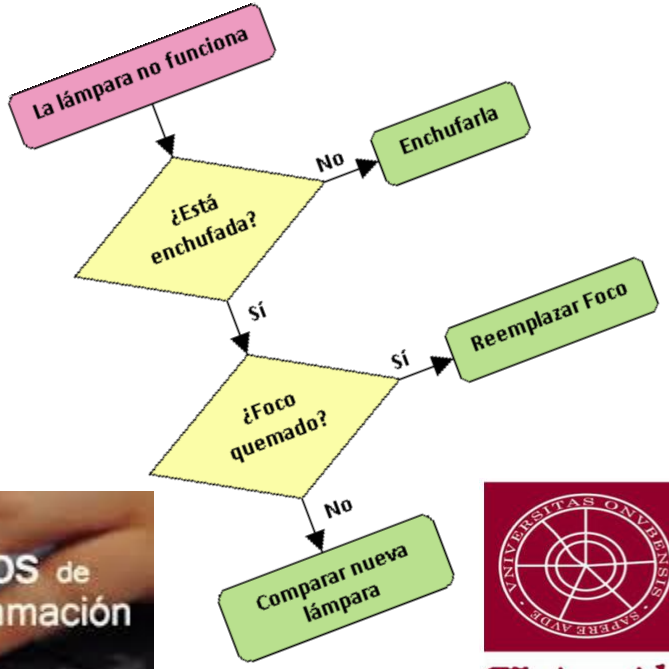
Tema 5.

Diseño Descendente.

Fundamentos de Programación
Grado en Ingeniería Informática

Profesores:
José Manuel Martín Ramos
Francisco Roche Beltrán.



ÍNDICE

1. Programación Estructurada.

- Un programa está compuesto de 1 a n ***funciones***.
- ***Función (subprograma):*** conjunto de acciones agrupadas bajo un nombre común.
- Las funciones:
  - Se escriben ***una sola vez*** y puede ser usadas ***varias veces en distintas partes*** del programa.
  - Hacen que un programa sea ***más corto***, ***fácil de entender y corregir*** y ***disminuye los errores***.
- Ventajas de uso de funciones:
  - **Modularización.**
  - **Ahorro** de memoria y **tiempo** de desarrollo.
  - **Independencia** de los datos y **ocultamiento** de la información.

Un programa está compuesto al menos de una función (main), pudiendo tener tantas funciones o clases como se crean convenientes.

```
#include <iostream>
using namespace std;
int main () {
    int b, e;
    double valor;
    cout << "pon el valor de la base: ";
    cin >> b;
    cout << "Pon el valor del exponente: ";
    cin >> e;
    //No se tiene en cuenta si es 0 elevado a cero.
    valor = 1;
    for (int i=0; i < e; i++)
        valor = valor * b;
    cout << b << " elevado a " << e << " es "
         << valor <<"\n";
    return 0;
}
```

```
#include <iostream>
using namespace std;
double potencia (){
    double  resultado;
    int base, exponente;
    cout << "Pon el valor de la base: ";
    cin >> base;
    cout << "Pon el valor del exponente: ";
    cin >> exponente;
    resultado = 1;
    for (int i=0; i < exponente; i++)
        resultado = resultado * base;
    return resultado;
}
int main () {
    double valor;
    cout << "Vamos a calcular el valor de una base "
         << "elevada a un exponente: \n";
    valor = potencia ();
    cout <<  "El resultado es " << valor <<"\n";
    //La utilizamos otra vez
    valor = potencia ();
    cout <<  "El resultado es " << valor <<"\n";
    return 0;
}
```

Las funciones se definen una vez y se pueden utilizar las veces que se desee, haciendo el programa más pequeño en tamaño y más fácil de entender y corregir.

```
#include<iostream>
#include <string.h>
using namespace std;
#define M 3
typedef char cadena[30];
class tpalabras {
    cadena tabla[M];
public:
 void leer();
 void buscar();
};
void tpalabras::leer() {
 //Esquema de recorrido
 for(int i = 0; i < M; i++){
   cout << "\nIntroduce la palabra " << i <<"\n";
   cin >> tabla[i];
   }
}
```

```
void tpalabras::buscar(){
 cadena leida;
 bool encontrado;
 int i;
 cout << "Introduce la palabra a buscar\n";
 cin >> leida;
 //Esquema de búsqueda
 i = 0;
 encontrado = false;
 while ( (i < M)  &&  (!encontrado) ) {
   if (strcmp(tabla[i], leida) == 0)
                encontrado = true;
   else i++;
   }
 if (encontrado) cout << "Se encontro en"
                   <<" la posicion " << i;
 else       cout << "No se encontro";
}
int main() {
 tpalabras palabras;
 tpalabras palabrasb[2];
 palabras.leer();
 palabras.buscar();
 for (int i=0; i < 2; i++)
     palabrasb[i].leer();
 //...
 return 0;
}
```

El método leer de la clase tpalabras se utiliza en varias ocasiones.

```
#include <iostream>
using namespace std;
double potencia ();
int main () {
    double valor;
    cout << "Vamos a calcular el valor de una base "
         << "elevada a un exponente: \n";
    valor = potencia ();
    cout <<  "El resultado es " << valor <<"\n";
    return 0;
}
double potencia (){
    double  resultado;
    int base, exponente;
    cout << "Pon el valor de la base: ";
    cin >> base;
    cout << "Pon el valor del exponente: ";
    cin >> exponente;
    resultado = 1;
    for (int i=0; i < exponente; i++)
        resultado = resultado * base;
    return resultado;
}
```

//Declaración de la función.

//Llamada a la función.

//Declaración de la función.

2. Variables Globales y Locales.

Variable local

- ***Es declarada dentro de un bloque o función***.
- Sólo puede ser usada dentro del bloque en el que está. No puede usarse fuera.
- Sólo existe durante la ejecución del bloque en el que está: se crea al entrar en el bloque y se destruye al salir.

Variable global

- ***Es declarada fuera de cualquier bloque o función***, al inicio del programa.
- Puede ser usada en cualquier parte del programa, incluyendo cualquier función (no hay que pasarla como parámetro).
- Existe durante toda la ejecución del programa.

Una variable local puede tener el mismo nombre que una global.

```
#include <iostream>
using namespace std;
class tfactorial{
  int n;
public:
  void informacion();
  long factorial();
};
void tfactorial::informacion(){
  do {
    cout << "Indique el valor de n...\n";
    cin >>  n;
   }
  while (n <0 || n >= 20);
}
long tfactorial::factorial(){
  long fact=1;
  for (int i = 2; i <= n; i++)
      fact = fact * i;
  return fact;
}
long factor;
tfactorial dato;
int  main(){
  dato.informacion();
  factor= dato.factorial();
  cout <<"\nEl factorial es ... " << factor <<"\n";
  return 0;
}
```

Una **variable global** es una variable que se declara fuera de cualquier función o bloque.

factor y dato son dos variables globales.

Se recomienda NO utilizar variables globales, ya que es difícil la corrección de errores al estar visibles desde cualquier función.

```
#include <iostream>
using namespace std;
class tfactorial{
  int n;
public:
  void informacion();
  long factorial();
};
void tfactorial::informacion(){
  do {
    cout << "Indique el valor de n...\n";
    cin >>  n;
   }
  while (n <0 || n >= 20);
}
long tfactorial::factorial(){
  long fact=1;
  for (int i = 2; i <= n; i++)
      fact = fact * i;
  return fact;
}
int  main(){
  long factor;
  tfactorial dato;
  dato.informacion();
  factor= dato.factorial();
  cout <<"\nEl factorial es ... " << factor <<"\n";
  return 0;
}
```

Una variable local es una variable que se declara dentro de un bloque, función o método.
Sólo existe durante la ejecución de la función o bloque que la declara.

long es una variable local en el método factorial

factor y dato son dos variables locales en la función main.

3. Variables Miembro Privadas y Públicas.

Variable miembro de clase

- *Es declarada dentro de una clase*

Variable miembro privada

- ***Es declarada en la parte private: de la clase***.
- No es visible fuera de la clase.
- Solo pueden ser utilizadas en los métodos de la clase.
- Al ser privadas un mal uso de las mismas es totalmente imposible.

Variable miembro pública

- ***Es declarada en la parte public: de la clase***.
- Es visible fuera de la clase.
- La puede usar los métodos de la clase y cualquier código que haga uso de los objetos declarados de dicha clase.
- No es muy aconsejable su uso.

```
#include <iostream>
using namespace std;
class tfactorial{
  int n;
public:
  void informacion();
  long factorial();
};
void tfactorial::informacion(){
  do {
    cout << "Indique el valor de n...\n";
    cin >>  n;
   }
  while (n <0 || n >= 20);
}
long tfactorial::factorial(){
  long fact=1;
  for (int i = 2; i <= n; i++)
      fact = fact * i;
  return fact;
}
int  main(){
  long factor;
  tfactorial dato;
  dato.informacion();
  factor= dato.factorial();
  cout <<"\nEl factorial es ... " << factor <<"\n";
  return 0;
}
```

Un atributo (variable declarada dentro de una clase) será público si se declara en la parte public de una clase, en caso contrario será privado.
Los atributos también reciben el nombre de variables miembro.

No se permitirá el uso de **atributos públicos.**

n es un **atributo privado**.

Los atributos privados sólo están visibles para los métodos de la propia clase.
Los métodos información y factorial utilizan el atributo n.

4. Parámetros por Valor y por Referencia.

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

int menu(){
    int opc;
    for (int i=0; i < 8; i++)  cout << "\n";
    cout << "\t\t\t   **********   MENU ***********\n";
    cout << "\t\t\t           1.- Leer.\n";
    cout << "\t\t\t           2.- Escribir.\n";
    cout << "\t\t\t           3.- Salir.\n";
    cout << "\t\t\t       Pon la opcion que deseas:";
    cin >> opc;
    return opc;
}

int main () {
    int opcion;
    opcion = menu();
    switch (opcion) {
        case 1: //......
                break;
        case 2:  //.....
                break;
        case 3:  //.......
                break;
        default: //....
                cout << "opcion incorrecta.";
        }
    return 0;
}
```

La única vía de comunicación que conocemos hasta el momento entre la llamada a la función y la propia función es a través de la instrucción return, que devuelve un valor al que lo llama.

Parámetros en Funciones y Métodos

```
void funcion(int x, int y, char c) // bien
void funcion(int x, y, char c)     // mal
```

- **Sintaxis Funciones:**

```
tipo nombre([tipo v1, ..., tipo vN]) {
    [declaraciones;]
    sentencias;
    [return valor_devuelto;]
}
```

- **Sintaxis Métodos:**

```
tipo nombre_clase::nombre_metodo([tipo v1, ..., tipo vN]) {
    [declaraciones;]
    sentencias;
    [return valor_devuelto;]
}
```

- Los ***parámetros*** son los valores que se le pasan a la función/método al ser llamada. Cada parámetro se comporta dentro de la función/método como una variable local.

- Los parámetros escritos en la sentencia de llamada a la función/método se llaman ***parámetros reales*** o ***argumentos***.

- Los que aparecen en la descripción de la función/método se llaman ***parámetros formales***.

- Los parámetros formales son sustituidos por los parámetros reales (argumentos) en el orden de llamada. Al emparejarse éstos deben coincidir en número y en tipo.

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

double potencia ( int base, int exponente){
    double  resultado;
    resultado = 1;
    for (int i=0; i < exponente; i++)
        resultado = resultado * base;
    return resultado;
}

int main () {
    int b, e;
    double valor;
    cout << "pon el valor de la base: ";
    cin >> b;
    cout << "Pon el valor del exponente: ";
    cin >> e;
    //No se tiene en cuenta si es 0 elevado a cero.
    valor = potencia (b, e);
    cout << b << " elevado a " << e << " es " << valor <<"\n";
    valor = potencia (3, 4);
    cout << 3 << " elevado a " << 4 << " es " << valor <<"\n";
    valor = potencia (2*b, e-1);
    cout << 2*b << " elevado a " << e-1 << " es " << valor <<"\n";
    return 0;
}
```

Los **parámetros** son los valores que se le pasan a la función o método al ser llamado.
Parámetros formales: base y exponente.

Cada parámetro se comporta dentro de la función o método como una variable local, existe sólo durante la ejecución de esa función.

Parámetros reales: b, e, 3, 4, 2*b y e-1.
Al emparejarse con los parámetros formales deben de coincidir en número y en tipo.

Parámetros por valor

- ***Los cambios producidos dentro de la función/método no afectan a la variable real usada como argumento***.
- En la llamada se pasa una copia del valor del argumento.
- El parámetro real puede ser una constante, variable o expresión del tipo indicado en el parámetro formal.

Parámetros por referencia

- ***Los cambios producidos dentro de la función/método afectan a la variable con la que se realiza la llamada***.
- En la llamada se transfiere la propia variable.
- El parámetro real sólo puede ser una variable, ya que se trasfiere ésta.
- Hay que anteponer el carácter & en el prototipo y en la definición de la función.

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

double potencia ( int base, int exponente){
     double  resultado;
     resultado = 1;
     for (int i=0; i < exponente; i++)
          resultado = resultado * base;
     return resultado;
}

int main () {
     int b, e;
     double valor;
     cout << "pon el valor de la base: ";
     cin >> b;
     cout << "Pon el valor del exponente: ";
     cin >> e;
     //No se tiene en cuenta si es 0 elevado a cero.
     valor = potencia (b, e);
     cout << b << " elevado a " << e << " es " << valor <<"\n";
     valor = potencia (3, 4);
     cout << 3 << " elevado a " << 4 << " es " << valor <<"\n";
     valor = potencia (2*b, e-1);
     cout << 2*b << " elevado a " << e-1 << " es " << valor <<"\n";
     return 0;
}
```

Los parámetros pueden ser por valor o por referencia.
Parámetros por valor: base y exponente.
Los cambios producidos dentro de la función o método **no afectan** al parámetro real.

El valor inicial de un parámetro formal es la **copia** del valor del parámetro formal.

Parámetros reales: b, e, 3, 4, 2*b y e-1.
Los parámetros reales pueden ser: una constante (3), una variable (b) o una expresión (2*b)

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

void cambiar ( int &a, int& b){
    int c;
    if ( a > b )    {
        c = a;
        a = b;
        b = c;
        }
}

int main () {
    int c, d;
    cout << "Pon el valor del primer numero: ";
    cin >> c;
    cout << "Pon el valor del segundo numero: ";
    cin >> d;
    cambiar (c, d);
    cout << " El valor del primer numero ahora es
         << c <<"\n";
    cout << " El valor del segundo numero ahora es
         << d <<"\n";
    return 0;
}
```

Parámetro por referencia: se pone un & entre el tipo de datos y el nombre del parámetro.
En este ejemplo a y b son parámetros formales por referencia.

El valor inicial de un parámetro por referencia es el propio parámetro formal, ¡No hay copia!
Por ello, los cambios producidos dentro de la función o método afectan al parámetro real.

El parámetro real sólo puede ser una variable (c).

```
#include <iostream>
using namespace std;
void permutar(int &a, int &b);
void cambiamal(int a, int b);
int main() {
    int i = 1, j = 2, x = 3, y = 4;
    cout << "\ni = " << i << ", j = " << j;
    cout << "\nx = " << x << ", y = " << y;
    permutar(i, j);
    cambiamal(x, y);
    cout << "\ni = " << i << ", j = " << j;
    cout << "\nx = " << x << ", y = " << y;
    return 0;
}
void permutar(int &a, int &b) {
    int temp;
    temp = a;
    a = b;
    b = temp;
}
void cambiamal(int a, int b) {
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Salida del programa por pantalla:

i = 1, j = 2
x = 3, y = 4

i = 2, j = 1
x = 3, y = 4

5. Constructores y Destructores.

Constructores

- ***Es un método que tiene el mismo nombre que la clase en la que está definido y no tiene ningún tipo devuelto (ni void).***
- Es llamado automáticamente cada vez que se crea un objeto de la clase.
- Puede tener parámetros y puede haber más de uno.
- Se usa para inicializar los atributos de la clase.

Destructores

- ***Es un método que tiene el mismo nombre que la clase en la que está definido pero su nombre está precedido por el carácter ~***
- Al igual que el constructor, no tienen nigún tipo devuelto.
- Es llamado automáticamente cuando el objeto va a dejar de existir.
- Es único y no tiene parámetros.
- Es llamado automáticamente antes de la destrucción del objeto.

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

class Punto {
    int X,Y,Z;
  public:
    Punto();
    Punto(int px);
    Punto(int px, int py);
    Punto(int px, int py, int pz);
    void Sumar(int px, int py, int pz);
    void Devolver(int &px, int &py, int &pz);
};

Punto::Punto() {
    X=Y=Z=0;
}

Punto::Punto(int px) {
    X=px;
    Y=Z=0;
}

Punto::Punto(int px, int py) {
    X=px;
    Y=py;
    Z=0;
}

Punto::Punto(int px, int py, int pz) {
    X=px;
    Y=py;
    Z=pz;
}
```

```cpp
void Punto::Sumar(int px, int py, int pz) {
    X+=px;
    Y=Y+py;
    Z=Z+pz;
}

void Punto::Devolver(int &px, int &py, int &pz) {
    px=X;
    py=Y;
    pz=Z;
}

int main() {
    Punto P1;
    Punto P2(1);
    Punto P3(2,5);
    Punto P4(3,4,5);

    int x,y,z;
    cout<< "\nIntroduce X,Y,Z de P5: ";
    cin>>x>>y>>z;

    Punto P5(x,y,z);
    P2.Devolver(x,y,z);
    P1.Sumar(x,y,z);
    P3.Devolver(x,y,z);
    P1.Sumar(x,y,z);
    P4.Devolver(x,y,z);
    P1.Sumar(x,y,z);
    P5.Devolver(x,y,z);
    P1.Sumar(x,y,z);
    P1.Devolver(x,y,z);
    cout<<"\nP1=P2+P3+P4+P5\n";
    cout<<"X="<<x<<" Y="<<y<<" Z="<<z<<endl;
    return 0;
}
```

Salida

```
Introduce X,Y,Z de P5:
1 2 3

P1=P2+P3+P4+P5
X=7 Y=11 Z=8
```

6. Paso de Estructuras a Funciones y Métodos

- Por defecto, todas las estructuras de datos que ***no sean vectores*** se pasan a las funciones/métodos ***por valor a no ser que se especifique el símbolo &*** en la definición del parámetro formal.
- ***Los vectores siempre*** se pasan a las funciones/métodos ***por referencia*** y por lo tanto ***no hay que especificar el símbolo &*** en la definición del parámetro formal.

```
class obj {
    int n;
public:
    obj(int i) { n = i; }
    void set(int i) { n = i; }
    int get() { return n; }
};

void cambia(obj a, int valor){
    a.set(valor);
    cout << "\na vale "<<a.get();
}

int main() {
    obj a(10); // a.n vale 10
    cambia(a,7);  // por valor
    cout <<"\na.n vale "<<a.get();
    system("Pause");
    return 0;
}
```

a vale 7
a.n vale 10

Si pasamos el objeto *por referencia* se modifica el original:

```
void cambia(obj &a, int valor){
    a.set(valor);
    cout << "\na vale " << a.get();
}
```

a vale 7
a.n vale 7

- Para pasar un ***vector*** a una función/método en la **llamada** (parámetro real) se pone el ***nombre del vector sin corchetes***.
- Para pasar ***un elemento*** de un ***vector*** éste se pasa ***por valor*** o ***por referencia*** dependiendo de la definición del parámetro formal.

x[0]	x[1]	x[2]	x[3]	x[4]
0	0	0	0	0
6	0	0	0	0
6	4	0	0	0
8	6	2	2	2
8	6	2	2	5

```
class obj {
    int n;
public:
    obj() {n = 0;}
    void set(int i) { n = i; }
    int get() { return n; }
};
void suma_uno(obj &a, int n) {
    a.set(a.get() + n);
}
void suma(obj a[5], int n) {
    for(int i=0; i < 5; i++)
        a[i].set(a[i].get() + n);
}
int main() {
    obj x[5]; //x[0], ... x[4]
    x[0].set(6);
    x[1].set(4);
    suma(x,2);
    suma_uno(x[4],3);
    return 0;
}
```

```
struct direccion {
    char calle[30];
    int num;
};
class datos_per {
    char nombre[35];
    direccion dir;
public:
    void setdir(direccion d) { dir = d; }
    void ver_datos();
    void leer_datos();
};
void datos_per::ver_datos() {
    cout << "\nNombre: " << nombre;
    cout << "\nCalle: " << dir.calle;
    cout << "\nNumero: " << dir.num;
}
void datos_per::leer_datos() {
    cout << "Nombre: "; cin >> nombre;
    cout << "Calle: "; cin >> dir.calle;
    cout << "Numero: "; cin >> dir.num;
}
```

```
void cargar_datos(datos_per p[5]) {
    for (int n = 0; n < 5; n++)
        p[n].leer_datos();
}
void lee_num(int& num) {
    cout << "Numero: ";
    cin >> num;
}
int main() {
    int n; char tecla;
    datos_per personas[5]; direccion d;
    cargar_datos(personas);
    do {
        do {
            cout << "num persona a ver: ";
            cin >> n;
        } while (n < 0 || n > 4);
        personas[n].ver_datos();
        lee_num(d.num);
        personas[n].setdir(d);
        cout << "\nVer otra persona (s/n)?";
        tecla = getch();
    } while (tecla == 's');
    return 0;
}
```

7. Sobrecarga de Funciones y Métodos.

- C++ permite definir varias funciones/métodos distintos con un mismo nombre, ***siempre que*** difieran en el número y/o el tipo de sus argumentos.
- Los argumentos de la llamada indicará la función/método a usar.

Si las funciones/métodos sólo difieren en el tipo de datos que devuelven ***no se pueden*** sobrecargar:

```
int funcion(int x);
float funcion(int x);
int main() {
  . . .          // ambigüedad
  funcion(2); // ¿a que función llama?
}
```

Si sólo difieren en el tipo de sus argumentos, puede ser ambiguo, aunque la función/método no sea ambigua:

```
float dividir(float x);
double dividir(double x);
int main() {
  . . .          // ambigüedad
  dividir(15); // 15 es double o float?
}
```

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

int vabs(int n) {
    return n < 0 ? -n : n;
}
long vabs(long n) {
    if (n < 0)
        return -n;
    return n;
}
double vabs(double n) {
    if (n < 0)
        n = -n;
    return n;
}

int main() {
    long b=-50000;
    cout << "\n|-6|: " << vabs(-6);
    cout << "\n|-50000|: " << vabs(b);
    cout << "\n|-6.7|: " << vabs(-6.7);
    return 0;
}
```

Salida:

|-6|: 6
|-50000|: 50000
|-6.7|: 6.7

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

void copiar(char C1[20], const char C2[20])
{
    strcpy(C1, C2);
}

void copiar(char C1[200], const char C2[20],
            int n)
{
    strncpy(C1, C2, n);
}

int main() {
  char cad_a[20], cad_b[200];
  copiar(cad_a, "Buen Dia\n");
  copiar(cad_b, "*", 8);
  cad_b[8] = '\0';
  cout << cad_a << cad_b<<endl;
  return 0;
}
```

Salida:

```
Buen Dia
********
```

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

class Punto {
    int X,Y,Z;
  public:
    Punto();
    Punto(int px);
    Punto(int px, int py);
    Punto(int px, int py, int pz);
    void Sumar(int px, int py, int pz);
    void Devolver(int &px, int &py, int &pz);
    void Mostrar(char texto[50]);
    Punto Sumar(Punto p);
};

Punto::Punto() {
    X=Y=Z=0;
}

Punto::Punto(int px) {
    X=px;
    Y=Z=0;
}

Punto::Punto(int px, int py) {
    X=px;
    Y=py;
    Z=0;
}

Punto::Punto(int px, int py, int pz) {
    X=px;
    Y=py;
    Z=pz;
}

void Punto::Sumar(int px, int py, int pz) {
    X+=px;
    Y=Y+py;
    Z=Z+pz;
}

void Punto::Devolver(int &px, int &py, int &pz) {
    px=X;
    py=Y;
    pz=Z;
}

void Punto::Mostrar(char texto[50]) {
    cout<<texto<<" X="<<X<<" Y="<<Y<<" Z="<<Z<<endl;
}

Punto Punto::Sumar(Punto p) {
    p.Sumar(X,Y,Z);
    return p;
}
```

```
int main()
{
    Punto P1;
    Punto P2(2,5);
    Punto P3(3,4,5);

    Punto Datos[100],SumaTotal;
    int Npuntos;
    int x,y,z;

    cout <<"P1 P2 P3\n";
    P1.Mostrar("P1:");
    P2.Mostrar("P2:");
    P3.Mostrar("P3:");

    cout <<"\nP1 = P2 + P3\n";
    P1=P2.Sumar(P3);
    P1.Mostrar("P1:");
```

```
    cout<<"Introduce el número de Puntos: ";
    cin>>Npuntos;
    for (int i=0; i<Npuntos; i++)
    {
        cout<< "Introduce X,Y,Z de P"<<i+1<<": ";
        cin>>x>>y>>z;
        Datos[i].Sumar(x,y,z);
    }

    for (int i=0; i<Npuntos; i++)
        SumaTotal=SumaTotal.Sumar(Datos[i]);

    SumaTotal.Mostrar("SumaTotal:");
    return 0;
}
```

Salida

```
P1=P2+P3+P4+P5
X=7 Y=11 Z=8

P1 P2 P3
P1: X=0 Y=0 Z=0
P2: X=2 Y=5 Z=0
P3: X=3 Y=4 Z=5

P1 = P2 + P3
P1: X=5 Y=9 Z=5
```

Salida

```
Introduce el número de
Puntos: 5
Introduce X,Y,Z de P1: 1 2 3
Introduce X,Y,Z de P2: 2 3 4
Introduce X,Y,Z de P3: 3 4 5
Introduce X,Y,Z de P4: 4 5 6
Introduce X,Y,Z de P5: 5 6 7
SumaTotal: X=15 Y=20 Z=25
```