
Capítulo 1

Introducción a la programación

1.1. Un ejemplo

Con el fin de exponer una noción de lo que es programar veamos el siguiente ejemplo, suponga que un familiar suyo estuvo de viaje, visitó Japón, y le trajo de presente un robot, que solamente atiende a los dos siguientes tipos de ordenes:

- *avanzar* X centímetros
- *girar* X grados.

Una secuencia de ellas es posible dárselas al robot, para que recorra un camino determinado. Si queremos indicarle al robot (la carita feliz de color turquesa) que aparece en la figura 1.1 que se desplace hasta donde está el objetivo debemos de algún modo "*decirle*" lo que debe hacer, si suponemos que cada rectángulo de la cuadrícula tiene diez centímetros de lado, las ordenes le daríamos a nuestro alegre amigo para alcanzar el objetivo podrían ser algo como:

Código 1 Ejemplo de instrucciones para llegar al objetivo.

```
1 avanzar 70cm.  
2 girar 90 grados a la izquierda.  
3 avanzar 250cm.  
4 avanzar 80 cm.
```

Aunque ahora es posible darle algunas instrucciones a las máquinas mediante la voz, por ahora se las daremos a la antigua, escribiéndolas,

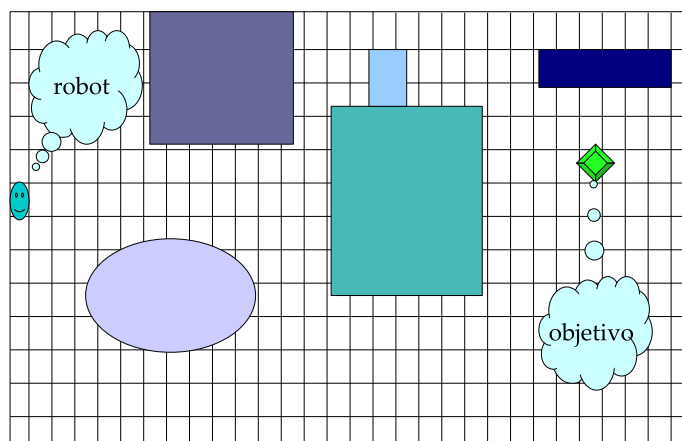


Figura 1.1: Un robot.

para hacerlo, debe existir algún tipo de teclado que nos permita digitarlas. Las órdenes se graban para que se ejecuten una a una.

Si el robot toma las instrucciones dadas anteriormente realizará un recorrido como el mostrado en la figura 1.2. Lo que se acaba de hacer es *programar*, la programación de sistemas reales no difiere mucho de lo aquí mostrado, bueno, posiblemente se tengan a la mano más instrucciones y un sistema que no se llame "smile".

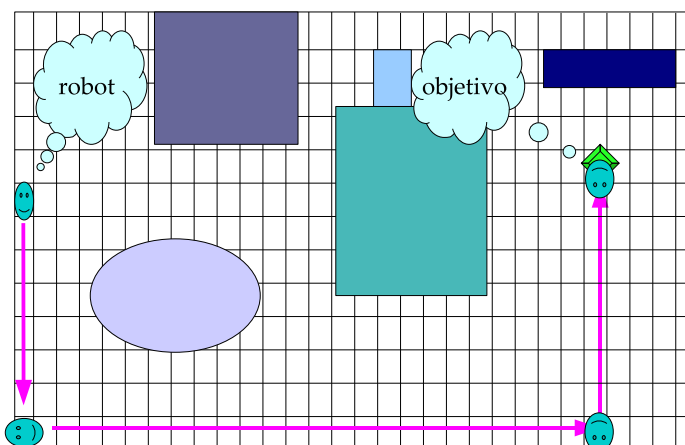


Figura 1.2: Ejecución de instrucciones.

1.2. Sistemas programables, algoritmos y programas

Hoy en día, las computadoras, ya sean de propósito general o específico están por todas partes, teléfonos, electrodomésticos, aviones, etc; y realizan tareas de todo tipo, desde reproducir vídeo hasta controlar trayectorias de disparo de tanques, todas esas máquinas de cómputo requieren, como cualquier máquina, que se enciendan y sean controladas para realizar una tarea específica, la diferencia entre una computadora y un tractor (sin computadora de abordo) es que al tractor lo controla una persona y a la computadora lo que denominamos un *programa*, también llamado *software*.

Las computadoras son un ejemplo de *sistemas basados en programa almacenado*, todos estos sistemas poseen un procesador central, cuya actividad de una forma simple puede resumirse a:

1. Obtener una instrucción.
2. Determinar que instrucción es.
3. Ejecutar la instrucción
4. Ir al paso número 1

El conjunto de instrucciones que se desea que el sistema ejecute se almacena en algún tipo de memoria, RAM o ROM, dependiendo del sistema, por ejemplo muchos de los microcontroladores el programa se almacena en ROM, mientras que en las computadoras los programas son cargados a memoria RAM por el sistema operativo para su ejecución. En la figura 1.3 se muestra un ejemplo de estructura de un sistema basado en procesador.

Todo programa comienza con idea, algo que se quiere hacer, generalmente ese algo resulta como solución a un problema específico, la solución de un problema requiere el diseño de un *algoritmo*.

Algoritmo Palabra que proviene del nombre de un matemático y astrónomo árabe *Al-Khôwarizmi* del siglo IX, que escribió un tratado sobre la manipulación de números y ecuaciones llamado *Kitab al-jabr w'almugabala*. Un algoritmo es una secuencia ordenada de pasos, no ambiguos, expresados en lenguaje natural que conducen a la solución de un problema dado.

Los algoritmos deben cumplir con algunas características:

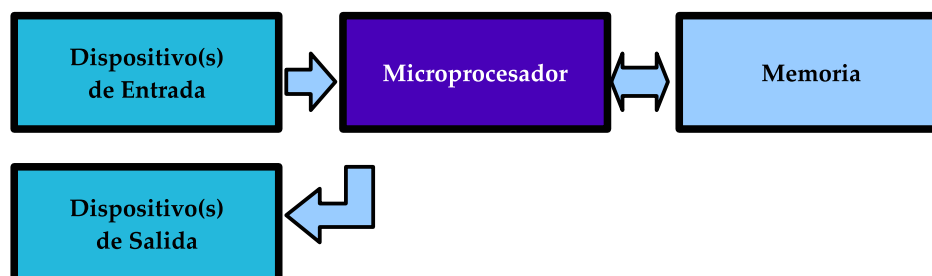


Figura 1.3: Sistema basado en procesador.

- *Preciso*. Indica el orden de realización de cada uno de los pasos.
- *Definido*. Si a un algoritmo se le suministra varias veces los mismos datos los resultados deben ser los mismos.
- *Finito*. El algoritmo debe terminar en algún momento.

Los algoritmos son soluciones abstractas a problemas, ellos generalmente son codificados en un lenguaje de programación y luego traducidos para que una computadora los pueda ejecutar y solucionar entonces un problema real. Los algoritmos son independientes del lenguaje de programación y de la máquina que lo ejecute, una analogía de la vida real [joyanes1989], la receta de un plato de cocina puede expresarse en inglés, francés o español e indicará la misma preparación independientemente del cocinero.

Lenguaje de programación Son conjuntos de instrucciones con que se pueden escribir los algoritmos para que un sistema lo ejecute. Existen múltiples tipos de lenguajes de programación:

- *Lenguaje de máquina*. Es un lenguaje compuesto por códigos binarios que un sistema puede ejecutar directamente, los programas ejecutables son precisamente secuencias de instrucciones en lenguaje de máquina, un ejemplo de instrucciones en lenguaje de máquina es:

| |
|----------------|
| 0011 0000 0001 |
| 0101 0001 0011 |

Las anteriores instrucciones le indican a un procesador que sume dos datos y que luego multipliquen ese resultado por otro. Las instrucciones en lenguaje de máquina están compuestas de un código que

identifica la instrucción (*opcode*) y uno o varios operandos (o referencias a los mismos). Depende del tipo de procesador sobre la cual se esté programando, Intel, Motorola, Atmel, etc, cada uno de ellos tiene códigos de operación diferentes.

- *Lenguajes ensambladores.* Escribir programas funcionales en lenguaje de máquina es una tarea que pocas personas desean hacer, pues es muy propenso a errores y tedioso, por ello a alguien se le ocurrió asociar símbolos o mnemonicos a las instrucciones que una máquina podía realizar, por ejemplo en algun lenguaje ensamblador las instrucciones en lenguaje de máquina antes mencionadas quedarían:

| |
|------------------------------------|
| <pre>add [0] [1] mul [1] [3]</pre> |
|------------------------------------|

Para convertir los programas en lenguaje ensamblador a código de máquina se usa un programa llamado *ensamblador*.

- *Lenguajes de alto nivel.* Son lenguajes que tienen conjuntos de instrucciones similares a las palabras del idioma ingles (o algún otro) que son más fáciles de entender por los seres humanos. En C, las instrucciones para hacer lo que hicimos en lenguaje de máquina y ensamblador serían:

| |
|-------------------------|
| <pre>res=(a+b)*c;</pre> |
|-------------------------|

Existen muchos lenguajes que se pueden llamar de alto nivel, Basic, Pascal, Fortran. Lenguaje C, comparado con el lenguaje ensamblador es de alto nivel, pero tiene la característica de permitir acceder a muchos de los recursos de hardware disponibles en un sistema, por lo que mucho lo clasifican como lenguaje de nivel medio. Para convertir los programas escritos en un lenguaje de alto nivel en código de máquina se tienen las siguientes opciones:

- *Intérpretes.* En los lenguajes interpretados, cuando se ejecuta un programa cada instrucción se traduce a lenguaje de máquina y a continuación se ejecuta, ejemplos de lenguajes interpretados son matlab, python, smalltalk.

- *Compiladores*. Traducen completamente los programa fuente a lenguaje de máquina, ejemplos de lenguajes compilados son, C/C++, Pascal, Fortran, COBOL.
- *Híbridos*. En los últimos años aparece una especie de lenguajes que combinan las dos opciones anteriores, existe un compilador que traduce el código fuente en un código intermedio que es ejecutado por un programa especial denominado máquina virtual. Ejemplos de esto son Java y los lenguajes de la plataforma .Net de Microsoft.

1.2.1. El proceso de traducción

Como se había mencionado antes, cuando se tiene un algoritmo listo, es posible programarlo en algún lenguaje de programación con el fin de ejecutarlo en una computadora, para ello, es necesario traducirlo a código de máquina, el proceso de generación de un programa puede dividirse en los siguientes pasos:

1. Se escribe el algoritmo en algún lenguaje de programación en un editor de texto y se guarda en un archivo.
2. Se utiliza un compilador para generar el ejecutable, en este paso si existen errores de sintaxis el compilador genera un mensaje de aviso.

En la figura 1.4 se muestra esquemáticamente el proceso de traducción de un programa escrito en un lenguaje de alto nivel a código entendible por una máquina.

1.2.2. Un poco de historia de los lenguajes de programación

Según lo que se dice, fue Charles Babage a mediados del siglo XIX el que creo el primer lenguaje de programación, dicho lenguaje era para una máquina diseñada por él, llamada la máquina analítica, como compañera de Babage estaba Ada Byron considerada como la primera persona que escribió un programa. La máquina de Babage solo se construyó alrededor de un siglo después. La primera computadora la ENIAC se programaba directamente modificando el hardware de la misma mediante cables, luego se usaron las ideas de Babage y se realizaban programas utilizando tarjetas perforadas.

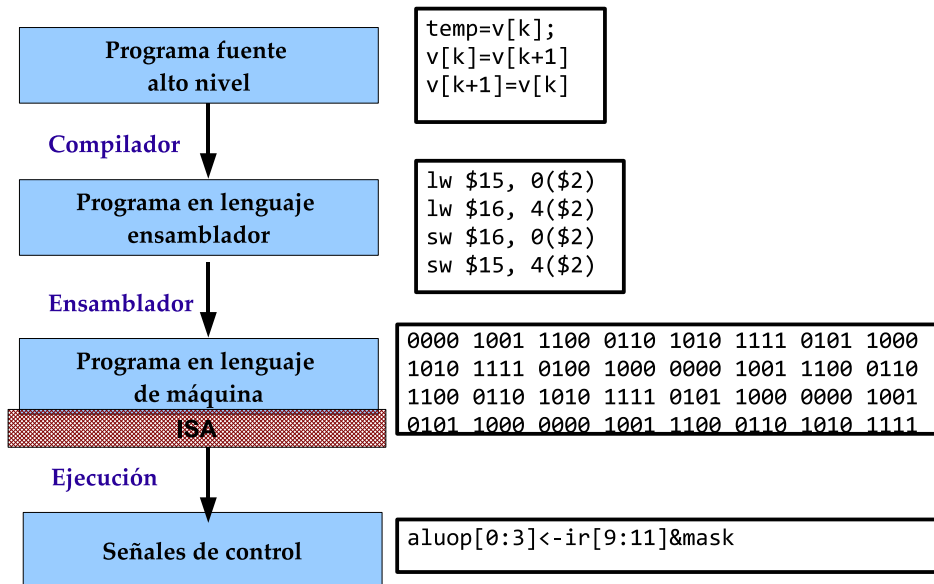


Figura 1.4: Traducción de un programa.

En la década de los 50 surgen los primeros ensambladores simbólicos, no era necesario indicar explícitamente las direcciones de memoria, si no que podíamos usar variables para referirnos a ellas.

En los 60 aparece la *programación estructurada*, comienza el uso de instrucciones de control de flujo de alto nivel (if, else while, etc, que empiezan a remplazar el uso del goto), funciones y procedimientos, algunos de los lenguajes que aparecieron en esa época son: Fortran IV, COBOL, Lisp, Algol 60, Basic, Simula-67 (precursor de lo que fue más adelante la programación orientada a objetos) entre otros.

En los 70 surge el diseño orientado a los *tipos abstractos de datos* y con ello lenguajes que permitían implementar aplicaciones así diseñadas, Pascal, Prolog, lenguaje B, lenguaje C (nuestro querido amigo). Modula-2.

En los 80 entra en escena la *programación orientada a objetos* y el lenguaje Smalltalk-80, C++, Object-Pascal, Oberon, Ada (no orientado a objetos del todo)

En los años noventas la revolución de las interfaces de usuario junto con la ya en carrera programación orientada a objetos hacen que los lenguajes y entornos de programación permitan el desarrollo de aplicaciones *controladas por eventos*, y aparecen lenguajes-entornos como Visual-

Basic, Delphi y el lenguaje Java.

En el año 2000 Microsoft lanza la plataforma .Net con varios lenguajes en ella, entre los que se encuentra C#, un lenguaje orientado a objetos y componentes del que hablaremos alguna páginas adelante.

1.2.3. Fases del desarrollo de software

Construir cualquier cosa requiere de un cierto proceso, un orden, no debemos pintar una pared antes de levantarla, no podemos, después de terminar una casa que fue planeada para ser de un piso, convertirla en un edificio, el proceso de crear un programa no es la excepción. Existen muchos textos en los cuales se exponen las fases del desarrollo de software, utilizando como referencia a [Booch1996] es posible indicar que la solución de problemas mediante programas de computadora (creación de un producto de software) puede dividirse en las siguientes fases:

- Análisis.
- Diseño.
- Implementación.
- Pruebas.
- Implantación.
- Mantenimiento.

Análisis. Aquí se asegura de que el problema esté completamente especificado y que se comprende completamente. Me parece muy útil citar la siguiente especificación de un problema [Bronson2000]:

Escriba un programa que proporcione la información que necesitamos acerca de los círculos. Termínelo para mañana.

-La gerencia.

Espero que ninguno de los lectores haya dejado de leer para sentarse en su computadora para trabajar en su lenguaje favorito. Una revisión rápida a la especificación del anterior problema nos indica que es vaga, pues no

se expresa de manera precisa qué es lo que se necesita a cerca de los círculos. Digamos que usted recibe la solicitud y mañana le lleva al gerente un programa que calcula el área un círculo dado su radio, pero el gerente le dice *"no! yo lo que quería era un programa que calculara el perímetro"* no le vaya a decir esto a su jefe, pero el problema fue que no le especificaron los requerimientos del programa. En últimas el análisis proporciona un modelo de la forma en la que un sistema se comporta.

Diseño. Si el programa lo que tiene que hacer es una sola cosa en esta fase se diseña el algoritmo que realiza el proceso, Si es un programa de mayor tamaño aquí se crea la arquitectura a usar para la implementación y se establecen las políticas tácticas comunes.

Implementación. Se realiza la implementación del programa en un lenguaje o lenguajes de programación.

Pruebas. Se corroborar que cumple con los requisitos.

Implantación. Llevar el sistema a los clientes.

Mantenimiento. Los programas del mundo real generalmente deben evolucionar con el tiempo con el fin de no volverse obsoletos con el tiempo, por ello es necesario actualizarlos cada cierto tiempo.

1.2.4. Qué se necesita para programar

La programación no es solo conocer un lenguaje y escribir aplicaciones, existen varios tipos de conocimiento necesarios que hacen que programar no sea solamente un arte sino que sea algo sistematico, dentro de los conocimientos necesarios en la formación en esta área, podemos citar [Villalobos2005]:

- Modelaje. Análisis y especificación de problemas.
 - Algorítmica. Diseño de algoritmos.
 - Tecnología y programación. Lenguajes de programación, modelaje etc.
 - Herramientas de programación. Editores, compiladores, depuradores, gestores de proyectos, etc.
-

- Procesos de software. División del proceso de programar en etapas claras, ciclo de vida del programa, formatos, entregables, estándares de documentación y codificación, técnicas de pruebas.
- Técnicas de programación y metodologías. Estrategias y guías que ayudan a crear un programa. Como se hacen las cosas.
- Elementos estructurales y arquitecturales. Estructura de la aplicación resultante, en terminos del problema y elementos del mundo del problema. Funciones, objetos, componentes, servicios, modelos, etc. La forma de la solución, responsabilidades y protocolos de comunicaciones.

Se recomienda hacer que la formación en programación ataque los elementos citados de manera uniforme, pero esto es más fácil decirlo que hacerlo.

1.3. Algoritmos

-TODO, tipos de dato, variables, expresiones, funciones.

1.4. Ejercicios

1. Escriba un algoritmo para preparar una taza de café.
2. Escriba un algoritmo para realizar una llamada telefónica desde un teléfono público.
3. Suponga que las instrucciones del robot *smile* en lenguaje de máquina son:

| <i>opcode</i> | <i>operando</i> | <i>significado</i> |
|---------------|-----------------|------------------------------|
| 0000 0001 | no tiene | rotar 90 grados a la derecha |
| 0000 0010 | no tiene | rotar 90 a la izquierda |
| 0000 0011 | xxxx xxxx (cm) | avanzar X centímetros |

Cuadro 1.1: Instrucciones de smile.

Escriba el programa que aparece en 1 en lenguaje de máquina usando la tabla anterior, necesita convertir los valores de centímetros a binario.

4. Inventese una instrucción para tomar un objeto y escriba un programa ir por el objetivo, tomarlo y regresar a la posición de origen.
 5. Dado que la instrucción avanzar tiene un operando que es de 8 bits de longitud, el máximo número de centímetros que se puede avanzar es de 255 (2^8), Escriba un programa para avanzar 325 centímetros en línea recta.
 6. Escriba un algoritmo para intercambiar el contenido de dos tazas de café. suponga que tiene una tercera taza disponible para realizar el proceso. Cada taza debe enjuagarse antes de recibir contenido nuevo.
 7. Escriba un algoritmo para encontrar el número más pequeño de un conjunto de tres.
 8. Escriba un algoritmo para encontrar el número de veces que aparece la letra *a* en una oración.
-

Capítulo 2

Introducción al lenguaje C

-¿Cómo, en pleno siglo 21 y nos vamos a gastar un capítulo en ese personaje del Jurásico?

-Si.

"El lenguaje C es como una gran arma, uno puede destruir casi cualquier cosa con ella, pero se puede volar los pies si no es cuidadoso".

Vamos a ver las características básicas del lenguaje, que nos permitan implementar algoritmos básicos.

2.1. ¿Por qué lenguaje C?

Una razón es por que no sé Pascal. Y la otra es por que es un lenguaje muy útil en tareas de programación de sistemas embebidos, aunque tal y como se muestra en la Figura 2.1 no es lo único que se puede hacer con él.

Lenguaje C tiene sus orígenes en un lenguaje llamado BCPL, desarrollado por Martin Richards. El cual fue la base para el lenguaje B creado por Ken Thomsom.

Un los años setenta un nuevo sistema operativo quería salir a la luz pero el lenguaje disponible en ese momento era el ensamblador para el DEC-PDP11, el cual dada la magnitud del proyecto no era la mejor opción, entonces Dennis Ritchie mezclo el dichoso lenguaje ensamblador con el lenguaje B y nació el lenguaje C.

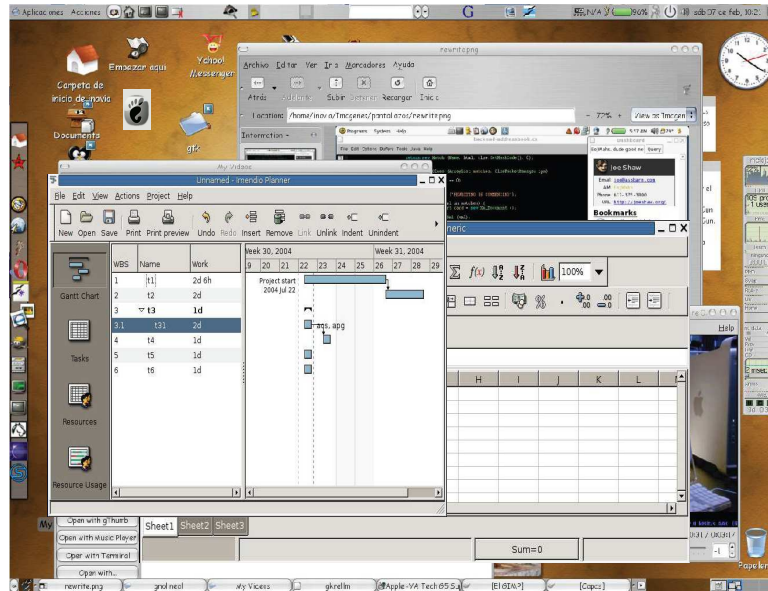


Figura 2.1: Programas hechos en C.

Lenguaje C tiene características muy especiales, la sintaxis es muy *"chick"*, con muy pocas palabras reservadas, 27 del estándar de Kerningam & Ritchie + 5 del estándar ANSI, compararlas por ejemplo con 159 del Basic de IBM.

Ah por si no recuerda el sistema operativo que estaba en fase embriónica era Unix. Fue el primer sistema operativo con compiladores de C.

2.2. Herramientas necesarias

Para implementar los ejemplos que vamos a realizar en este capítulo se puede utilizar cualquier compilador de C que cumpla con el estándar ANSI C, sin embargo vamos a utilizar una versión de windows del famoso GCC, el MinGW. Para usar MinGW, existen básicamente dos opciones, usarlo desde línea de comandos o desde un entorno de desarrollo que simplifique el proceso de compilación, enlazado y depuración. Hay varios entornos de desarrollo para trabajar con las herramientas mingw, entre los que podemos citar, DevC++, Eclipse, Ultimate++ y Code::Blocks, en las siguientes figuras podemos observar capturas de pantalla de algunos de ellos:

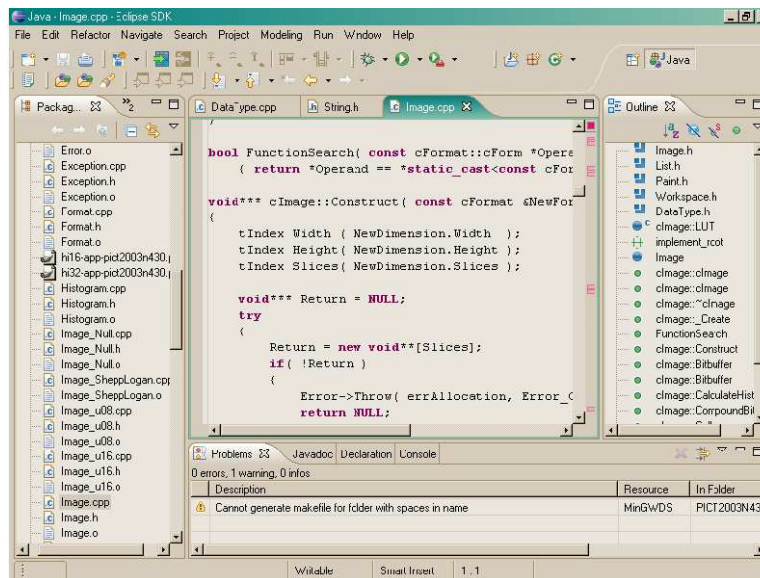


Figura 2.2: Eclipse para C/C++.

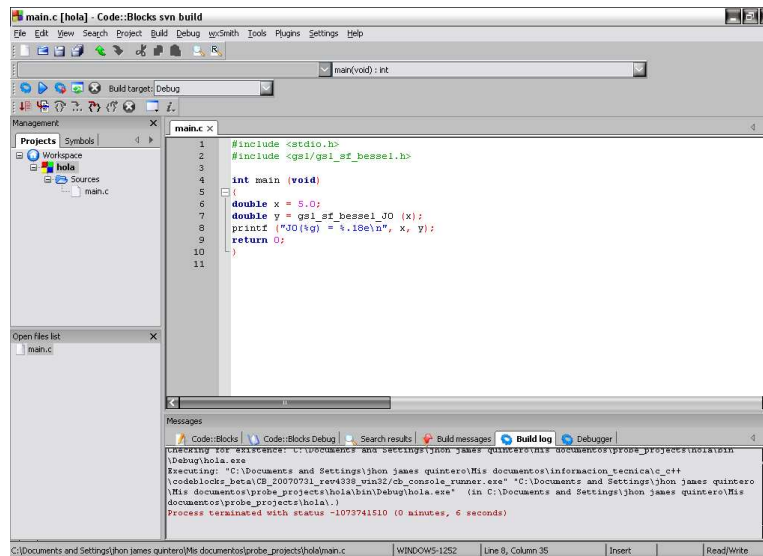


Figura 2.3: Code::Blocks.

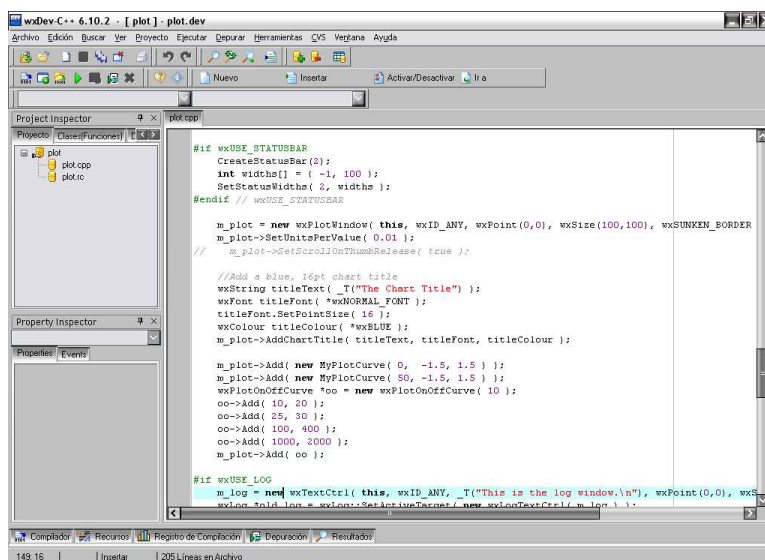


Figura 2.4: WxDevcpp.

2.2.1. Instalación del MinGW

Lo primero que hay que hacer es descargar de internet las herramientas, que consisten en, *binutils*, *w32api*, *gdb*, *mingw32-make* y *mingw-runtime*. Crear un directorio por ejemplo mingw en C:, y descomprimir allí los archivos mencionados quedando una estructura como la mostrada en la figura 2.5.

Ahora podemos abrir una consola de comandos de windows (inicio-ejecutar-cmd), nos dirigimos al directorio bin que aparece en el directorio donde instalamos mingw, allí aparecen varios archivos con extensión *exe*, digitemos mingw32-gcc o gcc seguido de enter y obtenemos algo como lo que aparece en la figura 2.6.

El anterior comando es el que se utiliza para compilar programas hechos en lenguaje C, pero dado que nuestros programas no deben ir en el directorio *bin* podemos agregar a la variable del sistema PATH, la ruta de los mencionados ejecutables (click derecho en mi pc, propiedades, opciones avanzadas, variables de entorno, seleccionar la variable del sistema PATH y modificar), de este modo podemos desde cualquier ubicación de nuestro sistema digitar gcc y esté se ejecutará.

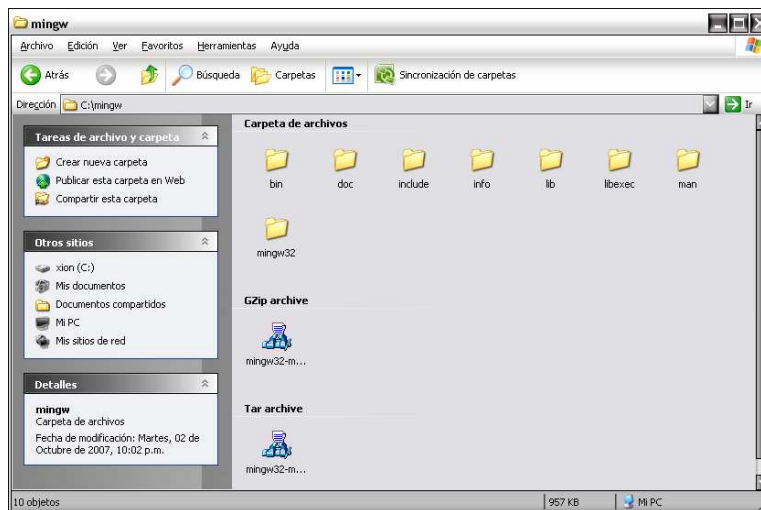


Figura 2.5: Directorio de instalación del mingw.

2.2.2. Primeros programas con GCC.

Ahora, creamos un directorio en la unidad C, digamos *projects*, *mkdir projects*, luego nos dirigimos ese directorio con *cd projects*, abrimos el editor de la consola de windows con *edit*, y aparece el editor de la figura (sin el código, claro) 2.7, bueno podemos usar también el netepad de windows o cualquier otro editor de texto.

Guardamos el archivo como *min.c*, luego digitamos *gcc min.c*, esto generará un archivo llamado *a.exe* que puede ser ejecutado tecleando su nombre

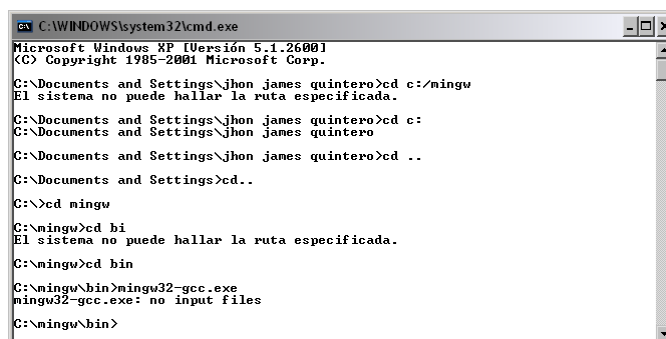


Figura 2.6: Comando para compilar.

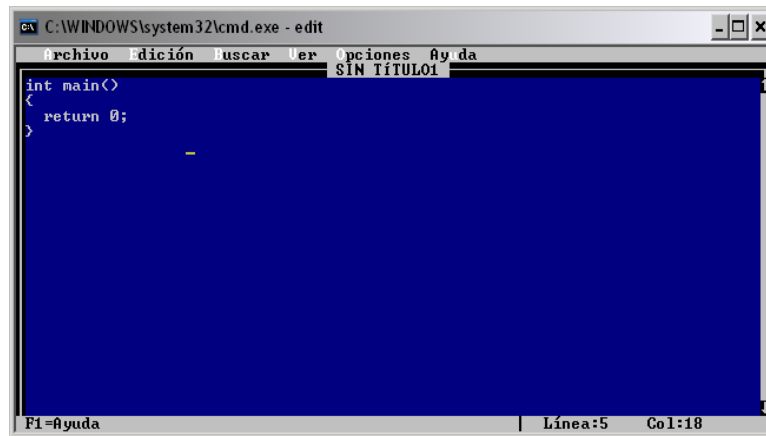


Figura 2.7: Programa mínimo en C.

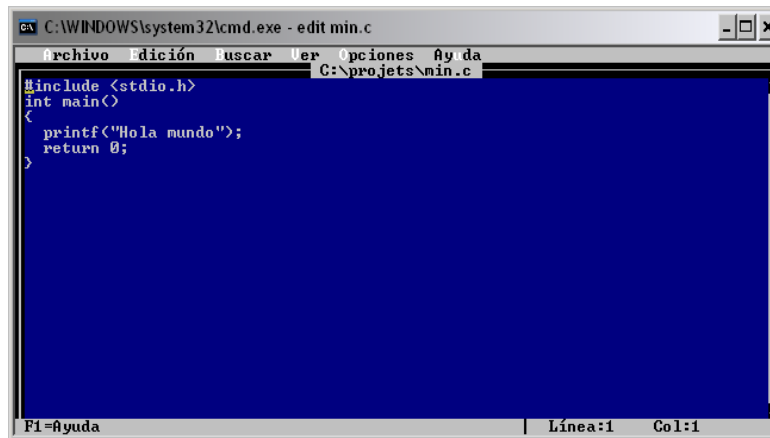
a seguido de enter (como el programa no hace nada, tampoco vemos nada).

El programa que se mostró es el programa mínimo que se puede hacer en lenguaje C, y tiene algo que tendrán todos los programas que hagamos en lenguaje C, la función *main* (posteriormente definiremos que es una función), allí se indica que es lo que queremos que el programa haga, en el caso del ejemplo solo queremos que le envíe un cero al sistema operativo y nada más.

Modifiquemos el programa para que luzca como se muestra en la figura 2.8.

Si volvemos a compilar el programa para generar el ejecutable y lo ejecutamos debemos obtener el famoso mensaje en nuestra consola. Este programa cuenta ya, con ingredientes adicionales, `#include <stdio.h>` y `printf("Hola mundo")`, la primera le indica al compilador que se van a usar instrucciones de entrada/salida estándar, y es lo que se llama inclusión de un archivo de cabecera y la segunda hace que se muestre en la pantalla el mensaje *hola mundo*.

Supongo que se habrá preguntado como hacer para que su programa ejecutable no se llame *a.exe*, para ello podemos usar la siguiente orden `gcc min.c -o min` de esta forma el archivo de salida se llamará *min.exe*.



```
#include <stdio.h>
int main()
{
    printf("Hola mundo");
    return 0;
}
```

Figura 2.8: Hola mundo.

2.2.3. Elementos básicos

Se mostrarán los elementos básicos del lenguaje y un par de ejemplos que incluya lo mencionado.

Identificadores

Cuando estamos escribiendo un programa es necesario que le demos nombre a las cosas con el fin de poder utilizarlas cuando las necesitemos, en lenguaje C existen ciertas reglas para los identificadores:

- Puede tener uno o varios caracteres.
- El primer carácter debe ser una letra o un guión bajo (_).
- No puede contener espacios.

Lenguaje C diferencia entre mayúsculas y minúsculas, es decir el identificador *x* es diferente de *X*.

Tipos de datos

En lenguaje C, los valores literales, las expresiones y las variables pueden tener los siguientes tipos de datos:

Enteros. Los datos de tipo entero son valores negativos, positivos o cero que no tienen parte decimal. Ejemplos de enteros validos en C son:

3, 12, -3659, +6954

Las variables enteras en lenguaje C se declaran como *int*. Los valores máximos y mínimos de que pueden almacenar dependen de la plataforma, aunque generalmente son de 16 o 32 bits.

Reales. Los tipos de datos reales permiten representar precisamente ese tipo de datos, en informática son denominados también números en punto flotante. Lenguaje C provee dos tipos de datos para representar reales:

- *float*. Números en punto flotante de 32 bits, para representar datos en el rango: $1.18 \cdot 10^{-38} < |X| < 3.40 \cdot 10^{38}$. Ejemplos de valores tipo float:

345.2f 45.6f 3.5e3f

La f al final indica que el número es un flotante de 32 bits.

- *double*. Números en punto flotante de 64 bits, para representar datos en el rango: $2.23 \cdot 10^{-308} < |X| < 1.79 \cdot 10^{308}$.

345.2f 45.6f 3.5e3f

Caracteres. Se usan para representar datos alfanuméricos, letras, símbolos, dígitos. Los literales tipo caracter se reconocen por que se encierran entre comillas simples `'a'`, `'@'`, `'5'`. Las variables se declaran como *char* y almacenan el código ASCII de los símbolos. Las variables tipo *char* pueden verse también como variables enteras de 8 bits.

void Es un tipo de datos útil para la comprobación de tipos y manejo de punteros, lo veremos más adelante, como dato curioso tiene 0 bits de longitud.

En el código 2 se muestran ejemplos de declaración de variables de los tipos comentados.

Código 2 Declaración de variables.

```
/* Esto es un comentario de varias lineas
   Este programa muestra como declarar
   de distintos tipos
*/
int main()
{
    int variable_entera; //una variable entera
    float flotante; // otro comentario de una línea
    double variable_doble;
    char car1,car2='@';

    variable_entera=325;
    variable_doble=3.564;
    flotante=3.2f;
    car1='a';
    return 0;
}
```

Expresiones

Las expresiones son combinaciones de operadores y operandos, Los operandos pueden ser literales, variables o el resultado de otra expresión. Los operandos pueden ser aritmeticos, de relación o lógicos. Según el resultado de la evaluación de una expresión, esas pueden ser aritmeticas o lógicas.

Expresiones aritméticas Dan como resultado un dato aritmetico, ya sea entero o real, dependiendo del tipo de dato que en ellas aparezca, por ejemplo

| |
|---|
| <pre>3*5+7 -->int 3.5f+7.0f -->float 3.5+7.0 -->double</pre> |
|---|

Expresiones lógicas

Variables

Modificadores de tipo

Entrada/Salida básica

Sentencias de control

2.3. Estructura de un programa en C

Los programas en lenguaje C pueden compuestos por uno o varios archivos que contienen código fuente, la estructura típica de un programa compuesto de un solo archivo es la mostrada en el listado 3.

Los programas inician su ejecución por la función `main()`.

Código 3 Ejemplo de la estructura de un programa en C.

```
#include <stdio.h>----->Archivos de cabecera
#include <stdlib.h>

#define SIZE 50 ----->Constantes simbolicas o macros
#define LONG 30

int variable_entera_global;
/*****
 * Prototipo de las funciones */
/*****/
/** Muestra un estudiante */
void mostrar_estudiante (int, char [][][LONG], unsigned long [],unsigned long[]);

/** Lista todos los estudiantes */
void listar_estudiantes (int, char [][][LONG], unsigned long [],unsigned long[]);
/** Ingresa un estudiante nuevo.*/
void ingresar_estudiante (int*, char [][][LONG], unsigned long [],unsigned long[]);

/** Función principal */
int main()
{
    int variable_entera_local;
    .....
    return 0;
}
/**
implementacion de las funciones
*/
void listar_estudiantes (int, char [][][LONG], unsigned long [],unsigned long[])
{
    int variable_entera_local;
    .....
}
void ingresar_estudiante (int*, char [][][LONG], unsigned long [],unsigned long[])
{
    .....
}
```

Capítulo 3

Estructuras

Las estructuras en lenguaje C son elementos que nos permiten organizar y manipular de una mejor manera la información con la que un programa debe tratar, permiten albergar bajo un mismo identificador información que a nivel informático es de tipo diferente, pero que posiblemente a nivel lógico está relacionada (la información de un libro en una biblioteca, de un vehículo en un taller, de un polígono en un programa de dibujo, etc), el objeto de éste capítulo es, mediante un ejemplo sencillo, mostrar la utilidad de las mismas.

Para realizar el estudio de las estructuras (llamadas también registros en otros lenguajes), se mostrará un ejemplo que sirva de motivación, y que permita ver las bondades de su uso en los programas, para ello se hace necesario un conocimiento mínimo de arreglos y funciones.

Supongamos que nos encargan escribir un programa simple para el manejo de la información de los niños de un jardín infantil, para ello nos solicitan que el programa como mínimo debe manejar la siguiente información por cada niño:

- El código.
- El nombre.
- Un teléfono de contacto.

¿ Cómo podemos almacenar dicha información ?, ¿ Cómo relacionamos esta información de tal modo que podamos buscar el teléfono de un niño o niña por su código o su nombre ?

Respondamos algunas preguntas antes de responder las anteriores, ¿

qué necesito para almacenar el *código* de un sólo niño ? ¿ qué necesito para el nombre ? bueno, pues si el código lo representamos con un número entero, podemos utilizar un dato tipo `long`, para el teléfono también es posible utilizar una variable de éste tipo, pero para el nombre es necesario una *cadena* . ¿ Si para almacenar el código de un solo niño usamos un `long` , qué usamos para almacenar varios códigos? un conjunto de variables tipo `long` sería una opción viable, es decir un arreglo, como es necesario tener no solo los códigos, sino también los nombres y los teléfonos, haremos uso de dos arreglos más, del modo en que se muestra en la figura 3.1.

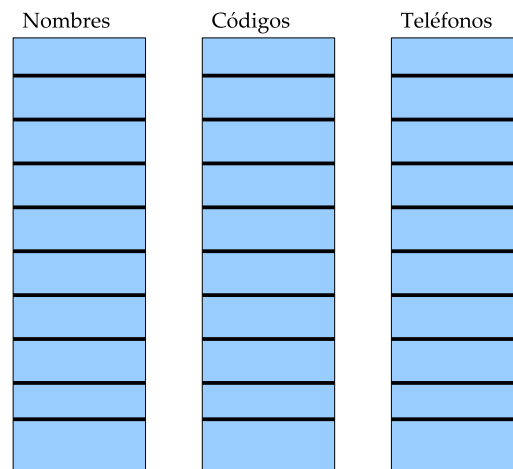


Figura 3.1: Arreglos paralelos.

Ya se sabe como representar la información, pero los arreglos no tienen relación alguna, ¿ cómo relacionamos el contenido de estos tres arreglos ? una alternativa consiste en ubicar la información de un niño en las mismas posiciones en cada uno de ellos, tal y como se muestra en la figura.

ahora ¿cómo manipulamos la información almacenada en estos tres arreglos ? con éste objetivo veamos el código 4, en él aparecen los prototipos de algunas funciones y un programa principal en el que se muestra como usarlas.

En el código 5 se muestra la implementación de las funciones antes mencionadas, es claro que estás funciones dan un soporte mínimo para empezar a construir el programa que nos proponemos realizar pero nos desviaremos de este camino y veremos como construir lo que hemos hecho

Las estructuras permiten unir información relacionada.

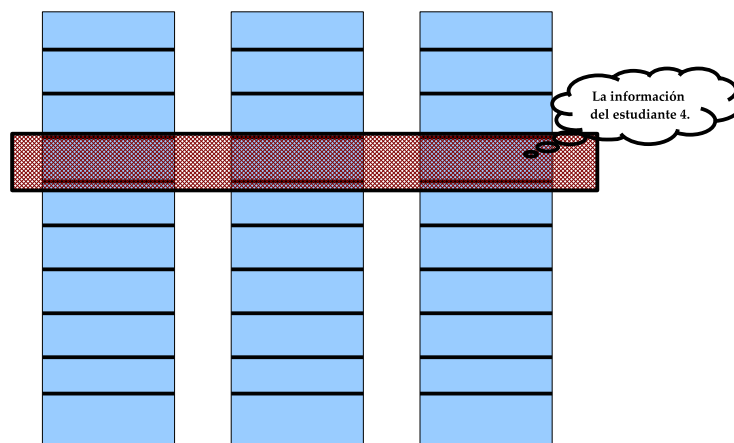


Figura 3.2: Ubicación de la información de un niño o niña.

| Estudiante |
|-----------------|
| +codigo: long |
| +nombre: char[] |
| +telefono: long |

Figura 3.3: La estructura Estudiante.

El objetivo de las estructuras (`struct`) es poder unir en un solo objeto, información que esta relacionada de alguna forma, con el fin de poder manipularla fácilmente mediante un mismo identificador, una representación gráfica de una estructura que representa un o una estudiante se muestra en la figura 3.3.

Generalmente cuando se crea una estructura se piensa en cuales son las características que nos interesan de un objeto dado, cada una de éstas características se implementa mediante lo que se denomina un *campo*, cada campo tiene un tipo (`int`, `float`, `..struct !...`), puede ser un escalar o un vector (llamemosle *dimensionalidad*) y además para poder diferenciarlo de otro tiene un nombre. Cuando se declara una estructura no se esta creando una variable en la cual guardar información, lo que se hace es indicar cuales van a ser las propiedades (campos) de las variables de ese tipo, por ejemplo, si pensamos en el conjunto de los números enteros sabemos que son todos aquellos números que no tienen parte dec-

Cuando se declara una estructura no se esta creando una variable.

imal, esa es una característica de ellos, pero no estamos hablando de un entero en particular, lo mismo ocurre con las estructuras.

Por ejemplo la implementación en lenguaje C de la estructura de la figura 3.3 puede verse en el código 6.

Comentemos algunas cosas de los códigos 6 y 7:

`typedef` permite crear un alias para un tipo de datos.

- La palabra clave `struct` se utiliza para declarar una estructura (*un tipo de datos nuevo*).
- `typedef` es utilizado para darle un alias a `struct _Estudiante`, de esta manera cuando declaramos una variable o utilizamos como parametro formal este tipo de datos no tenemos que escribir `struct _Estudiante x` sino `Estudiante x` (*esto no es necesario en C++*).
- Para acceder a cada uno de los campos de una variable tipo estructura se utiliza el operador punto (".") seguido por el nombre del campo.
- La implementación realizada con estructuras cumple las mismas funciones que la basada en arreglos paralelos, pero tiene las siguientes ventajas:
 - Los prototipos no dependen del número de campos.
 - Solo existe un arreglo, en el cual cada elemento es una estructura.
 - La información está más compacta para el programador (*una de las ventajas de trabajar con las estructuras!*).

Para aquellos que necesitan recordar algo sobre el paso de parámetros. ¿Por qué las funciones pueden modificar el contenido de la variable `estudiantes` si en C todas las variables se pasan por valor? La respuesta es que la variable es un arreglo y los arreglos se pasan a las funciones por puntero (la dirección de inicio del mismo).

Las variables tipo estructura funcionan como las variables de otros tipos, es decir, podemos igualar dos variables del mismo tipo, crear arreglos de ellas (en el ejemplo la variable `estudiantes` es un arreglo), pasarlas a funciones y retornarlas de las mismas, es necesario tener en cuenta que cuando se igualan dos estructuras el contenido de una se copia en la otra

campo a campo, lo cual es un problema que se debe tener en cuenta sobre todo cuando se trabaja en sistemas en los cuales los recursos son limitados (como sistemas embebidos o microcontroladores) donde el uso de la memoria y el tiempo de ejecución de los programas deben optimizarse, de este modo el uso de estructuras como argumentos o valores de retorno no son una práctica muy común, por ello es habitual utilizar punteros a las variables tipo estructura en lugar de los valores de las mismas, y utilizar el operador flecha "->" para acceder a los campos, en el código 8 se muestra lo que se acaba de indicar.

El uso de estructuras es amplio en el desarrollo de aplicaciones en lenguaje C, especialmente para la construcción de *tipos abstractos de datos*.

Las variables tipo estructura se usan como las de cualquier otro tipo.

3.1. Resumen

- Las estructuras se utilizan para almacenar información relacionada en un mismo objeto informático.
 - Los campos de una variable tipo estructura pueden ser de cualquier tipo incluso una estructura también (*mientras no sea del mismo tipo que la estructura contenedora pues se requeriría un tener un sistema con memoria infinita ¿?*).
 - Los campos de una variable tipo estructura se pueden usar como una variable normal, se pueden enviar a funciones, su valor puede ser retornado de una función, etc; y si estos son de los tipos básicos predefinidos (*enteros, flotantes, caracteres*) se pueden utilizar tanto en `printf` como en `scanf`.
 - Una variable tipo estructura puede ser enviada como argumento a una función.
 - Las funciones pueden retornar variables del tipo estructura.
 - Generalmente no se envían o reciben variables tipo estructura sino punteros a las mismas por cuestiones de eficiencia.
 - Para acceder a los campos de una variable tipo estructura se utiliza el operador punto (ejemplo: `variable.campo`).
-

- Para acceder a los campos de una variable tipo estructura mediante un puntero a ella es mejor utilizar el operador flecha (ejemplo: `puntero->campo`) en lugar del operador punto.
- Es posible crear arreglos de estructuras del mismo modo que con los tipos de datos predefinidos.

3.2. Ejercicios

1. Escriba estructuras para representar lo siguiente:
 - a) Resistencia
 - b) Dimensiones de un objeto
 - c) Vehículo en un taller
 - d) Vehículo en un parqueadero.
 - e) Libro en una biblioteca
 - f) Persona en un hospital
 - g) Persona en una universidad
 2. Escriba una estructura para representar números racionales ($n = \frac{a}{b}$) y escriba funciones para realizar lo siguiente:
 - a) Suma
 - b) Multiplicación por entero.
 - c) Multiplicación por otra fracción.
 - d) División
 - e) Potenciación
 - f) Simplificación (es necesario encontrar factores comunes)
 3. Utilizando como base el ejemplo planteado, escriba funciones adicionales para:
 - a) Mostrar de forma ordenada toda la información de un niño con un recuadro en la pantalla
-

- b) Agragar un niño al arreglo validando lo siguiente:
 - 1) El nombre no debe empezar por espacios y no debe quedar vacío
 - 2) El telefono de contacto no debe contener espacios ni letras.
 - 3) El código no debe tener espacios.
 - 4. Mostrar pantallazos con una lista bien presentada hasta mostrar todos los niños y niñas
 - 5. Buscar en el arreglo de niños y niñas uno que tenga un determinado código.
 - 6. Utilizando las funciones anteriores escribir un programa interactivo que permita administrar la información de los niños.
-

Código 4 Ejemplo de uso de arreglos paralelos

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 50 /*número máximo de niños */
#define LONG 30 /*longitud de las cadenas */

/*****
/* Prototipo de las funciones */
*****/
/*****
/* Muestra un estudiante */
*****/
void mostrar_estudiante (int, char [][][LONG], unsigned long [],unsigned long[]);

/****
/* Lista todos los estudiantes */
*****/
void listar_estudiantes (int, char [][][LONG], unsigned long [],unsigned long[]);
/****
/* Ingresa un estudiante nuevo.*/
*****/
void ingresar_estudiante (int*, char [][][LONG], unsigned long [],unsigned long[]);

/****
/* Función principal */
*****/
int main()
{
    /* Las variables con los datos de niños o niñas */
    unsigned long codigos[SIZE];
    unsigned long telefonos[SIZE];
    char nombres[SIZE][LONG];
    int cursor=0;

    /* leer un estudiante del teclado */
    ingresar_estudiante(&cursor,nombres, codigos,telefonos);

    /* leer otro estudiante del teclado */
    ingresar_estudiante(&cursor,nombres, codigos,telefonos);

    /* listar los estudiantes */
    listar_estudiantes(cursor,nombres, codigos,telefonos);

    /* vaciar el buffer de entrada para que lea bien el caracter */
    fflush(stdin);
    getchar();
    return 0;
}
```

Código 5 Ejemplo de uso de arreglos paralelos (continuación)

```
void mostrar_estudiante(int c, char noms[][LONG], unsigned long cods[], unsigned long tel[])
{
    printf("%ld\t%s\t\t\t%ld\n", cods[c], noms[c], tel[c]);
}
void listar_estudiantes(int c, char noms[][LONG], unsigned long cods[], unsigned long tel[])
{
    int i;
    printf("Codigo\tNombre\t\t\tTelefono\n");
    for (i=0; i<c; i++)
    {
        mostrar_estudiante(i, noms, cods, tel);
    }
}
void ingresar_estudiante(int* pc, char noms[][LONG], unsigned long cods[], unsigned long tels[])
{
    char buffer[255];
    printf("Nombre -->");
    fflush(stdin);
    gets(buffer);
    strncpy(noms[*pc], buffer, LONG);
    printf("Codigo -->");
    scanf("%ld", &cods[*pc]);
    printf("Telefono -->");
    scanf("%ld", &tels[*pc]);
    (*pc)++;
}
```

Código 6 Ejemplo con estructuras.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 50 /* número máximo de niños o niñas */
#define LONG 30 /* longitud de las cadenas */
struct _Estudiante
{
    char nombre[LONG];
    unsigned long codigo;
    unsigned long tel;
};
typedef struct _Estudiante Estudiante; /*Para no tener que usar struct siempre
(en C)*/
/*****
/* Prototipo de las funciones */
*****/
/**
Muestra un estudiante
*/
void mostrar_estudiante (Estudiante);
/**
Lista todos los estudiantes
*/
void listar_estudiantes (int, Estudiante[]);
/**
Ingresa un estudiante nuevo.
*/
void ingresar_estudiante (int*, Estudiante[]);
/**
Función principal
*/
int main()
{
    /* Las variables con los datos de niños o niñas */
    Estudiante mis_estudiantes[SIZE];
    int cursor=0;
    /* leer un estudiante del teclado */
    ingresar_estudiante(&cursor,mis_estudiantes);
    /* leer otro estudiante del teclado */
    ingresar_estudiante(&cursor,mis_estudiantes);
    /* listar los estudiantes */
    listar_estudiantes(cursor,mis_estudiantes);
    /* vaciar el buffer de entrada para que lea bien el caracter */
    fflush(stdin);
    getchar();
    return 0;
}
```

Código 7 Ejemplo con estructuras(continuación).

```
/**
Muestra un estudiante
*/
void mostrar_estudiante (Estudiante e)
{
printf("%lu\t%s\t\t\t%lu\n",e.codigo,e.nombre,e.tel);
}
/**
Lista todos los estudiantes
*/
void listar_estudiantes (int c, Estudiante estudiantes[])
{
int i;
printf("Codigo\tNombre\t\t\tTelefono\n");
for (i=0;i<c;i++)
{
mostrar_estudiante(estudiantes[i]);
}
}
/**
Ingresa un estudiante nuevo.
*/
void ingresar_estudiante (int* pc, Estudiante estudiantes[])
{
char buffer[255];
printf("Nombre -->");
fflush(stdin);
gets(buffer);
strncpy(estudiantes[*pc].nombre,buffer,LONG);
printf("Codigo -->");
scanf("%lu",&estudiantes[*pc].codigo);
printf("Telefono -->");
scanf("%lu",&estudiantes[*pc].tel);
(*pc)++;
}
```

Código 8 Punteros a estructuras como argumentos a funciones.

```
#include <stdio.h>

#define LONG 100

struct _Articulo
{
    unsigned long codigo;
    char nombre[LONG];
};

typedef struct _Articulo Articulo;

void articulo_mostrar(Articulo* arti)
{
    printf(" %lu\t%s\n",arti->codigo,arti->nombre);
}

void articulo_ingresar(Articulo* arti)
{
    char buffer[255];
    printf("Nombre -->");
    fflush(stdin);
    gets(buffer);

    /* arti->nombre remplaza (*arti).nombre */
    strncpy(arti->nombre,buffer,LONG);
    printf("Codigo -->");
    scanf(" %lu",arti->codigo);
}
}
```

Capítulo 4

Punteros y memoria dinámica

De los punteros se tocaron algunas características en el apartado de funciones, exactamente, cuando se describió la forma en la que se pasan los parametros a las mismas. Aquí no solo recordaremos eso, sino que tocaremos el tema que les concede toda la potencia y por ende todo el respeto, la *reserva dinámica de memoria*.

Para iniciar el estudio de punteros es necesario regresarnos un poco en el tiempo y recordar que es una variable, de una manera simple una variable es un espacio (*cajón*) en el cual podemos almacenar información, a cada variable que nosotros creamos le asignamos un nombre (*identificador*) que permite distinguirla de otras, y almacenar la información en el sitio correcto cada vez que lo necesitemos, una representación gráfica de lo dicho se muestra en la figura 4.1.

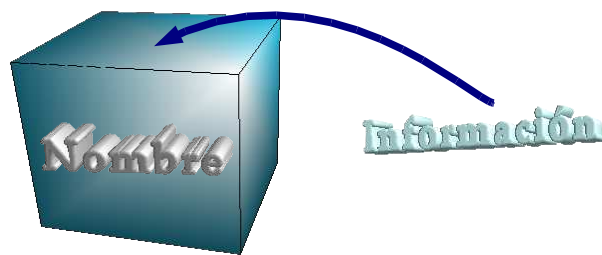


Figura 4.1: Una variable.

Además del nombre cada variable posee otras dos características im-

portantes, la *dirección* y el *dato* que contiene (existen lenguajes en los que las variables pueden no tener nombre). La dirección de memoria de una variable es como la coordenada en la que se encuentra dentro de la memoria del computador, y generalmente se dá en numeros hexacimales o binarios, arrancan en cero y su tamaño (número de bits) depende de la cantidad de memoria máxima que el procesador pueda direccionar.

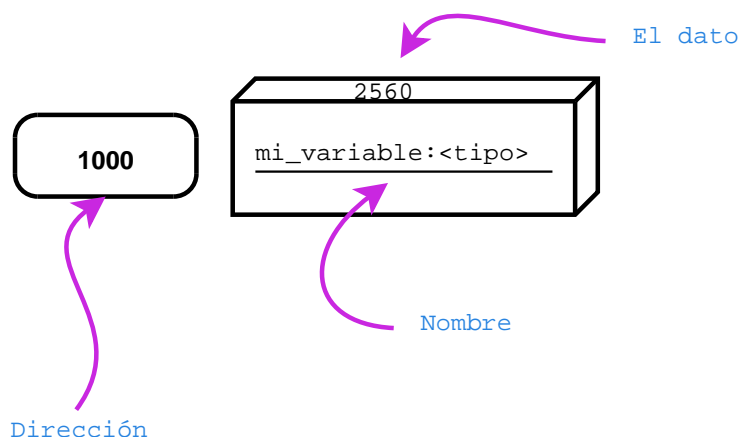


Figura 4.2: Las tres características de una variable.

Los *punteros* son variables cuyos contenidos son direcciones de memoria, en el código 9 se muestra como se declara una variable, un puntero y se imprime la dirección de la misma, para obtener la dirección de la variable se hace uso del operador `&`.

Los *punteros* son variables cuyos contenidos son direcciones de memoria.

La memoria de un sistema puede verse como conjunto de contenedores de información, cada contenedor tiene un cierto tamaño, que depende del sistema en particular y que generalmente es de 8,16, 32 o 64 bits, para las discusiones siguientes supondremos que cada contenedor es de 1 byte (8 bits), además de lo anterior cada contenedor tiene un número único que lo identifica llamado *dirección*.

Dependiendo del tipo de dato que una variable contenga ocupa una o mas posiciones de memoria, la dirección de una variable que ocupa más de una localidad es la dirección en la que inicia, en la figura 4.4 se muestra lo indicado. Recordemos que para saber el número de bytes que ocupa un tipo de dato podemos usar el operador *sizeof*.

Los espacios de memoria de un sistema se pueden dividir en básica-

Código 9 La dirección de una variable.

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

/* Ejemplo para mostrar la dirección de una variable */

int main()
{
    int a;      /* declaración de una variable entera */
    int* ptr;   /* declaración de una variable que es un puntero a entero,
                /*es decir puede almacenar la dirección de una variable tipo entero */
    a=10;
    printf("La direccion de la variable a es:%p y vale:%d\n",&a,a);
    ptr=&a;
    printf("El puntero almacena la direccion:%p\n",ptr);
    return 0;
}
```

mente tres grupos:

- Espacio global de aplicación.
- La pila o *stack*.
- El montículo o *heap*.

La figura 4.5 muestra la estructura de memoria de sistema típico.

4.1. Los espacios de memoria.

4.1.1. Espacio global.

Es el espacio en el que se almacenan todas las variables globales de la aplicación, recordemos que en lenguaje C, las variables globales son aquellas que se declaran fuera de todas las funciones, las variables de este tipo son asignadas en tiempo de compilación y duran todo el tiempo de vida de la aplicación.

4.1.2. La pila.

La pila es un espacio de memoria muy importante ya que es utilizada para, el paso de parámetros, la creación de variables locales, almacenar la

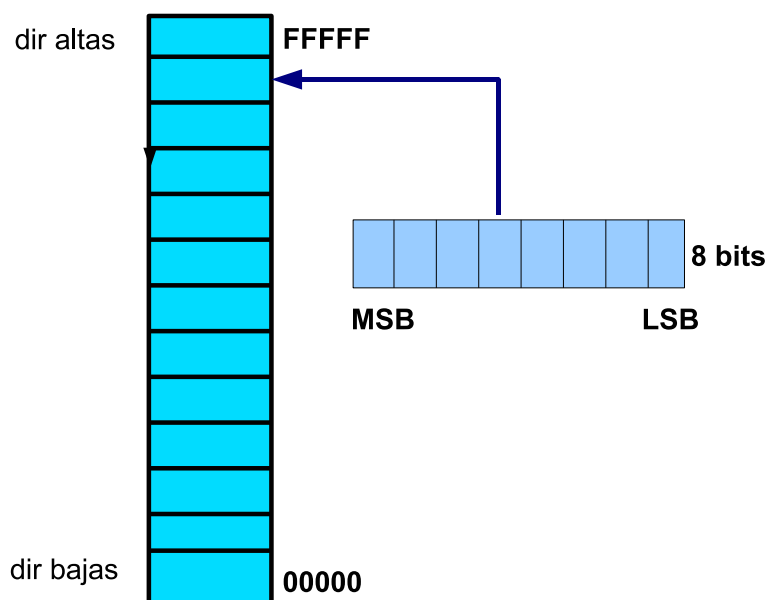


Figura 4.3: Mapa de memoria en un procesador 8086/8088.

dirección de retorno cuando se llama a una función y para que las funciones pongan allí sus valores de retorno, entre otras cosas.

Para aclarar lo que es una pila, se puede hacer uso de la analogía con una pila de platos para lavar, cada vez que se agrega un plato al conjunto de platos para lavar se pone sobre los que ya están (*apilar*) y cada vez que se necesita uno para lavarlo se toma de la parte superior del conjunto (*desapilar*). Una estructura de este tipo es llamada estructura *LIFO* (*last in first out*). Las pilas son estructuras muy usadas en computación y especialmente por los sistemas para almacenar información concerniente a la aplicación que se está ejecutando. Generalmente en la operación de una pila existe un índice o apuntador (*sp*) que se modifica cuando se apilan o desapilan datos en ella, aunque se ha dicho que las pilas son estructuras de datos LIFO, en la operación normal de un sistema se puede acceder a la zona de memoria de pila como si de un arreglo se tratara usando posiciones relativas al índice o apuntador de pila (*sp*). En la figura 4.6 se muestra un ejemplo de una pila.

Como se había mencionado los compiladores utilizan el espacio de pila generalmente para pasar argumentos a funciones, almacenar *marcos de pila* (información necesaria para que el procesador continúe su ejecución

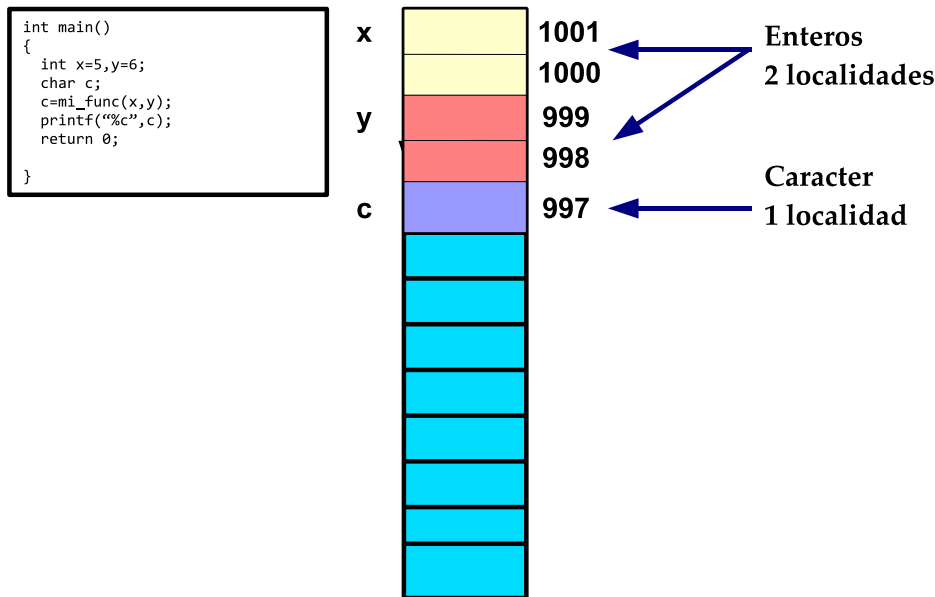


Figura 4.4: Tamaño de variables.

de modo normal después de retornar de la función llamada) y variables locales.

Lo siguiente es, a grandes rasgos, lo que ocurre cuando se realiza un llamado a una función en lenguaje C:

- Si la función tiene parámetros, los valores de estos se almacenan en la pila (ojo , los valores), en C se copia en la pila primero el último parámetro y por último el primero.
- Se almacena el *marco de pila* en la pila :). La dirección de retorno, y también algunos de los registros del procesador.
- Se crean las variables locales de la función, en la pila.
- La función realiza su trabajo ;).
- Si la función retorna algún valor pone el dato de retorno en la pila en un espacio previamente establecido para tal fin.
- La función retorna al lugar donde fue llamada.

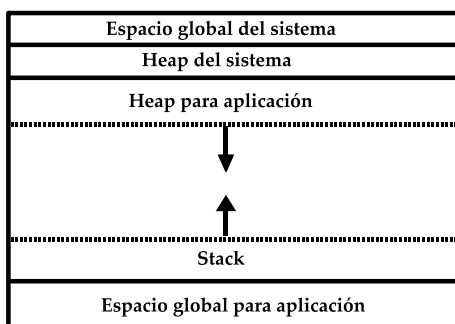


Figura 4.5: Estructura típica memoria.

Realmente la cosa es más compleja si se mira a un nivel más bajo, pero, lo que nos interesa aquí es ver que la pila es una zona de memoria muy importante, en la figura 4.7 se muestra un ejemplo de como se copian los datos en la pila cuando se realiza la llamada a una función en lenguaje C. Los valores de las variables x e y se copian en las posiciones de memoria utilizadas para recibir los parámetros (a y b), es claro que no se envían las variables originales se envían copias de los valores almacenados en ellas.

4.1.3. El heap o montículo.

Es la zona de memoria en la que se almacenan las *variables dinámicas*, aquellas que son creadas y destruidas por la aplicación en tiempo de ejecución. Es muy importante tener presente que las variables creadas en el heap deben ser destruidas de forma explícita por el programador si no la aplicación tendrá lo que se denomina una fuga de memoria, causa de muchos de los dolores de cabeza cuando se trabaja con memoria dinámica.

4.2. Los punteros.

Los punteros son utilizados generalmente para las siguientes cosas:

- Referenciar una variable automática¹, permite tener otro camino para interactuar con ella además de su identificador.

¹Variables que se declaran dentro de una función y se destruyen al terminar su ejecución

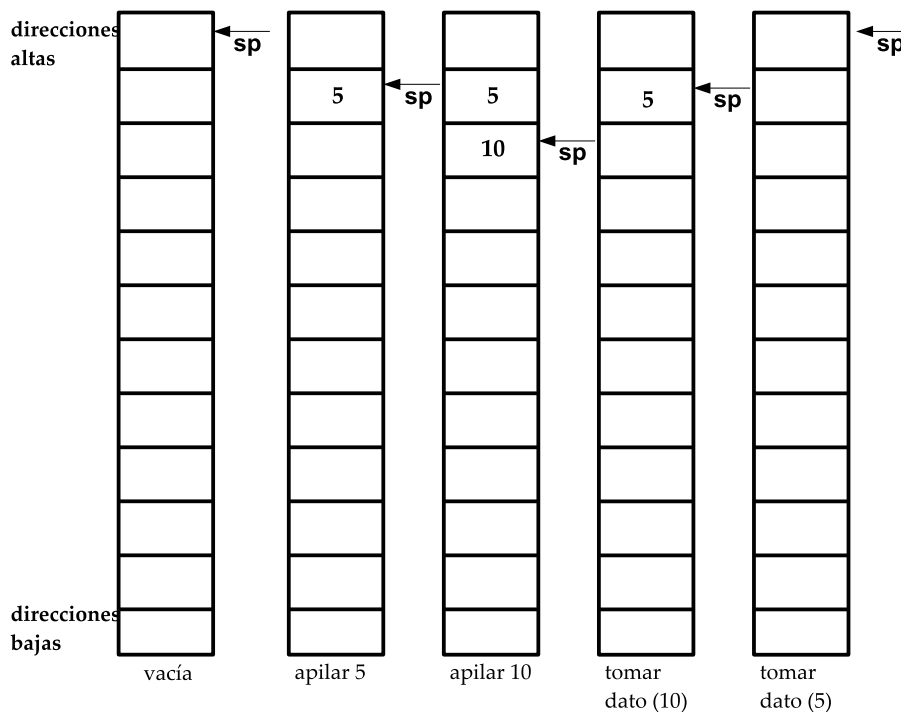


Figura 4.6: Ejemplo de operaciones con pila.

- Referenciar una zona de memoria reservada por el programador en tiempo de ejecución (*ver mas adelante*), es decir crear variables en tiempo de ejecución.

Una de las formas típicas de representar un puntero a una variable se muestra en la figura 4.8, además un puntero puede referenciar ya sea a una variable escalar o el inicio de un arreglo (es algo lógico, pues una variable escalar puede verse como un arreglo de un solo elemento), esto se puede observar en la figura 4.9.

Podemos modificar el valor de una variable mediante su identificador o mediante su dirección.

4.2.1. Punteros como referencia a variables.

Todas las variables creadas en un programa ya sean globales o locales a una función, tienen las características mencionadas anteriormente, *nombre*, *dato* y *dirección*, como la finalidad de los punteros es almacenar direcciones de memoria, podemos almacenar la dirección de una variable en un pun-

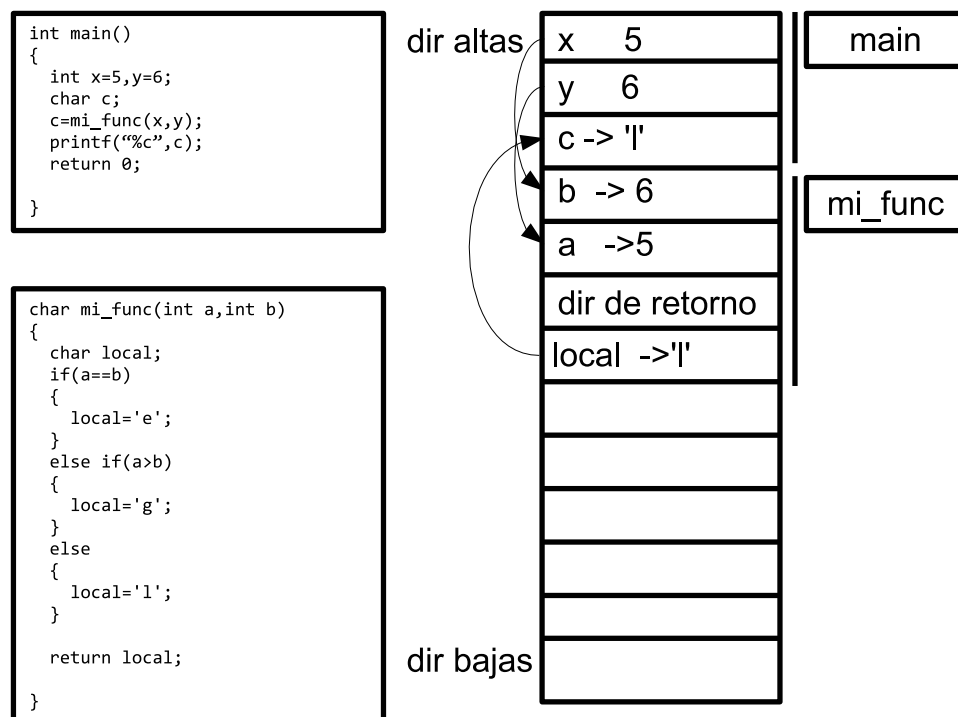


Figura 4.7: Paso de parámetros.

tero, y posteriormente usarlo para acceder a la misma de forma indirecta. Veamos un ejemplo de esto, en la figura 4.10 existen cuatro variables tipo entero, almacenadas en las direcciones (ficticias) 100 hasta 106 y una de tipo puntero a entero que apunta a la variable *b* y cuya dirección es 200, la figura muestra que en el puntero esta almacenada la dirección de la variable *b*.

Con el operador

"&" se obtiene la dirección de una variable y con el operador "*" se obtiene la variable a la que apunta un puntero.

En el código 10 se muestra como se utiliza un puntero para modificar el valor de, en primera instancia la variable *b* y luego de la variable *d*; es de notar que se utiliza el operador "&" para obtener la dirección de una variable, la cual es posible almacenar en un puntero, y luego, ya con el puntero referenciando la variable requerida, se utiliza el operador "*" sobre él para obtener acceso a la variable propiamente dicha.

Cuando se declara un puntero es necesario indicarle el tipo de dato al que va a apuntar (no necesariamente un tipo básico, puede ser tipo estructura), en el código de ejemplo se muestra que para declarar un puntero a un entero se utiliza `int* ptr`, que indica que apuntará a un entero, de

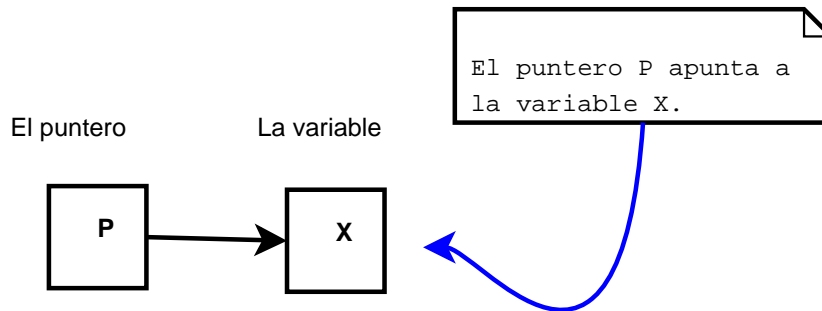


Figura 4.8: Representación típica de un puntero a una variable.

Código 10 Usando un puntero para alterar el contenido de una variable

```
#include <stdio.h>
int main()
{
    int a,b,c,d;
    int* ptr;
    a=10;
    b=15;
    c=20;
    d=30;
    printf("b vale: %d y d vale: %d\n",b,d);
    ptr=&b;
    *ptr=16;
    ptr=&d;
    *ptr=31;
    printf("b vale: %d y d vale: %d\n",b,d);
    return 0;
}
```

igual forma se hace para cualquier tipo de dato.

¿ Para que necesitamos otra forma de acceder al valor de una variable ? pues imaginemos que se requiere de una función que necesite retornar más de un valor a la función llamante, recordemos que en lenguaje C todos los argumentos a excepción de los arreglos, se pasan por valor, es decir se envía una copia del valor de la variable, lo cual indica que no podemos modificar el valor de la variable original, esta situación tiene varias soluciones:

- Que la función llamada retorne una estructura cuyos campos sean todos los datos que necesita la función llamante, el inconveniente de esta alternativa es que si esto lo necesitamos para muchas funciones, entonces, tendríamos que crear muchas estructuras con este fin y no

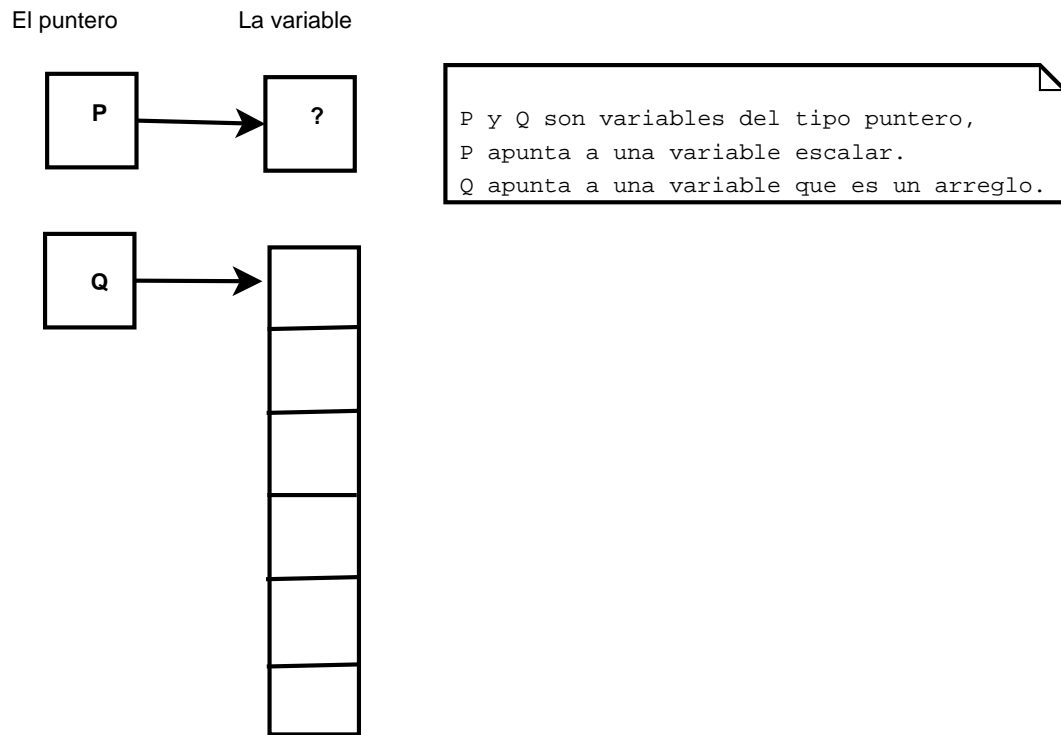


Figura 4.9: Puntero a escalar y puntero a arreglo.

con el de representar un objeto del mundo del problema (es decir nos llenaríamos de estructuras que no están relacionadas con el problema).

- La otra opción es enviar las direcciones de las variables, de tal modo que la función pueda modificar su valor y de esta manera simular el retorno de varios valores (*parámetros por puntero*), en el código 11 se muestra un ejemplo de una función en la se simula el retorno de tres valores.

4.2.2. Memoria dinámica.

La mayoría de aplicaciones reales no tienen *idea* de cuantos objetos van a necesitar para cumplir su labor, por ejemplo, ¿máximo, cuantas ventanas

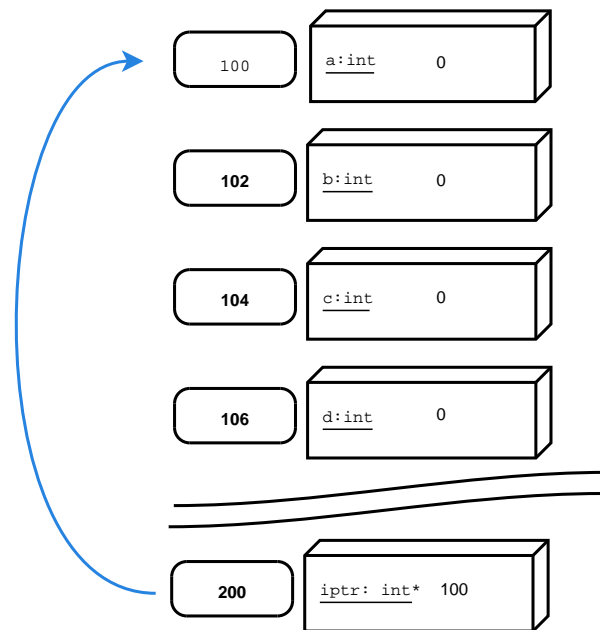


Figura 4.10: Un puntero a una variable.

de aplicación es posible tener abiertas en un PC?, el número debería depender de la cantidad de memoria RAM de la que disponga en la máquina y no del programa, esta es la justificación fundamental que se le da al uso de la memoria dinámica, veamos un problema típico, si estamos leyendo un archivo de texto y queremos almacenar todo su contenido en un arreglo, ¿qué tamaño debe tener este arreglo ?, ¿debe ser grande o pequeño ?.

Si es grande y leemos archivos pequeños estamos desperdiciando una buena cantidad de memoria que podría ser utilizada en otra cosa y si intentamos cargar archivos solo un poco más grandes que el tamaño que supusimos de los arreglos, estamos limitando el alcance del programa.

Para solucionar este tipo de problemas se recurre al uso de memoria dinámica, que no es otra cosa que crear variables, a medida que se van necesitando y no al inicio del programa como se hace generalmente.

Para crear una *variable dinámica* se le debe indicar al sistema que se necesita memoria para ella, y en caso de haber memoria disponible, él retornará la dirección de la variable. ¿Y que tipo de dato se necesita para almacenar una dirección? si, un puntero, por ello es necesario también

Código 11 Ejemplo de una función que *"retorna"* tres valores.

```
void mi_funcion(float* x, float* cuad, float* cub)
{
    *cuad=*x*x;
    *cub=*cuad*x;
}

int main()
{
    float a=7.0,b,c;
    mi_funcion(&a,&b,&c);
    printf("El valor%f, el cuadrado%f, el cubo%f",a,b,c);
    getchar();
    return 0;
}
```

declarar un puntero con el cual referenciar la variable creada.

Luego de creada la única forma de acceder a la variable creada, es mediante el puntero a ella (es decir mediante el uso de la dirección de la variable), utilizando para tal fin el operador de indirección *"*"*. Cuando la variable que se creó ya no es útil para el programa es necesario eliminarla de forma explícita. Los pasos típicos realizados con variables dinámicas se enumeran a continuación.

1. Declaración de un puntero (generalmente un variable automática) para referenciar la variable que se va a crear.
2. Creación de la variable y asociación al puntero del punto anterior.
3. Uso de la variable.
4. Borrado de la variable (no necesariamente en la misma función donde se creó).

Parece un poco ilógico que tengamos que crear una variable automática para crear una dinámica, ¿ por qué no sólo creamos la variable automática ?, si lo que queremos es crear una variable escalar realmente no estamos ganando gran cosa (a decir verdad si el puntero ocupa más espacio en memoria que la variable, estamos desperdiciando memoria), pero si lo que creamos es un arreglo si que estamos ganando, pues en primera instancia solo tendremos localizada una variable (el puntero) y luego con él accedemos al arreglo creado dinamicamente.

Veamos un ejemplo, vamos a crear una variable dinámica tipo `Articulo`, almacenamos algunos valores en sus campos, los mostraremos y luego eliminaremos la variable.

Código 12 Creación de una variable dinámica.

```
#include <stdlib.h>

struct _Articulo
{
    long codigo;
    char nombre[30];
    float precio;
    int iva;
};
typedef struct _Articulo Articulo;
int main()
{
    Articulo* articulo1=NULL; //inicialmente el puntero apunta a null.
    articulo = (Articulo*)malloc(sizeof(Articulo)); // Se crea una variable dinámica
                                                    // y se referencia con el puntero

    articulo->codigo=222424;
    strcpy(articulo->nombre,"puntillas");
    articulo->precio=2500;
    articulo->iva=0.16;

    .... // otros usos de la variable
    ....
    free(articulo); // Se borra (o libera el espacio ocupado por) la variable.
}
```

Comentemos algo de lo que aparece en el código 12, la función `malloc` se utiliza para reservar memoria para una variable, toma como parámetro el tamaño de la variable que se desea crear en bytes, para ello se utiliza el operador `sizeof`, el cual calcula el tamaño de un tipo de datos en bytes. La función `malloc` retorna un puntero tipo `void` (hablaremos de ellos más adelante) que en caso de haber memoria disponible apunta a la zona reservada de memoria para la variable que se deseaba crear, si el sistema no puede encontrar memoria para la variable entonces la función `malloc` retorna `NULL`, esto último es útil para saber si ocurre un error al momento de reservar memoria para una variable.

La otra función importante para trabajar con memoria dinámica es `free`, ella se encarga de *borrar* la variable referenciada por el puntero que se le pasa como parámetro, es muy importante que la zona de memoria se libere solo una vez, pues en cuando se libera la misma zona de memoria

más de una vez se corrompe la estructura de memoria, provocando que el programa que estamos haciendo tenga problemas.

Adicional a las funciones mencionadas existen otras funciones para trabajar con memoria dinamica en lenguaje C, *calloc*, *realloc*, la primera se usa también para reservar memoria y la segunda para cambiar el tamaño de una zona previamente reservada.

4.2.3. Punteros y arreglos.

En lenguaje C existe una estrecha relación entre los punteros y los arreglos, a para ser precisos se puede indicar que el identificador de un arreglo es un simbolo para representar la dirección en la que inicia el arreglo. Supongamos el arreglo mostrado en la figura 4.11

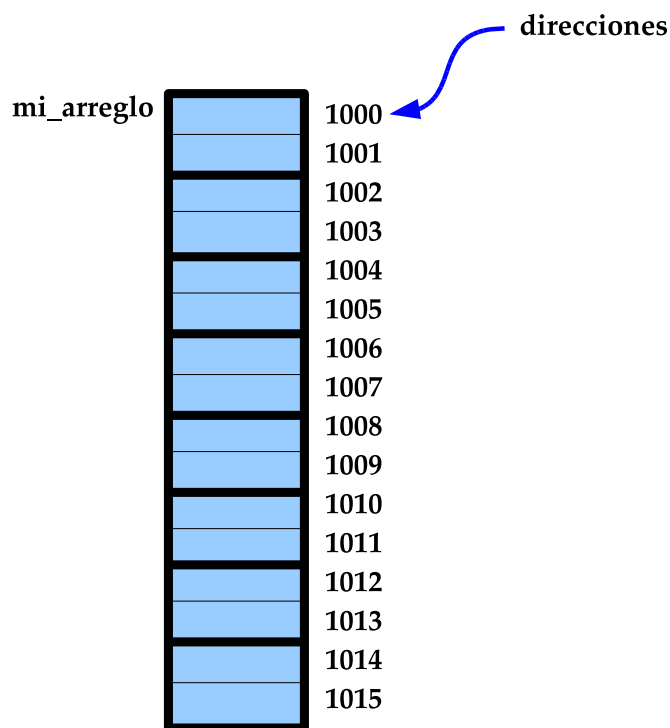


Figura 4.11: Un arreglo de enteros y sus direcciones.

Dicho arreglo inicia en la dirección de memoria 1000, cuando se escribe algo como:

```
x=mi_arreglo[3];
```

el compilador lo traduce como algo así:

```
x=*(1000+(3*sizeof(int)))
```

que es:

```
x=*(1006)
```

y significa que se esta accediendo al entero que inicia en la dirección 1006, que equivale a acceder al cuarto elemento del arreglo (el que tiene índice 3). En el caso anterior se supone que los enteros ocupan 2 bytes (`sizeof(int)` es 2) . Si ahora queremos utilizar un puntero para referenciar al arreglo podemos hacer uso de que *el nombre de un arreglo es un símbolo para su dirección de inicio*, para saber la dirección del mismo, del siguiente modo:

```
int* mi_puntero=mi_arreglo;
```

Lo anterior es equivalente a:

```
int* mi_puntero=&mi_arreglo[0];
```

Ahora podemos utilizar el puntero para acceder a cualquier elemento del arreglo, por ejemplo es posible escribir algo como lo siguiente:

```
printf("El dato en con el indice 3  
es %d",*(mi_puntero+3));
```

dado que el puntero tiene almacenado la dirección 1000, la línea anterior se traduce como:

```
printf("El dato en con el indice 3 es %d",  
*(1000+3*sizeof(int)));
```

Que como vemos es lo mismo que acceder mediante el identificador del arreglo al elemento con índice 3, es decir dado que `mi_puntero` apunta a `mi_arreglo`, entonces `mi_arreglo[3]` es equivalente a `*(mi_puntero+3)` y a `mi_puntero[3]`, si, aunque usted no lo crea.

4.2.4. Aritmética de punteros

Los valores que almacenan los punteros son direcciones de memoria, y admiten en lenguaje C las operaciones de suma y diferencia. Para que las operaciones anteriores existan es necesario que el operador *sizeof* este

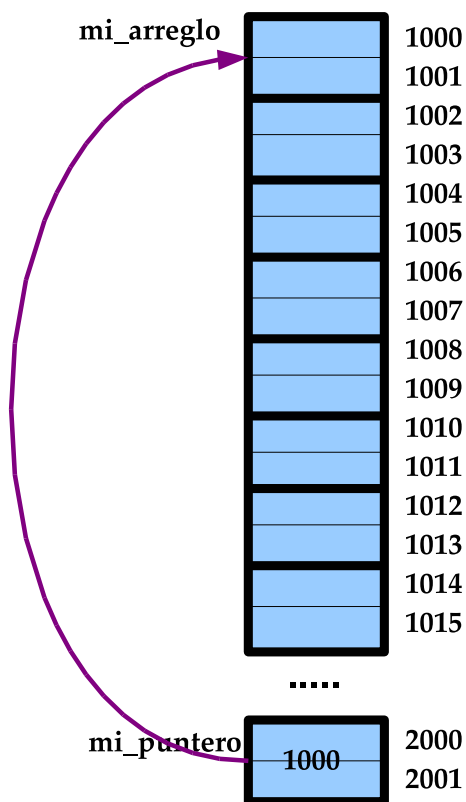


Figura 4.12: Puntero a un arreglo.

definido para el tipo de dato al que el puntero apunta, es decir que el tipo de dato tenga un tamaño en bytes, la razón de lo anterior es que una expresión como (suponiendo a p como un puntero a enteros):

$*(p+1)$

Se traduce como:

$*(p+1*\text{sizeof}(\text{int}))$

que indica que se obtenga la variable entera que aparece inmediatamente después de la que apunta p .

Con los punteros a void no es posible realizar aritmética de punteros. Pero es posible apuntar a variables de cualquier tipo.

Las operaciones de aritmética con punteros dan como resultado una dirección y dicho resultado depende del tipo de dato al que el puntero apunta (*el tipo de puntero*), por ejemplo si p es un puntero a enteros entonces $*(p+1)$, significa el entero siguiente al que apunta p (dos bytes

adelante), si p es un puntero a un flotante entonces significa el siguiente flotante (cuatro bytes adelante), etc.

También es posible usar expresiones como $p++$, $p--$, $++p$ o $--p$, sin embargo es importante notar que si p apunta a una variable dinámica estamos perdiendo la referencia a la misma al modificarlo, por ello si se piensan usar expresiones como las mencionadas, es mejor sacar una copia del contenido del puntero original en otro.

Existe un tipo de puntero especial, que puede ser usado para referenciar cualquier tipo de variable, pero que gracias a esto no existe una definición del operador `sizeof` para su tipo (no tiene tipo), el espécimen es el puntero al tipo `void`:

```
void* ptr;
```

Con los punteros tipo `void` podemos apuntar a variables de cualquier tipo pero no podemos realizar operaciones como las mostradas antes, es decir no se puede realizar aritmética de punteros. Sin embargo es posible realizar una conversión explícita (*cast*) del puntero tipo `void` a otro tipo y luego si realizar operaciones aritmeticas así:

Código 13 Ejemplo con puntero tipo void.

```
#include <stdio.h>
int main()
{
    void* ptr;
    int arreglo_enteros[]={5,6,7,8};
    char cadena[]="chanfle";
    ptr=arreglo_enteros;
    printf("primer elemento del arreglo de enteros%d\n",*(int*)ptr); /* 5 */
    ((int*)ptr)++;
    printf("segundo elemento del arreglo de enteros%d\n",*(int*)ptr); /* 6 */
    ptr=cadena;
    printf("primer caracter de la cadena%c\n",*(char*)ptr); /* c */
    ptr=(char*)ptr+3;
    printf("cuarto caracter de la cadena%c",*(char*)ptr); /* n */
    getchar();
    return 0;
}
```

Es importante indicar que las expresiones como $((int*)ptr)++$ solo funcionan en lenguaje C, los compiladores recientes de C++ no permiten realizar el incremento de una expresión como la que se muestra, por ello es recomendable escribir:

```
ptr=(int*)ptr+1;
```

Como se muestra en el código con un puntero tipo void es posible apuntar a variables de cualquier tipo, esto es útil cuando se escriben funciones o estructuras de datos genéricas.

4.2.5. Punteros y cadenas.

Lenguaje C no tiene soporte para el tipo de datos cadena, las cadenas son implementadas con arreglos de caracteres. *En lenguaje C las cadenas son arreglos de caracteres cuyo último elemento es el carácter ascii NULL ('\0').*

¿ Por qué es necesario usar un espacio adicional para el '\0' al final del arreglo ?. Es necesario hacer una distinción entre lo que es un arreglo de caracteres y una cadena, una cadena es un conjunto de caracteres que puede ser tratada como una unidad. Dado que en lenguaje C no es posible saber en tiempo de ejecución el tamaño de un arreglo que se pasa como argumento a una función (recordemos que lo que se pasa a la función es la dirección de inicio del mismo y nada más), es necesario tener algún mecanismo que permita saber donde termina la cadena, esto se logra precisamente con el terminador '\0'. En otros lenguajes las cadenas están implementadas de formas diferentes, por ejemplo en Pascal las cadenas son arreglos en los cuales el primer elemento del arreglo indica el número de caracteres que posee.

Sabiendo que una cadena se implementa con un arreglo de caracteres es posible utilizar punteros a caracteres para trabajar con ella, en el código 14 se muestra como se crea una cadena y luego se usa un puntero para imprimirla (posiblemente así este escrita la función puts), en este código se usa el hecho que el nombre del arreglo es la dirección del mismo para inicializar el puntero y además que las cadenas terminan con '\0' para saber hasta donde imprimir.

Es importante resaltar que los punteros a cadenas pueden ser inicializados a literales, esto se entiende mejor con un ejemplo

Código 14 Ejemplo con cadenas.

```
/*
Ejemplo del uso de un puntero a char
para acceder a los elementos de una
cadena.
*/
#include <stdio.h>
int main()
{
    char una_cadena[]="Esto es un mensaje";
    char* charptr=una_cadena;
    while(*charptr!='\0')
    {
        putchar(*charptr);
        charptr++;
    }
    getchar();
}
```

Código 15 Ejemplo de inicialización de puntero a cadena.

```
#include <stdio.h>
/*
ejemplo de inicialiacion de puntero a cadena
*/
int main()
{
    char* una_cadena="esto es un ejemplo";
    printf(una_cadena);
    getchar();
    return 0;
}
```

4.2.6. Arreglos de punteros.

Aquí no hay mucho que decir, dado que los punteros son variables es posible crear arreglos de punteros como si de cualquier otro tipo se tratara, el siguiente código muestra un ejemplo del uso de arreglos de punteros.

Código 16 Ejemplo con cadenas.

```
#include <stdio.h>
/*
ejemplo de uso de arreglos de punteros
*/
/*
Un arreglo de punteros inicializados a
cadenas
*/
char* app_msjs[]={ "ha ocurrido un error!\n",
                   "la aplicacion se torno inestable\n",
                   "todo bien, todo bien\n" };

int main()
{
    printf(app_msjs[0]);
    printf("%s",app_msjs[1]);
    puts(app_msjs[2]);
    getchar();
    return 0;
}
```

4.2.7. Punteros a punteros.

Igual que en la sección anterior, dado que los punteros son variables poseen las tres características de las mismas (a no ser que sean dinámicas en cuyo caso no tienen nombre). Como los punteros tienen direcciones no nos debe extrañar que podamos usar un puntero para almacenar su dirección. Lo anterior se muestra de forma gráfica en la siguiente figura.

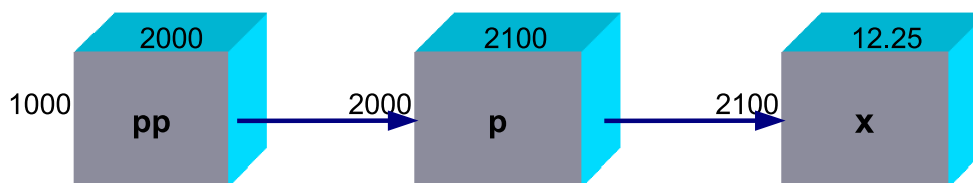


Figura 4.13: Representación de un puntero a puntero.

El código 17 implementa lo mostrado en la figura 4.13, allí se puede ver como se declaran tres variables automáticas, un flotante (x), un puntero a flotante (p) y un puntero a puntero a flotante (pp). Los punteros a punteros son variables que almacenan la dirección de memoria en la que está almacenado un puntero.

Los punteros a punteros son útiles para :

Código 17 Ejemplo puntero a puntero

```
#include <stdio.h>
/*
ejemplo puntero a puntero.
*/
int main()
{
float x=12.25;
float* p=&x; /* p apunta a x */
float** pp=&p; /* pp apunta a p */
printf("La dirección de x es%p\n",p);
printf("La dirección de p es%p\n",pp);
/* se muestra la dirección de x usando pp */
printf("La dirección de x es%p\n",*pp);
/* se muestra el dato de x usando pp */
printf("El valor de x ed%f\n",**pp);
getchar();
return 0;
}
```

- Modificar el contenido de un puntero mediante su dirección, por ejemplo para que una función modifique la zona a la que un puntero apunta.
- Crear arreglos dinámicos de punteros.

Es posible crear punteros triples (`float*** ppp=&pp`) y mayores pero generalmente no son tan usados como los punteros dobles.

4.2.8. Errores típicos trabajando con memoria dinámica

A continuación se listan algunos de los errores más frecuentes que se cometen cuando se trabaja con memoria dinámica en los lenguajes C y C++.

- Utilizar zonas no inicializadas.
 - No liberar zonas inicializadas.
 - Modificar el único puntero que tenemos para referirnos a una variable sin sacar una copia antes, perdiendo la referencia a la variable.
 - Liberar zonas no inicializadas.
-

Capítulo 5

Introducción a los Tipos Abstractos de Datos

...no me digas lo que eres, dime lo que tienes. Bertrand Meyer.

Comenzamos con cosas de mayor nivel de abstracción, este capítulo pretende introducir el concepto de tipo abstracto de dato y mostrar una de las formas más utilizadas para la implementación de éstos en lenguaje C.

5.1. Introducción

La programación de una aplicación con una complejidad moderada requiere (si se desea tener alta probabilidad de terminarlo), que se reutilice lo máximo posible el trabajo realizado por nosotros mismos o por otros programadores y que dicho código esté organizado en *partes* reutilizables, el objetivo de esto es no tener que reinventar la rueda siempre que se realiza un proyecto, y además que el programa se organice en módulos de tal manera que cada uno sea fácilmente codificado y puesto a punto. Las funciones o procedimientos son la abstracción básica de la programación procedimental, en la cual un gran programa se divide en pequeños subprogramas (funciones) que se encargan de una parte del problema. Otro paradigma se basa en la creación de tipos de datos a los cuales se les definen las *características* y las *operaciones* que tienen sentido para el tipo de

dato creado. Es posible decir que los tipos básicos de un lenguaje de programación dado son *TAD's*, los enteros, flotantes, cadenas, etc, dado que representan un conjunto de datos y tiene unas operaciones asociadas a ellos, sin embargo cuando necesitamos aumentar el nivel de abstracción es muy útil poder construir tipos propios.

5.2. Los TDA's

Supongamos que necesitamos hacer un programa que pregunte los lados de un triángulo rectángulo y muestre como resultado el valor de la hipotenusa. Es muy obvio que necesitamos tipos de datos con los que podamos realizar la siguiente operación: $h = \sqrt{x^2 + y^2}$ donde x e y son los catetos y h la hipotenusa. Sabemos que elevar una variable tipo caracter o cadena al cuadrado no tiene sentido para lo que tratamos de hacer, sin embargo es posible elevar una variable tipo *real* al cuadrado, dando como resultado un número *real* que puede sumarse con otro..., la idea es que el mejor tipo de datos para solucionar este problema es el tipo *real*. Dependiendo del problema a solucionar mejor tipo de datos puedo o no estar presente en el lenguaje en el que estemos trabajando y por ello es importante que el lenguaje nos permita crear nuestro propios tipos.

Hagamos unas definiciones para continuar con los TDA's de una manera más cómoda:

Abstracción: Cuando tratamos de entender desde un punto de vista matemático el mundo, necesitamos modelos que nos entreguen la información que requerimos en un momento dado, por ejemplo en el circuito de la figura 5.1 se establece que la corriente que circula por la resistencia es $I=V/R$, este modelo nos da la información del valor de la corriente en función del voltaje, despreciando las otras variables que puedan influir sobre este valor, ésto es una abstracción, según Rumbaugh y sus amigos es "...centrarse en los aspectos esenciales inherentes de una entidad, e ignorar sus propiedades accidentales".

- *Encapsulamiento:* Ocultar las cuestiones internas de implementación de una función u operación a los usuarios (posiblemente nosotros mismos) y mostrar sólo lo necesario para poder realizar la tarea de manera correcta. El conocimiento que se necesita para la realización de
-

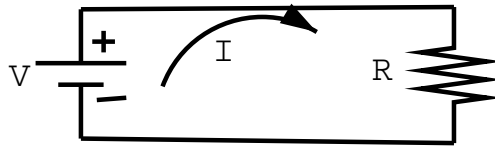


Figura 5.1: Circuito Simple.

cierta operación es lo que se denomina interfaz. Si un programador de un TDA decide cambiar la representación interna, no genera ningún traumatismo a los otros, si el TAD continua con la misma interfaz. En la figura 5.2 se muestra como las operaciones (acciones en el diagrama) son vistas desde afuera pero su implementación no.

- TAD: Definición de los atributos que posee (que nos interesan) y las operaciones (idem :) que se permiten realizar sobre un determinado objeto, por ejemplo, *las hojas de papel* tienen ciertas características, el gramaje, el color, el alto, el ancho, etc, pero la hoja de papel que usted esta leyendo (si no esta leyendo en un computador) en este momento tiene unos *valores* determinados para esas características, en ese sentido podemos decir que *hoja de papel* es un tipo y que esta hoja de papel es un objeto de ese tipo, las operaciones que permite una hoja son muchas, doblar, cortar, quemar, etc, algunas de ellas son útiles en ciertos contextos y en otros no, eso depende plenamente de la aplicación. En resumen un TDA es representa un conjunto de datos y las funciones para manipular esos datos.

Cuando se programa orientado a los tipos abstractos de datos, la preocupación inicial no es el control del programa sino más bien los datos con los que el programa tendrá que tratar, estos *tipos* generalmente permitirán operaciones de creación, de consulta, de actualización, de proceso y de destrucción (liberación); estas operaciones son las que permiten a los diferentes *objetos* interactuar durante la ejecución para realizar determinadas tareas. Los tipos de datos deben poder ser usados de una manera simple y natural, para ello los nombres de las funciones deben ser lo suficientemente claros como para saber con la lectura del código lo que realizan.

Del analisis del problema es de donde resultan los TAD's, los cuales tienen correspondencia con el mundo real (dominio de la aplicación), otros resultan después y no tienen correspondencia en el dominio de la apli-

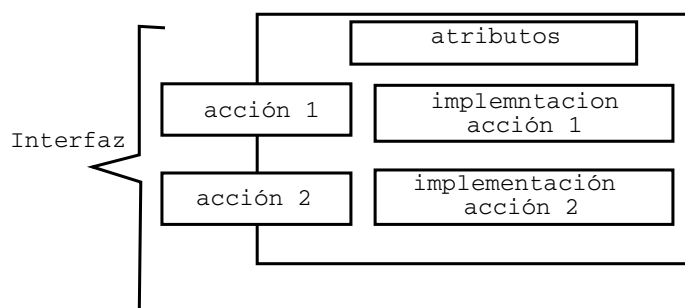


Figura 5.2: La interfaz de un TAD.

cación ya que son sólo existen dentro de la computadora como por ejemplo una lista enlazada. El objetivo de todo este rollo, es modelar de una forma más consistente el entorno y desarrollar programas más fácil, más rápido y más fáciles de mantener.

Veamos el siguiente ejemplo; se esta escribiendo una libreria para manipulación de la pantalla y nos han encargado escribir el TAD para representar los puntos:

TAD: Punto

Atributos:

```
int x;
int y;
```

Operaciones:

```
Punto* punto_crear(); crea una variable dinámica tipo Punto.
void punto_setX(Punto* p, int valorx); modifica el atributo x de p y lo pone a valorx.
void punto_setY(Punto* p, int valory); idem pero con el atributo y.
int punto_getX(Punto* p); funcion para obtener el valor de x del punto p.
int punto_getY(Punto* p); idem pero con el atributo y.
void punto_mover(Punto* a, float deltax,float deltay); función desplaza un punto un determinado valor en las coordenadas x e y.
int punto_distancia(Punto*a, Punto* b); función que calcula la distancia entre dos puntos.
void punto_dibujar(Punto* a); adivine :)
void punto_ocultar(Punto* a)
```

Pero que es Punto en todas esas funciones ?, pues es una estructura del siguiente tipo

```
typedef struct _Punto
{
    int x;
    int y;
} Punto;
```

y la implementación de la operación *mover* seria algo como lo que se muestra en el código 18

Código 18 Implementación de la operación *mover* del TAD Punto.

```
void punto_mover(Punto* punto,float deltax,float deltay)
{
    punto_ocultar(punto); //borramos el punto de la pantalla
    punto->x+=deltax;
    punto->y+=deltay;
    punto_dibujar(punto); //volvemos a mostrar el punto en la nueva ubicación
}
```

En la anterior definición del TAD Punto, pueden agregarse tantas operaciones (funciones) como se necesiten y además el TAD Punto puede implementarse con un arreglo de dos componentes, es libertad del programador del TAD decidir la mejor implementación. Vemos que todas las funciones con la que se implementan las operaciones que se definen para un TAD toman como primer parámetro un objeto tipo Punto (bueno un puntero a Punto) ya que esta es la forma de indicarle al lenguaje sobre que objeto en particular se desea realizar la operación, en lenguajes como C++ o Java esto no es necesario porque en ellos se puede trabajar con *Clases*, que son unas formas más avanzadas y cómodas de crear TAD's. Pero esto tan complicado para que, si es lo mismo que se hace siempre *funciones?*, - Si, es lo mismo, pero todas estas funciones son para manipular los puntos, es decir uno puede visualizar el TAD Punto como un todo.

Ahora suponga que se esta haciendo un software para dibujo llamado *repaint* y que a usted lo encargaron de crear un TAD llamado Poligono con la siguiente especificación:

TAD: Poligono**Atributos:**

int numPuntos; Atributo que dice cuantos puntos tiene un poligono.

int maxPuntos; Para controlar el tamaño del arreglo dinámico de puntos en caso de tener que reasignar memoria (muchos puntos).

Punto* puntos; Los puntos del poligono

Operaciones:

Poligono* poligono_crear(int maxPuntos); crea un poligono con un máximo de maxPuntos puntos.

int poligono_add_punto(Poligono* pol, Punto* p); agrega el punto p al poligono, retorna 1, si hay error.

void poligono_dibujar(Poligono* pol); pinta un poligono.

void poligono_liberar(Poligono* pol); Libera los componentes internos de un poligono.

Generalmente durante la ejecución de una aplicación se crean, modifican y destruyen objetos todo el tiempo y estos, tal y como se mencionó interactúan usando sus interfaces. Veamos la implementación de los tipos de datos que acabamos de ver.

Bibliografía

- [Meyer96] Construcción de software orientado a objetos.
- [1] [Coo94] William R. Cook. A comparison of objects and abstract data types, February 1994.
- [Villalobos2005] Jorge Villalobos, Rubby Casallas, Katalina Marcos . El reto de diseñar un primer curso de programación de computadores. Universidad de los Andes, Bogotá, Colombia, 2005.