Metodología de la Programación

MODULARIZACIÓN

1. MODULARIZACIÓN DE FUNCIONES

Francisco J. Cortijo Bon
cb@decsai.ugr.es

Curso 2021-2022

Resumen

La repetición de código entre diferentes proyectos (*copiar y pegar*) es una pésima manera de programar. Nunca conseguiremos versiones estables y coherentes de nuestras funciones y clases entre todos nuestros proyectos.

En este documento vamos a detallar la manera en la que se modularizará a nivel de **funciones** obteniendo un módulo con el **código objeto** de las funciones y el **fichero de cabecera** asociado. De esta manera podrá utilizarse en diferentes proyectos enlazando el código objeto obtenido.

En próximas versiones de este problema veremos cómo este código objeto podría incorporarse a **bibliotecas** y cómo se modularizarán **clases**. También aprenderemos a utilizar el programa `make` y a escribir ficheros `makefile` para la gestión automatizada de los proyectos que generemos.

Índice

1. Introducción

En el contexto de esta asignatura, un proyecto software complejo se compone de diferentes módulos de código que se *combinan* para generar ejecutables.

Los módulos que intervienen en el proyecto deberían tener, básicamente, dos propiedades:

1. Deberían ser **independientes**, en el sentido de que no se debería necesitar de otros módulos adicionales. De esta manera, su uso no requeriría la incorporación de un código demasiado extenso.
2. Deberían tener un alto grado de **cohesión**, o sea, ser *concretos*, en el sentido de que deberían tener una tarea única y bien definida.

El cumplimiento de estas dos propiedades:

1. favorece la **reutilización**, ya que el módulo estará listo para su uso allá donde sea necesario.
2. minimiza el **mantenimiento**, ya que los cambios se realizarán en un solo lugar y se actualizará todo el software para reflejar los cambios introducidos.

Un ejemplo fácil de entender y conocido es la generación de números aleatorios. En muchos y diferentes problemas surge la necesidad de trabajar con números aleatorios. No tiene sentido escribir el mismo código una y otra vez (entiéndase *copiar y pegar*) en todos y cada uno de los proyectos en los que se necesiten.

- Deberíamos tener una sola copia compilada del código y poder incorporarla a los proyectos cuando sea necesario, sin tener que *copiar y pegar* el código fuente, y después compilarlo junto al resto del proyecto.
- Como ventaja adicional, cualquier modificación del código que gestiona los números aleatorios se haría en un sólo sitio y los cambios se “propagarían” a todos los proyectos involucrados.

Otro ejemplo sería el de la clase `string`: se trata de una clase de la biblioteca estándar de C++ que usamos una y otra vez sin tener que hacer nada más (al menos aparentemente) que escribir la línea `#include<string>`.

En este documento vamos a detallar cómo se modularizará a nivel de **funciones**[1] obteniendo código objeto (que podría finalmente organizarse en forma de **bibliotecas**) con el objetivo de poder utilizarse en diferentes proyectos.

En este proceso cada módulo tendrá asociada dos ficheros de código fuente:

1. Un **fichero de cabecera** (extensión `.h`) con la **declaración** de las funciones.
2. Un **fichero de código fuente** (extensión `.cpp`) con la **definición** de las funciones.

2. Declaración/definición de funciones

La **declaración** de una función le sirve al compilador para conocer

1. su *nombre*,
2. su *tipo* (tipo del valor devuelto, o en su caso `void`), y
3. para cada argumento, su *tipo*.

Con esta información el compilador debería ser capaz de verificar que la llamada a la función es sintáctica y semánticamente correcta y saber qué función es la que debe ejecutarse cuando se realiza una llamada.

La sintaxis de una declaración de función es sencilla: se escribe la **cabecera** de la función seguida de un punto y coma. En la cabecera está toda la información que requiere el compilador: tipo y nombre de la función, y para cada argumento

[1] La modularización de clases la posponemos para una nueva versión del problema que nos sirve de ejemplo.

formal, su tipo. Suele emplearse también el término **prototipo**. Por ejemplo:

```
int MaximoValorPosible (int num);
```

es el prototipo de una función que devuelve un valor `int`, el número más grande que se puede formar con los dígitos de `num`.

En este contexto **no** es necesario escribir el nombre de los parámetros formales en la declaración (de hecho el compilador los ignora). Por ejemplo, este prototipo es idéntico al anterior:

```
int MaximoValorPosible (int);
```

En la **definición** de una función se especifica:

- La **cabecera** de la función. Incluye los elementos de la declaración aunque ahora **sí** hay que especificar, obligatoriamente, el *nombre* de los parámetros.
  Importante: *Si se hubiera dado nombre a los parámetros en la declaración, no tienen porqué coincidir con los que aparecen en la definición*.
- Las *instrucciones* que va a ejecutar la función (el **cuerpo** de la función). Las instrucciones están delimitadas por los caracteres `{` y `}` y se escriben tras la cabecera.

```
int MaximoValorPosible (int num)
{
   int digitos[10] = {0};

   while (num>0) {
      digitos[num%10]++;
      num = num/10;
   }

   int resultado = 0;

   for (int i=9; i>=0; i--)
      while (digitos[i]>0) {
         resultado=resultado*10 + i;
         digitos[i]--;
     }

   return resultado;
}
```

Los parámetros especificados en la cabecera de la función se conocen como **parámetros formales** (también llamados argumentos formales). Pueden usarse únicamente dentro de la función, como cualquier *variable local* de la función que se está definiendo. Dicho de otra manera, las instrucciones de una función podrá utilizar: 1) los datos locales declarados en ella y 2) los parámetros formales especificados en su cabecera. En el ejemplo anterior, `num` es el único parámetro formal, `digitos` y `resultado` son variables locales de la función e `i` es una variable local al ciclo `for`.

Los parámetros formales son un tipo especial de datos locales porque se *inicializan* con valores que se especifican en la llamada a la función, *antes* de que se ejecute el cuerpo de la función. Por ejemplo, en esta llamada:

```
int mayor = MaximoValorPosible (253);
```

cuando se empieza a ejecutar la función `MayorValorPosible` el parámetro formal `num` se inicializa con el valor `253`. Al finalizar su ejecución, la variable `mayor` toma el valor `532` (valor devuelto por la función).

Las funciones se comunican entre sí a través de:

1. los **argumentos**: la función *llamadora* -la que invoca la ejecución de la función *llamada*- proporciona valores a los parámetros formales de la función llamada, y
2. del **valor devuelto**: la función *llamada* proporciona un valor a la función *llamadora* (excepto si se trata de una función `void`).

Los **parámetros reales** (también llamados argumentos reales) son las *expresiones* que se emplean en la llamada a una función para inicializar los parámetros formales. Por ejemplo, en las llamadas:

```
int mayor = MaximoValorPosible (253);
cout << MaximoValorPosible (n*5);
```

`253` (literal) y la expresión `n*5` son los parámetros reales cuyos valores se emplean para inicializar el valor del parámetro formal `num` en cada llamada.

El número y tipo de los parámetros reales debe coincidir con el de los parámetros formales, y en el mismo orden. Se establece una correspondencia uno a uno entre ellos.

3. Ámbito de una función

El ámbito de una función incluye todas las funciones que están definidas por debajo de su declaración, hasta el final del fichero en el que está declarada.

Todas la funciones que son llamadas en un fichero han debido ser definidas antes de ser llamadas, o han sido declaradas previamente.

Para cumplir este requisito hay dos posibilidades:

1. **Definir la función antes de usarse**:

```
// Definición de la función
int MaximoValorPosible (int num)
{
   int digitos[10] = {0};
   ......
}
...
int main (void)
{
   ...
   cout << MaximoValorPosible (253);
   ...
}
```

   Esta es la manera de trabajar que hemos seguido hasta ahora (asignatura *Fundamentos de Programación*).

   - Obliga a ser muy cuidadoso con el orden en el que se definen las funciones.
   - Fuerza a que `main()` sea la última función.

2. **Declarar la función antes de usarse**:

```
// Declaración de la función
int MaximoValorPosible (int num);
...
int main (void)
{
   ...
   cout << MaximoValorPosible (253);
   ...
}
...
// Definición de la función
int MaximoValorPosible (int num)
{
   int digitos[10] = {0};
   ......
}
```

Con esta manera de trabajar, una vez declaradas las funciones **el orden en que se definen no es importante**. Además, es posible escribir la función `main()` en primer lugar, lo que facilita la lectura del código.

4. Modularización en ficheros

El hecho de poder escribir las declaraciones (prototipos) antes de la función `main` posibilita:

- Agrupar las **declaraciones** en un fichero de cabecera (con extensión `.h`) y sustituir todas las declaraciones por una línea `#include`, especificando el fichero de cabecera donde se encuentran las declaraciones.
- Agrupar las **definiciones** en un fichero independiente (con extensión `.cpp`) -añadiendo en él la correspondiente línea `#include`- y compilarlo independientemente para generar un módulo objeto (con extensión `.o`).

Una vez tengamos el fichero objeto con el código compilado de las funciones podremos enlazarlo con otros módulos objeto que usen/llamen a esas funciones *sin tener que repetir el código*.

Para que el compilador pueda generar código objeto deberemos *incluir* el fichero de cabecera en el que aparecen los prototipos de las funciones llamadas. Así podrá comprobarse la sintaxis y semántica de las llamadas a las funciones y sus valores de retorno.

5. Ejemplo: un proyecto completo (Kaprekar)

El proyecto presentado consiste en la creación de un ejecutable un ejecutable que soluciona el problema del **Proceso de Kaprekar**.

La descripción de este problema y la solución -no modularizada- está disponible en PRADO en el fichero `Kaprekar.cpp`. Es conveniente leerla para familiarizarse con la solución que vamos a desarrollar en esta sección.

Solución sin modularizar

Para crear el ejecutable a partir de `Kaprekar.cpp` bastaría con compilar y enlazar en un solo paso:

```
g++ -o bin/Kaprekar src/Kaprekar.cpp
```

ó en dos pasos:

```
g++ -c -o obj/Kaprekar.o src/Kaprekar.cpp
g++ -o bin/Kaprekar obj/Kaprekar.o
```

Solución modularizada

La estructura del proyecto (diagrama de dependencias) que nos ocupa ahora, resultado de la modularización será el que se indica en la figura 1.

- `main_Kaprekar.cpp` contendrá la función `main` únicamente.
- `funciones_Kaprekar.h` contendrá las **declaraciones** de las funciones.
- `funciones_Kaprekar.cpp` contendrá las **definiciones** de las funciones.

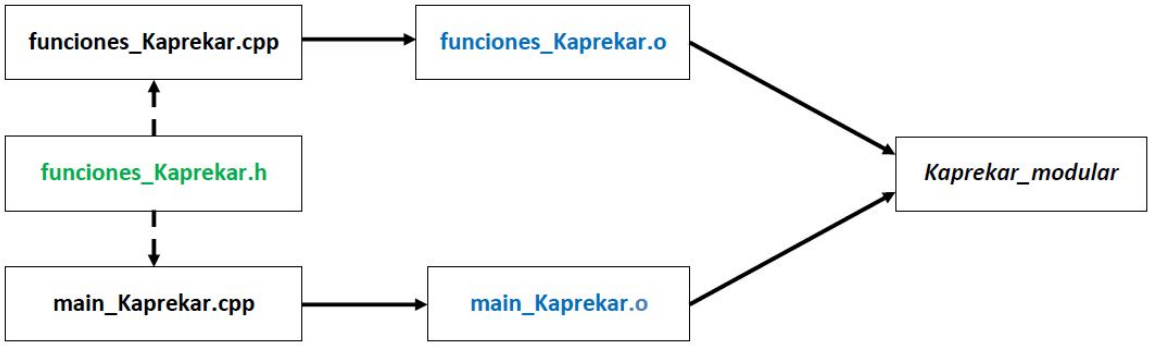

Figura 1: Diagrama de dependencias del proyecto completo

El código de `main_Kaprekar.cpp` (contiene únicamente la función `main`) será:

`main_Kaprekar.cpp`

```
//*************************************************************/
// METODOLOGIA DE LA PROGRAMACIÓN
//
// (C) FRANCISCO JOSÉ CORTIJO BON
// DEPTO. DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL
/*************************************************************/

#include <iostream>
#include <iomanip>

#include "funciones_Kaprekar.h"

using namespace std;

/*************************************************************/
/*************************************************************/

int main()
{
   cout.setf (ios::fixed);
   cout.setf (ios::showpoint);

   int num;

   // Lectura adelantada del número que se analiza

   cout << endl;
```

```
   cout << "Escriba un numero de cuatro o menos cifras"
        << " (negativo o cero para finalizar) = ";
   cin >> num;

   while (num > 0) {

      cout << endl;

      if (NumCifras(num)>4)

         cout << "Numero no valido (mas de cuatro cifras)." << endl;

      else {

         if (VerificaKaprekar(num)) {

            int iterac = IteracionesKaprekar(num);
            cout << "Se verifica la propiedad de Kaprekar." << endl;
            cout << "Iteraciones requeridas = ";
            cout << setw(2) << iterac << endl;
         }
         else  {

            cout << "No se verifica la propiedad de Kaprekar.";
            cout << endl;

            if (TodasCifrasIguales(num))
               cout << "Todas las cifras con iguales" << endl;
            else
               cout << "Demasiadas iteraciones" << endl;
         }
      }
      cout << endl;

      // Nueva lectura

      cout << endl;
      cout << "Escriba un numero de cuatro o menos cifras"
           << " (negativo o cero para finalizar) = ";
      cin >> num;

   } // while (num > 0)

   return 0;
}
/*****************************************************************/
/*****************************************************************/
```

El fichero de cabecera `funciones_Kaprekar.h` contendrá los **prototipos** de las funciones.

1. Observe la documentación -comentarios- en la cabecera de las funciones. La explicación detallada en el fichero `.h` ayuda a entender el propósito de la función y su interface.
2. Observe la manera en que se usa la construcción `#ifndef` / `#define` / `#endif` para evitar la inclusión repetida de código.

`funciones_Kaprekar.h`

```
//*************************************************************/
// METODOLOGIA DE LA PROGRAMACIÓN
//
// (C) FRANCISCO JOSÉ CORTIJO BON
// DEPTO. DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL
//
// Declaración de las funciones usadas en el problema de Kaprekar
// Fichero: funciones_Kaprekar.h
/*************************************************************/

#ifndef FUNCS_KAPREKAR
#define FUNCS_KAPREKAR

//*************************************************************/
// Función que devuelve el número de cifras de un número int.
// Entrada: num, el valor a estudiar.
// Devuelve:  número de cifras de "num".
// PRE: num>=0

int NumCifras (int num);

//*************************************************************/
// Calcula si todas las cifras de "num" son iguales.
// Entrada: num, el valor a estudiar.
// Devuelve: true, si todas las cifras de "num" son iguales.
// PRE: num>=0

bool TodasCifrasIguales (int num);

//*************************************************************/
// Calcula el número más pequeño posible con los dígitos de "num".
// Recibe: num, número a convertir.
// PRE: num>=0
```

```
int MinimoValorPosible (int num);

//****************************************************************/
// Calcula el número más grande posible con los dígitos de "num".
// Recibe: num, número a convertir.
// PRE: num>=0

int MaximoValorPosible (int num);

//****************************************************************/
// Calcula el número más grande posible con los dígitos de "num"
// y lo completa con ceros finales hasta alcanzar "num_digitos"
//  digitos.
// Si num_digitos <= NumDigitos(num) no se modifica el número
// de dígitos.
// Recibe: num, número a convertir.
//     num_digitos, número de dígitos del resultado.
// PRE: num>=0

int MaximoValorPosible (int num, int num_digitos);

//****************************************************************/
// Calcula el número de iteraciones que necesita el proceso Kaprekar
// para converger.
// Recibe: num, número de cuatro o menos cifras a verificar.
// Devuelve: el número de iteraciones empleadas en llegar
// a CTE_KAPREKAR.
//
// PRE: VerificaKaprekar(num) == true
// PRE: num>=0

int IteracionesKaprekar (int num);

//****************************************************************/
// Calcula si el número de iteraciones que requiere el proceso de
// Kaprekar hasta converger es correcto.
// Recibe: num, número de cuatro o menos cifras a verificar.
// Devuelve: true si el número de iteraciones empleadas es correcto.
//
// PRE: NumCifras(num) <= 4
// PRE: num>=0

bool VerificaKaprekar (int num);

//****************************************************************/

#endif
```

El fichero `funciones_Kaprekar.cpp` contendrá las **definiciones** de las funciones. Se repiten los comentarios en las cabeceras de las funciones, que ayudan a entender el código y a informar acerca de cómo se usan las funciones.

`funciones_Kaprekar.cpp`

```
//*****************************************************************/
// METODOLOGIA DE LA PROGRAMACIÓN
//
// (C) FRANCISCO JOSÉ CORTIJO BON
// DEPTO. DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL
//
// Definición de las funciones usadas en el problema de Kaprekar
//
// Fichero: funciones_Kaprekar.cpp
//
/*****************************************************************/

#include "funciones_Kaprekar.h"

using namespace std;

//*****************************************************************/
// Función que devuelve el número de cifras de un número int.
// Entrada: num, el valor a estudiar.
// Devuelve:  número de cifras de "num".
// PRE: num>=0

int NumCifras (int num)
{
   int cont = 0; // Contador de apariciones

   while (num != 0) {
      num = num /10; // Descartamos la cifra menos significativa
      cont++;
   }
   return (cont);
}

//*****************************************************************/
// Calcula si todas las cifras de "num" son iguales.
// Entrada: num, el valor a estudiar.
// Devuelve: true, si todas las cifras de "num" son iguales.
// PRE: num>=0
```

```
bool TodasCifrasIguales (int num)
{
   int ref = num % 10; // Cifra menos significativa (referencia)
   num = num / 10;  // Descartamos la cifra menos significativa

   bool todas_iguales = true;

   while (num != 0 && todas_iguales) {

      if (ref != num % 10) todas_iguales=false;
      else num = num / 10; // Descartamos la cifra menos significativa
   }
   return todas_iguales;
}

//*************************************************************/
// Calcula el número más pequeño posible con los dígitos de "num".
// Recibe: num, número a convertir.
// PRE: num>=0

int MinimoValorPosible (int num)
{
   int digitos[10] = {0};

   while (num>0) {
      digitos[num%10]++;
      num = num/10;
   }

   int resultado = 0;

   for (int i=1; i<10; i++)

      while (digitos[i]>0) {
         resultado=resultado*10 + i;
         digitos[i]--;
      }
   return resultado;
}

//*************************************************************/
// Calcula el número más grande posible con los dígitos de "num".
// Recibe: num, número a convertir.
// PRE: num>=0

int MaximoValorPosible (int num)
{
   int digitos[10] = {0};
```

```
   while (num>0) {
      digitos[num%10]++;
      num = num/10;
   }

   int resultado = 0;

   for (int i=9; i>=0; i--)

      while (digitos[i]>0) {
         resultado=resultado*10 + i;
         digitos[i]--;
      }
   return resultado;
}

//*************************************************************/
// Calcula el número más grande posible con los dígitos de "num"
// y lo completa con ceros finales hasta alcanzar "num_digitos"
//  digitos.
// Si num_digitos <= NumDigitos(num) no se modifica el número
// de dígitos.
// Recibe: num, número a convertir.
//     num_digitos, número de dígitos del resultado.
// PRE: num>=0

int MaximoValorPosible (int num, int num_digitos)
{
   int resultado = MaximoValorPosible (num);

   int digitos_faltan = num_digitos-NumCifras(num);

   for (int i=0; i<digitos_faltan; i++)
      resultado = resultado*10;

   return resultado;
}

//*************************************************************/
// Calcula el número de iteraciones que necesita el proceso Kaprekar
// para converger.
// Recibe: num, número de cuatro o menos cifras a verificar.
// Devuelve: el número de iteraciones empleadas en llegar
// a CTE_KAPREKAR.
//
// PRE: VerificaKaprekar(num) == true
// PRE: num>=0
```

```
int IteracionesKaprekar (int num)
{
   const int CTE_KAPREKAR = 6174;

   // Como se cumplen las precondiciones se asegura la convergencia.
   // En cada paso se genera un valor "num" de acuerdo a las
   // indicaciones y en algún paso llegará a valer "CTE_KAPREKAR".

   int iteraciones = 0;

   while (num != CTE_KAPREKAR) {

      iteraciones++; // Se ha completado una iteración

      int menor = MinimoValorPosible(num);
      int mayor = MaximoValorPosible(num, 4);

      num = mayor-menor; // Nuevo valor de "num"

   } // while (num != CTE_KAPREKAR)

   return iteraciones;
}

//*************************************************************/
// Calcula si el número de iteraciones que requiere el proceso de
// Kaprekar hasta converger es correcto.
// Recibe: num, número de cuatro o menos cifras a verificar.
// Devuelve: true si el número de iteraciones empleadas es correcto.
//
// PRE: NumCifras(num) <= 4
// PRE: num>=0

bool VerificaKaprekar (int num)
{
   bool verifica = true;

   if (NumCifras(num)==4 && TodasCifrasIguales(num))

      verifica = false;

   else {

      const int MAX_ITERACIONES_KAPREKAR = 7;

      int iterac_para_num = IteracionesKaprekar(num);
```

```
        if (iterac_para_num>MAX_ITERACIONES_KAPREKAR)
           verifica = false;
    }

    return verifica;
}

//*************************************************************/
```

Para la generación del ejecutable nos basamos en el esquema de dependencias del proyecto (figura 1 en el que enumeramos los pasos a seguir. En la figura 2 indicamos los pasos a seguir. Observe que los pasos 1 y 2 son intercambiables.

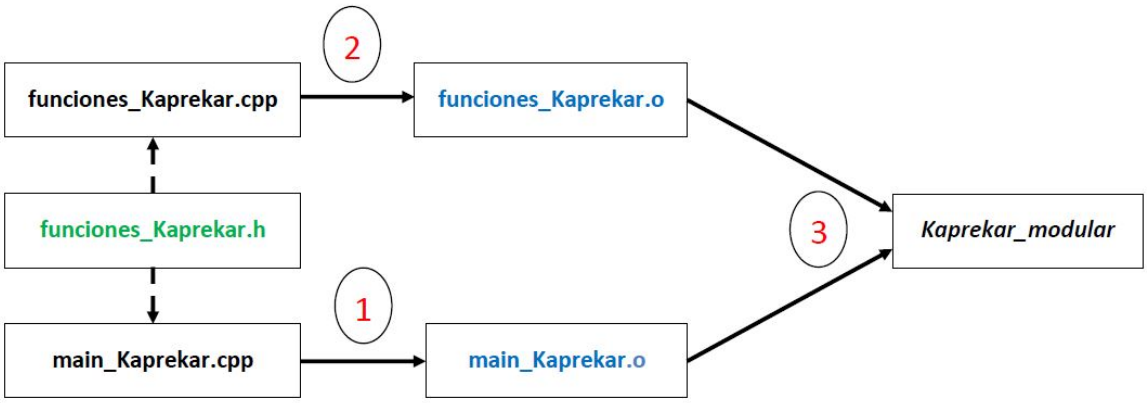

Figura 2: Diagrama de dependencias del proyecto completo

1. `g++ -c -o obj/main_Kaprekar.o src/main_Kaprekar.cpp -I./include`
2. `g++ -c -o obj/funciones_Kaprekar.o src/funciones_Kaprekar.cpp -I./include`
3. `g++ -o bin/Kaprekar_modular.o obj/main_Kaprekar.o obj/funciones_Kaprekar.o`