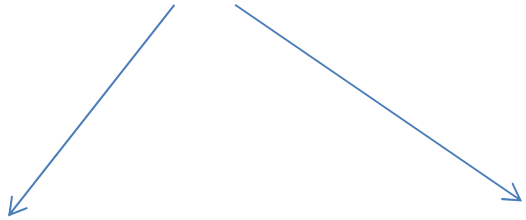
Diseño Modular

Resolver un problema

- Definir el problema
- Desarrollar una solución



Algoritmo Estructura de datos

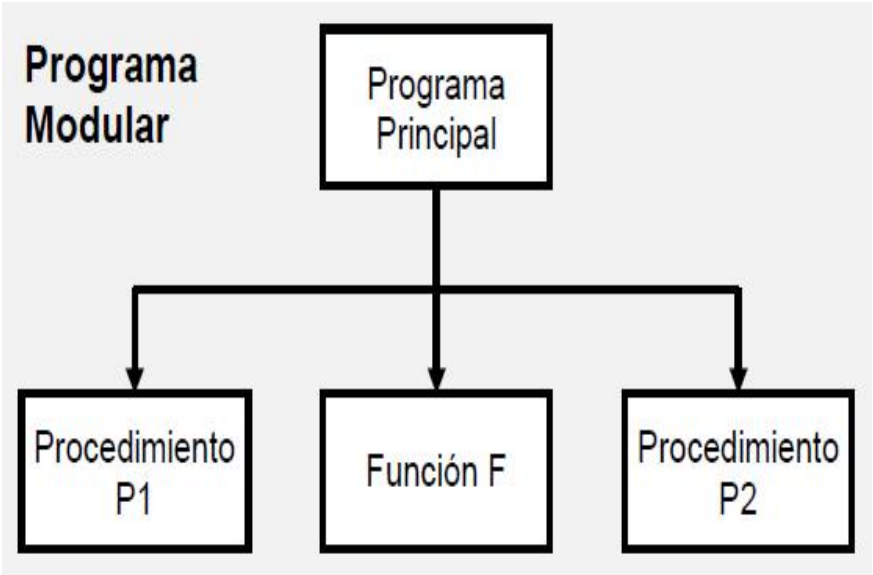
Programación tradicional

Desventajas:

- rigidez e inflexibilidad de los programas,
- pérdida excesiva de tiempo en la corrección de errores
- documentación deficiente e ineficiente, incluso mala,
- imposibilidad de reutilizar el código o fragmentos del mismo en proyectos futuros

Diseño Top-Down

- Consiste en llevar a cabo una tarea mediante pasos sucesivos a un nivel de detalle cada vez más bajo.
- Los módulos diseñados son independientes entre sí excepto por la interfaz que los comunica.



Abstracción procedural

- Cada algoritmo comienza como una caja negra.
- A medida que la resolución del problema avanza, se definen gradualmente las cajas negras hasta que se implementan las acciones que especifican en algún lenguaje de programación.
- Cada caja negra especifica **qué** hace, no **cómo** se hace.
- Ninguna caja negra debe saber cómo otra caja negra realiza una tarea, sino sólo qué acción realiza.
- Típicamente, esas cajas negras se implementan como subprogramas. La **abstracción procedural** separa el propósito de un programa de su implementación.

Ejemplo con GobStones

```
procedure CuidarCanteroAnidado()
{
   if (hayFlor())
   {
      if (not hayFertilizante())
      {
        Fertilizar()
       }
   }
}
```

Otro ejemplo GobStones

```
function hayAlcanzable()
{
   IrAPrimerCelda(Este, Norte)
   while(not esUltimaCelda(Este, Norte) && not
esAlcanzable())
   {
      PasarASiguienteCelda(Este, Norte)
   }
   return(esAlcanzable())
}
```

La **modularidad** y la **abstracción procedural** se complementan una a otra.

La modularidad implica dividir una acción en módulos; la abstracción procedural implica **especificar** cada módulo claramente antes de ser implementado.

Estos principios permiten la **modificación** de partes de la solución sin afectar a otras partes de la misma.

Ocultación de información

- La abstracción permite hacer públicas características de funcionamiento (qué hace) de los módulos, pero también establece mecanismos para **ocultar** detalles de los módulos que deben ser privados (sobre cómo hace tal cosa).
- Si **P** realiza una ordenación de elementos sobre un vector de enteros, de no más de **Max** componentes, **Q** no debe saber cómo **P** realiza la ordenación pero sí que debe "pasarle" un vector de enteros y de longitud **Max**.

Modularidad (Top-Down)

- **Construcción del programa**. Se reduce a escribir varios programas pequeños. Permite trabajar en módulos independientes.

- **Depuración de un programa**. La tarea de depurar un programa muy grande se reduce a la depuración de varios programas pequeños. Pruebas modulares y de integración.

- **Lectura de un programa**. La modularidad aumenta la legibilidad y comprensión de un programa. Un módulo bien escrito debe ser inteligible a partir de su nombre, los comentarios escritos en su cabecera y los nombres de los módulos que los llaman.

Modularidad (Top-Down)

- **Modificación de un programa**. Un pequeño cambio en los requerimientos de un programa debería implicar sólo un pequeño cambio del código. Un programa modular requerirá cambios sólo en unos pocos módulos. La **modularidad aísla las modificaciones**.

- **Eliminación de la redundancia de código**. Se pueden localizar operaciones que ocurren en diferentes lugares de un programa e implementarlas en subprogramas.

  Esto significa que el código de una operación aparecerá una sola vez, aumentando así tanto la legibilidad como la modificabilidad del programa.

Criterios de Modularidad

- No existen algoritmos formales para determinar cómo descomponer un problema, es totalmente subjetivo.
- Algunos criterios:
  - Acoplamiento
  - Cohesión

Criterios de Modularidad

Acoplamiento:

Es el grado de interconexión entre los módulos. Se debe **minimizar**.

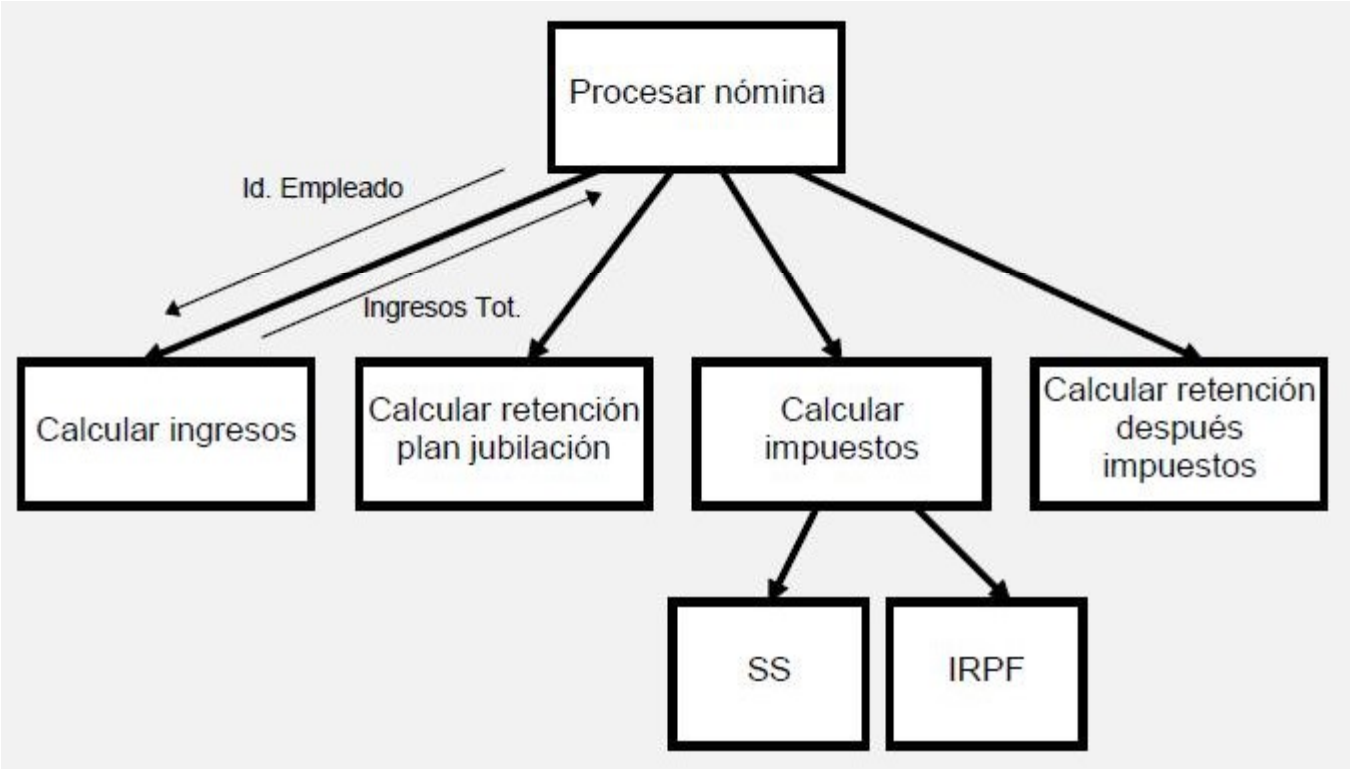
2 tipos: de control y de datos.

De control: implica la transferencia de control de un módulo a otro (ej. Llamada/retorno de subprogramas)

De datos: compartir datos entre los módulos.

Criterios de Modularidad

Acoplamiento



Criterios de Modularidad

Acoplamiento

- **Acoplamiento implícito:** se produce cuando se utilizan datos globales dentro de varios módulos.
- Algunos módulos pueden alterar la información de una forma no prevista por el resto del sistema, con posibles resultados imprevistos o desastrosos.
- **Solución:** no emplear variables globales.

  (Dentro de lo posible.)

Criterios de Modularidad

Cohesión

- Es el grado de interrelación entre las partes internas de un módulo. Hay que **maximizarla**.
- Hay 2 tipos: lógica y funcional.
- **Cohesión lógica:** consiste en agrupar dentro del mismo módulo elementos que realizan operaciones de similar naturaleza (es un cohesión débil).
- **Cohesión funcional:** consiste en que todas las partes del módulo están encaminadas a realizar una sola actividad (cohesión más fuerte).

Ventajas de la modularización

- Son una potente herramienta para desarrollar grandes programas. Fácil depuración.
- Los programas son fáciles de modificar.
- Programas más portables. Se pueden incorporar en un módulo los detalles pendientes de la máquina.
- Hace posible la compilación separada.
- Permite desarrollar bibliotecas con código reutilizable.
- El código generado es de más fácil comprensión. Está mejor documentado.

Módulos de biblioteca

- Se usan para exportar e importar recursos a otros módulos.
- Consta de 2 partes, la **parte de definición** y la **parte de implementación**.
- En C++ no están bien definidas como en otros lenguajes.
- En C++ usaremos los archivos de cabecera como .hpp y los módulos de implementación como .cpp.

Módulos de biblioteca

En los archivos de cabecera "*.hpp*" incluiremos:

1. las definiciones de constantes (la cláusula *#define* del preprocesador);
2. variables globales;
3. la descripción del programa; y
4. los prototipos de las funciones que aparecen en el programa.

Módulos de biblioteca

Archivo Funciones.hpp

```
#ifndef Funciones_hpp
//pregunta si no existe la definición
   Funciones_hpp
#define Funciones_hpp
//si no existe, la crea
int digVerif(char[9]);
//Prototipo de una función
#endif
```

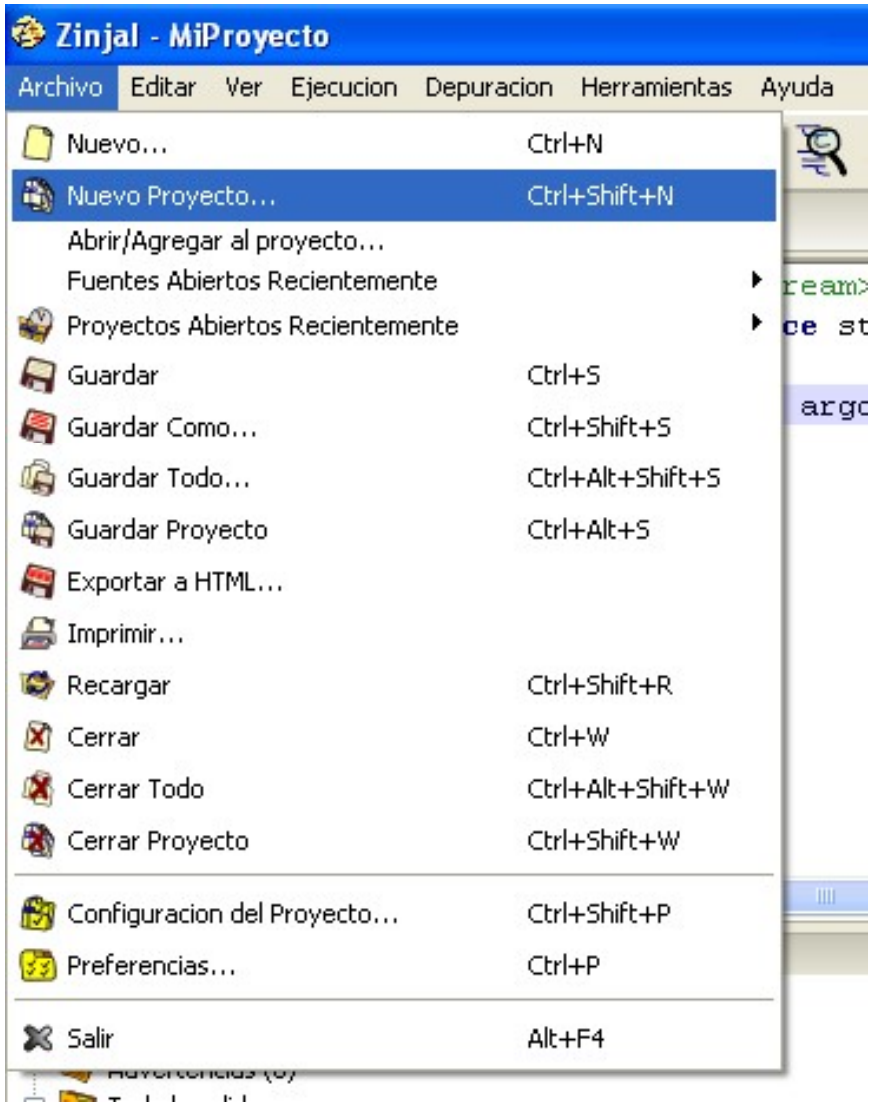
Módulos de biblioteca

Archivo Funciones.cpp

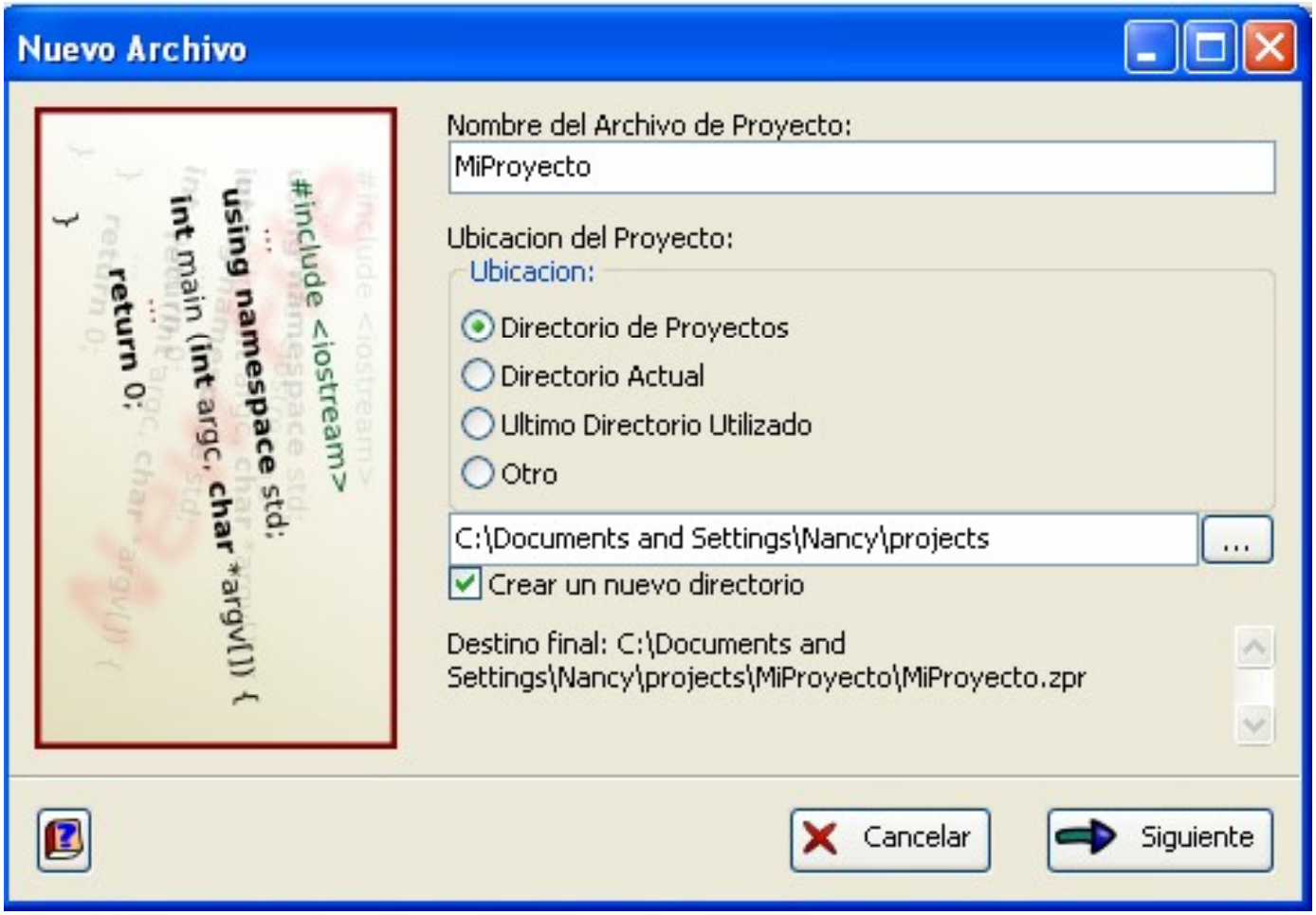
```
#include <conio.h>
#include <iostream.h>
#include "Funciones.hpp“

int digVerif(char ced[9])
{
  int suma=0, i, num[7]={2,9,8,7,6,3,4};
  for(i=0; i<7; i++)
    suma=suma+(((int)ced[i])-48)*num[i];
  suma=10-(suma%10);
  if(suma==(int)(ced[7]-48))
    return 1; //cédula válida
  else
    return 0;  //cédula inválida
}
```

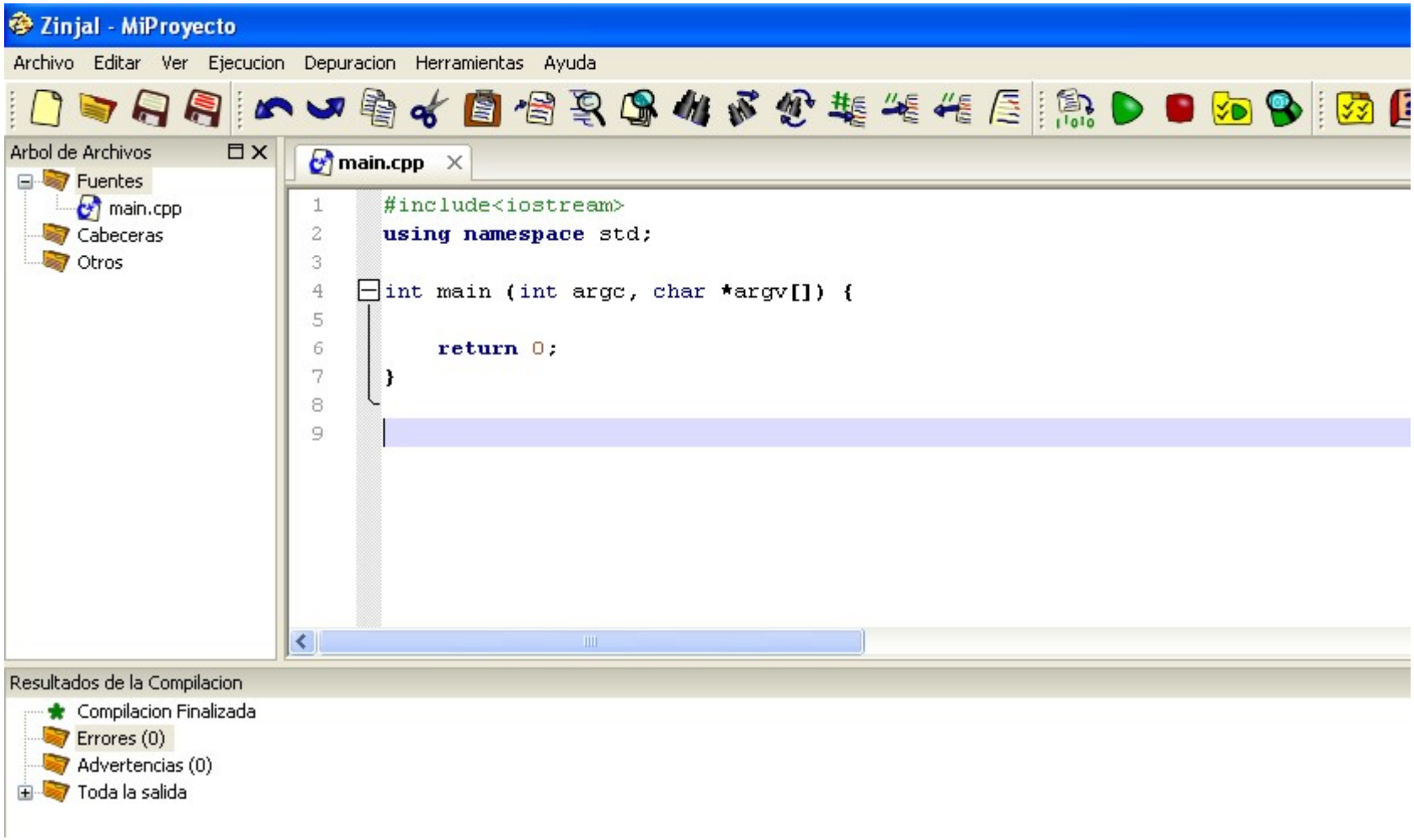
Modularización en Zinjal



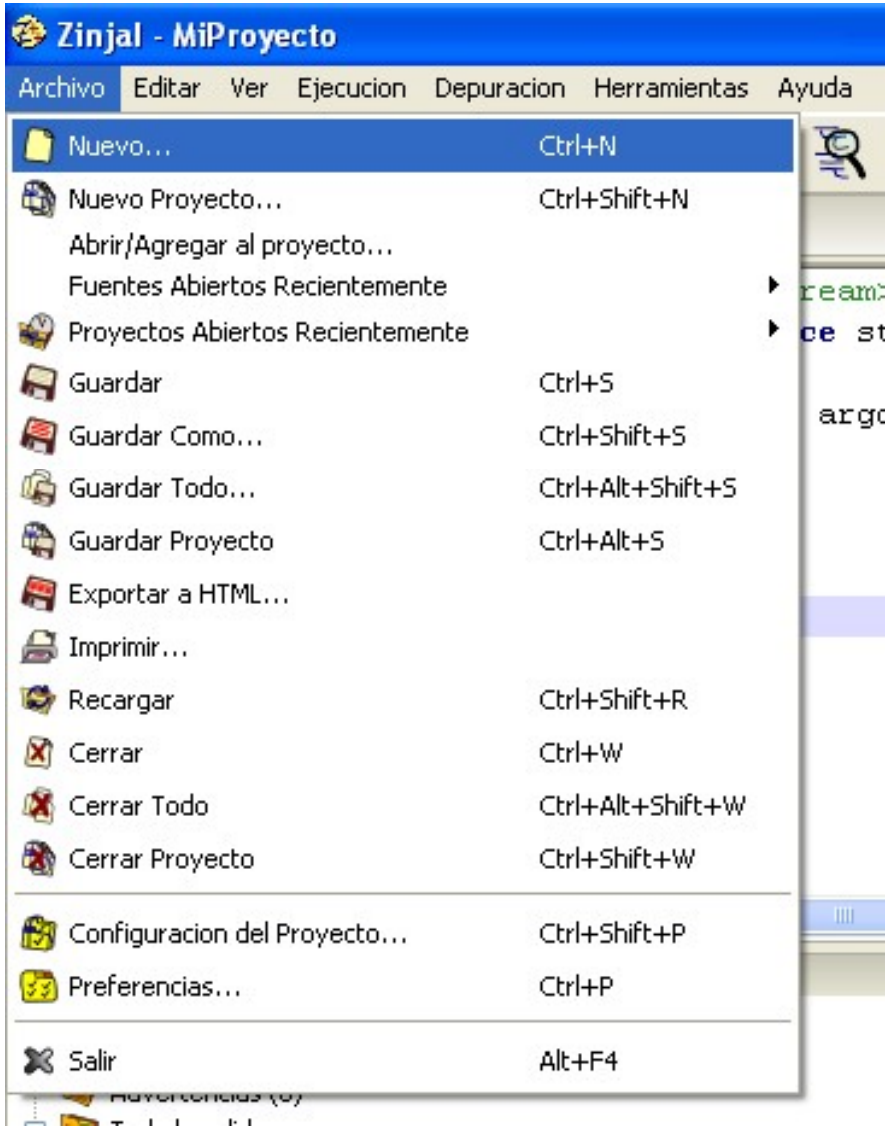
Modularización en Zinjal



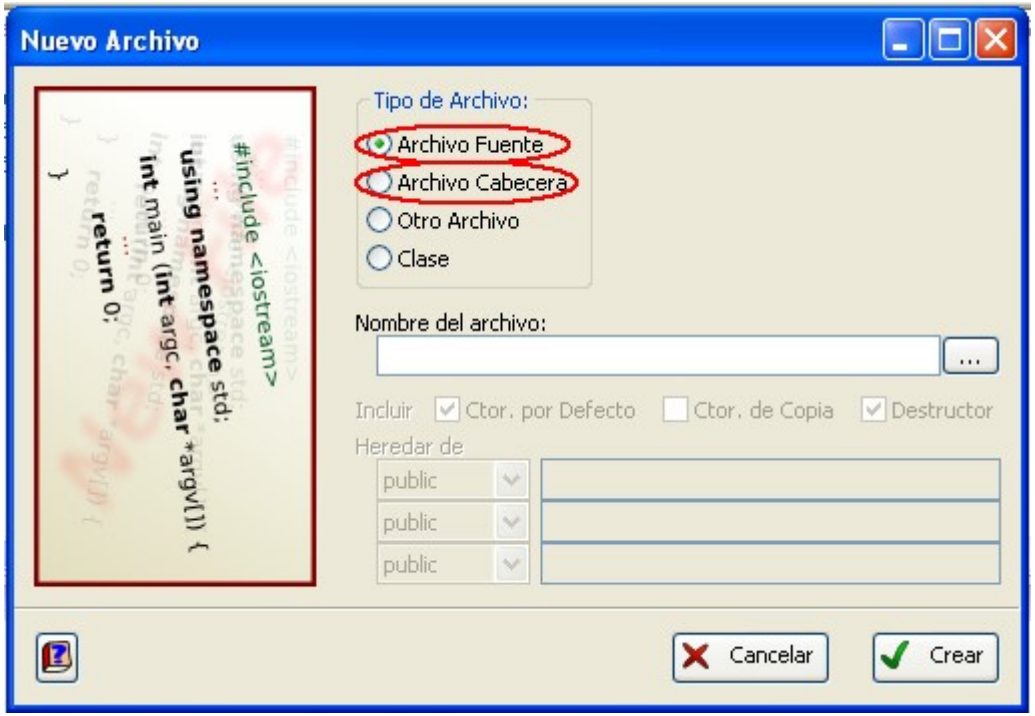
Modularización en Zinjal



Modularización en Zinjal



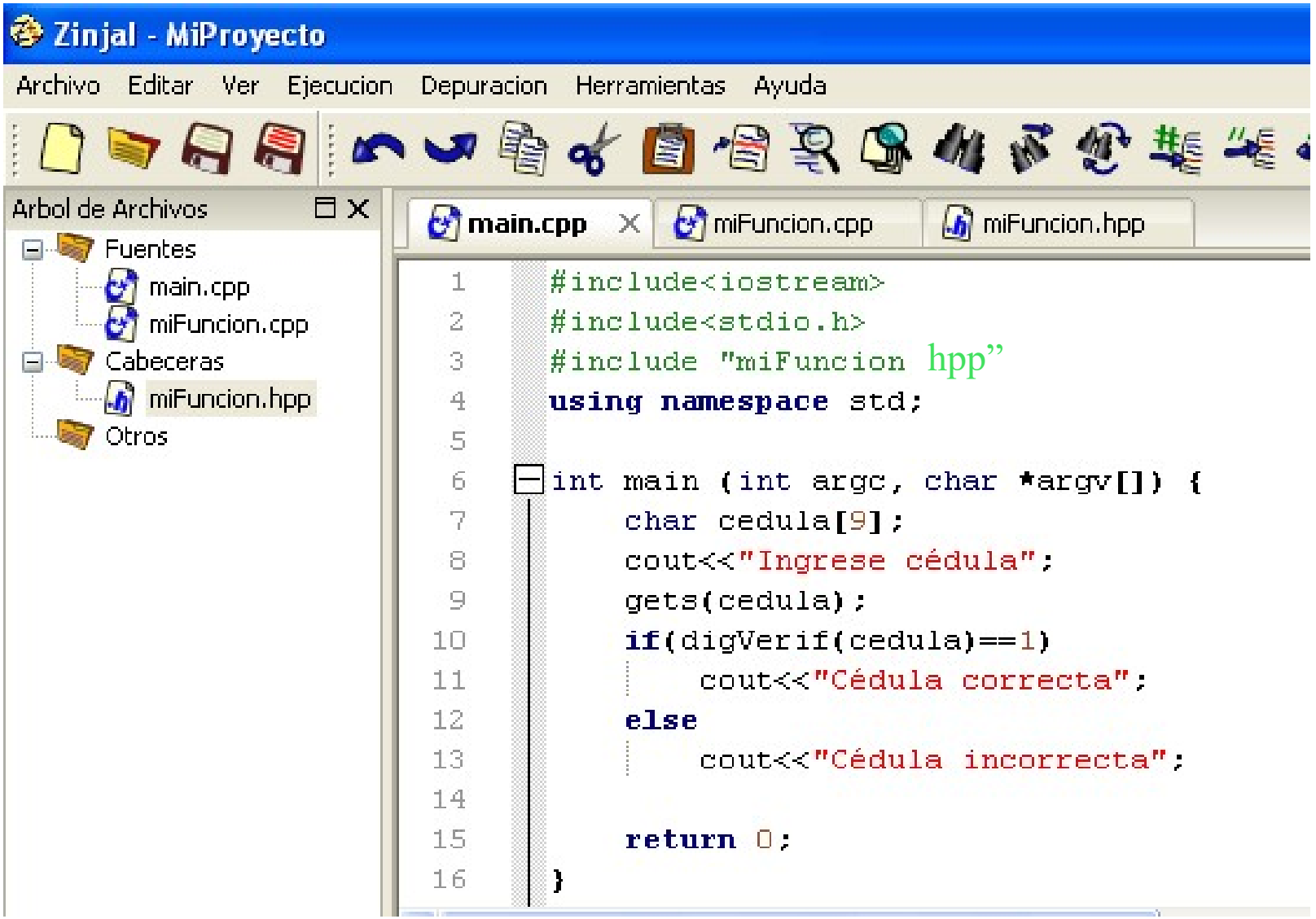
Modularización en Zinjal



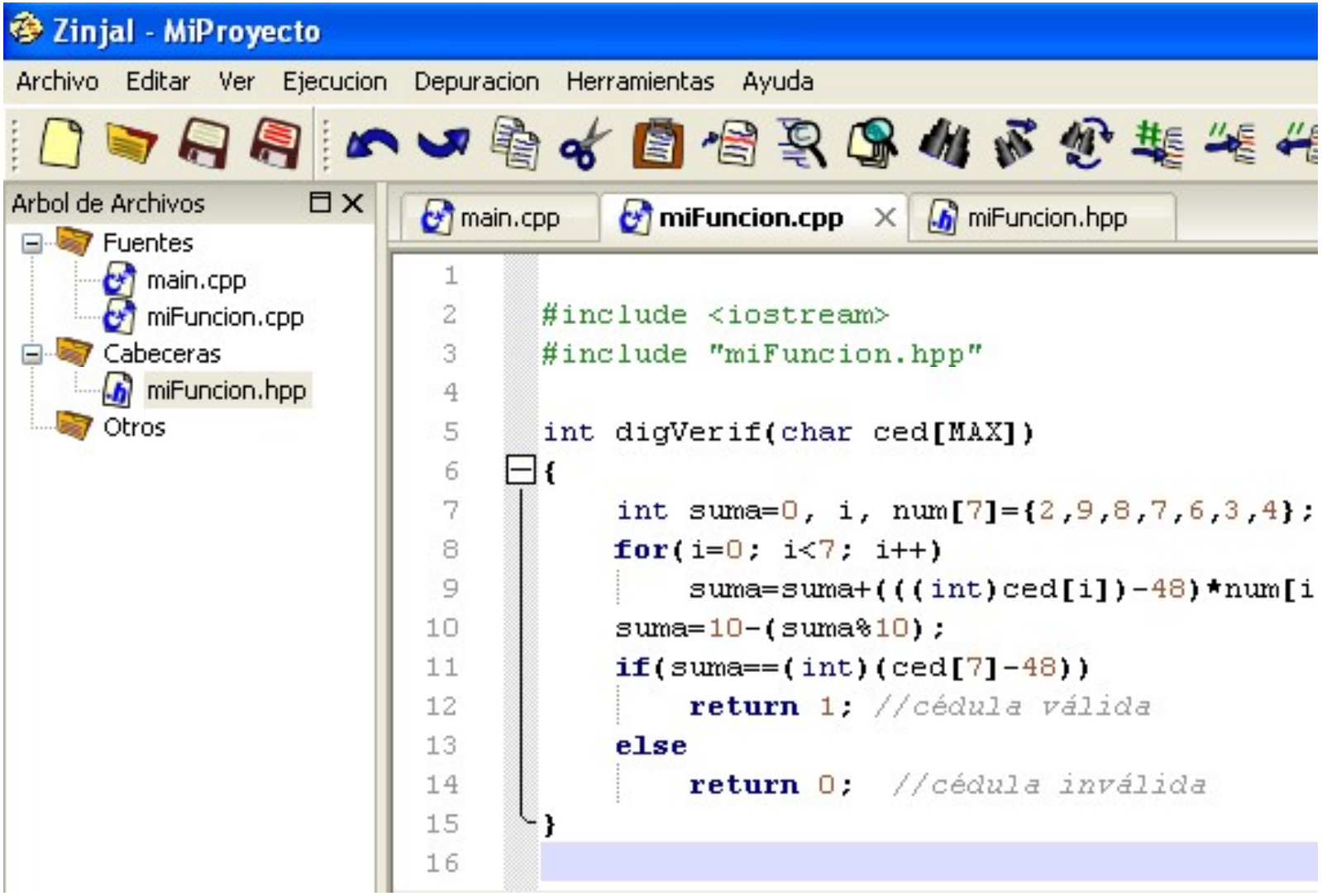
Archivo fuente:
.cpp

Archivo cabecera:
.hpp

Modularización en Zinjal



Modularización en Zinjal



Modularización en Zinjal