a situación final na memoria amósase na Figura 10.3.

Se agora se libera a memoria ocupada por **pEnteiro1**, o punteiro **pEnteiro2** continúa apuntando á mesma posición de memoria, pero perdemos toda referencia a ela, posto que se liberou e xa non contén ningún dato válido. Ademais, se accedemos ao dato apuntado por **pEnteiro2** logo da liberación, pódese producir un erro de segmentación, e seguramente o seu valor cambiou con respecto ao que tiña antes da liberación.

Por iso, é recomendable ter especial coidado á hora de traballar con memoria dinámica e verificar que os punteiros non apuntan a direccións de memoria non válidas. Hai que evitar tanto deixar posicións de memoria inaccesibles, como acceder a posicións de memoria incorrectas.

## 10.3 Operacións con punteiros

Os punteiros son variables que almacenan direccións de memoria, polo que non son variables como as que se estudaron ata o de agora. Porén é posible realizar con eles determinadas operacións. Así, poderán facerse operacións de direccionamento e indireccionamento, de asignación, de comparación de punteiros, e operacións aritméticas.

### 10.3.1 Direccionamento/Indireccionamento

Dado que os punteiros conteñen direccións de memoria que poden apuntar a datos do mesmo tipo que os punteiros, haberá que ter en conta que, para unha variable punteiro, poderase acceder tanto á dirección de memoria que contén como ao dato que hai almacenado na mesma.

Mediante o direccionamento devólvese a dirección de memoria. Emprégase o operador &.

Mediante indireccionamento accedemos ao contido, o valor da variable ubicada na dirección de memoria especificada. Para iso, temos o operador de contido ou indirección *. Non hai que confundir o * que aparece na declaración co operador de indirección.

En C++, farase do seguinte modo:

```
tipo *p;
tipo v = valor;
..........
p = &v;
cout << p       // Dirección de memoria de p
cout << *p;     // Contido da dirección de memoria á que apunta p
```

***Exemplo***: uso de operadores de direccionamento e contido en C++.

```
int *p;
int v = 5;
........
p = &v;
cout << p;      // Dirección de memoria
cout << *p;     // 5
```

### 10.3.2 Inicialización

Despois de declarar un punteiro, este contén un valor indeterminado. Por iso, non é correcto usalo antes de asignarlle un valor. Unha primeira forma de inicializar un punteiro é asignarlle o valor nulo. Así, o punteiro non estará apuntando a ningures. Para iso pódese empregar a constante NULL ou 0. Existe tamén o punteiro xenérico, que se declara como **void**, e non se asigna a ningún tipo específico de dato.

Para inicializar un punteiro apuntando a algunha parte, asígnaselle a dirección doutra variable previamente declarada, ou ben faise que non apunte a nada.

Hai que ter en conta que os punteiros deben apuntar sempre ao tipo de dato correcto, xa que o compilador vai realizar todas as operacións que se indiquen sobre elas en función dos seus tipos base.

Farase do seguinte modo:

```
tipo *p1, *p2;
tipo v = valor;
..........
p1 = NULL;
p2 = &v;
```

Nestes casos realizouse unha inicialización estática, a asignación de memoria empregada para almacenar o valor é fixa, e non pode desaparecer. Cando a variable está definida, o compilador reserva memoria para almacenar este tipo de dato, e esta reserva mantense durante toda a execución do programa. É dicir, esta memoria non se pode liberar. A dirección á que apunta o punteiro pode variar, pero a reserva de memoria feita para o punteiro non.

### 10.3.3 Asignación

Pódese asignar un punteiro a outro, se os dous son do mesmo tipo. Farase a través do operador de asignación:

```
tipo *p1, *p2;
tipo v = valor;
..........
p1 = NULL;
p2 = &v;
p1 = p2;
```

### 10.3.4 Comparación

Para comparar punteiros empréganse os correspondentes operadores relacionais. A comparación aplícase ás direccións almacenadas nestes punteiros.

En C++ empréganse os operadores $==, !=, >, >=, <, <=$

Existe ademais outra comparación moi habitual, que xa vimos neste tema na reserva de memoria: a comparación co punteiro NULO ou NULL.

### 10.3.5 Aritmética de punteiros

Existen dous tipos de operacións aritméticas con punteiros. No primero deles un dos operandos é un punteiro, e o outro un enteiro. No segundo tipo, ambos os dous operandos son punteiros. Pódense realizar as seguintes operacións:

#### Suma/resta de enteiros a punteiros

Realízase mediante os operadores +, -, ++, –, +=, -=, e as instrucións:

```
tipo *p1, *p2;
tipo v = valor;
..........
```

```
p1 = &v;
p2 = &v;
p2 = p2 + v;
```

**Resta dun punteiro a outro**

Só está permitida a resta, xa que a suma non tén sentido. A resta dará a distancia (diferenza en posicións de memoria) que separa os dous punteiros.

Empréganse as instrucións:

```
tipo *p1, *p2;
..........
cout << p1 - p2;
```

***Exemplo***: aritmética de punteiros en C++.

```
int *p1, *p2;
int v = 8;
........
p = &v;              // Contido de p1 e 8
p2 += v;             // p2 salta 8 posicións
cout << p1 - p2;     // Distancia entre p1 e p2
```

Veremos máis adiante que os arrais e os punteiros están fortemente relacionados. Deste modo, a aritmética de punteiros só ten sentido se se leva a cabo sobre os elementos dun arrai, cuxo nome non é máis que un punteiro constante (o seu valor non se pode modificar) ao primeiro elemento do arrai.

## 10.4 Punteiros e funcións

Ata este momento estudáronse punteiros a variables de tipo simple. Non obstante, é posible declarar punteiros que apunten a calquera outro tipo de variable, como a estruturas ou arrais, e a funcións.

Xa sabemos que unha función unicamente pode devolver un valor, independentemente do número de parámetros que posúa. Porén o emprego de punteiros a funcións permitirá establecer, durante a execución da función, os valores deses parámetros, e mantelos unha vez a execución finalice e se devolva o control do fluxo ao programa principal.

De forma similar ás variables, cualquera que sexa o seu tipo, as funcións almacénanse en memoria, o que significa que tamén teñen dirección. Os punteiros a funcións apuntan a trozos de código executable. Deste modo, un punteiro a unha función é un punteiro cuxo valor apunta á dirección do nome da función, isto é, a un punteiro constante (Figura 10.4).

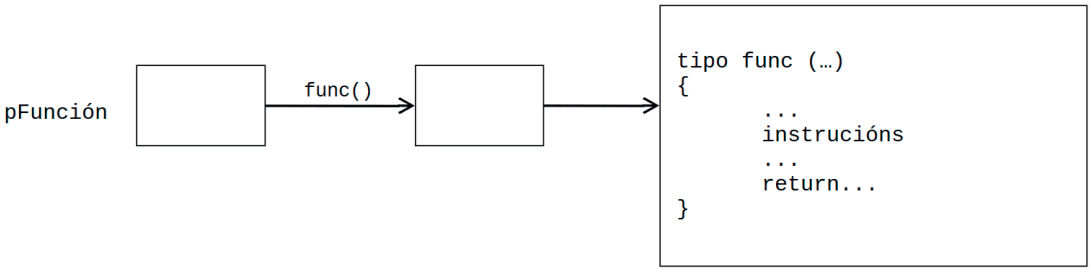



Figura 10.4: Representación dun punteiro a unha función

A declaración dun punteiro a unha función faise de forma similar á declaración dos punteiros:

```
tipo (*nomeFuncion)(parametros);
```

***Exemplo***: declaración dun punteiro a unha función en C++ (Programa 10.1).

Programa 10.1: Exemplo de uso dun punteiro a unha función en C++

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

#define PI 3.1416

float (*pFuncion) (float r);

float calcularArea (float r);
float calcularPerimetro (float r);

int main()
{
    float radio, area, perimetro;

    cout << "\nIntroduce radio da circunferencia: ";
    cin >> radio;                                    //radio = 1

    pFuncion = calcularArea;
    area = pFuncion(radio);

    pFuncion = calcularPerimetro;
    perimetro = pFuncion(radio);

    cout << "Area: " << area << endl;               // Resultado: 3.1416
    cout << "Perimetro: " << perimetro << endl;     // Resultado: 6.2832

    return 0;
}

float calcularArea (float r)
{
    return PI * r * r;
}

float calcularPerimetro (float r)
{
    return 2 * PI * r;
}
```

A utilidade dos punteiros á funcións emprégase fundamentalmente cando se desexa elixir entre varias funcións, e nos que a función elexida é invocada varias veces. Empregando punteiros, a asignación faise unicamente despois de asignar a función seleccionada a un punteiro, a través do cal pode ser chamada.

Finalmente, os punteiros tamén poden pasarse como argumentos de funcións de modo análogo ao resto de variables, xa estudado.

## 10.5 Punteiros e estruturas

Aínda que o emprego de punteiros con variables de tipos de datos simples é unha xestión dinámica de memoria, xa que se reserva e destrúe en tempo de execución, non aporta grandes vantaxes ao emprego de variables estáticas.

Así e todo, cando se empregan tipos de datos estruturados, a situación é completamente diferente. Nestes casos, estes datos poden chegar a ocupar espazos considerables de memoria, e a xestión por medio de punteiros pode resultar moito máis eficiente.

Os punteiros pódense declarar de calquera tipo base, e apuntar a calquera dato do mesmo tipo. Por iso, é posible declarar punteiros a estruturas, e reservar e liberar memoria, de forma similar a calquera outro tipo de dato.

As sentenzas que se deben empregar son:

```
struct [NomeEstrutura] {
    tipo1 nomeMembro1;
    tipo2 nomeMembro2;
    tipo3 nomeMembro3;
    ..........
    tipoN nomeMembroN;
};

NomeEstrutura *nomePunteiro;
```

Sabemos que, para facer referencia a un membro da estrutura utilizando o nome desta, debemos empregar o operador ".". Cando se accede aos membros dunha estrutura a través dun punteiro, será necesario utilizar o operador $\rightarrow$ (ou alternativamente (*`nomePunteiro`)).

***Exemplo***: declaración de punteiros a unha estrutura, e acceso a sus membros en C++.

```
struct Libro {
    char titulo[100];
    char autor[100];
    float prezo;
    int numPaxinas;
};
..........
Libro *pl;

if ((pl = new Libro) == NULL) {
    cout << "Erro asignacion\n";
}
else {
    pl->prezo = 32.5;
    pl->numPaxinas = 65;    // Alternativa: (*pl).numPaxinas = 65
    delete pl;
}
```

No exemplo anterior, `pl` é un punteiro a unha estrutura de tipo `Libro`. Cando se asigna a memoria a `pl`, este apuntará a unha posición de memoria na que se almacenan os datos correspondentes aos membros que contén un elemento da estrutura de tipo `Libro` (Figura 10.5).

Vemos así que o tipo base dun punteiro pode ser tan complexo como se queira, incluíndo estruturas ou arrais, os cales poden tamén incluír outras estruturas ou arrais. De forma similar a estes últimos, que podían ser campos de estruturas, é tamén posible declarar membros das mesmas que sexan punteiros.

## 10.6 Punteiros e arrais

Ao estudar a aritmética de punteiros indicouse que hai operacións que unicamente teñen sentido se se realizan sobre os elementos dun arrai, e que estes e os punteiros están intimamente relacionados. Tanto é así que se poden direccionar punteiros como se fosen arrais, e á inversa.

Un nome dun arrai é simplemente un punteiro ao primeiro elemento do mesmo. Así, cando declaramos un arrai, estamos a declarar implicitamente un punteiro do mesmo tipo que os elementos do arrai. De

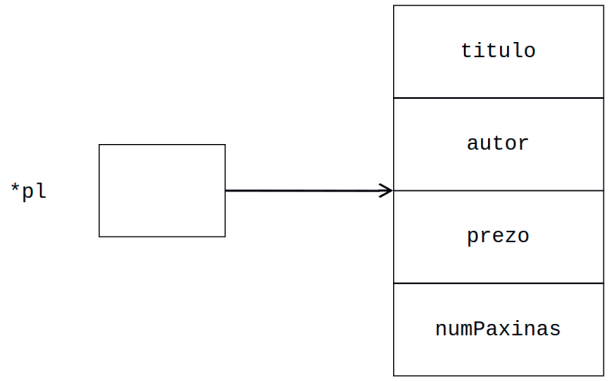



Figura 10.5: Representación dun punteiro a unha estrutura

feito, resérvase memoria para todos os elementos do arrai, e inicialízase o punteiro de xeito que apunte ao primeiro elemento deste.

Supoñamos, por exemplo, que temos definido en C++ un arrai de números enteiros, de 5 elementos (do 1 ao 5), da forma:

```
int ordenado [5];
```

A representación gráfica na memoria da computadora amósase na Figura 10.6.

```
ordenado[0]  1  *(ordenado + 0) // ordenado
ordenado[1]  2  *(ordenado + 1)
ordenado[2]  3  *(ordenado + 2)
ordenado[3]  4  *(ordenado + 3)
ordenado[4]  5  *(ordenado + 4)
```

Figura 10.6: Representación dun arrai na memoria

Vimos xa que para acceder ao primeiro elemento do arrai empregamos a expresión `ordenado[0]`. Pero o nome do arrai é un punteiro ao primeiro elemento, Polo tanto, o acceso a este primeiro elemento pódese facer tamén simplemente mediante o nome: `ordenado` (equivalente a `*(ordenado + 0)`).

Deste modo, para acceder aos elementos dun arrai é posible facelo a través do operador de indexación `[]`, como vimos facendo ata o de agora, ou por medio do contido do punteiro correspondente, mediante o operador `*`.

Para asignar un punteiro a un arrai, faise da mesma forma na que se asignan outras variables. Supondo declarado o arrai anterior ordenado, pódese definir un punteiro do mesmo tipo e apuntalo ao arrai da forma seguinte:

```
tipo_int *p_ordenado;
pOrdenado = &ordenado[0];   // Ou ben pOrdenado = ordenado
```

Desta forma, o punteiro `pOrdenado` apunta ao primeiro elemento do arrai ordenado, de tal forma que o valor que conterá esa posición de memoria será 1 (Figura 10.7).

En moitas ocasións poderemos declarar un dato como arrai e percorrelo usando punteiros. En xeral, e sempre e cando o punteiro apunte á posición de memoria do primeiro elemento do arrai, o acceso a unha posición `i` pode realizarse mediante: `ordenado[i]`, `*(ordenado + i)`, `*(pOrdenado + i)`, `pOrdenado[i]` Se o punteiro non apunta á dirección de comezo do arrai (e apunta, por exemplo, á terceira posición do

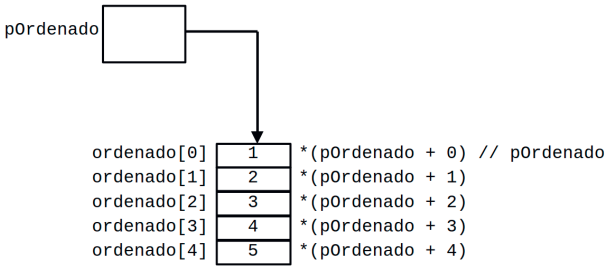



Figura 10.7: Punteiro que apunta á primera posición de arrai

mesmo), non se poden empregar as expresións anteriores, xa que haberá un desprazamento de posicións entre ambos os dous.

Hai que recordar tamén que nunca se poderá utilizar un índice negativo cun arrai, xa que estariamos accedendo a unha zona de memoria que non pertence ao arrai, pero iso non ten por que ser certo con punteiros. Vimos que unha das operacións de punteiros permitidas para estes é a resta, e permite desprazarnos un determinado número de posicións cara atrás na memoria, con respecto á posición á que apunta o punteiro. Por tanto, se o noso punteiro `pOrdenado` apunta, por exemplo, ao terceiro elemento de ordenado, `pOrdenado[-3]` apuntará ao primeiro elemento, isto é, ao principio do arrai ordenado.

***Exemplo***: emprego de punteiros e arrais en C++.

```
    ..........

    int main()
    {
        int ordenado[5] = {1, 2, 3, 4, 5};
        int *pOrdenado = &ordenado[0];

        cout << pOrdenado[2];       // 3
        cout << *(ordenado + 2);    // 3
        pOrdenado = &ordenado [1];
        cout << pOrdenado[2]        // 4
        cout << *(ordenado + 2);    // 3
}
```

A pesar das grandes similitudes, arrais e punteiros non son exactamente o mesmo. As principais diferenzas son:

- O identificador dun arrai compórtase como un punteiro constante (non se pode facer que apunte a outra dirección de memoria, declárase con `cons`).
- O compilador asocia unha zona de memoria para os elementos do arrai (non o fai para os elementos apuntados por un punteiro).

Cando se traballa con punteiros, existe ademais a posibilidade de empregar arrais de punteiros, isto é, un arrai onde cada un dos seus elementos é un punteiro que apuntará a un determinado tipo de dato. Polo tanto, nun arrai de punteiros, cada elemento contén unha dirección de memoria.

***Exemplo***: unha matriz de enteiros pode considerarse como un arrai de punteiros. O número de elementos virá determinado polo número de filas, onde cada unha delas se corresponde cun arrai con tantos elementos como columnas teña a matriz.

***Exemplo***: arrai representando as catro estacións do ano, da forma:

```
char *estacions[4] = {"primavera", "veran", "outono", "inverno"};
```

## 10.7 Arrais dinámicos

Ata o de agora, traballamos con arrais estáticos, cuxo tamaño e tipo son declarados en tempo de compilación, polo que non poden ser modificados durante a execución do programa. Non obstante, en ocasións non se coñece a priori o número de elementos dun arrai, e pode suceder que se reserve demasiada memoria de forma innecesaria, ou ben que non se reserve unha cantidade suficiente de memoria para o arrai. No primeiro caso, estaremos desperdiciando recursos. No segundo caso, se seguimos almacenando elementos do arrai en posicións de memoria non reservadas (dado que non se fai comprobación dos límites do arrai) é posible que xurdan problemas e inconvenientes. Por exemplo, se reservamos un arrai de enteiros `conxunto[10]`, e necesitamos máis espazo e asignamos un determinado valor á posición `conxunto[15]`, o programa pode producir un erro de segmentación, e en todo caso estará ocupando memoria non reservada, de xeito que pode sobrescribir outras variables e dar erros lóxicos.

Tampouco sería posible (no C++ estándar), dado que a memoria se reserva en tempo de compilación, deixar o tamaño do arrai determinado por unha variable tamaño, `conxunto[TAM]`, de tal modo que o valor de tamaño se introduce en tempo de execución.

Para poder traballar con arrais dinámicos será necesario poder asignar memoria en tempo de execución. Farase mediante os xa coñecidos operadores `new` e `delete`. Inicialmente, o arrai definirase como un punteiro.

```
tipo *arrai;

if((arrai = new tipo [tamano]) == NULL) {   // Reservar memoria
    cout << "Erro de asignacion de memoria\n";
}
else {
    ..........
    instrucions;
    delete [] arrai;
}
```

***Exemplo***: arrais dinámicos en C++. Ollo!: son arrais dinámicos, por iso empregamos o operador `.`, non `->`.

```
struct Libro {
    char titulo[100];
    char autor[100];
    float prezo;
    int numPaxinas;
};
..........

Libro *biblioteca;

if ((biblioteca = new Libro [5]) == NULL) {
    cout << "Erro asignacion\n";
}
else {
    for (int i = 0; i < 5; i++) {
        cin >> biblioteca[i].titulo;
        cin >> biblioteca[i].autor;
        cin >> biblioteca[i].prezo;
        cin >> biblioteca[i].numPaxinas;
    }
    delete [] biblioteca;
}
```

## 10.8 Exercicios propostos

1. Escribir un algoritmo que declare dúas variables de tipo punteiro a enteiro, `p1` e `p2` (esta apuntando a NULL), e unha variable enteira `v`, con valor 8. Apuntarase o primeiro punteiro a `v`, modificarase posteriormente o seu contido a 10. Visualizaranse todos os contidos e direccións de memoria posibles ata ese momento. Agora, igualarase este punteiro ao segundo. Visualizaranse de novo todas as variables e as súas direccións de memoria.

2. Cal é a saída do seguinte código?

```
...
int fun (int *a);

int main ()
{
    int a = 3, b = 6;

    cout << fun (&b) << endl;
    cout << b;
    return 0;
}

int fun (int *a)
{
    *a = *a + 5;
    return (*a)*(*a);
}
```

3. Se tecleamos os valores 1, 1, 2000, cal é a saída do seguinte código?

```
...
struct Data {
    int dia;
    int mes;
    int ano;
};

int fun (int tip, int *dato);

int main ()
{
    Data d = {0,0,0};

    cout << "Dame data: ";
    fun (1, &d.dia);
    fun (2, &d.mes);
    d.ano = fun (3, &d.mes);

    cout << d.dia << endl;
    cout << d.mes << endl;
    cout << d.ano << endl;
    return 0;
}

int fun (int tip, int *dato)
{
    (tip == 1) ? cout << "Dia ": (tip == 2 ? cout << "Mes ": cout << "Ano ");
    cin >> *dato;
    return *dato;
}
```

4. Que se imprime na pantalla ao executar o seguinte programa?

```
...
int fun (int n, int *vector);

int main ()
{
    int vector[] = {1, 2, 3, 4, 5};

    cout << fun (3, vector) << endl;
    return 0;
}

int fun (int n, int *v)
{
    int a = 0;
    while  (n-- > 0) {
        a += *v++;
    }
    return a;
}
```

5. Supondo que `a = 5`, `b = 4`, que imprime por pantalla o seguinte código?

```
...
void fun (int &f1, int *f2);
int main ()
{
    int a, b;
    cin >> a >> b;
    fun (a, &b);
    cout << a << ", " << b << endl;
    return 0;
}

void fun (int &f1, int *f2) {
    int f = f1 - *f2;
    f1 += *f2;
    *f2 = f;
}
```

6. Implementar un algoritmo que, mediante un punteiro a un vector de 10 enteiros, cuxos valores son introducidos por teclado, obteña o número de valores iguais a 1 que contén o vector. Non se empregará o operador `[]` para percorrer nin o vector nin o punteiro.
7. Dadas as variables enteiras `n = 5`, `v= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`, e un punteiro a enteiro `*pe`, codificar un algoritmo que apunte o punteiro a `n` e amose a dirección de `n` e o seu valor por medio da variable e do punteiro. Despois débese percorrer o vector a través do punteiro e amosar os seus elementos por pantalla.
8. Supoñamos dous vectores de números enteiros, cuxo tamaño e valores son coñecidos en tempo de execución. Obter outro vector resultante cuxa lonxitude será a suma dos dous anteriores, e os seus valores corresponderán aos do primeiro e segundo vector concatenados. Débense empregar subprogramas para introducir datos, e para obter o vector resultante. Débense tratar os vectores como punteiros.
9. Escribir un programa que permita almacenar a información dun clube de ciclismo que ten un número variable de ciclistas, coñecido en tempo de execución. Para cada ciclista deberase almacenar o nome, DNI, data de nacemento e mellor marca (tempo) da tempada. Débense escribir subprogramas para ler e visualizar os datos de cada ciclista.
10. Desexamos almacenar nun vector de tamaño variable, coñecido en tempo de execución, unha colección de números complexos. Deseñar a estrutura de datos máis adecuada. Con esta estrutura, escribir un programa que almacene un vector de números complexos e inclúa un subprograma

para calcular o módulo de cada un deles.

11. Implementar o código necesario para almacenar a información correspondente a un número variable de atletas (nome completo, nacionalidade e mellor marca -horas, minutos, segundos-). Escribir unha función para ler a información dun conxunto de atletas cuxo número introducimos por teclado, e visualizala por pantalla. Finalmente, deberá liberarse a memoria empregada.
12. Supoñamos unha táboa que contén os datos nome, materia e cualificación final dun grupo variable de estudantes, coñecido en tempo de execución. Deseñar un algoritmo que inclúa funcións para: introducir os datos de todos eles dende teclado, visualizalos, e determinar o estudante que obtivo unha maior cualificación.

# Capítulo 11

# Cadeas de caracteres

## 11.1 Conceptos xerais

As linguaxes de programación empregan determinados xogos de caracteres para comunicarse coa computadora. Estes xogos foron evolucionando dende o código máquina, baseado no sistema binario, ata os códigos actuais. Entre eles destacan o código ASCII (American Standard Code for Information Interchange), o EBDIC (Extended Binary Coded Decimal Interchange Code) e o Unicode.

**Código ASCII**

O código básico emprega 7 *bits* para representar cada carácter, isto tradúcese en 128 ($2^7$) caracteres diferentes. O ASCII estendido, amplamente utilizado, emprega 8 *bits*, polo que se poden representar 256 caracteres ($2^8$). Os primeiros 128 son estándar, os seguintes ocúpanse con caracteres específicos de cada máquina en particular. Está composto polos seguintes caracteres:

- **Alfabéticos** (a, b, ..., z, A, B, ..., Z).
- **Numéricos** (0, 1, 2, ..., 9).
- **Especiais** (+, -, *, /, ¡, . . . ).
- **Control**: non se poden imprimir (funcións de escritura, separación de arquivos,...):
  - DEL: borrar.
  - STX: inicio de texto.
  - LF: avance de liña.
  - FF: avance de páxina.
  - CR: retorno de carro.

**Código EBDIC**

É un código similar a ASCII, e utiliza 8 *bits* para representar cada carácter, polo tanto, de novo 256 caracteres diferentes. Inclúe tamén caracteres alfanuméricos especiais.

**Unicode**

É un estándar internacional, con aplicacións en Internet e alfabetos internacionais. Emprega 2 *bytes* (16 *bits*), e permite representar 65536 ($2^{16}$) caracteres diferentes.

**Secuencias de escape**

Ademais dos xogos de caracteres, é necesario que unha linguaxe de programación poida representar aqueles caracteres que non se poden introducir tal e como son dende o teclado, como pode ser un tabulador, ou un retorno de carro. Estas representacións denomínanse secuencias de escape, e están formadas por:

- **Carácter escape**: símbolo que indica ao compilador que hai que traducir o carácter de xeito especial. Exemplo: '\' en C++.
- **Valor de tradución**: valor ao que representa. Pódense utilizar valores de xogos de caracteres. Os Unicode son os máis sinxelos. Deben especificarse como a letra u seguida por un número de catro díxitos en hexadecimal. Exemplo: ('\u000D' : retorno de carro).

Algunhas das secuencias de escape máis habituais son as seguintes:

- \b: retroceso.
- \t: tabulación.
- \n: nova liña.
- \r: retorno de carro.

Todas as anteriores secuencias (ademais doutras) pódense utilizar ao programar en case calquera linguaxe de programación, como por exemplo en C++.

Ademais de poder ser empregados nestas secuencias de escape, que representan caracteres especiais, os diferentes xogos de caracteres permiten traballar con cadeas.

Unha **cadea de caracteres** é un conxunto ordenado de caracteres (incluído o espazo en branco) que pertencen a un código determinado e están almacenados en áreas contiguas da memoria.

Denomínase **lonxitude da cadea** ao número de caracteres que contén. Existe tamén a **cadea baleira** ou **nula**, que é aquela que non contén ningún carácter: lonxitude cero.

As cadeas represéntanse mediante comiñas dobres `" "` ou simples `' '`, dependendo da linguaxe de programación. En C ++ emprégase `" "`.

Existen cadeas constantes e variables.

**Cadeas constantes**

Son conxuntos de caracteres válidos encerrados entre comiñas.

**Cadeas variables**

Son variables cuxo valor é unha cadea de caracteres, e teñen unha lonxitude determinada. Como veremos, non son máis que vectores de caracteres. Deste modo, se consideramos que a lonxitude da cadea é lon, cada un dos seus elementos pode ser accedido igual que cada elemento dun vector: `cad[i]`, onde `i` varía entre [0, lon-1]. En función do método de almacenamento, distínguense tres tipos:

- **Cadeas de lonxitude fixa**: vectores de lonxitude declarada, con brancos á esquerda ou dereita se a cadea non alcanza esta lonxitude (Figura 11.1).

| c | a | d | e | a | | d | e | | l | o | n | x | i | t | u | d | e | | f | i | x | a | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Figura 11.1: Representación das cadeas de lonxitude fixa

- **Cadeas de lonxitude variable cun máximo**: teñen unha lonxitude variable (cun máximo). Considérase un punteiro con dous campos (con lonxitude máxima actual utilízase unha marca para indicar a fin da cadea ('\0') ) (Figura 11.2).

| c | a | d | e | a | | v | a | r | i | a | b | l | e | \0 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Figura 11.2: Representación das cadeas de lonxitude variable cun máximo

- **Cadeas de lonxitude indefinida**: represéntanse mediante listas enlazadas (estruturas dinámicas, listas unidas mediante punteiros) (Figura 11.3).


Figura 11.3: Representación das cadeas de lonxitude indefinida

As cadeas deben ser declaradas como calquera outra variable. Cando se traballa con cadeas, e dependendo da linguaxe de programación empregada, existen numerosas funcións que se poden utilizar para o tratamento das mesmas, e que permitirán a súa manipulación con maior ou menor facilidade.

Entre as operacións con cadeas máis habituais están: acceso aos seus membros, lectura e escritura, asignación, cálculo da lonxitude, comparación, concatenación, extracción de subcadeas, busca, inserción, borrado, substitución, e conversión entre cadeas e números. Veremos a continuación o tratamento de cadeas en C++.

A linguaxe de programación C++ proporciona unha serie de funcións para o manexo de cadeas. Pódense tratar de dúas formas diferentes. A primeira delas considera que son vectores de caracteres, polo que o seu manexo, elemento a elemento, será similar ao de calquera vector, que xa coñecemos. A segunda opción permite o tratamento mediante a clase `String`, ausente en C (que só permite tratar as cadeas como vectores de caracteres). Veremos a continuación cada unha destas dúas opcións.

Coas cadeas, ao igual que co resto de variables, pódense realizar operacións de E/S, polo que poden ser lidas dende teclado ou dende un dispositivo de entrada, e enviadas a unha saída como unha pantalla, ou a un ficheiro.

## 11.2 Tratamento mediante vectores de caracteres

### 11.2.1 Declaración e inicialización

En C++, unha cadea é un vector de caracteres parcialmente cheo, onde o carácter nulo ('\0') indica o final da mesma, e se engade automaticamente. Polo tanto, ao declararmos unha cadea temos que reservar espazo para un carácter máis que o requirido pola cadea de maior tamaño que se desexa almacenar.

Faise mediante a instrución:

```
char nomeCadea [tamMaximoCadea + 1];
```

***Exemplo***: declaración de diferentes cadeas en C++.

```
char cad1[16] = "Cadea en C++";
char cad2[] = "Cadea en C++";
char cad3[50] = "Cadea en C++";
```

En relación coa súa inicialización, pódese inicializar unha cadea mediante o operador de asignación, das seguintes formas:

- Dándolle valor a cada elemento do vector:

```
char nomeCadea1[11] = {'a','b','c','d',...,'\0'};
```

- Asignándolle directamente unha constante de cadea de caracteres. Non é necesario especificar o carácter nulo; este engádese automaticamente á fin da cadea.

```
char nomeCadea2[11] = "Caracteres asignados á cadea";
```

- Mediante un arrai indeterminado: defínese automaticamente un vector de tamaño suficientemente grande para almacenar a cadea xunto con seu carácter nulo.

```
char nomeCadea3[] = "Caracteres asignados á cadea";
```

***Exemplo***: asignación de cadeas en C++.

```
char cad1[] = "Cadea en C++";
char cad2[10] = {'O','l','a','\0'};
char cad3[] = "Cadea de proba";
```

### 11.2.2 Acceso aos elementos

Dado que unha cadea non é máis que un vector de caracteres, é posible acceder a cada elemento dunha cadea mediante o operador de indexación `[]`. Polo tanto, a indexación comezará en 0 e terminará na lonxitude da cadea menos un. Ao igual que nos arrais, non se revisa se o acceso a unha determinada posición é valido.

***Exemplo***:

```
char cadea = "cadea como vector de caracteres";
cadea[0]; // Conten 'c'
```

### 11.2.3 Lectura

En C++ existen diversas formas de ler unha cadea.

Unha forma é ler carácter a carácter, igual que fariamos con cada compoñente dun vector.

Pódese realizar tamén a lectura dunha cadea de caracteres a unha variable tipo cadea.

É posible ler unha cadea usando o operador de entrada >>, pero o espazo en branco (espazos e saltos de liña) delimita a fin dunha cadea. Polo tanto, con este operador lerase unicamente a porción dunha cadea ata chegar ao primeiro espazo en branco.

```
char nomeCadea[tamMaximoCadea + 1];
cin >> nomeCadea:                          // Lese ata o primeiro espazo en branco
```

Se se desexa ler toda unha liña de entrada e colocala nunha variable de cadea de caracteres, úsase a función predefinida **`getline()`**. Esta función ten dous parámetros: o primeiro é a variable cadea de caracteres que recibe a entrada, e o segundo é un enteiro que, normalmente, denota o tamaño co que se declarou a variable. Este segundo argumento serve para indicar o número máximo de elementos do vector (variable) que **`getline()`** pode encher con caracteres. A sintaxe é:

```
char nomeCadea[tamMaximoCadea + 1];
cin.getline(nomeCadea, tamMaximoCadea);     // Lese toda a liña
```

Cando se especifica o segundo argumento de **`getline()`**, debe terse en conta que o carácter nulo ocupa unha posición do vector.

Coa función **`getline()`** a lectura remata nas seguintes situacións:

- Cando termina a liña, aínda que a cadea resultante sexa máis curta que o número de caracteres especificados polo segundo argumento.
- Cando colocou no vector o número de caracteres especificados polo segundo argumento, aínda que non se chegara ao final da liña.

### 11.2.4 Escritura

Para escribir unha cadea, o procedemento é similar ao empregado cos demáis tipos de variables.

```
cout << nomeCadea;
```

***Exemplo***: lectura e escritura de cadeas en C++.

```
char cad1[] = "Cadea en C++";
char cad2[10];
char cad3[50];
..........

cin >> cad2;
getline(cad3, 49);
cout << cad1;
```

### 11.2.5 Asignación

En C++ existen varias formas de asignar valores a unha cadea. A asignación directa unicamente se pode realizar cando se fai ao declarar a cadea. Xa a vimos na inicialización de cadeas.

Dado que as cadeas son tratadas como vectores de caracteres, e non como valores simples, non se lles pode aplicar o operador de asignación ($=$) para realizar unha asignación fóra da súa declaración. A función predefinida que permite asignar unha cadea é:

```
strcpy(cadeaDestino, cadeaOrixe)
```

Copia a cadea contida en **cadeaOrixe** en **cadeaDestino**, sen comprobar se esta cadea ten suficiente espazo. E engade o carácter '\0' ao final da cadea resultante.

### 11.2.6 Cálculo da lonxitude

Existe unha función que permite calcular o número de caracteres dunha cadea, e vén dada por:

```
strlen(cadeaOrixe)
```

Devolve un enteiro que representa a lonxitude de **cadeaOrixe**, sen ter en conta o carácter nulo '\0'.

Para empregar esta función é necesario incluír a directiva de preprocesador `#include <cstring>` ou ben `#include "string.h"`. O mesmo pasa coas funcións para tratamento de cadeas que veremos a continuación.

### 11.2.7 Comparación

Dado que as cadeas son vectores de caracteres, do mesmo modo que non se pode utilizar con elas o operador de asignación, tampouco se poden utilizar os de comparación ($==, !=, >, >=, <, <=$). A función que realiza esta operación é a seguinte:

```
strcmp(cadea1, cadea2)
```

Compara as cadeas contidas en **cadea1** e **cadea2**. A comparación realízase analizando os dous primeiros caracteres das dúas cadeas. Se coinciden, pásase aos dous seguintes, e así sucesivamente ata que se atopan dous caracteres distintos na mesma posición nas dúas cadeas, ou se atopa a marca de fin de cadea '\0' nunha das dúas.

Esta función devolve un dos seguintes valores:

- 0 se as dúas cadeas son iguais.
- Un enteiro menor que 0 se **cadea1** é menor que **cadea2**.
- Un enteiro maior que 0 se **cadea1** é maior que **cadea2**.

### 11.2.8 Concatenación

É posible concatenar cadeas mediante funcións predefinidas en C++. Empregaremos:

```
strcat(cadeaDestino, cadeaOrixe)
```

Concatena a **cadeaOrixe** ao final da cadea que está en **cadeaDestino**, sen comprobar se esta cadea ten suficiente espazo para o resultado da concatenación.

### 11.2.9 Busca

É posible buscar caracteres e cadeas noutras cadeas. Unha das funcións máis habituais é:

```
strstr(cadea1, cadea2)
```

Busca **cadea2** en **cadea1**, e devolve un punteiro ao primeiro carácter de **cadea1** que coincide con **cadea2**. Se **cadea2** non está en **cadea1**, devolve NULL.

### 11.2.10 Conversión entre cadeas e números

Podemos converter cadeas a números. Empréganse:

```
atoi(cadIni) // Resultado: número enteiro que resulta de converter en díxitos cadIni
atof(cadIni) // Resultado: número real que resulta de converter en díxitos cadIni
```

***Exemplo***: emprego de funcións de cadeas como vectores de caracteres en C++ (Programa 11.1).

Programa 11.1: Emprego de funcións de cadeas como vectores de caracteres en C++

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

int main()
{
    char cad1[]= "Programacion";
    char cad2[]=" I";
    char cad3[50];                  // Ilegal: char cad[]
    int lon;

    strcpy (cad3, cad1);            // Asignación: cad3: "Programacion"
    lon = strlen(cad1);             // Calcular lonxitude
    cout << "Lonxitude: " << lon ;  // Saída: 12

    if (!strcmp(cad1, cad3)) {      //  if(strcmp(cad1, cad3) == 0) // Comparación
        cout << "Son iguais\n";
    }
    else {
        cout << "Son distintas\n";
    }

    strcat(cad3, cad2);             // Concatenación
    cout << cad3;                   // Resultado: "Programacion I"
```

```
    cout << strstr(cad1, "ma");     // Busca : Resultado: "macion"

    atoi("123")                     // Saída: 123 (numérico)
    atof("12.3")                    // Saída: 12.3 (numérico)

    return 0;
}
```

## 11.3 A clase `String` de C++

Ademais de tratar as cadeas como vectores de caracteres, en C++ existe unha clase propia, denominada **`String`**, que permite traballar coas cadeas dunha forma moi sinxela e eficiente, creando obxectos (tipo de datos abstracto definidos por quen programa) desta clase. Para iso, é necesario incorporar á cabeceira dos programas a biblioteca **`string.h`**.

### 11.3.1 Declaración e inicialización

Farase da seguinte forma:

```
string identificadorObxecto;
```

De forma predeterminada, **`identificadorObxecto`** inicialízase á cadea baleira, aínda que se lle pode asignar un valor concreto:

- Se se usa un só argumento: constante de cadea de caracteres.

  ```
  string nomeString("cadea do string");
  ```

  Créase **`nomeString`**, que contén os caracteres que están na constante de cadea de caracteres `"cadea do string"`.

- Se se usan os dous argumentos, o primeiro deles indica o número de caracteres, e o segundo o carácter que compón esta cadea.

  ```
  string nomeString(num, 'carácter');
  ```

  Créase **`nomeString`** con **`num`** caracteres iguais.

Un obxecto da clase **`String`** tamén pode ser inicializado mediante o operador de asignación $=$.

```
string nomeString = "Cadea do string";
```

No é posible converter **`int`** ou **`char`** a **`string`** durante a definición do obxecto **`String`**. Así, son ilegais as instrucións: **`string s1 = 'y'`**, **`string s2 = 5`**, **`string s3 = ('y')`**, **`string s4(3)`**.

Si é posible, en cambio, asignar un só carácter a un obxecto **`String`** nunha sentenza de asignación.

***Exemplo***:

```
string s1; s1 = 'n';
```

### 11.3.2 Acceso aos elementos

É posible acceder a cada elemento dun obxecto **`String`** mediante o operador de indexación `[]`, de forma similar ao acceso aos elementos dun vector. Polo tanto, a indexación comezará en 0 e terminará na lonxitude do obxecto **`String`** menos un. Ao igual que nos arrais, non se revisa se o acceso a unha determinada posición é valido.

***Exemplo***:

```
string nomeString = "cadea de string";
nomeString[0]; // Contén 'c'
```

Outra forma de acceder a cada elemento do obxecto `String` é mediante a función membro `at()`, que proporciona acceso a caracteres individuais con verificación de intervalo. É dicir, se tentamos acceder fóra do intervalo, a función fará que o programa termine anormalmente.

```
string nomeString = "cadea de string";
nomeString.at(0);   // Contén 'c'
```

### 11.3.3 Lectura

Emprégase o operador de extracción >>, delimitado por caracteres de espazo en branco. Se se desexa ler unha liña completa (delimitada por '\n') será necesario usar a función independente `getline()`, onde o primeiro argumento é un fluxo de entrada e o segundo un obxecto `String` (o carácter '\n' elimínase da entrada). Usando esta mesma función podemos determinar o final da entrada mediante un terceiro argumento, que é un carácter cuxa presenza finaliza a entrada.

### 11.3.4 Escritura

Para escribir unha cadea, o procedemento é similar ao empregado cos demáis tipos de variables.

```
cout << nomeString;
```

***Exemplo***: lectura e escritura de `String` en C++.

```
string nomeString;

cin >> nomeString;              // Le palabra
getline(cin, nomeString);       // Le liña completa
getline(cin, nomeString, '*');  // Le liña ata chegar a '*'

cout << nomeString;
```

### 11.3.5 Asignación

Existen dúas formas:

- Mediante o operador de asignación =.

```
string nomeString1;
string nomeString2 = "cadea de string2";
nomeString1 = nomeString2;
```

- Mediante a función membro `assign()`.

```
string nomeString1;
string nomeString2 = "cadea de string2";
nomeString1.assign(nomeString2);
```

Unha versión da función `assign()` permite copiar nun obxecto `String` un número especificado de caracteres procedente doutro obxecto `String`. Para iso usa tres argumentos: o primeiro é o obxecto `String` que se vai copiar, o segundo o subíndice inicial, e o último o número de caracteres que se desexan copiar.

### 11.3.6 Cálculo da lonxitude

Almacénase no obxecto e pode ser recuperado mediante a función membro `length()`.

```
string nomeString;
int lon;
lon = nomeString.length();
```

### 11.3.7 Comparación

Existen dúas formas:

- Mediante os operadores de comparación ($==, !=, >, >=, <, <=$): devolven valores *bool*:

  ```
  nomeString1 == nomeString2;
  nomeString1 != nomeString2;
  nomeString1 > nomeString2;
  ...
  ```

- Mediante a función membro `compare()`: devolve 0 se ambos os dous obxectos `String` son equivalentes; Un número positivo se o primeiro obxecto `String` é lexicograficamente maior que o segundo; e un número negativo se o primeiro obxecto `String` é lexicograficamente menor que o segundo.

  ```
  nomeString1.compare(nomeString2);
  ```

### 11.3.8 Concatenación

Ao igual que a comparación, pódese realizar de dúas formas:

- Mediante o operador +:

  ```
  string nomeString1, nomeString2;
  getline(cin, nomeString1);
  getline(cin, nomeString2);
  nomeString1 += nomeString2;
  ```

- Mediante a función membro `append()`:

  ```
  string nomeString1, nomeString2;
  getline(cin, nomeString1);
  getline(cin, nomeString2);
  nomeString1.append(nomeString2);
  ```

  Unha versión da función `append()` permite concatenar a un obxecto `String` un determinado número de caracteres doutro obxecto `String`. Para iso usa tres argumentos: o primeiro corresponde ao obxecto `String` dende o que se copia, o segundo indica o carácter dende o cal concatena, e o último argumento indica o número de caracteres que concatena.

  ```
  string nomeString1 = "cadea de string1";
  string nomeString2 = "cadea de string2";
  //Concatena en nomeString1 num caracteres de nomeString2, contados dende pos
  nomeString1.append(nomeString2, pos, num);
  ```

### 11.3.9 Busca

Mediante a función membro `substr()` podemos buscar e recuperar unha subcadea dun obxecto `String`. O seu primeiro argumento especifica o subíndice inicial da subcadea, e o segundo o número de caracteres que se queren extraer.

```
string nomeString1;
nomeString1.substr(pos, n2);    // Extrae n2 caracteres dende pos
```

### 11.3.10 Intercambio

Pódense intercambiar dous obxectos `String` mediante a función membro `swap()`:

```
string nomeString1;
string nomeString2;
nomeString1.swap(nomeString2);  // Intercambia os dous strings
```

***Exemplo***: emprego de funcións de cadeas como obxectos `String` en C++ (Programa 11.2).

Programa 11.2: Emprego de funcións de cadeas como `Strings` en C++

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

int main()
{
    string s1 = "Programacion";
    string s2;
    string s3;
    int lon;

    getline(cin, s2);               // Lectura: " I"

    s3 = s1;                        // Asignación: cad3: "Programacion"

    cout << s3[0];                  // Acceso a elementos. Saída: P

    lon = s1.lentgh();              // Calcular lonxitude
    cout << "Lonxitude: " << lon ;  // Saída: 12

    if (!s1.compare(s3)) {          // Comparación
        cout << "Son iguais\n";
    }
    else {
        cout << "Son distintas\n";
    }

    s3 = s1 + s2;                   // Concatenación
    cout << s3;                     // Resultado: "Programacion I"

    s1.substr(0, 3);                // Extracción de 3 caracteres dende pos = 0: "Pro"

    swap(s1, s2)                    // Saída: s1: " I", s2: "Programacion"

    return 0;
}
```

## 11.4 Exercicios propostos

1. Que fai a seguinte función?

```
void fun (char cad1[], char cad2[])
{
```

```
    int i = 0;
    while (cad2[i] != '\0'){
        cad1[i] = cad2[i];
        i++;
    }
}
```

2. Que fai a seguinte función?

```
void fun (char *cad1, char *cad2)
{
    while (*cad2++ = *cad1++);
}
```

3. Que se imprime ao executar a instrución `fun ("casa")`?

```
void fun (char cad[])
{
    int i = 0, n = 5;
    char p[5];
    strcpy (p, cad);
    while (p[i] != '\0') {
        i++;
    }
    cout << n << p;
}
```

4. Escribir un programa que pida unha frase por teclado e, mediante un subprograma, conte o número de vocais que posúe.
5. Escribir un subprograma que reciba como entrada unha frase e un par de caracteres, e conte o número de ocorrencias do par de caracteres na cadea.
6. Codificar un algoritmo que solicite por teclado unha cadea e dous caracteres por teclado, e substitúa a aparición do primeiro carácter na cadea polo segundo.
7. Escribir un programa que pida por teclado o seu nome e apelidos, e os amose coas maiúsculas e minúsculas correctas (aparecerán en maiúsculas a primeira letra e as que haxa tras cada espazo; as demais aparecerán en minúsculas).
8. Repetir o Exercicio 7 empregando punteiros.
9. Repetir o Exercicio 5 empregando punteiros para a cadea e o par de caracteres.
10. Codificar unha función para verificar se dúas palabras, tratadas como vectores de caracteres, son anagramas (cambios na orde das letras dunha palabra ou frase, que da lugar a outra palabra ou frase diferente, como por exemplo casa e saca).
11. Dado un `String s`, que fai o seguinte fragmento de código?

```
for (int i = 0; i < s.length(); i++) {
     if (s.at(i) == ' ') {
        s.at(i) = '*';
    }
}
cout << s << endl;
```

12. Supondo definidos os `String s1 = "Outra vaca no millo"`, e `s2 = "Nunca choveu que non escampara "`, que se almacena en `s2` despois da execución das seguintes instrucións?

```
for (int i = 0; i < s1.length(); i++) {
    if (i %2 == 0) {
        s2.at(i) = s1.at(i);
    }
}
```

13. Supondo definidos os `String s1 = "Outra vaca no millo"`, `s2 = "centeo"`, e `s3`, que se almacena en `s3` despois da execución das seguintes instrucións?

```
s3.append(s1, 0, 14);
s3 += s2;
```

14. Empregando funcións típicas de `String`, implementar un programa que conteña unha función que conte o números de vocais existentes nunha frase introducida por teclado.
15. Empregando `String`, escribir un programa que lea por teclado o nome e os apelidos dunha persoa, e os combine nunha única cadea, separando cada palabra por un espazo en branco.
16. Escribir un programa que pida por teclado un `String` e substitúa todos os espazos en branco por un asterisco.
17. Deseñar un algoritmo recursivo que pida o seu nome ao usuario, almacenado como `String`, e o escriba ao revés.

# Fontes bibliográficas


[1] Betancourt Uscátegui, J.F., Polanco Guzmán, I.Y., *115 Ejercicios Resueltos de Programación C++*, 1ª ed, Ra-Ma. ISBN: 978-84-18551-29-1 (2021).

[2] Brookshear, J. Glenn. *Introducción a la Computación*, 12ª ed. Pearson Educación. ISBN: 978-84-7829-139-7 (2013).

[3] Ceballos Sierra, F.J. *C/C++ Curso de Programación*, 5ª ed, Ra-Ma. ISBN: 978-84-9964-812-5 (2019).

[4] Hernández Orallo, E., Hernández Orallo, J. Juan Lizandra, M.C. *C++ Estándar*, Ediciones Thomson-Paraninfo, ISBN: 84-9732-040-9 (2002).

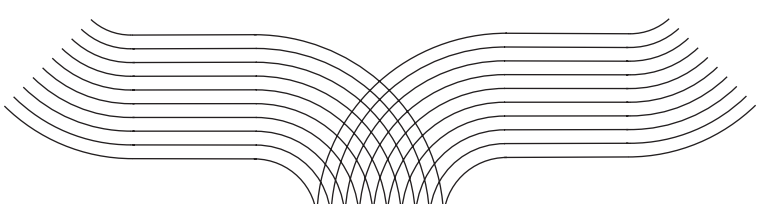
[5] Joyanes Aguilar, L. *Fundamentos de Programación*, 5ª ed. McGraw Hill. ISBN: 978-607-15-1468-4 (2020).

[6] Joyanes Aguilar, L., Rodríguez Baena, L., Fernández Azuela, M. *Fundamentos de Programación. Libro de Problemas*, 2ª ed. McGraw Hill. ISBN: 84-481-3986-0 (2003).

[7] Joyanes Aguilar, L., Zahonero Martínez, I. *Programación en C: Metodología, Algoritmos y Estructuras de Datos*, 2ª ed. McGraw Hill, ISBN: 84-481-9844-1 (2005).

[8] Prieto Espinosa, A., Lloris Ruiz, A., Torres Cantero, J.C. *Introducción a la Informática*, 4ª ed. McGraw-Hill. ISBN: 84-481-4624-7 (2006).

[9] Virgós Bel, F., Segura Casanova, J. *Fundamentos de Informática: En el Marco del Espacio Europeo de Enseñanza Superior*, 1ª ed. McGraw-Hill. ISBN: 84-481-6747-3 (2008).



# Manuais

Serie de manuais didácticos

## Últimas publicacions na colección

*Economates. As Matemáticas na Economía (2024)*
Carmen Vázquez Pampín

*Fundamentos de Meteoroloxía en 180 preguntas (2024)*
Luis Gimeno, Raquel Nieto, Marta Vázquez, Rogert Sorí e Luis Gimeno-Sotelo.

*Manual de cultivo e asentamento do ourizo de mar. Paracentrotus Lividus.(2024)*
Estefanía Paredes Rosendo e Alba Lago Dopico

*Problemas de Ocenografía Física: Ejercicios resueltos con y sin solución numérica (2024)*
Gabriel Rosón Porto

*Pensamento, sociedade e cultura. A relevancia da filosofía na educación (2024)*
Abraham Rubín Álvarez e Brais González Arribas

9 7 8 8 4 1 1 8 8 0 3 4 3

# Introdución á programación

*en Linguaxe C*

A programación é fundamental para o desenvolvemento de aplicacións cun impacto elevado en diferentes sectores, como a saúde, a educación ou as finanzas. Na actualidade, a demanda de profesionais con competencias no desenvolvemento de ferramentas software ten experimentado un aumento significativo.

Deste xeito, a programación convértese nunha habilidade esencial para o estudantado de diversas Enxeñarías, como pode ser a Enxeñaría Informática, que estará capacitado para a creación de software e solucións tecnolóxicas para problemas de diferente natureza. Ademais, se fomenta tamén o pensamento crítico, a creatividade e a capacidade de innovación.

Unha das linguaxes máis amplamente empregadas para a aprendizaxe de programación é C++, linguaxe de propósito xeral que prepara ao estudantado para enfrontar desafíos no mundo real, e proporciona as ferramentas necesarias para o desenvolvemento de aplicacións robustas e eficientes.

Neste libro preséntanse os fundamentos da programación procedimental en C++. O enfoque é eminentemente práctico, e inclúe numerosos exemplos de código explicados, así como exercicios ao remate de cada capítulo.

Abórdanse con detalle diferentes contidos, dende os conceptos máis básicos, como a terminoloxía empregada en tarefas de programación ou a definición de variables e constantes, ata temas máis avanzados, como o manexo de arquivos, a xestión dinámica de memoria, ou o tratamento de cadeas de caracteres, a programación estruturada, os vectores ou as matrices.

Preténdese que os contidos estudados permitan unha aprendizaxe sinxela desta linguaxe de programación, de xeito que se facilite unha posterior transición a outras linguaxes de programación, como Java o Python

Servizo de Publicacións
Universidade Vigo