UNLaM
Dto. Ingeniería e Investigaciones Tecnológicas

Objetivo: Mejorar la organización del código, proponer una estructura del espacio de trabajo y facilitar la reutilización de código.

Temas: Nociones básicas de la compilación de un programa en C. Instrucciones al preprocesador.

Organización del código

Cuando estamos aprendiendo a programar, nos acostumbramos a hacer proyectos de programas pequeños, que generalmente tienen una sola función (**`main`**) y que rara vez voy a ver el código repetido. Pero ¿qué pasa cuando los programas empiezan a crecer y se comienza a ver que se repiten las acciones?

Lo primero que solemos hacer cuando nos damos cuenta de que una parte del problema que estamos resolviendo ya lo resolvimos en otra parte y en el apuro por terminar el proyecto copiamos y pegamos código. Pero esto trae consigo que, al descubrir algún error en el código, para resolverlo en todos lados debería modificar todas las porciones de código que haya copiado y pegado.

El lenguaje C, al ser un lenguaje estructurado, nos propone que separemos todos los procedimientos que podamos identificar de nuestro programa, en funciones. Hacer esto nos va a aportar claridad al código que estemos escribiendo y, además, la posibilidad de ejecutar en varias partes de nuestro programa ese funcionamiento sin duplicar el código.

Luego, a medida que nuestro programa sigue creciendo, empieza a ser difícil de seguir si está todo escrito en un solo archivo. Es en parte por esto que debemos empezar a agrupar las funciones que hayamos escrito, valga la redundancia, por funcionalidad. Podemos encontrar ejemplos en las bibliotecas que vienen con el lenguaje; **`stdio.h`**, tiene todas las funciones de entrada-salida (mediante streams o flujos); **`string.h`**,

agrupa las funciones de manejo de cadenas; **`ctype.h`**, las de manejo de caracteres; **`math.h`** las de matemáticas.

Es importante que sigamos esta forma de organización, para que nuestro programa sea entendible y no repita código. Y aún podemos ir más lejos, si creamos nuestras propias bibliotecas para no tener que repetir el código entre varios de nuestros programas.

Pero empecemos por separarnos en archivos.

Programas en varios archivos

Hay algunos conceptos que tenemos que entender, o problemas que tenemos que resolver para poder crear un programa en varios archivos.

Para empezar, deberíamos indicarle a nuestro compilador que este programa consta de varios archivos de código. Eso vamos a dejar que lo resuelva nuestro IDE (Integrated Development Environment o Entorno Integrado de Desarrollo), que es un software, con un editor de texto integrado, que nos da todas las herramientas que necesitamos para el desarrollo. En este caso, vamos a usar Code::Blocks.

Por otro lado, tenemos que entender cómo se van a comunicar los archivos de nuestro programa entre sí. Vamos a eso.

Conceptos básicos de funciones

Supongamos que escribimos un programa que resuelve el cálculo de factorial de un número que se ingresa por teclado, y lo muestra por pantalla.

```
#include<stdlib.h>
#include<stdio.h>

int main()
{
    int i,
        n,
        fact = 1;

    printf("Ingrese un numero entero: ");
```

UNLaM
Dto. Ingeniería e Investigaciones Tecnológicas

```
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
        fact *= i;
    printf("El factorial de %d es %d\n", n, fact);
    return 0;
}
```

Agregamos validaciones:

```
#include<stdlib.h>
#include<stdio.h>

int main ()
{
    int i,
        n,
        fact = 1;

    printf("Ingrese un numero entero: ");
    scanf("%d", &n);
    if(n < 0)
    {
        printf("No se puede calcular el factorial de un numero negativo.");
        return 1;
    }
    for (i = 1; i <= n; i++)
        fact *= i;
    printf("El factorial de %d es %d\n", n, fact);
    return 0;
}
```

Nos damos cuenta de que necesitamos que `fact` sea `double` por el tamaño de los factoriales:

```
#include<stdlib.h>
#include<stdio.h>
int main()
{
    int    i,
           n;
    double fact = 1;

    printf("Ingrese un numero entero: ");
```

UNLaM
Dto. Ingeniería e Investigaciones Tecnológicas

```
    scanf("%d", &n);
    if(n < 0)
    {
        printf("No se puede calcular el factorial de un numero negativo.\n");
        return 1;
    }
    for (i = 1; i <= n; i++)
        fact *= i;
    printf("El factorial de %d es %.0lf\n", n, fact);
    return 0;
}
```

Cuando calculamos el factorial de otro número . . .

```
#include<stdlib.h>
#include<stdio.h>
int main()
{
    int    i,
           n;
    double fact = 1;

    printf("Ingrese un numero entero: ");
    scanf("%d", &n);
    if(n < 0)
    {
        printf("No se puede calcular el factorial de un numero negativo.\n");
        return 1;
    }
    for (i = 1; i <= n; i++)
        fact *= i;
    printf("El factorial de %d es %.0lf\n", n, fact);

    fact = 1;
    printf("Ingrese otro numero entero: ");
    scanf("%d", &n);
    if(n < 0)
    {
        printf("No se puede calcular el factorial de un numero negativo.\n");
        return 1;
    }
    for (i = 1; i <= n; i++)
        fact *= i;
    printf("El factorial de %d es %.0lf\n", n, fact);
    return 0;
}
```

UNLaM
Dto. Ingeniería e Investigaciones Tecnológicas

. . . nos damos cuenta de que estamos repitiendo código, así que lo separamos en una función, teniendo en cuenta que en la función sólo queremos calcular el factorial de un número, no queremos incluir la obtención de ese número, para que esa función la podamos reutilizar, independientemente de si el dato se ingresa por teclado o desde un archivo, o viene de un cálculo previo del programa, etc.

Cambiamos un poco el código . . .

```
#include<stdlib.h>
#include<stdio.h>

double factorial(int n)
{
    double fact = 1;
    if (n < 0)
        return 0;
    for (; n > 1; n--)
        fact *= n;
    return fact;
}

int main ()
{
    int i, n;
    printf("Ingrese un numero entero: ");
    scanf("%d", &n);
    if (n < 0)
    {
        printf("No se puede calcular el factorial de un numero negativo.\n");
        return 1;
    }
    printf("El factorial de %d es %.0lf\n", n, factorial(n));

    printf("\nIngrese otro numero entero: ");
    scanf("%d", &n);
    if (n < 0)
    {
        printf("No se puede calcular el factorial de un numero negativo.\n");
        return 1;
    }
    printf("El factorial de %d es %.0lf\n", n, factorial(n));

    return 0;
}
```

UNLaM
Dto. Ingeniería e Investigaciones Tecnológicas

Agregamos el prototipo de la función factorial, así podemos moverla abajo.

```
#include <stdlib.h>
#include <stdio.h>

double factorial(int n);///Prototipo

int main ()
{
    int i,
        n;

    printf("Ingrese un numero entero: ");
    scanf("%d", &n);
    if (n < 0)
    {
        printf("No se puede calcular el factorial de un numero negativo.\n");
        return 1;
    }
    printf("El factorial de %d es %.0lf\n", n, factorial(n));

    printf("\nIngrese otro numero entero: ");
    scanf("%d", &n);
    if (n<0)
    {
        printf("No se puede calcular el factorial de un numero negativo.\n");
        return 1;
    }
    printf("El factorial de %d es %.0lf\n", n, factorial(n));

    return 0;
}

double factorial(int n)
{
    double fact = 1;
    if (n<0)
        return 0;
    for (; n>1; n--)
        fact*=n;
    return fact;
}
```

Podemos usar las funciones para aportarle claridad a nuestro código, separando en una función algún módulo de la solución que estamos desarrollando. O podemos generar funciones "de biblioteca", que serían funciones pensadas para ser utilizadas varias veces, y que resuelven un problema en particular.

Un ejemplo del primer caso puede ser, en un programa que presenta navegación por menús y submenús, la impresión y el comportamiento de cada submenú de ese programa estaría separado en funciones distintas. Así es que tendríamos la función "**`menu_principal`**", la función "**`submenu_opcion1`**", "**`submenu_opcion2`**", etc.

Y para el segundo caso, el ejemplo de cálculo de factorial nos viene bien, que es una función que resuelve un problema bastante general, y que, por lo tanto, va a ser utilizado muchas veces. En esta situación en particular, tenemos que intentar que nuestra función resuelva exclusivamente un problema planteado, y que todos los datos que necesita para funcionar vengan provistos en los parámetros. Así nos aseguraríamos de que podemos reutilizar la función en otro contexto. Un ejemplo de reutilización del factorial, sería el cálculo del número combinatorio:

$$\binom{m}{n} = \frac{m!}{n!(m-n)!}$$

```
int combinatorio(int n, int m)
{
    return factorial(m) / (factorial(n) * factorial(m - n));
}
```

Separando en archivos

En la medida que nuestro programa sigue creciendo, se hace necesario empezar a separar nuestro programa en archivos, para que nuestro código no quede en un solo archivo y sea imposible de seguir.

UNLaM
Dto. Ingeniería e Investigaciones Tecnológicas

Por otro lado, puede suceder que en otras soluciones que desarrollemos, empecemos a reutilizar funciones que hayamos generado antes. Por ejemplo, la validación de una fecha es algo que podríamos llegar a usar varias veces. Para este caso también es necesario que separemos en archivos esas funciones. Si tomamos como ejemplo las bibliotecas del lenguaje C, todas sus funciones están agrupadas en distintos archivos por tema, así que vamos a encontrar **`string.h`** que resuelve funciones de cadenas de caracteres, **`stdio.h`** todo lo que es entrada salida**`, math.h`**, matemática, etc.

Para ambos casos, el conocimiento básico que tenemos que saber es el mismo. Empecemos por los tipos de archivos en C.

Hay 2 tipos de archivos en C.

Los archivos de código, que tienen extensión **`.c`**, y es en donde ponemos la funcionalidad de nuestros programas en sí. Acá escribimos lo que hacen las funciones, el código propiamente dicho. Son los archivos que venimos utilizando normalmente.

Y los archivos de encabezados o "headers", que son los archivos donde vamos a informar cómo se usan nuestras funciones, qué tipos de datos manejamos, etc. En definitiva, aquí vamos a poner nuestros **`typedef`** y nuestros prototipos de funciones para quien quiera usar nuestras funciones sepa cómo hacerlo.

Recordando lo visto anteriormente con el factorial, cuando un programa en C es compilado, necesita conocer el prototipo de una función al ser llamada, así puede validar que los argumentos que se están pasando, y el valor de retorno que se espera sean correctos.

Pero ¿cómo se usan estos archivos "headers"? Lo bueno es que ya lo estuvimos haciendo cuando incluíamos las bibliotecas de C en nuestros programas con la directiva **`#include <stdlib.h>`**. ¡Y esa es la forma! La instrucción **`include`** es una instrucción al preprocesador que indica que se va a utilizar ese archivo de headers en el archivo actual.

Instrucciones (o directivas), al preprocesador de C

Son todas las instrucciones que vienen precedidas por un **#**. Las más conocidas son el **`include`** y el **`define`**, pero hay algunas más que son muy útiles también.

Pero primero, ¿qué es el preprocesador o pre compilador?

El preprocesador es un programa que se ejecuta antes de compilar nuestro programa, que se encarga de interpretar las directivas precedidas por el **#** y dejar nuestros archivos del programa listos para ser compilados.

Vamos a ver algunas de las instrucciones que nos van a ser útiles para el desarrollo de la cursada:

- Include: **`#include<archivo.h> o #include "archivo.h".`**
  - Esta instrucción indica al preprocesador que reemplace esta instrucción por todo el contenido del archivo que le pasamos por parámetro. Es decir, solo copia, sin ningún otro tipo de lógica.
  - La referencia al archivo es pasada como la ruta (o *path*)[1] relativa al mismo, pero varía el punto de inicio de búsqueda dependiendo si se usó <> o “”.
    - <>: La búsqueda se realiza entre todos los directorios marcados por defecto para el compilador para la búsqueda de bibliotecas de C.
    - **`""`**: La búsqueda se realiza a partir de la ubicación del archivo en el que se llamó a la instrucción include.
- Define: **#**define **`REM valor o #define REM(P1) valor`**
  - Esta instrucción indica el remplazo de todas las ocurrencias de la clave de remplazo, por el valor definido. Se pueden definir parámetros para ser remplazados en la expresión del valor.

[1] una ruta (path, en inglés) es la forma de referenciar un archivo informático o directorio en un sistema de archivos de un sistema operativo determinado. Una ruta señala la localización exacta de un archivo o directorio mediante una cadena de caracteres concreta. Puede ser relativa a la carpeta o directorio en el que estamos posicionados en este momento, o absoluta a partir del directorio raíz del volumen montado.

- De nuevo, define es solo un remplazo de texto con algo de lógica en el remplazo de los parámetros.
- Revisar el apunte de Macros o macro remplazos para más información.
  - If y sus variantes:

```
#if condición
      //sentencias en c para el verdadero
#else
      //más sentencias para el falso
#endif
```

- Es una expresión condicional para el preprocesador, y nos permite controlar qué código queremos que el compilador vea, dependiendo de una condición. Si la condición se cumple, el compilador solo verá las instrucciones que estén en la parte verdadera del if, si no, solo las que estén en la parte falsa
- En este curso, particularmente, nos vamos a enfocar en el uso del **`#ifndef MACRO`** que valida solamente si la macro especificada no fue definida, es decir, que no se haya ejecutado antes un **`#define MACRO`**.

Separamos nuestro programa en archivos

Vamos a crear un archivo de *headers* que sea **`matematica.h`** donde vamos a agrupar todos nuestros prototipos de funciones que resuelven algún problema matemático,

`matematica.h:`

```
#ifndef MATEMATICA_H_INCLUDED
#define MATEMATICA_H_INCLUDED

double factorial(int num);
int combinatorio(int n, int m);

#endif // MATEMATICA_H_INCLUDED
```

Toda la expresión del **`#ifndef`** con el **`#define`**, lo que hace es validar que este archivo no se haya incluido antes. Recordando lo que hace el **`include`**, que es únicamente una

copia del contenido del archivo al lugar donde se está ejecutando ese **include**, vemos que si en el archivo ya estaba definida la macro **MATEMATICA_H_INCLUDED** vemos que este código no va a incluir las definiciones que están dentro del **#ifndef**. Por ende, no se va a incluir dos veces el mismo archivo. Esto nos brinda la comodidad de incluir bibliotecas sin tener que estar pensando si las que incluí anteriormente ya tenían a la biblioteca.

Luego creamos **matematica.c**, donde vamos a poner las definiciones de las funciones que anteriormente declaramos, y además podemos poner las funciones internas del archivo que no queremos que nadie más que nuestro archivo conozca y use.

matematica.c

```c
#include "matematica.h"
int funcion_interna();

double factorial(int n)
{
    double fact = 1;
    if( n < 0)
    return 0;
    for(; n > 1; n--)
    fact *= n;
    return fact;
}

int combinatorio(int n, int m)
{
    return factorial(m) / (factorial(n) * factorial(m - n));
}

int funcion_interna()
{
    //comprtamiento de la función interna
    return 0;
}
```

Vemos que en nuestro archivo incluimos **matematica.h**. Esto no es estrictamente necesario, pero si es muy útil para no tener que pensar en el orden en el que

UNLaM
Dto. Ingeniería e Investigaciones Tecnológicas

desarrollamos las funciones en nuestro archivo **.c**, o mejor aún, ordenarlas como a nosotros nos resulte más fácil de leer. Más allá de eso podemos notar que en el **include** usamos las comillas **""**, para indicar el nombre del **.h**. Esto quiere decir que el archivo **.h** va a ser buscado en el mismo **path** de acceso que nuestro **archivo.c** (donde estamos haciendo el **include**).

Por último, modificamos nuestro archivo de **main** para que use las funciones de nuestros nuevos archivos.

main.c

```c
#include<stdlib.h>
#include<stdio.h>
#include "matematica.h"


int main ()
{
    int i,
        n;

    printf("Ingrese un numero entero: ");
    scanf("%d", &n);
    if (n < 0)
    {
        printf("No se puede calcular el factorial de un numero negativo.\n");
        return 1;
    }
    printf("El factorial de %d es %.0lf", n, factorial(n));

    return 0;
}
```

Nuevamente, para incluir el **matematica.h** usamos las comillas porque pusimos todos nuestros archivos en la misma carpeta.

En este paso habría que indicarle al compilador que tiene que compilar estos archivos que creamos, pero, como dijimos anteriormente, en este curso vamos a dejarle al IDE que haga todo lo necesario para poder compilar nuestros archivos, lo único que tenemos

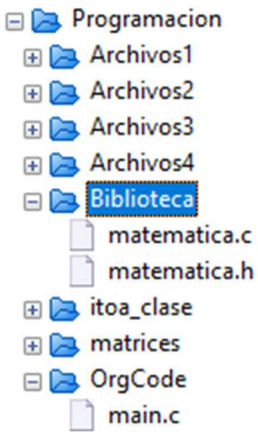
que ocuparnos para que Code::Blocks sepa que archivos debe compilar, es ponerlos todos en el mismo proyecto.

Para este momento, ya somos capaces de separar en distintos archivos nuestro código, para tenerlo mejor organizado. Pero ¿qué pasa cuando empezamos a encontrar el caso de que, en distintos programas, distintos proyectos empezamos a usar las mismas funciones? Un ejemplo trivial sería que para un programa desarrollamos una función que valida una fecha, y luego, lo empezamos a necesitar en la mayoría de nuestros programas que tienen una interfaz de usuario.

Armando el espacio de trabajo

Podemos crear nuestro propio espacio de trabajo, donde vamos a poner todos nuestros proyectos. Y dentro de ese espacio, separar una carpeta o directorio para los archivos con funciones genéricas, que empezamos a usar en más de un proyecto. Estamos armando nuestra biblioteca personal de código.

No hay una única forma de hacer esto, pero aquí va una propuesta:



Y lo único que debería cambiar con esa organización es el **`include`** de **`main.c`**, para que referencia a la ubicación correcta del archivo.

```
#include "../Biblioteca/matematica.h"
```

Noten que el **`include`** de **`matematica.h`** que pusimos en **`matematica.c`**, no debe ser cambiado, ya que la ruta del **`include`** se refiere al archivo en el que fue escrito y no al proyecto.

Conclusión

Es importante identificar el código que se repite y encapsularlo en funciones, así como organizarlo y reutilizarlo entre varios proyectos, esto va a ayudar en gran manera a la mantenibilidad y comprensibilidad de nuestro código, y además a agilizar nuestra manera de programar.

Práctica

Crear un programa que, a través de un menú, nos permita acceder y probar todos los ejercicios de la práctica 1. Separar la resolución de los ejercicios de la práctica en archivos por tema (matemática, vectores, matrices y fechas), como vimos en este apunte.