Programación con GPUs

Teórica 01 - Lenguaje C

Segundo Semestre, 2025

Contents

1.1 Introducción 4
1.2 ¿Cómo instalo un compilador de C? 4
1.2.1 Crear el contenedor 4
1.3 Hello World en C++ 5
1.4 Corriendo programas en C 6
1.5 Repaso de C y C 7
1.5.1 Compilación y ejecución 7
1.5.2 Sintaxis básica 7
1.5.3 Variables y tipos de datos 8
1.5.4 Operadores 8
1.5.5 Estructuras de control 9
1.5.6 Punteros 10
1.5.7 Funciones 11
1.5.8 Manejo de memoria 11
1.5.9 Matrices y Vectores 12
1.5.10 Entrada y salida de datos 13
1.5.11 Estructuras 15
1.5.12 Cadenas 16
1.6 Scopes 18
1.7 Errores 20
1.7.1 Errores de compilación 20
1.7.2 Errores de ejecución 21

¡Bienvenidos a Programación con Algoritmos 1!

Esta es la primera guía de la materia y tiene dos objetivos principales que son:

- **Aprender a compilar y ejecutar código en C / C**
- **Repasar los conceptos básicos de C / C** que son necesarios para la materia.

Esta práctica es el PRIMER paso para poder cursar la materia Algoritmos 1, porque lo que es importante que tengas todos estos conceptos incorporados en las primeras semanas de clases.
¡Recordá que es fundamental que preguntes y consultes todo lo que necesites!

1.1 Introducción

Para poder trabajar en la materia vas a tener que tener instalado un compilador C si querés probar tus propios ejercicios y problemas. Si sólo vas a correr los ejemplos y problemas vistos en clase podés utilizar simplemente la plataforma de programación competitiva de la universidad en: UP Competitive Programming Platform.

1.2 ¿Cómo instalo un compilador de C?

La forma más sencilla es utilizando `docker`. Docker Desktop es una aplicación que permite crear y administrar contenedores de Docker en tu computadora. Un contenedor de Docker es una instancia de una imagen de docker (muy similar a una máquina virtual) que se ejecuta de manera aislada del sistema operativo subyacente, lo que permite ejecutar aplicaciones y servicios de manera consistente en diferentes entornos.

La ventaja de utilizar docker es que no estás instalando nada más que docker (que es muy utilizada en entornos profesionales, con lo cual seguramente lo sigas usando), y luego todo lo que se instala dentro del contenedor es totalmente efímero y si un día querés eliminarlo, simplemente eliminás el contenedor y listo, sin tener que preocuparte por desinstalar nada ni por conflictos de versiones de software.

1.2.1 Crear el contenedor

La primera cosa que tenés que hacer es crear el contenedor de docker con el compilador que vamos a utilizar en la materia. Para eso tenés que ir al directorio `docker/` que se encuentra en este repositorio y ejecutar:

```
docker build -t algoritmos_1 .
```

Esto va a crear una imagen de docker con el nombre `algoritmos_1` que tiene instalado el compilador de C y todas las herramientas necesarias para poder compilar y ejecutar código en C.

¡Con esto ya podés correr los ejemplos y ejercicios de la materia!

1.3 Hello World en C++

En general uno asocia C con la programación orientada a objetos, sin embargo en esta materia, vamos a utilizar el lenguaje C como un C potenciado. Es decir, vamos a utilizar la sintaxis de C pero sin utilizar las características de la programación orientada a objetos, pero vamos a tomar ventaja de las bibliotecas de alto nivel que ofrece, como las estructuras de datos ya incluídas y sus algoritmos para poder tener un código que se ejecute en alto nivel que, además, pueda compilar C que es algo que verán en el resto de la carrera.

El código de ejemplo para imprimir un Hello World es un poco diferente en C que en C, porque la interfaz de entrada y salida de datos es diferente. En C se utiliza la biblioteca `iostream` para la entrada y salida:

```cpp
#include <iostream>  // esto es lo mismo que en C, pero sin la extensión .h

int main() {
  // std::cout es el objeto de salida estándar en C++, y std::endl es un
    salto de línea
  std::cout << "Hola mundo.... desde C++!!!" << std::endl;
  return 0;
}
```

Listing 1: Código de ejemplo Hello World en C++

En general el estudio de Algoritmos se puede hacer en cualquier lenguaje, pero C es un lenguaje de muy bajo nivel que no tiene en su biblioteca estándar algunas funcionalidades que necesitamos para poder estudiar cierto tipo de algoritmos.

1.4 Corriendo programas en C

Para ejecutar programas en C pueden utilizar una instalación local, pueden usar 'docker' como ya vimos en la sección anterior 1.2.1, pueden utilizar la plataforma de programación competitiva de la universidad, o pueden utilizar alguna plataforma como OneCompiler o OnlineGDB.

En cualquier caso si van a utilizar un contenedor de docker o una instalación local de `g++` para compilar el código, el comando para compilar un programa en C es el siguiente:

```
g++ -Wall -o programa programa.cc
```

Donde `-Wall` es un flag que le indica al compilador que muestre todas las advertencias, `-o programa` es el flag que le indica al compilador que el archivo ejecutable se llame `programa`, y `programa.cc` es el archivo de código fuente que queremos compilar.

1.5 Repaso de C y C

ATENCIÓN
En esta sección vamos a repasar los conceptos básicos de C, algunas cosas no van a estar explicadas porque el objetivo de esta materia no es enseñar C, con lo cual si alguna de las cosas que están escritas acá no se entiende, podrás preguntar en clase o investigar un poco.
Los conceptos IMPORTANTES van a estar explicados, pero la sintaxis del lenguaje no va a estar explicada en detalle, porque el libro de C en sí mismo es un libro de más de 600 páginas y no es obligatorio para la materia, simplemente aprendé la sintaxis.
Es cierto que escribir `print "hola"` puede resultar más ameno que escribir `std::cout « "hola" « std::endl`, pero en definitiva es una cuestión exclusivamente de sintaxis.
Igual iremos explicando lo que significa cada parte del código, pero lo importante será enfocarnos en algoritmos.

1.5.1 Compilación y ejecución

Este apartado está repetido en la práctica para que puedas ver cómo se compila y ejecuta un programa en C. Supongamos que tenés el siguiente código en un archivo llamado `cuadrado.cc`:

```cpp
#include <iostream>

using namespace std; // Esto no es obligatorio en versiones nuevas de C++

int main() {
  int n;

  cin >> n; // Lee un número desde la entrada estándar (similar a scanf en C)
  cout << n*n << endl; // Imprime el cuadrado del número (similar a printf en C)

  return 0;
}
```

Listing 2: Cuadrado de un número

```
g++ -Wall -o cuadrado cuadrado.cc
./cuadrado
5
25
```

La idea es que cada vez que ejecutemos este programa, se quedará esperando a que ingresemos un número por pantalla y luego imprimirá el cuadrado de ese número.

1.5.2 Sintaxis básica

El lenguaje C es un lenguaje de programación de propósito general, que se caracteriza por su simplicidad y eficiencia. Un programa en C se compone de una o más funciones, siendo la

función `main` la función principal que se ejecuta al iniciar el programa. La sintaxis básica es la siguiente:

```cpp
// #include sirve para incluir bibliotecas
#include <iostream>  // Biblioteca para entrada y salida de datos
#include <algorithm> // Biblioteca para algoritmos predefinidos
#include <vector>    // Biblioteca para arrays dinámicos (vectores)
// etc.

// main es la funcion principal del programa que DEBE estar presente
// argc y argv son los argumentos de la linea de comandos
int main(int argc, char *argv[]) {
  cout << "Hola mundo.... desde C++!!!" << endl; // Imprime un mensaje en
    la consola
  return 0;
}
```

Listing 3: Sintaxis básica de un programa en C

1.5.3 Variables y tipos de datos

En C, las variables se utilizan para almacenar datos y deben ser declaradas antes de ser utilizadas. Los tipos de datos más comunes en C son:

- `int`: Enteros (números enteros).
- `float`: Números de punto flotante (decimales).
- `double`: Números de punto flotante de doble precisión.
- `char`: Caracteres individuales.
- `void`: Tipo de dato vacío, utilizado para funciones que no retornan un valor.

Además se pueden utilizar modificadores de tipo como `unsigned` para enteros sin signo, o `long` para enteros de mayor tamaño. Por ejemplo, `unsigned int` es un entero sin signo que puede almacenar valores desde 0 hasta 2^{32-1}, mientras que `long int` es un entero de mayor tamaño que puede almacenar valores desde -2^{63} hasta 2^{63-1}.

1.5.4 Operadores

C cuenta con una variedad de operadores que se utilizan para realizar operaciones aritméticas, lógicas y de comparación. Los operadores más comunes son:

- Aritméticos: `+`, `-`, `*`, `/`, `%`.
- Logicos: `&&` (AND), `||` (OR), `!` (NOT).
- De comparacion: `==`, `!=`, `<`, `>`, `<=`, `>=`.
- Asignación: `=`, `+=`, `-=`, `*=`, `/=`, `%=`.

- Bit a bit: `&`, `|`, `^`, `~`, `«`, `»`.

Los operadores aritméticos no rquieren una explicación adicional, con excepción del operador `%` que es el operador de módulo que nos devuelve el resto de una división entera. Por ejemplo, si tenemos `5 % 2` el resultado es `1` porque `5 / 2` da como resultado `2` y el resto es `1`.

Los operadores lógicos se utilizan para combinar expresiones que devuelve un valor de verdad (`true` o `false`. **No olvidar que FALSE en C es**: `NULL`, `0` o `false`, todo lo demás se considera `true`.

Los operadores de comparación se utilizan para comparar dos valores y devolver un valor de verdad.

Los operadores de asignación se utilizan para asignar un valor a una variable. Los operadores de asignación compuestos (`+=`, `-=`, `*=`, `/=`, `%=`) son una forma abreviada de escribir una operación de asignación combinada con una operación aritmética. Por ejemplo, `x += 5` es equivalente a `x = x + 5`. Recordar que en la asignación existe el `lvalue` y el `rvalue`, donde el `lvalue` es la variable donde se va a almacenar el resultado de la operación y el `rvalue` es el valor que se va a asignar. Esta es la razón por la cual no se puede hacer una asignación como `5 = x` ya que `5` no es un `lvalue`, lo que significa que no se puede asignar un valor a él.

Por último, los operadores bit a bit se utilizan para realizar operaciones a nivel de bits en números enteros.

- `&`: `5 & 3` da como resultado `1` porque en binario `5` es `101` y `3` es `011`, y al hacer la operación AND bit a bit, el resultado es `001`.
- `|`: `5 |3` da como resultado `7` porque en binario `5` es `101` y `3` es `011`, y al hacer la operación OR bit a bit, el resultado es `111`.
- `^`: `5 ^3` da como resultado `6` porque en binario `5` es `101` y `3` es `011`, y al hacer la operación XOR bit a bit, el resultado es `110`.
- `~`: `~` es el operador de complemento a uno, que invierte todos los bits de un número. Por ejemplo, `~5` da como resultado `-6`.
- `«`: `5 « 1` desplaza los bits de `5` una posición a la izquierda, lo que equivale a multiplicar por 2, dando como resultado `10`.
- `»`: `5 » 1` desplaza los bits de `5` una posición a la derecha, lo que equivale a la división entera por 2, dando como resultado `2`.

1.5.5 Estructuras de control

C cuenta con estructuras de control que permiten tomar decisiones y repetir bloques de código. Las estructuras más comunes son:

- Condicionales: `if`, `else if`, `else`
- Bucles: `for`, `while`
- Saltos: `break`, `continue`, `return`.

Ejemplos:

```c
#include <stdio.h>

int main() {
  // Bucle for
  for (int x = -10; x < 11; x++) {
    if((x > 5) && (x % 3 == 0)) {
      printf("x=%d es mayor a 5 y multiplo de 3\n", x);
    }

    // Condicional if
    if (x > 0) {
      printf("x=%d es positivo\n", x);
    } else if (x < 0) {
      printf("x=%d es negativo\n", x);
    } else {
      printf("x=%d es cero\n", x);
    }
  }

  // Bucle while
  int j = 0;
  while (j < 5) {
    printf("Iteracion %d\n", j);
    j++;
  }

  return 0;
}
```

Listing 4: Estructuras de control en C

1.5.6 Punteros

Los punteros son una característica poderosa de C que permite manipular direcciones de memoria directamente. Un puntero es una variable que almacena la dirección de memoria de otra variable. Se declaran utilizando el operador `*` y se accede al valor al que apuntan utilizando el operador `&`. Por ejemplo:

```c
#include <stdio.h>

int main() {
  int x = 10; // Declaracion de una variable entera
  int *p = &x; // Declaracion de un puntero que apunta a la direccion de x

  printf("Valor de x: %d\n", x); // Imprime el valor de x
  printf("Direccion de x: %p\n", (void*)&x); // Imprime la direccion de x
  printf("Valor al que apunta p: %d\n", *p); // Imprime el valor al que
    apunta p (valor de x)
  printf("Direccion almacenada en p: %p\n", (void*)p); // Imprime la
    direccion almacenada en p

  return 0;
```

```
}
```

Listing 5: Ejemplo de punteros en C

1.5.7 Funciones

Las funciones en C son bloques de código que realizan una tarea específica y pueden ser reutilizadas en diferentes partes del programa. Se declaran utilizando la siguiente sintaxis:

```c
int sumar_enteros(int a, int b) {
  return a + b; // Retorna la suma de a y b
}
```

Listing 6: Declaración de una función en C

Para llamar a una función, simplemente se utiliza su nombre seguido de los argumentos entre paréntesis:

```c
#include <stdio.h>

int sumar_enteros(int a, int b) {
  return a + b;
}

int main() {
  int resultado = sumar_enteros(5, 10); // Llama a la funcion sumar_enteros
  printf("Resultado: %d\n", resultado); // Imprime el resultado
  return 0;
}
```

Listing 7: Llamada a una función en C

1.5.8 Manejo de memoria

En C, el manejo de memoria básico de C se realiza utilizando las funciones `malloc` y `free` de la biblioteca `stdlib.h`. La función `malloc` se utiliza para asignar memoria dinámica y la función `free` se utiliza para liberar la memoria asignada. Por ejemplo:

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

int main() {
  int *arr = (int*)malloc(5 * sizeof(int)); // Asigna memoria para un
    arreglo de 5 enteros
  if (arr == NULL) {
    printf("Error al asignar memoria\n");
    return 1; // Sale del programa si no se pudo asignar memoria
  }

  for (int i = 0; i < 5; i++) {
    arr[i] = i * 10; // Inicializa el arreglo
  }

  for (int i = 0; i < 5; i++) {
```

```
    printf("%d ", arr[i]); // Imprime el arreglo
  }
  printf("\n");

  free(arr); // Libera la memoria asignada
  return 0;
}
```

Listing 8: Manejo de memoria en C

1.5.9 Matrices y Vectores

Al pedir memoria con `malloc` C nos retorna un puntero a la memoria asignada, y esta memoria asignada es contigua, con lo cual podemos utilizar punteros o subíndices para acceder a los elementos de la memoria asignada. Por ejemplo si tenemos un vector de tamaño 100 y queremos acceder al elemento 47 podemos hacerlo de la siguientes maneras:

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

int main() {
  int *vector = (int*)malloc(100 * sizeof(int)); // Asigna memoria para un
    vector de 100 enteros
  if (vector == NULL) {
    printf("Error al asignar memoria\n");
    return 1; // Sale del programa si no se pudo asignar memoria
  }

  // Acceso al elemento 47 del vector usando subindice
  vector[47] = 42;
  printf("Elemento en la posicion 47: %d\n", vector[47]);

  *(vector + 47) += 1; // Acceso al elemento 47 del vector usando puntero
  printf("Elemento en la posicion 47: %d\n", *(vector + 47));

  free(vector); // Libera la memoria asignada
  return 0;
}
```

Listing 9: Acceso a elementos de un vector en C

Supongamos que ahora tenés que trabajar con una matriz de 100x200 enteros (20.000 elementos), en este caso lo que se puede hacer es calcular la dirección de memoria del elemento en la fila `fila` y la columna `col` de la matriz haciendo lo siguiente:

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

int main() {
  int filas = 100;
  int columnas = 200;
  int *matriz = (int*)malloc(filas * columnas * sizeof(int)); // Asigna
    memoria para una matriz de 100x100 enteros
  if (matriz == NULL) {
```

```c
    printf("Error al asignar memoria\n");
    return 1; // Sale del programa si no se pudo asignar memoria
  }

  int fila = 47;
  int col = 42;

  // Acceso al elemento en la fila y columna especificadas
  int posicion_memoria = fila * columnas + col; // Calcula la posicion en
   memoria
  matriz[posicion_memoria] = 1234; // Acceso usando subindice
  printf("Elemento en la posicion [%d][%d]: %d\n", fila, col, matriz[
   posicion_memoria]);

  *(matriz + posicion_memoria) += 1; // Acceso usando puntero
  printf("Elemento en la posicion [%d][%d]: %d\n", fila, col, *(matriz +
   posicion_memoria));

  free(matriz); // Libera la memoria asignada
  return 0;
}
```

Listing 10: Acceso a elementos de una matriz en C

1.5.10 Entrada y salida de datos

C proporciona funciones para la entrada y salida de datos a través de la biblioteca `stdio.h`. Las funciones más comunes son:

- `printf`: Para imprimir datos en la consola.
- `scanf`: Para leer datos desde la entrada estándar (teclado).
- `getchar`: Para leer un carácter desde la entrada estándar.
- `putchar`: Para imprimir un carácter en la salida estándar.
- `fgets`: Para leer una línea completa desde la entrada estándar.
- `fputs`: Para imprimir una línea completa en la salida estándar.

Estas funciones son ideales para interactuar con el usuario, pero también funcionan para obtener datos de STDIN y enviarlo a STDOUT, que es lo que se puede utilizar luego para redirigir la entrada y salida de datos desde y hacia archivos.

Supongamos que tenemos un matriz de `filas x columnas` enteros y queremos leer los datos desde la entrada estándar, podemos crear un archivo así:

```
3 4
1 2 3 4
5 6 7 8
9 10 11 12
```


Y luego leerlo desde un programa en C de la siguiente manera:

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

int main() {
  int filas, columnas;
  scanf("%d %d", &filas, &columnas); // Lee las dimensiones de la matriz

  int **matriz = (int**)malloc(filas * sizeof(int*)); // Asigna memoria para las filas
  for (int i = 0; i < filas; i++) {
    matriz[i] = (int*)malloc(columnas * sizeof(int)); // Asigna memoria para las columnas
  }

  // Lee los datos de la matriz
  for (int i = 0; i < filas; i++) {
    for (int j = 0; j < columnas; j++) {
      scanf("%d", &matriz[i][j]);
    }
  }

  // Imprime la matriz
  printf("Matriz:\n");
  for (int i = 0; i < filas; i++) {
    for (int j = 0; j < columnas; j++) {
      printf("%d ", matriz[i][j]);
    }
    printf("\n");
  }

  return 0;
}
```

Listing 11: Lectura de datos desde la entrada estándar en C

1.5.11 Estructuras

Las estructuras en C son una forma de agrupar diferentes tipos de datos bajo un mismo nombre. Se declaran utilizando la palabra clave `struct` y se pueden utilizar para crear tipos de datos personalizados. Por ejemplo:

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

struct Persona {
  char nombre[50]; // Nombre de la persona
  int edad; // Edad de la persona
};

int main() {
  struct Persona* persona = (struct Persona*)malloc(sizeof(struct Persona))
    ; // Asigna memoria para una estructura
  if (persona == NULL) {
    printf("Error al asignar memoria\n");
    return 1; // Sale del programa si no se pudo asignar memoria
  }

  // Inicializa la estructura
  snprintf(persona->nombre, sizeof(persona->nombre), "Juan Perez"); //
    Asigna el nombre
  persona->edad = 30; // Asigna la edad
  printf("Nombre: %s, Edad: %d\n", persona->nombre, persona->edad); //
    Imprime los datos de la persona
  free(persona); // Libera la memoria asignada
  return 0;
}
```

Listing 12: Declaración de una estructura en C

1.5.12 Cadenas

En C, las cadenas de caracteres se representan como arreglos de caracteres terminados en un carácter nulo (`NULL`). Es **muy** importante recordar que para ser una cadena válida, el último carácter debe ser el carácter nulo, ya que muchas de las funciones de manipulación de cadenas en C dependen de este carácter para determinar el final de la cadena.

El caracter nulo en C es el caracter con número ASCII 0, y se representa como `0` o `NULL`.

Si uno utiliza una cadena en C, el compilador automáticamente agrega el carácter nulo al final de la cadena, por lo que no es necesario agregarlo. Pero si estamos utilizando memoria dinámica para almacenar una cadena, debemos asegurarnos nosotros de que el último carácter sea el carácter nulo, para que las funciones de manipulación de cadenas funcionen bien.

Por ejemplo:

```c
#include <stdio.h>
#include <string.h> // Incluye la biblioteca para manipulacion de cadenas

int main() {
  char cadena[20]; // Declara un arreglo de 20 caracteres

  cadena[0] = 'H'; // Asigna el primer caracter
  cadena[1] = 'o'; // Asigna el segundo caracter
  cadena[2] = 'l'; // Asigna el tercer caracter
  cadena[3] = 'a'; // Asigna el cuarto caracter

  printf("El largo de cadena es: %d\n", strlen(cadena)); // 'cadena' NO termina en NULO!!!, CUIDADO!!

  char* cadena2 = "Hola"; // Declara una cadena literal, termina en NULO automaticamente
  printf("El largo de cadena2 es: %d\n", strlen(cadena2)); // cadena2 termina en NULO

  return 0;
}
```

Listing 13: Ejemplo de cadenas en C

La salida de este programa (puede) ser:

```
El largo de cadena es: 9
El largo de cadena2 es: 4
```

¿Por qué la cadena `cadena` tiene un largo de 9? Porque no termina en NULO, y por lo tanto la función `strlen` cuenta todos los caracteres hasta que encuentra un caracter `NULL`. Esto puede llevar a comportamientos inesperados, ya que si la cadena no termina en NULO, la función `strlen` seguirá contando hasta que encuentre un caracter `NULL` en memoria o hasta que se salga del rango de memoria asignada al programa y termine con un `Segmentation Fault`.

Para ello se utiliza la función `strnlen` que permite especificar el tamaño máximo de la cadena a contar, de esta forma evitamos tener problemas de memoria, ya que podemos especificar cuántos caracteres vamos a contar como máximo antes de que la función se salga del rango de memoria asignada al programa.

```c
#include <stdio.h>
#include <string.h> // Incluye la biblioteca para manejo de cadenas

int main() {
  char cadena[20]; // Declara un arreglo de 20 caracteres

  cadena[0] = 'H'; // Asigna el primer caracter
  cadena[1] = 'o'; // Asigna el segundo caracter
  cadena[2] = 'l'; // Asigna el tercer caracter
  cadena[3] = 'a'; // Asigna el cuarto caracter

  printf("El largo de cadena es: %d\n", strnlen(cadena, sizeof(cadena)));
   // Ahora si termina en NULO!!!

  return 0;
}
```

Listing 14: Ejemplo de cadenas en C con strnlen

Importante: Esto NO elimina el problema de que la cadena no termine en `NULL`, para que sólo cuente 4 caracteres deberíamos haber agregado `cadena[4] = NULL;`, para hacer que nustra cadena `Hola` termine en el caracter nulo.

1.6 Scopes

En C, el **scope** (o ámbito) de una variable se refiere a la región del código donde la variable es accesible. Existen diferentes tipos de scopes en C:

- **Scope global**: Las variables declaradas fuera de cualquier función tienen un scope global y son accesibles desde cualquier parte del programa.
- **Scope local**: Las variables declaradas dentro de una función tienen un scope local y sólo son accesibles dentro de esa función.
- **Scope de bloque**: Las variables declaradas dentro de un bloque (por ejemplo, dentro de llaves `{}`) tienen un scope limitado a ese bloque.

Saber utilizar los scopes correctamente es fundamental para evitar errores de acceso a variables, por ejemplo supongamos el siguiente código:

```c
#include <stdio.h>

int x = 10;

void asignar_valor_a_x(int nuevo_x) {
  int x = nuevo_x;
}

int main() {
  int x = 30;
  asignar_valor_a_x(20);
  printf("%d\n", x);
  return 0;
}
```

Listing 15: Ejemplo de scopes en C

En este caso la salida del programa será `30`. Y esto se debe a que la variable que se le pasa al `printf` es una variable local al *scope* de la función `main`.

El scope global no tiene ninguna sintaxis particular, simplemente se declara una variable fuera de cualquier función y en el mismo archivo que estamos trabajando [1]. El resto de los scopes se declaran dentro de llaves `{}`, ya sea scope de funciones o scope de bloques.

Si el scope tiene una sola línea de código, se puede omitir las llaves, excepto en el caso de que se necesite declarar una función. Ejemplos:

```c
#include <stdio.h>

int x = 10; // scope global

int main() {
  printf("%d\n", x); // imprime el valor de x en el scope global

  int x = 30; // scope local a la funcion main
```

[1] En C además existe `extern` que permite declarar variables globales en otros archivos, pero no es necesario para esta materia.

```c
  printf("%d\n", x); // imprime el valor de x en el scope local de main

  for(int i = 0; i < 5; i++) // scope de bloque
    printf("iteracion %d\n", i); // scope de una linea

  int j = 0;
  while(j < 5) { // scope de bloque
    int cuadrado = j * j; // scope local a la iteracion del while
    printf("cuadrado de %d es %d\n", j, cuadrado);
    j++;
  }

  return 0;
}
```

Listing 16: Ejemplo de scopes en C

En las versiones antiguas de C, las variables había que declararlas al principio del bloque. En general se recomienda declarar las variables lo más cerca posible de su uso, para evitar confusiones y mejorar la legibilidad del código.

1.7 Errores

Algo fundamental al programar es poder **entender** y **corregir** los errores que se vayan presentando. De hecho, considero que la habilidad de entender los errores y warnings de un compilador es una habilidad que se debe intentar desarrollar, ya que es un componente clave del proceso de programación.

1.7.1 Errores de compilación

Por ejemplo, supongamos el siguiente código:

```c
#include <stdio.h>

int main() {
  int n = 10;
  int* ptr = (int*)malloc(n * sizeof(int)); // Error: falta el include <
   stdlib.h>

  for(int i = 0; i < n; i++) {
    printf("Iteracion %d\n", i);
  }
}
```

Listing 17: Código con error de compilación

Al compilar este código, el compilador nos dará un error similar al siguiente:

```
ejemplo.c:5:20: error: call to undeclared library function 'malloc'
                with type 'void *(unsigned long)'; ISO C99 and later do not s
upport implicit function declarations [-Wimplicit-function-declaration]
  int* ptr = (int*)malloc(n * sizeof(int)); // Error: falta el include <stdlib.h>
                   ^
ejemplo.c:5:20: note: include the header <stdlib.h> or explicitly provide a declaration
```

En este caso, el error nos indica que la función `malloc` no está declarada, lo que significa que no hemos incluido la biblioteca `stdlib.h` que contiene la declaración de la función `malloc`. Para corregir el error, simplemente debemos incluir la biblioteca al principio del código:

```c
#include <stdio.h>
#include <stdlib.h> // Agregamos la biblioteca stdlib.h

int main() {
  int n = 10;
  int* ptr = (int*)malloc(n * sizeof(int)); // Ahora no hay error

  for(int i = 0; i < n; i++) {
    printf("Iteracion %d\n", i);
  }

  free(ptr); // No olvides liberar la memoria asignada
}
```

Listing 18: Código corregido

A esto se le llama **error de compilación**. Son los errores más fáciles de detectar porque el compilador nos informa de ellos antes de ejecutar el programa, sólo tenemos que ser cuidadosos sobre el mensaje de error que nos entrega el compilador.

1.7.2 Errores de ejecución

Los errores de ejecución son aquellos que ocurren mientras el programa está ejecutando, con lo cual son errores mucho más difíciles de detectar, porque pueden ocurrir en cualquier momento y no siempre son evidentes. Por ejemplo, si tenemos el siguiente código:

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

int main() {
  int m = 10;
  int n = 10000;
  int* ptr = (int*)malloc(n * m * sizeof(int));

  for(int i = 0; i < n*n; i++) { // deberia ser n * m
    ptr[i] = i;
  }

  free(ptr);
}
```

Listing 19: Código con error de ejecución

Al ejecutar este código, el programa puede fallar con un error de segmentación (segmentation fault) porque estamos intentando acceder a una posición de memoria que no hemos asignado. El error de segmentación ocurre cuando el programa intenta acceder a una dirección de memoria que no le corresponde y el sistema operativo lo detecta, terminando el programa abruptamente.

```
$ ./ejemplo
[1]    29897 segmentation fault  ./ejemplo
```