Objectius de la pràctica:
- Saber aplicar el mètode de disseny descendent per a obtenir programes modulars.
- Emprar les funcions i la modularització de programes correctament.
- Reconèixer la conveniència d'aplicar el pas de paràmetres a les funcions per valor o per referència.
- Entendre el concepte d'àmbit de les variables i la seua aplicació al disseny de subprogrames.
- Saber usar les funcions de llibreria.

Programació modular

Els problemes més complexos es resolen millor quan es divideixen en subproblemes o mòduls. La resolució per separat de cadascun d'aquests subproblemes és més fàcil que la resolució de tot el problema com una unitat indivisible. Aquesta estratègia s'anomena programació modular o descendent.

En el cas de la programació, la tècnica de disseny descendent (*top-down*) de programes divideix successivament el problema inicial en subproblemes més senzills fins que aquesta subdivisió arriba a un nivell de dificultat mínim en el qual la conversió del subproblema en un subprograma siga més directa.

Funcions i procediments

Existeixen dos tipus de mòduls independents o subprogrames: els procediments i les funcions. Als subprogrames que retornen un valors se'ls anomena FUNCIONS. Si un subprograma no retorna cap valor rep el nom de PROCEDIMENT. Tanmateix, el llenguatge C no fa aquesta distinció. En C només existeixen les funcions i els procediments s'interpreten com un exemple particular de funció que no retorna cap valor.

Les funcions en C/C++

Un programa en C no és més que una col·lecció de funcions, entre les quals sempre hi trobarem la funció principal (*main*).

Què és una funció en C/C++?

Una funció és un mòdul independent de codi, dissenyat per a realitzar una tasca específica i que té assignat un nom com a identificador, mitjançant el qual altres funcions poden invocar-lo i llançar la seua execució.

Definició de funcions

Per a crear una nova funció cal definir-la. Dins de la definició fixarem el nom de la funció, el tipus del resultat que retorna, la llista d'arguments i el cos de la funció, d'acord amb la sintaxi següent:

Sintaxi:

```
tipus_valor_retorn nom_funció (llista_de_paràmetres)
{
    Cos de la funció
}
```

On:

- `tipus_valor_retorn` (tipus de la funció): és el tipus del valor que retorna la funció com a resultat de la seua execució mitjançant la sentència `return`. Si la funció no retorna

cap valor, açò és, si no existeix cap sentència `return` dins del seu cos, el tipus de la funció haurà de ser `void`.

- `nom_funció`: és l'identificador de la funció.
- `llista_de_paràmetres`: és una llista de variables precedides pel seu tipus i separades per comes. Aquestes variables copien els valors dels arguments que utilitzem quan cridem a la funció. Quan la funció no té arguments cal deixar buit l'espai entre parèntesis, o bé posar entre ells la paraula reservada `void`, per a indicar exactament que la funció està buida d'arguments.
- `Cos de la funció`: és el bloc de sentències que defineixen el que fa la funció. Es troba tancat entre claus.

Exemple de funció i de procediment:

```
int major(float num1, float num2)
{
    float aux;

    if (num1 > num2)
        aux = num1;
    else
        aux = num2;

    return aux;
}
```

Un procediment té a void el tipus de dades que retorna:

```
void salutacio()
{
    cout << “Hola, com va?\n”;
}
```

Crida a les funcions

Per a utilitzar les funcions que hem declarat només hem de cridar a la funció pel seu nom, tot passant-li entre parèntesis tants valors (arguments) com paràmetres formals hàgem definits. Les funcions poden ser cridades des de qualsevol punt del programa, siga des de un altre subprograma (o funció), siga des de la funció principal.

Els valors que emprarem en la crida es copiaran sobre les variables definides com a paràmetres. Llavors, ha d'existir una correspondència, no sols entre el nombre de valors i el nombre de paràmetres sinó també entre els seus tipus.

Exemple de crida:

```
int main()
{
    float num1, num2, resul;
    cout << “Introdueix dos valors reals\n”;
    cin >> num1 >> num2;
    resul = major(num1, num2);
    salutacio();
    return 0;
}
```

Paràmetres formals i reals

Per paràmetres formals entenem els paràmetres que emprem en la codificació d'un programa. Per exemple, en la funció:

```
int major(float num1, float num2)
{
    ...
}
```

els paràmetres `num1` i `num2` reben el nom de paràmetres formals. Pel contrari, quan la funció main anterior cridava a la funció `major`, les variables prenien un valor concret i rebien el nom de paràmetres reals.

La sentència return

Podem retornar un valor que es troba a dins del cos de la funció mitjançant la sentència `return`. A més a més, la sentència `return` suposa realitzar un salt en la execució que provoca l'eixida immediata de la funció, independentment de la seua posició relativa dins del cos de la funció. Així, doncs, una funció acaba quan s'executa la sentència `return` o bé, en el cas que no retorne cap cosa, quan acabe el bloc de sentències.

Sintaxi: `return (valor);`

El valor que acompanya a la sentència `return` ha de ser del mateix tipus que el tipus de retorn de la funció. No res impedeix que dins del cos d'una sentència existeixen diverses sentències `return`, tot i que només una s'executarà, ja que amb aquesta sentencia finalitza l'execució de la funció.

Àmbit de les variables

Recordem que hem definit les funcions com subprogrames independents que realitzen una determinada tasca especificada en el seu cos. Aquestes funcions treballaran amb dades emmagatzemades en variables que podran ser de classes diferents en funció del lloc on estiguen declarades:

- Variables globals: Estan declarades fora del cos de qualsevol funció i abans que siguen emprades per cap d'aquelles. Es recomana que es declaren al començament del programa, abans de les definicions de les funcions. L'àmbit d'aquestes variables és global, açò és, són visibles des de qualsevol funció i totes elles poden modificar el seu valor. La vida d'aquestes variables va lligada a la del programa: es creen quan comença l'execució del programa i poden ser utilitzades fins al seu acabament.
- Variables locals: Estan declarades dins de les funcions, siga en el seu cos o en els seus paràmetres formals. L'àmbit de la variable està limitat a la funció on estiga declarada i fora d'aquesta no és visible. Llavors, només pot ser utilitzada per la funció on estiga declarada. Cap subprograma és capaç d'accedir a les variables locals d'un altre subprograma. Aquest fenomen, del qual el programador pot traure molt de profit, s'anomena “encapsulació” de les dades.

Pas de paràmetres per valor i pas per referència

Els paràmetres d'una funció es poden definir de dues maneres diferent, aspecte que determina la manera en què es passaran els arguments a les funcions. Es poden passar arguments per valor o per referència.

Pas per valor

Aquest mètode copia el valor dels arguments sobre el paràmetres formals, de manera que els canvis de valor dels paràmetres no afecten a les variables utilitzades com a arguments en la crida a la funció. Allò més important en el pas per valor és el contingut de l'argument. És per aquesta raó que serà indiferent si el que es passa com a argument és una variable, una constant o una expressió. Per exemple:

```
#include <iostream>
#include <math.h>
using namespace std;

/* Prototips de les funcions */
void ender (float param1, int param2, int p3);

/* Definició */
void ender (float param1, int param2, int p3)
{
    p3 = param1 * param2 – 5;
}

int main (void)
{
    int a = 5, b;
    float h = 9.0;

    b = -1;
    ender (2 * sqrt(h), 3 * 7 + a, b);

    /* Després de la crida, b continua essent -1 */
    return 0;
}
```

En l'exemple anterior, la crida a la funció `ender` li passa tres valors. Els tres valors es copien sobre els tres paràmetres de la funció:

```
param1 = 2 * sqrt(h);         /* param1 = 6 */
param2 = 3 * 7 + a;           /* param2 = 26 */
p3 = b;                       /* p3 = -1 */
```

Tot i que dins de la funció el contingut del paràmetre `p3` es veu alterat, açò no modifica el valor de la variable emprada com a argument (`b`).

En resum, en el pas per valor els paràmetres copien el valor dels arguments. Encara que aquests paràmetres canvien de valor dins de la funció, el valor de les variables utilitzades en la crida no es veu modificat. Per tant, els paràmetres sols serveixen com a entrada de dades a la funció.

Veiem un altre exemple de pas per valor:

```
void funcioA (void)
{
    int a;
    a = 5;
    cout << “Sóc la funció A i vaig a passar un “ << a << “ a la funció B.\n”;
    funcioB(a);
    cout << “Sóc de nou la funció A i a val ” << a << endl;
}
```

```
void funcioB (int b)
{
    cout << “Sóc la funció B i m'han passat un ” << b << endl;
    b = 2;
    cout << “Sóc de nou la funció B i he canviat el paràmetre a ” << b << endl;
}
```

L'exemple anterior mostrarà els següents missatges per pantalla:

```
Sóc la funció A i vaig a passar un 5 a la funció B.
Sóc la funció B i m'han passat un 5
Sóc de nou la funció B i he canviat el paràmetre a 2
Sóc de nou la funció A i a val 5
```

En l'exemple anterior hem passat la variable `a` com a argument per valor. Açò significa que la `funcioB` pot conèixer i emprar el valor de `a`, però no pot modificar-lo.

Pas per referència

Al contrari del pas per valor, en el pas per referència els paràmetres no copien el valor de l'argument sinó que comparteixen el seu valor. Aleshores, quan canvia el valor del paràmetre també canvia el valor de la variable utilitzada com a argument en la crida.

Llavors, els paràmetres definits per referència es poden utilitzar tant per a l'entrada com per a l'eixida de dades. A banda de la sentència return, aquest és l'altre mecanisme que posseeixen les funcions per poder retornar valors.

Una conseqüència evident és que, en la crida a una funció, els paràmetres per referencia han de ser sempre variables (no poden ser ni constants ni expressions).

Executeu el següent codi i comproveu el resultat:

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

/* Prototips de les funcions */
void intercanvi (float & x, float & y);

/* Definició */
void intercanvi (float & x, float & y)
{
    float t;

    t = x;
    x = y;
    y = t;
}

int main (void)
{
    float h, j;

    cout << “Introdueix dos nombres: “ << endl;
    cin >> h >> j;
    cout << “La primera variable val “ << h << “ i la segona “ << j << endl;
    intercanvi(h, j);
    cout << “Ara la primera variable val “ << h << “ i la segona “ << j << endl;
    return 0;
}
```

Un altre exemple de l'efecte del pas per referència és el següent:

```
void funcioA (void)
{
    int a;
    a = 5;
    cout << “Sóc la funció A i vaig a passar un “ << a << “ a la funció B.\n”;
    funcioB(a);
    cout << “Sóc de nou la funció A i a val ” << a << endl;
}

void funcioB (int & b)
{
    cout << “Sóc la funció B i m'han passat un ” << b << endl;
    b = 2;
    cout << “Sóc de nou la funció B i he canviat el paràmetre a ” << b << endl;
}
```

L'exemple anterior mostrarà els següents missatges per pantalla:

```
Sóc la funció A i vaig a passar un 5 a la funció B.
Sóc la funció B i m'han passat un 5
Sóc de nou la funció B i he canviat el paràmetre a 2
Sóc de nou la funció A i a val 2
```

Com hem passat l'argument a la `funcioB` per referència, aquesta funció pot canviar-lo.

En conclusió, quan cridem a un subprograma i li passem un paràmetre per valor, realment li estem passant una còpia del valor a una variable local del subprograma. Aquesta còpia es destruirà quan acabe el seu àmbit, açò és, quan acabe l'execució del subprograma i, per tant, com que n'és una còpia, la variable que proporcionà originàriament el seu valor no modificarà el seu contingut. En el pas per referència, en canvi, no es passa una còpia de la variable que dóna el valor sinó que es passa la pròpia variable. Així, doncs, quan acaba l'execució del subprograma, la variable que fou passada com a paràmetre real conservarà les modificacions potencialment realitzades pel subprograma.

Exemple de sintaxi per a cadascun d'aquests passos

```
void pasParametres(int param1, int & param2)
{
    param1 = 5;
    param2 = 7;
}
```

El paràmetre `param1` està passat per valor i el paràmetre paràmetre `param2` està passat per referència. Observeu com la diferència està en què aquell que està passat per referència està precedit d'un símbol &.

Si férem la següent crida:

```
int main()
{
    int x, y;
    x = 8;
    y = 9;
    pasParametres(x, y);
    cout << “x = “ << x << endl << “y = ” << y << endl;
    return 0;
}
```

L'eixida per pantalla seria:

```
x = 8
y = 7
```

Com que param1 està passat per valor, la variable que fa de paràmetre real no es modifica. En canvi, la que es passa per referència sí que ho fa.

Funcions de llibreria en C++

De la mateixa manera que les funcions d'entrada-eixida per consola es troben en la llibreria "`iostream`", existeixen altres llibreries en C++ que arrepleguen el seu propi tipus de funcions. La millor manera de conèixer-les és fer una ullada al manual de C++.

Les capçaleres d'algunes llibreries d'utilitat són:

- "`iostream`": Conté els objectes `cin`, `cout` i `cerr` que corresponen respectivament als fluxos d'entrada estàndard, d'eixida estàndard i d'error. Aquesta llibreria proporciona funcions d'entrada i eixida amb i sense format.
- "`stdlib.h`": Conté moltes funcions útils com ara: `system`, `itoa`, `ultoa`, `atoi`,... (aquestes últimes converteixen nombres a cadenes de caràcters i viceversa).
- "`string`": Conté les funcions de tractament de cadenes de caràcters (strings).
- "`ctype.h`": Defineix funcions per a operar sobre els caràcter i determinar de quin tipus són: `isupper` (majúscules), `islower` (minúscules), `isdigit` (nombre)...
- "`math.h`": Conté funcions per a càlculs matemàtics. Algunes de les més útils són:

Funcions aritmètiques:

Prototip	**Descripció**	**Capçalera**
`int abs (int x);`	Valor absolut de x.	stdlib.h
`long labs (long x);`	Valor absolut de x.	stdlib.h
`double fabs (double x);`	Valor absolut de x.	math.h
`double sqrt (double x);`	Arrel quadrada de x.	math.h
`double pow (double x, double y);`	Retorna el primer argument (x) elevat a la potència del segon argument (y).	math.h
`double exp (double x);`	Retorna *e* (base dels logarismes naturals) elevat a la potència del seu argument x.	math.h
`double log (double x);`	Logarisme natural (ln) de x.	math.h
`double log10 (double x);`	Logarisme en base 10 de x.	math.h
`double ceil (double x);`	Retorna l'enter més xicotet, major o igual que x.	math.h
`double floor (double x);`	Retorna l'enter més gran, menor o igual que x.	math.h

Funcions trigonomètriques:

Prototip	**Descripció**	**Capçalera**
`double acos (double x);`	Arc cosinus.	math.h
`double asin (double x);`	Arc sinus.	math.h

`double atan (double x);`	Arc tangent.	math.h
`double cos (double x);`	Cosinus de x en radians.	math.h
`double cosh (double x);`	Cosinus hiperbòlic de x.	math.h
`double sin (double x);`	Sinus de x en radians.	math.h
`double sinh (double x);`	Sinus hiperbòlic de x.	math.h
`double tan (double x);`	Tangent de x.	math.h
`double tanh (double x);`	Tangent hiperbòlica de x.	math.h

Generador de nombres aleatoris:

Prototip	**Descripció**	**Capçalera**
`int random (int);`	La crida a *random(n)* retorna un enter pseudoaleatori major o igual a 0 i menor que n-1. (No està disponible en totes les implementacions. Si no està disponible cal emprar la funció *rand*).	stdlib.h
`int rand ();`	La crida a *rand()* retorna un enter pseudoaleatori major o igual a 0 i menor o igual al valor de la constant RAND_MAX. RAND_MAX és una constant entera predefinida en la llibreria stdlib.h el valor de la qual depén de la implementació però, almenys, és 32767.	stdlib.h
`void srand (unsigned int);`	Reinicialitza el generador de nombres aleatoris. L'argument que rep aquesta funció rep el nom de llavor. La crida a srand diverses vegades amb el mateix argument fa que les funcions *rand* o *random* produeixen la mateixa succesió de nombres pseudoaleatoris. Si *rand* o *random* s'invoquen sense una crida prèvia a srand, la succesió de nombres que es produeix és la mateixa que si s'haguera invocat *srand* amb un argument igual a 1.	stdlib.h

Notes:

- Si volem obenir un nombre aleatori que estiga dins d'un rang determinat ([R1, R2]) podem cridar la funció rand i, en acabant, modificar el valor de retorn com es mostra a continuació:

$$y = \frac{x \cdot (R2 - R1)}{RAND_{MAX}} + R1$$ on x = rand(), x ∈ [0, RAND_MAX], y ∈ [R1, R2]

- Si es desitja iniciar el generador de nombres aleatoris sense necessitat de proporcionar una llavor diferent cada vegada, es pot fer la següent crida: `srand(time(NULL))`, on `time(NULL)` retorna l'hora actual en dècimes de segon.

- Quan es necessite cridar a una funció matemàtica, a més a més d'incloure el fitxer math.h, cal enllaçar la llibreria amb l'opció (-lm) quan es genera l'executable (en Dev C++ açò es fa de forma automàtica).

Comentaris:

1. Com a documentació del programa, per a cada funció caldrà escriure un comentari amb el seu nom, els paràmetres d'entrada i els d'eixida que rep i una breu descripció del que fa. Per exemple:

```
/*************************************************************/
/* Funció: factorial                                         */
/*                                                           */
/* Descripció: Calcula el factorial d'un nombre              */
/*                                                           */
/* Paràmetres                                                */
/* Nom      I/O      Descripció                              */
/* ---      ---      ----------                              */
/*  n        I                                               */
/*                                                           */
/* Valor de retorn:                                          */
/* double  És un tipus real per a evitar l'overflow          */
/*************************************************************/

double factorial(int n)
{
    ...
}
```

   La descripció dels paràmetres només caldrà quan no estiga clar el seu significat. En una funció poden existir diverses eixides. L'elecció de quina d'aquestes eixides correspondrà amb el resultat de la funció es farà quan s'implemente la funció i no en la documentació del programa.

2. Les funcions ha de ser el més independents possible de la resta del programa, ja que només es podran comunicar amb la resta del programa mitjançant paràmetres. **Recordeu que NO es poden fer servir variables globals.**