

TSSI

# Programación II

---

Ing. Pinkas Leonardo

Ing. Mostovoi Alejandro

Año 2019

## Índice

<b>Tipo Abstracto de Datos (TAD)</b>	<b>4</b>
El rol de la abstracción	4
Tipo abstracto de dato	4
Ventajas de los TAD	5
Especificación de un TAD	6
Especificación informal	6
Implementación de un TAD en C	7
Archivos de cabecera (parte pública)	7
Implementación de funciones (parte privada)	8
Aplicación de un TAD	8
<b>Bibliotecas</b>	<b>9</b>
¿Qué es una biblioteca?	9
Tipos, Similitudes y Diferencias	9
Bibliotecas estáticas	9
Bibliotecas dinámicas	9
Creación de bibliotecas con CodeBlocks	10
Creación	10
Utilización	11
<b>Punteros</b>	<b>14</b>
Declaración e Inicialización	14
Indirección de punteros	15
Punteros y Vectores	16
Punteros constantes y Punteros a constantes	17
Aritmética de punteros	19
<b>Cadenas</b>	<b>20</b>
Declaración e inicialización de variables de cadena	20
Biblioteca string.h	21
<b>Depuración</b>	<b>25</b>
Bug	25
Falla	25
Depuradores	25
<b>Errores fáciles de corregir</b>	<b>25</b>
<b>Errores difíciles de corregir</b>	<b>27</b>
<b>Tipos de Errores</b>	<b>28</b>
Errores Sintácticos	28

Errores Lógicos	28
Errores de Regresión	28
<b>Depurando con CodeBlocks</b>	<b>29</b>
Configurar CodeBlocks para depurar	29
Compilar para depurar	30
Establecer un Punto de Interrupción (breakpoint)	30
Observar el valor de una variable	30
Ver la pila de ejecución	31
<b>Bibliografía</b>	<b>32</b>

# Unidad Temática N°1

~ TAD ~

"Si ha elegido las estructuras de datos correctas y las cosas están bien organizadas, los algoritmos casi siempre serán evidentes."

~ Rob Pike

## Tipo Abstracto de Datos (TAD)

### El rol de la abstracción

A lo largo de la historia, las personas han tenido que tratar con problemas complejos, problemas que involucran muchas variables, escenarios posibles, etc. Tratar de encontrar una solución a esos problemas prestando atención a cada detalle, realmente es una tarea compleja es por eso que surge la *abstracción* como la “herramienta” utilizada para tratar este tipo de problemas. La abstracción resulta ser la herramienta más eficaz para lidiar con problemas complejos ya que nos permite concentrarnos en los aspectos fundamentales del problema a resolver, dejando de lado los detalles no esenciales.

<b>Definición</b>	La abstracción es la capacidad para encapsular y aislar la información, de diseño y ejecución
-------------------	---

Desde el surgimiento del software y los lenguajes de programación, podemos ver que la abstracción se repite una y otra vez hasta nuestros días.

Inicialmente la programación se realizaba a nivel de hardware, luego se comenzó a programar microprocesadores con instrucciones de *secuencia* y *salto*. Posteriormente, con el advenimiento de los lenguajes de alto nivel surgieron instrucciones que permiten controlar el flujo de ejecución con estructuras de control y repetición (if, while, for, etc), a esto se lo llama **abstracción de control** y también surgieron los tipos de datos, que permitió crear variables de un tipo de dato particular, este cambio permitió dar los primeros pasos en la **abstracción de datos**.

Un refinamiento posterior dentro de la abstracción de control es la **abstracción procedimental**. Esta técnica permite diseñar software de forma modular. Se basa en la utilización de procedimientos o funciones sin preocuparse de cómo están implementadas, es decir, sólo conociendo qué hacen, qué entrada requieren y qué salida tienen.

### Tipo abstracto de dato

Con todo lo mencionado anteriormente, los programadores comenzaron a darse cuenta de que para crear software de calidad y confiable, la solución era modularizar el código.

Pero...¿Qué es un módulo?

<b>Definición</b>	Modularizar: es una técnica de construcción de software que permite separar los datos y procedimientos en una parte privada - sólo accesible dentro del módulo - y una parte pública - accesible por fuera del módulo -.
-------------------	--

Entonces un módulo es una pieza de software diseñada con esta separación de datos y procedimientos públicos y privados presente desde el primer momento.

### ¿Y un tipo abstracto de dato?

Debemos recordar que un tipo de dato, en general, queda definido por un conjunto los valores (dominio) y las operaciones definidas sobre dichos valores.

Ahora bien, un tipo de dato abstracto o TAD es entonces un tipo de dato definido por el programador, que se puede utilizar de forma similar a los tipos de datos definidos por el sistema.

Por ejemplo, si definimos el tipo de dato *vectorEntero*, sus posibles valores serían números enteros y las operaciones podrían ser *cargarVector*, *ordenarVector*, *buscarEnVector*, etc.

La modularización se utiliza frecuentemente como una técnica para implementar los TAD ya que para poder construir un TAD, se debe poder:

- Exponer una definición del tipo (formal o informal).
- Hacer disponible un conjunto de operaciones que permitan operar sobre los datos.
- Proteger los datos de forma tal que sólo se pueda interactuar con ellos a través de las operaciones definidas.
- Poder generar múltiples instancias del tipo.

### Ventajas de los TAD

La utilización de TAD para el desarrollo de software nos proporciona varias ventajas (respecto de no hacerlo), mencionamos algunas:

1. Mejora la conceptualización, comprensión y modelización del mundo real. Ayuda a clarificar los objetos (del mundo real) que intervienen en el problema y sus comportamientos.
2. Mejora la robustez del sistema. Nos permite realizar una verificación de tipos y evitar errores en tiempo de ejecución.
3. Separa la implementación de la especificación. Esto permite modificar la implementación (parte privada) sin cambiar la interfaz (parte pública) del tipo de dato.

4. Permite la extensibilidad del sistema. La utilización (y reutilización) de módulos permiten crear y mantener software con mayor facilidad
5. Agrupan y localizan las operaciones y la representación de atributos mejorando la semántica del tipo.

Un programa que utiliza un TAD, lo hace teniendo en cuenta las operaciones que tiene sin interesarse en la representación física de los datos ni en la implementación de las operaciones, es decir, utilizan el TAD a partir de su interfaz. Esto permite que se modifique la implementación sin alterar el funcionamiento de los programas que lo utilizan.

## Especificación de un TAD

Como mencionamos previamente, un TAD está formado por un conjunto de datos y operaciones. La especificación de un TAD consta de 2 partes, la descripción del conjunto de datos y las operaciones definidas.

El objetivo es poder describir el comportamiento del TAD.

La especificación del TAD puede tener un enfoque informal, donde se describe tanto los datos como las operaciones con un lenguaje natural o bien puede tener un enfoque formal donde se debe suministrar un conjunto de axiomas que describen las operaciones sintáctica y semánticamente.

En este curso sólo veremos la especificación informal.

### Especificación informal

El formato que generalmente se utiliza es especificar el nombre del TAD y los datos:

TAD <nombre del tipo> (<valores y su descripción>)

y a continuación cada una de las operaciones:

<nombreOperación>(<lista\_de\_argumentos>). <Descripción funcional>.

### Ejemplo: TAD “Conjunto”

TAD Conjunto (colección de elementos sin duplicados, que pueden estar en cualquier orden, utilizado para representar los conjuntos matemáticos y sus operaciones).

**Operaciones:**

- ❖ conjuntoVacio. Crea un conjunto sin elementos.
- ❖ añadir(conjunto, elemento). Comprueba si el elemento pertenece al conjunto. En caso negativo lo añade.
- ❖ eliminar(conjunto, elemento). Comprueba si el elemento pertenece al conjunto. En caso afirmativo lo elimina.
- ❖ pertenece(conjunto, elemento). Comprueba si el elementos pertenece al conjunto. En caso afirmativo devuelve Verdadero.
- ❖ cardinal(conjunto). Devuelve la cantidad de elementos que posee el conjunto.

**Implementación de un TAD en C**

Las características del lenguaje que nos permiten implementar un TAD son: las estructuras, las funciones y los archivos de cabecera (.h).

Una estructura es una agrupación de campos (variables) de cualquier tipo predefinido, usuario o sistema, que guardan cierta relación lógica entre sí. Las estructuras nos permiten tratar a los campos que la componen como una sola unidad, como un todo.

Por ejemplo, si quisiéramos definir el TAD “Punto”, los datos a permiten representar un punto son sus coordenadas X e Y, Para esto podemos definir una estructura de la siguiente manera:

```
typedef struct {
    float x;
    float y;
} punto2D;

// definimos una variable de tipo punto2D
punto2D miPunto;
```

**Archivos de cabecera (parte pública)**

Los archivos de cabecera representan la parte pública de un TAD, su interfaz. Debemos pensar y definir correctamente las estructuras y operaciones antes de sentarnos a codificar ya que una vez que nuestro TAD esté en uso será muy difícil cambiar las interfaces.

Estos archivos los utilizaremos para agrupar declaraciones de estructuras de datos comunes y prototipos de funciones.

**Definición**

Prototipo de función: nos especifica el nombre de la función, los tipos de datos que recibe por parámetro y el tipo de dato que devuelve. Ejemplo:



	punto2D crearPunto(int x, int y)
--	----------------------------------

Estos archivos deben ser incluidos en los archivos que contengan la implementación de las funciones y también en los archivos de código que hagan referencia a algún elemento aquí declarado (.c o .cpp).

### Implementación de funciones (parte privada)

La implementación de funciones la realizaremos en un archivo con extensión .c o .cpp. Estos archivos representan la parte privada del TAD. En estos archivos se debe implementar cada una de las funciones declaradas en el archivo de cabecera definido previamente.

Por convención el archivo de cabecera y el archivo donde se realiza la implementación de las funciones deben tener el mismo nombre. Ejemplo:

- punto2D.h → archivo de cabecera del TAD Punto
- punto2D.cpp → archivo de implementación de funciones del TAD Punto

### Aplicación de un TAD

A modo de resumen, veremos un ejemplo simplificado de la distribución de código y archivos en la implementación y utilización de un TAD. Tomaremos como ejemplo el TAD punto.

punto2D.h	punto2D.cpp	main.cpp
<pre>typedef struct {     float x;     float y; } punto2D;  punto2D crearPunto(int, int);  float distancia(punto2D, punto2D);</pre>	<pre>#include "punto2D.h"  punto2D crearPunto(int X, int Y){     punto2D punto;     punto.x = x;     punto.y = y;      return punto; }  float distancia(punto2D p1, punto2D p2){     float d = sqrt( pow(p2.x-p1.x, 2) + pow(p2.y-p1.y, 2) );      return d; }</pre>	<pre>#include "punto2D.h"  punto2D p1; punto2D p2;  p1 = crearPunto(1,2); p2 = crearPunto(1,2);  cout &lt;&lt; distancia(p1, p2) &lt;&lt; endl;</pre>

## Bibliotecas

### ¿Qué es una biblioteca?

Es un conjunto de funcionalidades que ofrece la posibilidad de ser utilizadas mediante la definición de una interfaz bien definida.

En términos más técnicos, podemos decir que es la implementación de un conjunto de funciones, por lo general agrupadas de forma lógica en un archivo según su funcionalidad, que permiten ser utilizadas desde otras bibliotecas o programas.

Como dijimos anteriormente, una biblioteca es un conjunto de funciones, por lo tanto no tiene un punto de entrada como si lo tiene un programa.

### Tipos, Similitudes y Diferencias

Existen dos tipos de bibliotecas, las bibliotecas estáticas y las bibliotecas dinámicas.

#### Bibliotecas estáticas

Las bibliotecas estáticas deben su nombre a que las referencias desde un programa a las funciones que posee la biblioteca se resuelven en tiempo de compilación. Esto significa que al momento de compilar nuestro programa, se realiza el enlace a todas las funciones de nuestra biblioteca, y todo queda incluido en el archivo ejecutable que se obtiene como resultado del proceso de compilación.

Las ventajas de realizar un enlace estático es que facilita la distribución de nuestro archivo ejecutable ya que el mismo posee todos los símbolos y funciones necesarios para ser ejecutado.

Las desventajas es que el proceso de compilación puede durar más tiempo que con bibliotecas dinámicas. Otra desventaja es que un cambio en cualquier biblioteca, fuerza la recompilación de todo el ejecutable.

#### Bibliotecas dinámicas

Las bibliotecas dinámicas, deben su nombre a que las referencias desde un programa a las funciones que posee la biblioteca se resuelven en tiempo de ejecución. Esto significa que al momento de ejecutar nuestro programa, se realizará la carga en memoria (en caso de que no estuviera) de la biblioteca requerida.

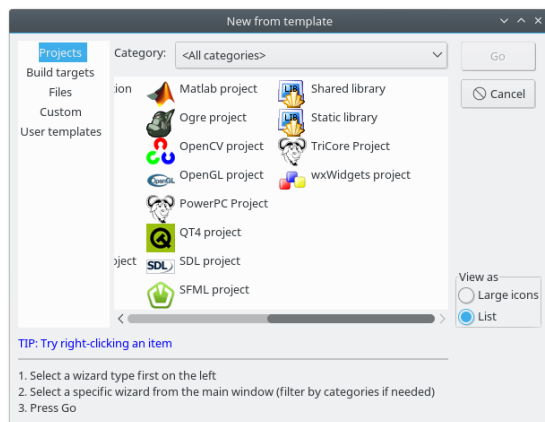
Las ventajas de realizar un enlace dinámico es que se reduce el tiempo de compilación de programas grandes, debido a que lleva menos tiempo compilar muchos fuentes pequeños que uno grande. Por otra parte un cambio en una biblioteca no fuerza la recompilación de todo el programa (salvo que esa modificación cambie las interfaces). Otra ventaja es que se reduce el espacio utilizado en disco, debido a que puedo enlazar varios programas con la misma biblioteca, no es necesario copiarla 2 veces.

Como desventajas de este tipo de enlace, podemos mencionar que se pueden producir problemas de dependencias, en caso de no distribuir con nuestro programa las bibliotecas requeridas y que se requiere un tiempo extra para cargar las bibliotecas la primera vez que son utilizadas por el programa.

## Creación de bibliotecas con CodeBlocks

### Creación

1. Crear nuevo proyecto
  1. Seleccionar la opción Shared Library



2. Crear archivo de encabezados (extensión .h) para incluir la declaración de constantes, tipos y funciones.
3. Crear archivo para implementar las funciones declaradas en el archivo de encabezados. Este archivo debe incluir el archivo de encabezados declarado anteriormente.
4. Compilar el proyecto.

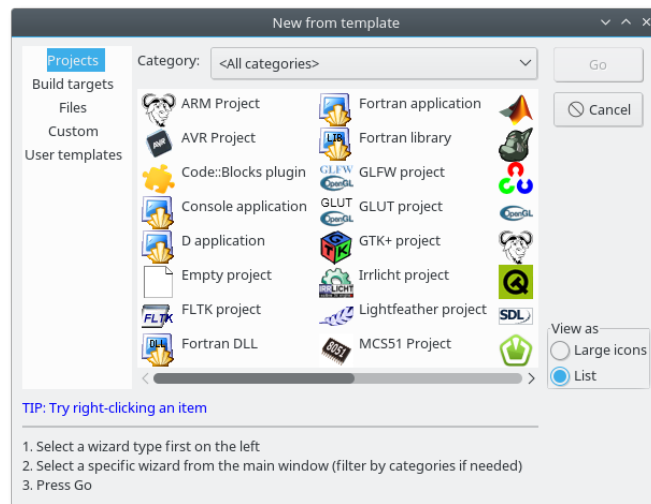
Luego de compilar el proyecto vemos que en el directorio *bin/Release* se creó un archivo con el nombre *lib<nombre del proyecto>.so*.

Esa es nuestra biblioteca.

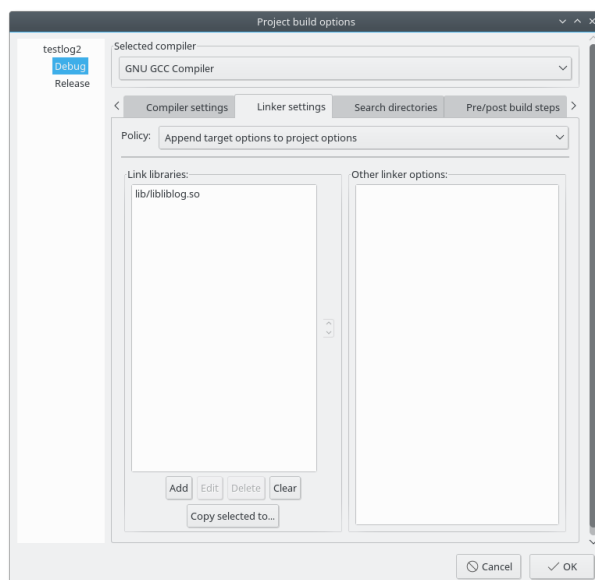
## Utilización

Para poder utilizar las bibliotecas que creamos, debemos hacer lo siguiente:

1. Crear nuevo proyecto
  1. Seleccionar la opción Console application

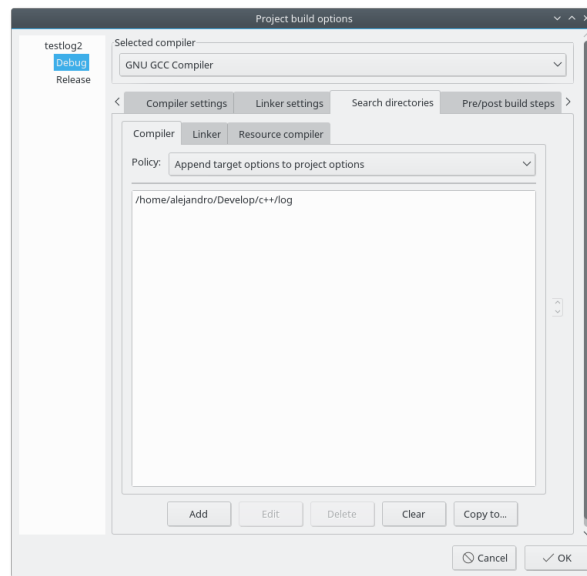


2. Establecer las opciones de compilación para el proyecto
  1. Para esto ingresamos en Project → Build Options → Linker Settings



2. Presionar el botón *add* y a continuación buscar el archivo \*.so que representa la biblioteca creada.

3. En la pestaña Search directories debemos indicarle al compilador en qué directorios se encuentran los archivos de encabezados a utilizar



3. En nuestro archivo main.cpp, debemos incluir el archivo de encabezados de la biblioteca a utilizar.
4. Presionar el botón add y buscar el directorio donde se encuentra el archivo de encabezados de la biblioteca a utilizar.
5. En nuestro archivo main.cpp, debemos incluir el archivo de encabezados de la biblioteca a utilizar.
6. Una vez terminada la codificación, compilar el proyecto

Luego de compilar el proyecto vemos que en el directorio *bin/Release* se creó un archivo con el nombre *<nombre del proyecto>*. Ese es nuestro ejecutable, que estará utilizando la biblioteca dinámica creada.

# Unidad Temática N°2

~ Punteros y Cadenas ~

"No hay nada en computación que no pueda ser roto por otro nivel de direccionamiento indirecto".

~ Rob Pike

## Punteros

Al ejecutarse nuestro programa, por cada variable declarada, se reserva un espacio de memoria para almacenar el contenido de la misma. A cada variable declarada se le asocian 3 atributos fundamentales que son: su nombre, su tipo y su dirección de memoria.

Cuando desde nuestro programa queremos acceder a alguna de las variables, lo hacemos a través de su nombre, pero la máquina accede directamente a través de su dirección de memoria.

Ejemplo:

<pre>#include &lt;stdio.h&gt;  int main(){     int n = 10;     printf("n = %d\n", n);     printf("Dir. de memoria de n = %p\n",     &amp;n);     return 0; }</pre>	<p>Ejecución</p> <p>n = 10; Dir. de memoria de n = 0xff123098</p>
--	---

Nosotros podemos hacer lo mismo mediante el uso de punteros.

Un puntero es una variable como cualquier otra que hayamos utilizado, sólo que sus valores serán direcciones de memoria de otras variables.

### Declaración e Inicialización

Para utilizar un puntero, al igual que las otras variables, debemos declararlo e inicializarlo.

Un puntero se declara de la siguiente manera:

<code>&lt;tipo de dato&gt; *&lt;nombre&gt;;</code>
--

Donde *<tipo de dato>* deberá corresponder con el tipo de dato de la variable apuntada, es decir, la variable cuya dirección de memoria vamos a contener en el puntero.

Para inicializar un puntero utilizaremos la constante NULL.

Ejemplos:

<pre>int *ptrInt = NULL; // puntero a entero, inicializado en NULL float *ptrFloat;    // puntero a float, sin inicializar</pre>
--

```
char *ptrCadena;    // puntero a char, sin inicializar
```

Previamente vimos un ejemplo donde se imprimía la dirección de memoria de una variable, y para esto usamos el signo & (ampersand) delante del nombre de la variable. El signo & (ampersand) es un operador que nos devuelve la dirección de memoria de una variable. Este operador podría ser utilizado para inicializar un puntero con la dirección de memoria de una variable o bien, para asignarle la dirección de memoria de una variable a un puntero en cualquier momento.

```
// Ejemplo 1
int n = 10;
int *ptrInt = &n; // puntero a entero, inicializado con la dirección de n

// Ejemplo 2
char letra = 'p';
char *ptr = &letra;

// Ejemplo 3
char palabra[] = "Hola";
char *ptrPalabra = palabra;
```

## Indirección de punteros

Una vez que definimos e inicializamos una variable de tipo puntero, el próximo paso es utilizarlo, ya sea para asignar un valor o bien para leer un valor desde una dirección de memoria. Para poder realizar estas acciones vamos a utilizar el operador \*.

### Ejemplo

<u>Asignación</u>	<u>Lectura</u>
<pre>int n = 0; int *ptr = &amp;n; // inicializa ptr con la dir. de n  *ptr = 20; printf("n = %d\n", n); // imprime 20</pre>	<pre>int n = 10; int * ptr = &amp;n;  printf("*ptr = %d\n", *ptr); // imprime 10;</pre>



Tener cuidado al realizar una asignación a un puntero, verificar que el puntero esté apuntando a alguna variable.

```
int *ptr;

*ptr = 20; // Esto es un error ya que ptr está apuntando a nada.
```

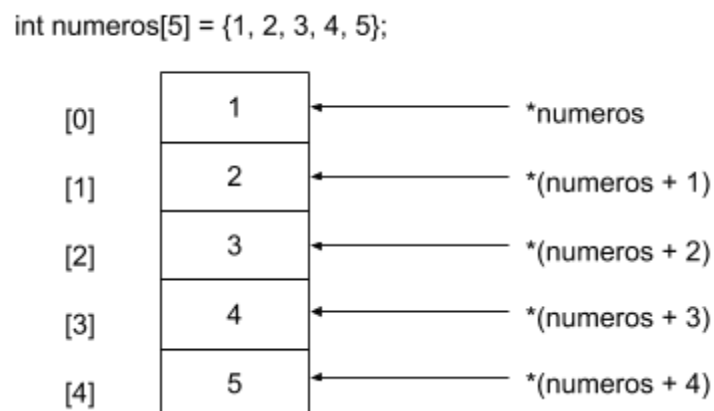
Resumiendo

Operador	Descripción
*	Definición de una variable puntero
&	Obtiene la dirección de memoria de una variable
*	Obtiene y también permite asignar el contenido de una variable puntero

## Punteros y Vectores

En C, los punteros y vectores (arrays) están fuertemente relacionados. Se pueden direccionar arrays como si fueran punteros y punteros como si fueran arrays.

El nombre de un array, es un puntero a la primer posición del mismo.



Es por esta relación que para visualizar, almacenar o asignar un valor a un elemento de un array se puede utilizar notación de subíndices o *notación de punteros*.

El nombre de un array es un puntero al primer elemento del array, es decir, contiene la dirección de memoria del primer elemento del array. Se dice que un array es un puntero constante ya que no se puede modificar la dirección a la que apunta, sólo se puede modificar su contenido.

```
#include <stdio.h>

int main(){
    int lista[] = {1,2,3};
    printf("%d\n", *lista);
    printf("%d\n", *lista+1);
    printf("%d\n", *lista+2);

    return 0;
}
```

Del ejemplo anterior sabemos que *lista* es un array de enteros, o lo que es lo mismo decir un puntero constante a enteros. Si quisiéramos hacer que *lista* apunte a otra dirección de memoria obtendríamos un error.

```
#include <stdio.h>

int main(){
    int lista[] = {1,2,3};
    int x = 10;
    lista = &x; // ESTO NO SE PUEDE HACER PORQUE lista ES UN PUNTERO CONSTANTE

    return 0;
}
```

## Punteros constantes y Punteros a constantes

### Definición

**Puntero constante:** es un puntero que no puede cambiar la dirección de memoria a la que apunta, pero sí puede cambiar el contenido de aquello a lo que apunta.

Para definir un puntero constante, lo realizamos de la siguiente manera:

```
<tipo de dato> *const <nombre>;
```

### Ejemplo

```
int x = 10;
int y = 20;
int *const p1 = &x;

*p1 = y; // esto es legal ya que cambiamos el contenido de p1.

p1 = &y; // esto no es legal ya que intenta cambiar el valor de p1.
```

#### Definición

**Puntero a constante:** es un puntero que puede cambiar la dirección de memoria a la que apunta, pero no puede cambiar el contenido de aquello a lo que apunta.

Para definir un puntero a constante, lo realizamos de la siguiente manera:

```
const <tipo de dato> * <nombre>;
```

### Ejemplo

```
int x = 10;
int y = 20;
const int * p1 = &x;

*p1 = y; // esto no es legal ya que intenta cambiar el contenido de p1.

p1 = &y; // esto es legal ya que cambia el valor de p1 pero no su contenido.
```

Los punteros a constantes son muy utilizados cuando definimos los parámetros de una función, para protegernos de cambiar el contenido de lo apuntado por el puntero dentro de la función.

Ejemplo:

```
#include <stdio.h>

int longitudPalabra(const char *palabra){
    int longitud = 0;
    while(*(palabra++){
        longitud++;
    }
    return longitud;
}

int main(){
    char pal[] = "hola mundo";
    printf("%d\n", longitudPalabra(pal));
    return 0;
}
```

## Aritmética de punteros

Si el puntero no es un puntero constante, el mismo puede ser modificado. Para modificar un puntero, podemos, además de asignarle la dirección de memoria de distintas variables, realizar operaciones aritméticas como sumas o restas de enteros para que el puntero apunte a  $n$  direcciones posteriores o anteriores.

```
#include <stdio.h>

int main(){
    char pal[] = "hola mundo";
    char *p;
    p = pal;
    p = p + 2; // p apunta a la "l" de hola
    return 0;
}
```

### Recuerde

- No se puede sumar 2 punteros
- No se puede multiplicar 2 punteros
- No se puede dividir 2 punteros.

## Cadenas

En C, una *cadena* o *cadena de caracteres* es un array de caracteres (char) finalizados con el caracter '\0' (contrabarra cero).

h	o	l	a	\0
---	---	---	---	----

### Declaración e inicialización de variables de cadena

Para declarar una variable de tipo cadena, podemos utilizar una sintaxis similar a la utilizada para declarar arrays o bien, podemos utilizar una sintaxis similar a la utilizada para declarar punteros.

Ejemplo:

```
a) _ char palabra[] = "hola mundo";  
b) _ char palabra2[11] = "hola mundo";  
c) _ char *palabra = "hola mundo";
```

En el caso 'a', la dimensión de la cadena se asigna automáticamente a partir del contenido asignado.

En el caso 'b' lo definimos al declarar la variable. Si bien "hola mundo" posee 10 caracteres, en la declaración debemos reservar espacio para un caracter más que el caracter de finalización de la cadena, el caracter '\0'.

En el caso 'c' el sistema reserva memoria automáticamente al igual que en el caso 'a'.

### ¿Por qué no podemos declarar una variable de tipo cadena y luego asignarle valor?

Esto se debe a que una variable de tipo cadena es un *puntero constante* y por lo tanto no se puede cambiar la dirección de aquello a lo que apunta.

Es por este motivo que para declarar y luego asignarle valor a una variable de tipo cadena se utilizan funciones de la biblioteca string.h.

## Biblioteca string.h

Nombres	Descripción
<b>memcpy</b>	copia n bytes entre dos áreas de memoria que no deben solaparse
<b>memset</b>	sobre escribe un área de memoria con un patrón de bytes dado
<b>strcat</b>	añade una cadena al final de otra
<b>strncat</b>	añade los n primeros caracteres de una cadena al final de otra
<b>strchr</b>	localiza un carácter en una cadena, buscando desde el principio
<b>strrchr</b>	localiza un carácter en una cadena, buscando desde el final
<b>strcmp</b>	compara dos cadenas alfabéticamente ('a'!='A')
<b>strncmp</b>	compara los n primeros caracteres de dos cadenas numéricamente ('a'!='A')
<b>strcpy</b>	copia una cadena en otra
<b>strncpy</b>	copia los n primeros caracteres de una cadena en otra
<b>strlen</b>	devuelve la longitud de una cadena
<b>strspn</b>	devuelve la posición del primer carácter de una cadena que no coincide con ninguno de los caracteres de otra cadena dada
<b>strstr</b>	busca una cadena dentro de otra
<b>strtok</b>	parte una cadena en una secuencia de tokens

Para más información ver <http://www.cplusplus.com/reference/cstring/>

# Unidad Temática N°10

~ Depuración ~

“Depurar es como ser el detective en una película de crimen en la que tú también eres el asesino.”

~ Filipe Fortes

## Depuración

### Bug

Es un defecto en el código de un programa que desencadena en una falla del producto.

### Falla

Es la manifestación de un defecto en el código.

### Depuradores

Son programas que permiten, como su nombre lo indica, depurar el código fuente. Muchas veces incluyen interfaces gráficas y se encuentran integrados en un entorno de desarrollo. Proveen herramientas que permiten:

- Recorrer paso a paso un programa.
- Detenerse en una línea particular del código, esto se denomina breakpoint o punto de interrupción. Los mismos pueden ser condicionales.
- Inspeccionar el valor de las variables locales y globales
- Observar la secuencia de ejecución del programa (stack trace)

La prueba a ciegas con un depurador no suele ser productiva. Es más útil usarla para descubrir el estado del programa cuando falla, y luego pensar sobre cómo pudo haber ocurrido dicha falla.

### Errores fáciles de corregir

La depuración implica razonamiento en retrospectiva. Se debe empezar el análisis desde el punto donde surgió la falla e ir hacia atrás para descubrir las causas. Cuando se tenga una explicación completa, sabrá que arreglar.

#### Busque patrones familiares

Analice si el error no lo vió antes. Los errores comunes tienen síntomas característicos.

- Errores de conversión de tipos
- Acceso a memoria fuera de los límites
- No inicializar variables

Muchos de estos errores pueden ser detectados por el compilador si se habilitan las verificaciones correspondientes.



### Examine los cambios más recientes

Una práctica recomendada es cambiar una cosa a la vez. Si sigue esta regla, es muy probable que el error esté en el nuevo código o ha sido puesto en evidencia por lo último que haya hecho.

Una revisión cuidadosa ayuda a localizar el problema.

En este punto son de gran ayuda los sistemas de control de código (git, svn, etc.)

### No cometa dos veces el mismo error.

Después de haber encontrado y corregido un error, revise el código para ver si el mismo error se encuentra en otra parte del código.

### Corríjalo ahora, no después

Cuando probando su programa detecte un error, aunque no sea fatal, corríjalo en el momento en que lo detectó, dejarlo para después puede ser más crítico y costoso de corregir.

### Obtenga una reconstrucción de la pila de ejecución

Un uso muy común de los depuradores es que permiten hacer un análisis post-mortem. Para esto se obtiene y analiza la pila de ejecución del programa (*stack trace*) la cuál nos brinda la línea de código donde se produjo la falla, además también muestra todas las llamadas a funciones que se fueron sucediendo.

Ejemplo:

```
Program received signal SIGSEGV, Segmentation fault. At ejercicio3.cpp:62

[debug]#0  0x000000000040124c in ejercicio3::ejecutar () at ejercicio3.cpp:62
[debug]#1  0x0000000000401552 in main () at main.cpp:25
[debug]>>>>>cb_gdb:
```

La pila de ejecución debe leerse desde abajo hacia arriba para seguir el flujo de ejecución del programa.

En este ejemplo vemos que:

- lo primero que se ejecutó es la instrucción que se encuentra en la función `main()` en la línea 25 del archivo `main.cpp`
- lo segundo que se ejecutó es la instrucción que se encuentra en la función `ejecutar()` en la línea 62 del archivo `ejercicio3.cpp`

El mensaje siguiente nos está indicando que se produjo un error en dicha línea.

## Errores difíciles de corregir

Hay ocasiones cuando no se tiene la menor pista acerca del por qué de la falla detectada. A eso nos referimos con *errores difíciles de corregir*. Algunas recomendaciones al respecto son:

### Trate de hacer que el error sea reproducible

El primer paso es asegurarse de que puede reproducir el error cuando lo necesite. Diseñe casos de prueba, parámetros y entradas, que provoquen la falla.

Esto ayudará a analizar el problema, encontrar su solución y verificar que no vuelva a ocurrir.

### Divide y vencerás

Esta técnica apunta a acotar el fragmento del código donde se genera el error. Para esto suele ser útil saber cuál fue el flujo de ejecución del código, ya que esto nos permitirá descartar algunas partes del mismo. Este accionar se puede repetir hasta dar con la línea que genera el error.

### Estudie la numerología de los errores

A veces un patrón en la numerología de los casos de prueba diseñados, que permiten reproducir la falla, ofrece una pista que nos permite enfocar la búsqueda. Analizar los datos de entrada que provocan la falla y su relación puede llevarnos directamente a detectar el error.

### Despliegue salidas para situar la búsqueda del error

Cuando no se entiende lo que el programa está haciendo, agregar instrucciones para mostrar más información puede ser una forma sencilla y efectiva de averiguarlo. Esta técnica permitirá mejorar su entendimiento sobre el flujo de ejecución del código.

Si va a imprimir el valor de una variable, déle siempre el mismo formato.

Si muestra punteros, hágalo utilizando el formato %x(hexadecimal) o %p(puntero). Esto le ayudará a ver si dos punteros están relacionados.

### Escriba un archivo de registro (log)

Este punto es similar al anterior, sólo que enviando la salida a un archivo que contenga la información en un formato fijo. Cuando el programa termina de forma repentina, el registro contendrá la información hasta ese punto.

Asegúrese de vaciar los buffers de E/S para que los registros finales aparezcan en el archivo. En C, una llamada a fflush garantiza que toda la salida se escriba antes de que falle el programa.

## Tipos de Errores

A modo de resumen y desde un punto de vista conceptual, podemos agrupar los errores en tres tipos: sintácticos, lógicos y de regresión.

### Errores Sintácticos

Son aquellos que se producen cuando el programa viola la sintaxis, es decir las reglas de gramática del lenguaje.

Estos errores suelen ser detectados por el compilador y son fáciles de corregir. Algunos ejemplos de estos errores son:

- Omisión de punto y coma al final de una sentencia.
- Comentarios multilínea sin cerrar. Falta de la secuencia \*/
- Olvido de doble comillas para cerrar una cadena
- Etc.

### Errores Lógicos

Este tipo de error, representa un error del programador en el diseño del algoritmo. Son, por lo general, difíciles de encontrar y aislar. Estos errores no son detectados por el compilador.

Algunos de estos errores pueden producirse por:

- Falta de paréntesis en sentencias que calculan algo.
- Falta de inicialización de variables
- Falta de verificación en los datos de entrada.
- Etc.

Ejemplos:

```
// el error está en que el tercer * debería ser un +
double peso = densidad * 5.25 * PI * pow(longitud, 5)/4.0

// esta fórmula no es la correcta
grados_centigrados = grados_fahrenheit * temperatura_centigrados
```

### Errores de Regresión

Son errores que se crean de forma accidental cuando se intenta corregir otro error.

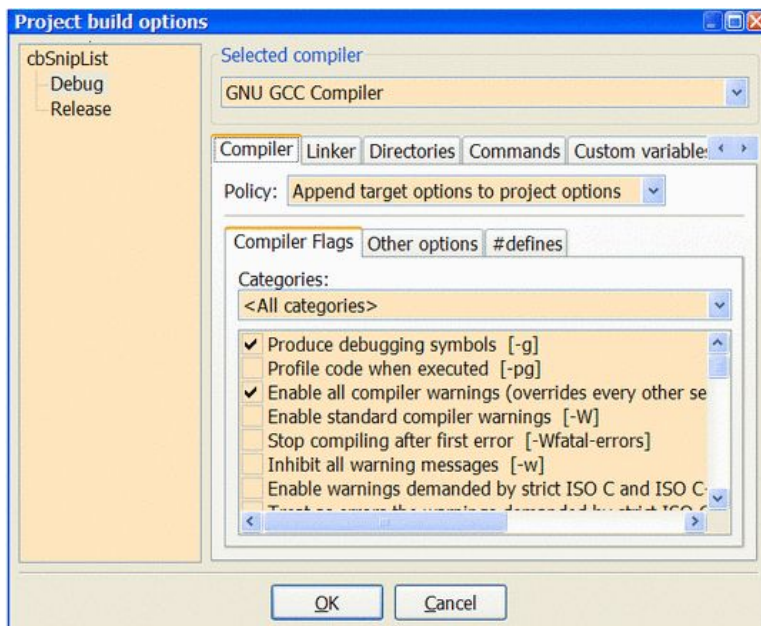
Como regla, y como ya mencionamos antes, antes de compruebe que el error no se produce en otra parte del código y que realmente fue corregido en su totalidad.

## Depurando con CodeBlocks

### Configurar CodeBlocks para depurar

El primer paso antes de poder depurar nuestro código es configurar el IDE para tal fin. Para ello debemos acceder al menú *Proyecto*→*Opciones de Compilación* (*Project* → *Build Options*)

Allí veremos una pantalla como la siguiente:



En el cuadro de la izquierda vemos el listado de modos de compilación, Debug (Depuración) y Release (Lanzamiento).

Debemos seleccionar la opción *Debug* y en el cuadro de la derecha vemos las opciones de compilación (Compiler Flags)

#### Opciones para construir una versión de depuración

Para poder generar una versión de depuración de su código, debe asegurarse que las siguientes opciones estén activas:

- Obligatorias
  - -g: compila el código fuente generando información de depuración que es utilizada por el depurador.
  - Opcionales pero útiles
    - -Wall: habilita todas las advertencias sobre construcciones comunes de error.

- -Wfatal-errors: frena el proceso de compilación ante el primer error
- Sólo Opcionales
  - -pedantic: habilita las advertencias exigidas por el estándar ISO
  - -pedantic-errors: trata como un error las advertencias exigidas por el estándar ISO

## Compilar para depurar

Una vez activadas las opciones mencionadas, ya podemos generar nuestra versión de depuración.

Para ello, desde la ventana principal, accedemos al menú *Depurar* → *Comenzar/Continuar*.

Recuerde que si hizo cambios o previamente había compilado en modo *Lanzamiento (Release)* deberá recompilar los fuentes.

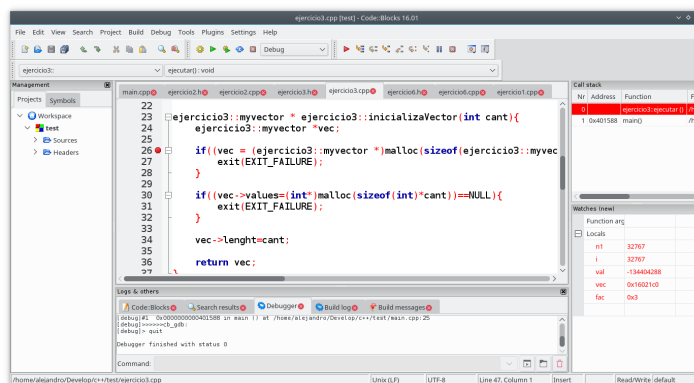
## Establecer un Punto de Interrupción (breakpoint)

Un breakpoint o punto de interrupción es una marca que se establece sobre una línea del código para que al momento de depurar, el flujo de ejecución se detenga en la misma.

Para generar un punto de interrupción podemos:


- Acceder al menú *Depurar* → *Establecer Punto de interrupción*
- O bien hacer click sobre el margen izquierdo (al lado del número de línea)

Una vez establecido el punto de interrupción, presionamos el botón *Depurar* y el programa se ejecutará hasta el primer punto de interrupción encontrado.



## Observar el valor de una variable

Cuando estamos depurando el código, suele ser útil conocer el valor de las variables en un determinado momento. Para esto podemos agregar la ventana *Watches* la cuál nos mostrará las variables locales y globales y su valor en cada momento.



Para agregar la ventana *Watches*, debemos acceder al menu *Depuración* → *Ventanas de Depuración* → *Watches*.

En la ventana *Watches*, también podemos evaluar expresiones que involucren 1 o más variables. Para esto, en la última fila de la ventan, hacemos doble click y escribimos la expresión.

#### Ver la pila de ejecución

Para agregar la ventana *Pila de ejecución*, debemos acceder al menu *Depuración* → *Ventanas de Depuración* → *Pila de ejecución*.

## Bibliografía

- Luis Joyanes Aguilar, Ignacio Zahonero Martínez. Programación en C. Segunda Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA. S.A.U., 2005. ISBN-84: 481-9844-1.
- Brian W. Kernighan, Rob Pike. La práctica de la programación. Pearson Educación. Méjico (2000)
- Debugging with CodeBlocks:  
[http://wiki.codeblocks.org/index.php?title=Debugging\\_with\\_Code::Blocks](http://wiki.codeblocks.org/index.php?title=Debugging_with_Code::Blocks)
- Bibliotecas.
  - [https://es.wikipedia.org/wiki/Biblioteca\\_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Biblioteca_(inform%C3%A1tica))
- Enlace dinámico
  - [https://es.wikipedia.org/wiki/Enlace\\_din%C3%A1mico](https://es.wikipedia.org/wiki/Enlace_din%C3%A1mico)
- Enlace estático
  - [https://es.wikipedia.org/wiki/Enlace\\_est%C3%A1tico](https://es.wikipedia.org/wiki/Enlace_est%C3%A1tico)

