1 – Bienvenidos

- Presentación de los docentes del curso.
- Breve explicación de las pautas generales de la materia a discutir más exhaustivamente en el siguiente día de clase.
- Libros de consulta: Kernighan y Richie y Deitel y Deitel.
- Días de cursada.
- Uso de plataforma Miel.
- Explicar que necesitan tener instalado para la próxima clase y que leer.
- Breve repaso de conceptos adquiridos en el curso anterior de Programación, e introducción de nuevos conceptos.
- La función main, funciones e identificadores, palabras reservadas. Tipos de datos y tipos de variables propios del Lenguaje C. Constantes y constantes literales. Macroreemplazos. Tipos de datos, typedef y struct. Tipos de datos enum. Uniones. Operadores, precedencia de operadores. Control de flujo de ejecución : if, switch, while, for y do while, break; continue y goto. Punteros a una variable, a un array, arrays de punteros, argumentos de main, argumentos de funciones, por valor, por puntero, punteros a estructuras, punteros a arrays de estructuras.

¿ Qué es el Code::blocks?

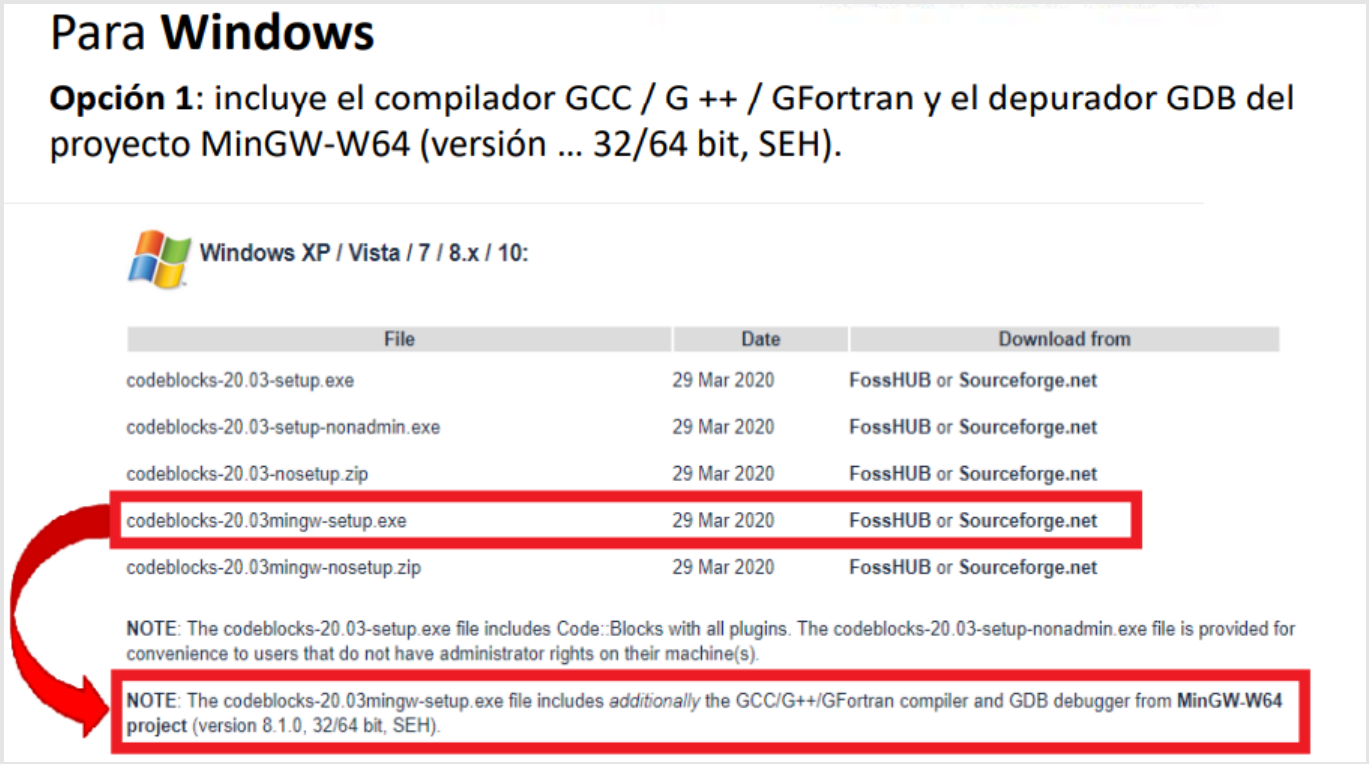
- **Es un entorno de desarrollo integrado (IDE, por sus siglas en inglés)** . Un entorno de desarrollo integrado (IDE) es una aplicación de software que ayuda a los programadores a desarrollar código de software de manera eficiente. Aumenta la productividad de los desarrolladores al combinar capacidades como editar, crear, probar y empaquetar software en una aplicación fácil de usar. Así como los escritores utilizan editores de texto y los contables, hojas de cálculo, los desarrolladores de software utilizan IDE para facilitar su trabajo.
- **De código abierto y multiplataforma** diseñado principalmente para la programación en C, C++ y Fortran. Nosotros en esta materia solo vamos a programar en lenguaje C.
- Proporciona herramientas y características para facilitar el proceso de desarrollo de software al ofrecer un conjunto de herramientas en una interfaz gráfica unificada.
- Soporta la utilización de múltiples compiladores . Nosotros utilizaremos el

GCC.

Características de **Code::Blocks**:

1. **Editor de Código:** Ofrece un editor de código con resaltado de sintaxis y otras características útiles para escribir y editar código fuente.
2. **Compilación y Depuración:** Code::Blocks integra un sistema de compilación y depuración que permite compilar y ejecutar programas directamente desde la interfaz. Además, proporciona herramientas de depuración para rastrear y solucionar errores en el código.
3. **Gestión de Proyectos:** Permite crear, administrar y trabajar con proyectos. Los proyectos pueden contener múltiples archivos de código fuente y configuraciones de compilación.
4. **Extensible:** Se puede ampliar mediante la instalación de complementos y extensiones que agregan funcionalidades adicionales, como soporte para otros lenguajes de programación, herramientas de análisis estático y más.
5. **Multiplataforma:** Code::Blocks es compatible con varias plataformas, lo que significa que puedes utilizarlo en sistemas operativos como Windows, macOS y Linux.
6. **Interfaz Personalizable:** Puedes personalizar la interfaz gráfica de Code::Blocks según tus preferencias, lo que incluye la disposición de ventanas y la configuración de colores y temas.
7. **Soporte para Diversos Compiladores:** Puede ser configurado para trabajar con diferentes compiladores, como GCC (GNU Compiler Collection) y otros, lo que brinda flexibilidad en la elección de herramientas de desarrollo.

Para instalarlo, podemos entrar a la pagina de codeblocks y elegir la opción que tiene **Mingw**



En esta materia siempre vamos a crear proyectos. La creación de proyectos es una buena práctica y tiene varias ventajas como las que se citan a continuación:

1. **Organización del Código:** Un proyecto proporciona una estructura organizada para tus archivos de código fuente. Se puede agrupar archivos relacionados en carpetas y organizarlos de manera lógica, lo que facilita la gestión y el mantenimiento del código.
2. **Gestión de Dependencias:** Los proyectos permiten gestionar las dependencias de tu código. Se pueden incluir bibliotecas, módulos o archivos externos necesarios para la aplicación y asegurar de que estén disponibles y se integren correctamente.
3. **Compilación Eficiente:** Al trabajar en un proyecto, se puede especificar configuraciones de compilación específicas. Esto incluye opciones de compilación, rutas de inclusión y otros ajustes que son esenciales para generar ejecutables de manera eficiente y sin errores.
4. **Facilita la Colaboración:** Si uno trabaja en equipo, un proyecto es esencial para facilitar la colaboración. Un proyecto bien estructurado y documentado permite a otros entender rápidamente la arquitectura y la lógica de tu código.
5. **Gestión de Recursos:** Un proyecto puede incluir recursos como imágenes, archivos de configuración, archivos de datos, etc. La gestión de estos recursos dentro del proyecto simplifica su manipulación y distribución junto con la aplicación.
6. **Configuración de Entorno de Desarrollo:** Los proyectos suelen incluir

información sobre el entorno de desarrollo, como configuraciones de compilación, configuraciones de depuración y otros ajustes específicos del proyecto. Esto ayuda a mantener coherencia entre los diferentes miembros del equipo.

7. **Depuración Efectiva:** Al trabajar en un proyecto, se puede configurar fácilmente puntos de interrupción y realizar operaciones de depuración de manera más eficiente. El IDE puede comprender la estructura del proyecto y proporcionar información relevante durante la depuración.
8. **Pruebas Unitarias y Automatización:** Los proyectos permiten integrar fácilmente pruebas unitarias y herramientas de automatización en el flujo de trabajo. Esto es esencial para mantener la calidad del código y facilitar la detección temprana de errores.
9. **Escalabilidad:** Un proyecto bien diseñado es escalable. Se pueden agregar nuevas características, módulos o componentes sin perder la visión general del proyecto. Esto es esencial a medida que el código crece en complejidad.
10. **Mantenimiento a Largo Plazo:** Un proyecto bien estructurado facilita el mantenimiento a largo plazo.

En resumen, crear un proyecto cuando programas proporciona una estructura organizativa, facilita la colaboración, mejora la gestión de dependencias y recursos, y contribuye a un desarrollo más eficiente y mantenible a largo plazo. Es una práctica que ayuda a mantener la calidad del código y a facilitar el trabajo tanto en proyectos pequeños como en grandes.

Puntualmente en el Codeblocks, si no creo un proyecto **no puedo** usar el debugger. Para crear un proyecto en este IDE, vamos a generar 3 archivos:

cabeceras.h : va a contener todas las bibliotecas,estructuras de datos y prototipos de las funciones necesarias para su funcionamiento.

funciones.c : va a contener el desarrollo de todas las funciones que está implementando o sea las declaradas en el archivo «**.h**» correspondiente.

main.c : va a contener todo nuestro programa.

Cuando hago #include con <…> indica que los archivos se encuentran en la ruta de instalación del compilador.

Mientras que cuando hago #include con **comillas dobles** indica que los archivos se encuentran en la ruta del proyecto actual o la ruta que se especifique. Se aconseja que para evitar problemas se eviten los espaciados.

Variables: Son espacios reservados de memoria, pueden variar su contenido a lo largo de la ejecución de un programa. Posee un nombre que sirve para identificarla y tiene asociado un tipo de dato que sirve para interpretar su contenido.

Constante: Es un valor que no puede ser alterado durante la ejecución de un programa. Corresponde a una longitud fija de un área reservada de la memoria principal del ordenador, donde el programa almacena valores fijos. Por ejemplo:

```
#include <stdio.h>
#define PI 3.1415926
int main ()
{
printf (“Pi vale %d”,PI);
return 0;
}
```

En el ejemplo anterior, escribimos **int main()** ya que el programa principal devuelve un entero (0 si terminó correctamente y distinto de 0 si no pudo terminar correctamente).

Literales: Tipo de constante, son las más usadas, toman valores tales como 4 5 -3 2 5 6 4. Las constantes definidas son identificadores que se asocian con valores literales constantes y que toman determinados nombre. Sus valores se almacenan en la memoria, pero no se pueden modificar.

Operando: Un operando es un término que se refiere a los valores con los que se realiza una operación matemática o lógica. En otras palabras, los operandos son los números, variables o constantes sobre los cuales se aplica un operador para realizar una operación. Por ejemplo, en la expresión matemática «3 + 5», los números «3» y «5» son los operandos de la operación de suma.

Operador: Un operador es un símbolo o un término que indica qué tipo de operación se debe realizar entre uno o más operandos. Los operadores pueden ser símbolos matemáticos, como + (suma), – (resta), * (multiplicación), / (división), etc., o símbolos lógicos, como && (AND lógico), || (OR lógico), ! (NOT lógico), etc. Los operadores determinan cómo se combinan o manipulan los operandos para obtener un resultado. En el ejemplo anterior, el símbolo «+» es el operador que se utiliza para sumar los operandos «3» y «5».

Tipos de Datos

Int: Su tamaño depende de la arquitectura y del compilador. Representa números enteros con signo.

Los int de 2 bytes están comprendidos entre -32768 y +32767, son comunes, como lo son los de 4 bytes que están comprendidos entre -2147483648 y +2147483647.

Los de 2 bytes se corresponden a una arquitectura de 16 bits.

Los de 4 bytes se corresponden a una arquitectura de 32 bits.

- **Long**: entero largo, pesa 4 bytes. **(es un modificador)**
- **Long Long**: entero largo, pesa 8 bytes. **(es un modificador)**
- **Short**: entero corto, pesa 2 bytes. **(es un modificador)**

Por que existe e long si pesa lo mismo que el int? xq antes había distintas arquitecturas, ahora el compilador es de 32bits por ende es lo mismo escribir long que int.

Unsigned : Permite trabajar únicamente con valores positivos. **(es un modificador)**

Char: También son de tipo entero, su tamaño es de 1 byte y además es interpretado como carácter (Código ascii) -0 al 255-

Float: Guarda números reales con precisión de 6 dígitos decimales, su tamaño es de 4 bytes.

- **Double**: Es un valor real de doble precisión, igual que float pero con mayor precisión (hasta 15 dígitos decimales), pesa 8 bytes. (es un modificador)
- **Long double**: Mucho mayor en decimales incluso mayor que double , pesa 12 bytes. (es un modificador)

Void: Significa nulo y vacío cuando retorna una función, también puede ser un genérico, cualquier tipo de dato.

typedef: me permite definir mis propios tipos de datos. Su función es asignar un nombre alternativo a tipos existentes, a menudo cuando su declaración normal es aparatosa, potencialmente confusa o probablemente variable de una implementación a otra.

Ejemplo:

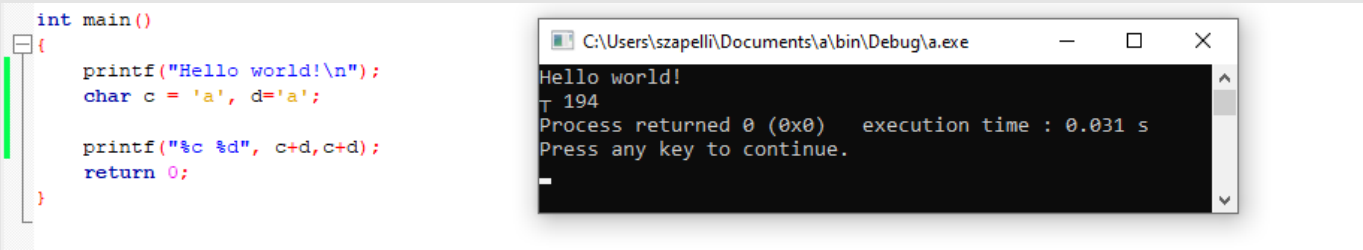
```
typedef int entero;
int main()
{

entero a;
a=4;
printf("%d", a);
return 0;
}
```

typedef no es lo mismo que **struct**. **Struct** solo crea estrcturas que pueden estar formadas por elementos de distintos tipos. Tienen un determinado orden y define un conjunto de elementos.

Ahora dijimos que **char es un entero** porque podemos almacenar un valor entero comprendido entre -128 y 127.

Los valores enteros positivos de las variables de tipo char están relacionadas con la tabla de caracteres ASCII.

Por ejemplo:



La suma dio 194 porque el ASCII del carácter a es 97

enum: se utiliza para definir un nuevo tipo de dato que puede almacenar un conjunto de constantes enteras con nombres simbólicos. Esto proporciona una manera conveniente de asignar nombres legibles a valores constantes. Ejemplo:

```
// Definición de enum
enum dias_semana {
LUNES,
MARTES,
MIERCOLES,
JUEVES,
VIERNES,
SABADO,
```

```
DOMINGO
};

int main() {
// Declaración de una variable de tipo enum
enum dias_semana hoy;

// Asignación de un valor del enum a la variable
hoy = MARTES;

// Acceso al valor de la variable enum
printf("Hoy es %d\n", hoy); // Imprimirá: Hoy es 1

return 0;
}
```

En este ejemplo, se ha definido un tipo de datos enum llamado dias_semana, que contiene siete constantes enteras. Estas constantes tienen valores predeterminados que van desde 0 hasta 6, a menos que se especifique explícitamente. En el ejemplo, MARTES tiene el valor 1.

Operadores

En el lenguaje de programación C, los operadores son símbolos especiales que se utilizan para realizar diversas operaciones en variables y valores. Los operadores en C se dividen en varias categorías según sus funciones y prioridades.

1. **Operadores Aritméticos:**
    - + (suma)
    - - (resta)
    - * (multiplicación)
    - / (división)
    - % (módulo o resto)

2. **Operadores de Asignación:**
    - = (asignación simple)
    - += (suma y asignación)
    - -= (resta y asignación)

    - *= (multiplicación y asignación)
    - /= (división y asignación)
    - %= (módulo y asignación)
    - …

3. **Operadores de Comparación:**
    - == (igual a)
    - != (no igual a)
    - < (menor que)
    - > (mayor que)
    - <= (menor o igual que)
    - >= (mayor o igual que)

4. **Operadores Lógicos:**
    - && (AND lógico)
    - || (OR lógico)
    - ! (NOT lógico)

5. **Operadores de Incremento y Decremento:**
    - ++ (incremento)
    - -- (decremento)

6. **Operadores Bit a Bit:**
    - & (AND a nivel de bits)
    - | (OR a nivel de bits)
    - ^ (XOR a nivel de bits)
    - ~ (complemento a nivel de bits)
    - << (desplazamiento a la izquierda)
    - >> (desplazamiento a la derecha)

7. **Operadores de Punteros:**
    - * (operador de indirección de puntero)
    - & (operador de dirección de puntero)

8. **Operadores de Ternario:**
    - condición ? expresión_verdadera : expresión_falsa

9. **Operadores de Miembro y Puntero a Miembro (para estructuras y clases):**
    - . (operador de miembro)
    - -> (operador de puntero a miembro)

Estos son solo algunos ejemplos de los operadores disponibles en C. Cada operador tiene una prioridad y asociatividad específicas que determinan el orden en que se evalúan las expresiones. Es importante tener en cuenta estas prioridades al escribir expresiones complejas para asegurarse de que se evalúen de la manera deseada.

Especificadores de Formato

Los especificadores de formato son códigos que se utilizan en las funciones de entrada y salida en C (como printf y scanf) para indicar cómo se deben interpretar y mostrar los datos. Estos especificadores permiten controlar el formato en el que se presentan los valores numéricos, de caracteres y otros tipos de datos. Aquí hay una lista de algunos de los especificadores de formato más comunes en C:

1. **Enteros:**
    - %d: Entero con signo en base decimal.
    - %u: Entero sin signo en base decimal.
    - %o: Entero en base octal.
    - %x o %X: Entero en base hexadecimal (minúsculas o mayúsculas).

2. **Punto flotante:**
    - %f: Número de punto flotante en notación decimal.
    - %e o %E: Número de punto flotante en notación científica (exponencial).
    - %g o %G: Usado para formatear en %f o %e según sea más apropiado.
    - %lf: para mostrar double.

3. **Caracteres y cadenas:**
    - %c: Carácter.
    - %s: Cadena de caracteres (array de caracteres).

4. **Punteros:**
    - %p: Puntero (dirección de memoria en hexadecimal).

5. **Valores de caracteres:**
    - %x o %X: Valor numérico de un carácter en base hexadecimal.

6. **Anchura y precisión:**
    - %n: Almacena el número de caracteres escritos hasta este punto.
    - %m.n: Especifica la anchura mínima y la precisión para números de punto flotante y cadenas.

7. **Otros:**
    - %%: Muestra un signo de porcentaje literal (%).

Prioridad de los Operadores

En C, los operadores tienen una prioridad o precedencia que determina el orden en el que se evalúan las expresiones. Cuando tienes una expresión con múltiples operadores, los operadores con mayor prioridad se evalúan antes que los operadores con menor prioridad. Si dos operadores tienen la misma prioridad, la asociatividad decide el orden en que se evalúan. Aquí tienes una lista de operadores comunes en C, ordenados de mayor a menor prioridad:

1. **Operadores de Paréntesis:**
    - (): Paréntesis que agrupan expresiones.

2. **Operadores de Puntero:**
    - *: Operador de indirección de puntero.
    - &: Operador de dirección de puntero.

3. **Operadores de Incremento/Decremento:**
    - ++ (prefijo y sufijo)
    - -- (prefijo y sufijo)

4. **Operadores Unarios:**
    - + (positivo)
    - - (negativo)
    - ! (negación lógica)
    - ~ (complemento a nivel de bits)

5. **Operadores de Cast (Conversión de Tipo):**
    - (tipo) (casting)

6. **Operadores de Multiplicación/División/Resto:**
    - * (multiplicación)
    - / (división)
    - % (resto o módulo)

7. **Operadores de Suma/Resta:**
    - + (suma)
    - - (resta)

8. **Operadores de Desplazamiento:**
    - << (izquierda)
    - >> (derecha)

9. **Operadores Relacionales:**

    - < (menor que)
    - > (mayor que)
    - <= (menor o igual que)
    - >= (mayor o igual que)

10. **Operadores de Igualdad:**
    - == (igual a)
    - != (no igual a)

11. **Operadores Bit a Bit:**
    - & (AND a nivel de bits)
    - ^ (XOR a nivel de bits)
    - | (OR a nivel de bits)

12. **Operadores Lógicos:**
    - && (AND lógico)
    - || (OR lógico)

13. **Operador Ternario:**
    - condición ? expresión_verdadera : expresión_falsa

14. **Operador de Asignación:**
    - = (asignación simple)
    - +=, -=, *=, /=, %= y otros (asignaciones compuestas)

La comprensión de la prioridad y la asociatividad de los operadores es esencial para escribir expresiones correctas y evitar resultados inesperados. Si es necesario, puedes usar paréntesis para controlar explícitamente el orden de evaluación en una expresión.

Ejercicio:

main.c

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

int main()
{
    char c;
    int d;

    scanf("%c",&c);
    scanf("%d",&d);

    printf("%c", c+d);
    return 0;
}
```

```
C:\Users\szapelli\Documents\a\bin\Debug\a.exe
a
5
f
Process returned 0 (0x0)   execution time : 10.489 s
Press any key to continue.
```

En este ejemplo, me sumo 5 al carácter a, por eso me muestra f.

Ahora si yo quiero sumar un número + un carácter (en formato número).

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

int main()
{
    char c;
    int d;

    scanf("%c",&c);
    scanf("%d",&d);

    printf("%d", c+d);
    return 0;
}
```

```
C:\Users\szapelli\Documents\a\bin\Debug\a.exe
1
3
52
Process returned 0 (0x0)   execution time : 1.748 s
Press any key to continue.
```

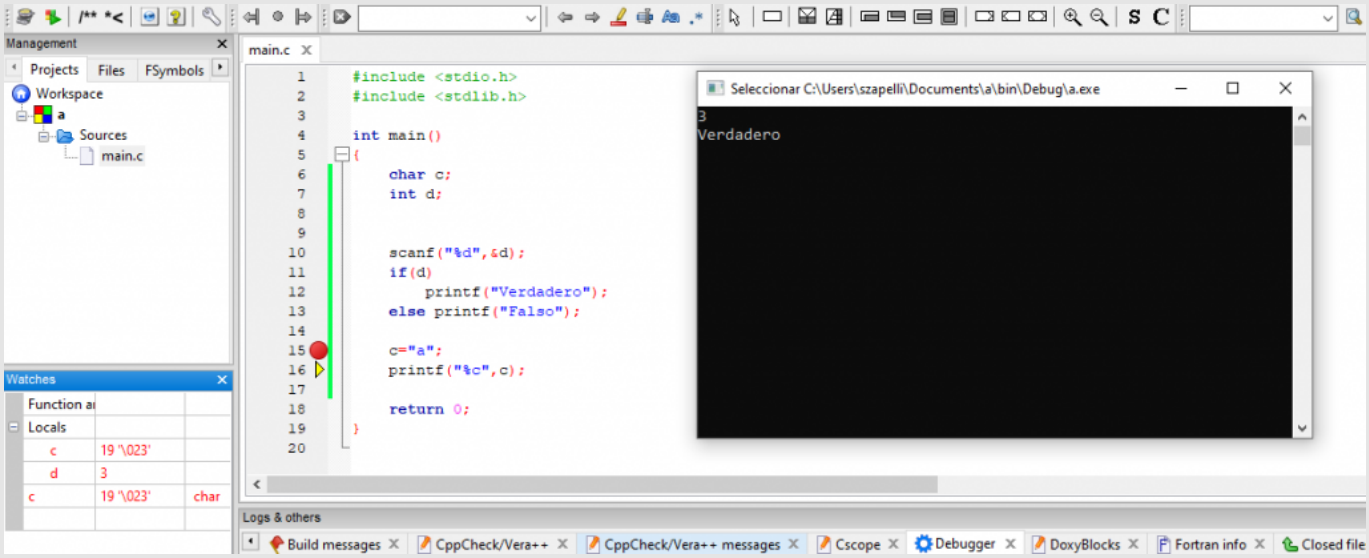
Va a convertir el 1 en ascii y le va a sumar 3. Para que yo pueda sumar efectivamente el 1 y el 3 al 1 deberia restarle -‘0’. Se le resta el caracter ‘0’ porque el ‘3’ esta en la posicion 51 del codigo ascii y el ‘0’ en la posición 48. Lo estaria «casteando» a entero.

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

int main()
{
    char c;
    int d;

    scanf("%c",&c);
    scanf("%d",&d);

    printf("%d", c-'0'+d);
    return 0;
}
```

```
C:\Users\szapelli\Documents\a\bin\Debug\a.exe
1
3
4
Process returned 0 (0x0)   execution time : 1.868 s
Press any key to continue.
```

Ahora, intentemos asignarle a c = «a» y vemos como funciona el debug y los

errores.



Ejemplo División de Enteros

Se tienen 3 variables enteras y 3 flotantes. En R y Z1 y Z2 se van a guardar los resultados.

```
int r, x2=2, y2=5;
float z1, z2 , z3=5;

z1 = y2/x2;
r = y2/x2;

printf("%f %d\n", z1,r);
```

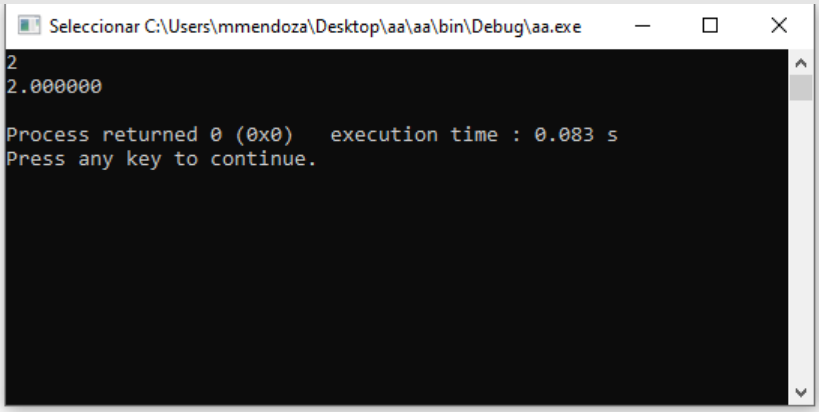
Cuando intentamos ver el resultado que se guarda en z1 y r , muestra que en z1 guarda el valor pero solo la parte entera y la parte flotante guarda 0 y en r guarda solo la parte entera. La solución a esto es lo que se llama **casteo**.

El casting o simplemente casteo **nos permite hacer una conversión explícita de un tipo de dato a otro, a criterio del programador siempre y cuando estos tipos sean compatibles**.

```
z2 =  (float)y2/x2;
printf("%f\n", z2);
z2 =  y2/(float)x2;
printf("%f\n", z2);
z2 =  (float)y2/(float)x2;
printf("%f\n", z2);
```

Ahora, yo podria hacer lo siguiente:

```c
int r, x2=2, y2=5;
float z1, z2 , z3=5;



r = (float)(y2/x2);
printf("%d\n", r);


z3 = r;
printf("%f\n", z3);

return 0;
```



Ejercicio para hacer en clase: Pedir un numero por teclado y calcular el factorial del mismo.

Recordar que el factorial era:

N! -> es 1 si N=0

-> N* (N-1)! si n>0