Tecnología de la programación

Bibliografía:

- Tecnología de la Programación, Título de Grado en Ingeniería Informática, Departamento de Ingeniería de la Información y las Comunicaciones, Universidad de Murcia (Curso 2017–18).
- Wikipedia, la Enciclopedia Libre (https://es.wikipedia.org/).

El lenguaje C

C es un lenguaje de programación creado en 1972 por Dennis M. Ritchie orientado a implementar el sistema operativo Unix. El primer estándar de C fue creado por ANSI en 1989 («C89»), y al año siguiente fue ratificado como estándar ISO. A mediados de los 80 se crea el C++, una extensión de C orientada a objetos, que se convierte a estándar ISO en 1998. El último estándar de C fue creado en 1999 («C99»), pues los nuevos cambios son incorporados en C++ y no en C. Usaremos el estándar C99 con extensiones de GNU.

C es un lenguaje de **medio nivel**, pues proporciona cierta abstracción de lenguajes de alto nivel pero con la eficiencia del lenguaje máquina. Es **débilmente tipado**, pues permite conversiones implícitas entre tipos. No permite encapsulado (aunque se puede simular), y permite la programación estructurada. Tiene como ventajas el ser muy transportable, flexible y expresivo, además de generar código eficiente, si bien es poco modular, realiza pocas comprobaciones y es más difícil escribir y leer código en comparación con otros lenguajes.

La compilación de un programa en C la realizan los siguientes programas:

1. **Preprocesador:** Elimina los comentarios del código (.c), incluye código de otros archivos (.h), sustituye las macros y permite la inclusión de código de forma condicional.
2. **Compilador:** Analiza el código resultante y lo convierte a lenguaje ensamblador (.s).
3. **Ensamblador:** Convierte el código ensamblador en código máquina dentro de un fichero objeto (.o).
4. **Enlazador:** Crea el ejecutable final a partir de ficheros objeto y las bibliotecas de código.

Los **comentarios** se usan para explicar alguna parte del código del programa, y no afectan a su significado. Se usan como `/* comentario */` o `// comentario de una línea (sólo C99)`.

0.1. Manipulación básica de datos

Tipos básicos:

- **Enteros:** Se escriben como (expresión regular) `-?[1-9][0-9]*` (base decimal), `0[0-7]*` (octal) o `0x[0-9a-fA-F]` (hexadecimal). Se declaran con `int`, que puede ir precedido por `long`, `long long` o `short` para establecer el tamaño en bits, que normalmente es, respectivamente, de 32, 64 o 16 bits, y si no se especifica es de 32 bits. Precediendo a esto puede ir `signed` o `unsigned` para indicar si el entero tiene signo o no (por defecto en general tiene). De hecho, si se indica cualquiera de los sufijos se puede omitir el `int`.

- **Reales en coma flotante:** Se escriben como `-?[0-9]*\.[0-9]+([eE]-?[0-9][1-9]*)?`. Se declaran con `float`, `double` o `long double`, que en general se corresponden con un tamaño de 32, 64 y 128 bits, respectivamente.

- **Caracteres:** Se escriben entre comillas simples como `'c'`, si bien algunos requieren secuencias de escape como `'\''` (comilla simple), `'\n'` (salto de línea), `'\t'` (tabulador), etc. También pueden funcionar como enteros (por lo general de 8 bits). Se declaran con `char`, que puede ir precedido de `signed` o `unsigned`.

- **Cadenas de caracteres:** Se escriben entre comillas dobles, con secuencias de escape como las de los caracteres sueltos (para las comillas dobles se usa `\"`). Realmente son listas de caracteres, que terminan con el caracter nulo (`'\0'`, que se corresponde con el entero 0).

Los **identificadores** nombran variables, funciones, tipos de datos definidos por el programador o macros del preprocesador. Su formato es `[a-zA-Z_][a-zA-Z0-9_]*`, distinguen mayúsculas de minúsculas y no pueden ser **palabras reservadas** por el compilador, que en C89 son: `auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while`. C99 añade además las palabras reservadas: `inline _Bool _Complex _Imaginary`.

Las variables se definen con la sintaxis *`tipo identificador`*`;`. Si antes del tipo se usa la palabra `const`, la variable es de sólo lectura.

Una **expresión** consiste en al menos un operando y cero o más operadores, donde los operandos pueden ser literales (valores escritos tal cual), variables o llamadas a funciones que devuelven un valor. Los operadores son (si no se dice nada son binarios):

1. Paréntesis: `(...)` para agrupar una expresión.

2. Operadores aritméticos: Suma (`+`), resta (`-`), producto (`*`), división (`/`), resto (`%`). C no detecta desbordamientos, por lo que si el resultado está fuera del rango, el valor es erróneo.

3. Operadores relacionales: Igual (`==`), distinto (`!=`), mayor (`>`), menor (`<`), mayor o igual (`>=`), menor o igual (`<=`). Devuelven 1 si la expresión es verdadera y 0 si es falsa.

4. Operadores lógicos: Conjunción (`&&`), disyunción (`||`), negación (unario, `!`). Interpretan el 0 como falso y cualquier otro valor como cierto, y devuelven 1 si la expresión es cierta y 0 si es falsa.

5. Asignación (*`dest`*`=`*`expr`*). También se puede usar, por ejemplo, *`a`*`*=`*`b`*, como abreviatura de *`a`*`=`*`a`*`*`*`b`*. Además, *`x`*`++` y `++`*`x`* ambas añaden 1 a la variable *`x`*, pero *`x`*`++` devuelve el valor antes de modificarlo (postincremento) mientras que `++`*`x`* lo devuelve después (preincremento). Lo mismo sucede con *`x`*`--` (postdecremento, resta 1) y `--`*`x`* (predecremento).

6. Obtención del tamaño de un tipo: `sizeof(`*`tipo`*`)` o `sizeof` *`expr`* devuelve el tamaño (en bytes) de un tipo de dato o resultado de una expresión.

En C, la asociatividad (en general) es de izquierda a derecha, y la precedencia es, de mayor a menor, la siguiente:

1. `()`, `[]`, `->`, `..`
2. Unarios `!`, `~`, `+`, `-`, `++`, `--`, `&`, `*`, `sizeof()`, (*tipo*).
3. `*` (binario), `/`, `%`.
4. `+`, `-` (binarios).
5. `<<`, `>>`.
6. `<`, `<=`, `>`, `>=`.
7. `==`, `!=`.
8. `&` (binario).
9. `^`.
10. `|`.
11. `&&`.
12. `||`.
13. `?:`.
14. `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `&=`, `^=`, `|=`, `<<=`, `>>=`.
15. `,`.

Si el resultado de una operación con enteros no es entero, se redondea a la baja. En la evaluación de una expresión pueden producirse conversiones de tipo, lo que se llama **promoción** si es hacia un tipo de mayor rango o **degradación** si es a uno de menor rango, y pueden ser **implícitas** si las hace automáticamente el compilador o **explícitas** (*casting*) si las indica el programador mediante la sintaxis `(`*tipo*`)`*expr*. Para la conversión implícita:

1. `char` y `short` se convierten a `int`.
2. `unsigned char` y `unsigned short` se convierten a `unsigned`.
3. Los operandos de menor rango se promueven al de mayor rango. Los tipos son, de menor a mayor rango, `int`, `unsigned`, `long`, `unsigned long`, `float` y `double`.
4. El resultado se degrada al asignarlo si es necesario.

0.2. Tipos de datos compuestos

Enumerados El tipo se define con `enum` *nombre* `{` *id1* `,` *id2* `,` *...* `}` *var1* `,` *var2* `,` *...* `;`, donde no es necesario definir las variables *var1* `,` *var2* `,` *...* en la definición, sino que podemos definirlas después con el tipo `enum` *nombre*. Una variable de este tipo puede tomar como valor cualquiera entre *id1* `,` *id2* `,` *...*, que asignamos como *var* `=` *id*.

Estructuras Las estructuras están formadas por una serie de campos de varios tipos, y se definen como `struct` *`nombre`* `{` *`tipo1 campo1`*`;` *`tipo2 campo2`*`;` `...` `}` *`var1`*`,` *`var2`*`,` `...`;, igual que con los enumerados. El tipo es `struct` *`nombre`*, accedemos a los campos con la sintaxis *`variable`*`.`*`campo`* y las inicializamos como `struct` *`nombre var`* `=` `{`*`valor_campo1`*`,` `...`}. Podemos asignar una variable de un tipo `struct` a otra del mismo, pero no a otra de un tipo `struct` de distinto nombre.

Uniones Similares a las estructuras, pero todos los campos se almacenan en el mismo espacio, con lo que no debemos acceder a uno distinto al que fue asignado. Se declaran con `union` *`nombre`* `{` *`tipo1 campo1`*`;` `...`} *`var1`*`,` `...`;. El tipo es `union` *`nombre`*, y se inicializan con `union` *`nombre var`* `=` {*`.campo = valor`*} o con `union` *`nombre var`* `=` {*`campo`*`:` *`valor`*}.

Tablas Las tablas o *arrays* son estructuras que permiten almacenar elementos del mismo tipo consecutivamente. Se definen con *`tipo nombre`*`[`*`n1`*`][...][`*`nt`*`]`, donde los índices son enteros no negativos constantes que indican el tamaño, y se accede a los elementos con *`nombre`*`[`*`j1`*`][...][`*`jt`*`]`, donde $0 \leq j_i \leq n_i - 1$. Las tablas unidimensionales se pueden inicializar como {*`val0`*`,` *`val1`*`,` `...`,` *`valN-1`*}, y de hecho podemos definirlas como *`nombre`*`[]` si la inicializamos en la misma definición de esta forma. También, en esta sintaxis, se pueden inicializar dentro de las llaves elementos sueltos con `[`*`i`*`]` *`valor`* o `[`*`i`*`]` `=` *`valor`* y rangos con `[`*`i0`* `...` *`if`*`]` `=` *`valor`*. No podemos asignar una tabla a otra directamente, sino que para ello hemos de copiar sus elementos.

Renombrado de tipos Permiten dar un nuevo nombre a un tipo mediante `typedef` *`nombre_tipo nuevo_nombre`*`;`, y este nuevo nombre de tipo se usa sin ningún prefijo (seguimos pudiendo usar el nombre antiguo). Al definir un enumerado, estructura o unión no es necesario decir su nombre (se dice que es un tipo **anónimo**), y de hecho tampoco hace falta definirlo antes de usarlo (se dice que es un tipo **incompleto**) salvo si fuera necesario conocer su estructura o tamaño, de modo que con frecuencia el renombrado de tipos ocurre con tipos anónimos (con la definición «dentro» del `typedef`) o incompletos.

0.3. Ámbito y extensión

El **ámbito** de un identificador es el contexto del programa en que este puede ser utilizado, mientras que la **duración** de una variable es el tiempo desde que empieza a existir hasta que deja de existir, liberándose el espacio que ocupaba. Así:

- Una **variable global** o **estática** (definida fuera de cualquier bloque) tiene como ámbito desde que se declara hasta el final del módulo. La declaración puede ser una definición o una declaración de su existencia en otro módulo, que es como su definición pero con el prefijo `extern`. Sin embargo, si la variable fue definida en su módulo con el prefijo `static`, no se puede usar en otros módulos. Se almacena en el **segmento de datos** del programa, por lo que su extensión es la del proceso en que se ejecuta el programa. Por defecto se inicializan a 0.
- Una **variable local** o **automática** (declarada dentro de un bloque o función) tiene como ámbito desde que se declara hasta el final del bloque en que se ha declarado, y oculta

a cualquier otra variable del mismo nombre global o definida en un bloque superior. Un parámetro a función actúa como variable local a esta. Se almacena en el **segmento de pila** o en los registros de la CPU, por lo que su extensión es desde que se declara hasta el final del bloque, salvo si se define con el prefijo `static`, cuyo efecto en una variable local es guardar el valor de la variable en el segmento de datos, conservando su valor entre llamadas a la función o ejecuciones del bloque.

- Una **variable dinámica** es accesible mediante apuntadores, y se construye y destruye en tiempo de ejecución. Su ámbito es el de la variable que la apunta (puede haber varias). Se almacena en el **segmento montón** (*heap*), y como se construyen y destruyen explícitamente mediante llamadas a funciones de la biblioteca estándar, su extensión viene dada por estas llamadas.

Por su parte, el ámbito de un tipo es similar al de una variable, y el de una función es siempre global.

0.4. Apuntadores

Son variables que almacenan una dirección de memoria. Se usan para paso de parámetros por referencia, implementación de ciertas estructuras, tratamiento de tablas (y por tanto cadenas de caracteres) y gestión de la memoria dinámica. Un apuntador a una variable de un cierto tipo se indica con *`tipo *ptr;`*. También se puede definir con `void *ptr;` si no se quiere indicar el tipo al que apunta.

Para acceder al valor al que apunta se usa el **operador de desreferencia** o **indirección** *`*ptr`*, y para obtener la dirección de memoria de una variable se usa el **operador de referencia** *`&var`*.

El apuntador nulo es aquel con valor 0, y la biblioteca estándar `stdio.h` define la macro `NULL` como 0 para referirse a este. No apunta a una variable válida, y se usa para inicializar apuntadores o como marca de fin o de error.

Podemos sumar y restar enteros a apuntadores mediante `+`, `-`, `++`, `--`, `+=` y `-=`, en cuyo caso el resultado es una dirección de memoria igual a la inicial más (o menos) el entero multiplicado por el tamaño del tipo de dato al que apunta si este se ha especificado, o por 1 si el tipo es `void*`. Además, podemos comparar apuntadores con `==`, `<=`, `>=`, `<`, `>` y `!=`.

Una tabla es una constante cuyo valor es la dirección del primer elemento, con lo que, en una tabla unidimensional, *`v[i]`* equivale a *`*(v+i)`*. Si *`v`* es un array de dos dimensiones, entonces `v[i][j]` equivale a `*(*(v+i)+j)`, y algo similar ocurre con más dimensiones. No obstante, un array bidimensional no puede ser asignado a un apuntador a apuntador porque lo último no asume la contigüidad de los datos.

Un apuntador se puede asignar a otro, si bien para un apuntador de un tipo a otro (o a ninguno) el compilador da un aviso si no se realiza una conversión explícita. Para apuntadores a estructuras, la sintaxis *`ptr->campo`* equivale a *`(*ptr).campo`*.

0.5. Sentencias

Una **sentencia simple** es una expresión (terminada en `;`), etiqueta, sentencia vacía (`;` «suelto», no hace nada) o cualquiera de las que veremos a continuación, mientras que una

sentencia compuesta o **bloque** está formada por varias sentencias entre llaves. Sentencias simples:

`if` `if (`*cond*`)` *then_clause* `[else` *else_clause*`]`. Evalúa la expresión *cond* y si el resultado es distinto de cero, se ejecuta *then_clause*, de lo contrario, si está presente, se ejecuta *else_clause*.

`switch` `switch (`*expr*`) { (case` *comp_expr*`:` *clause**`)+ [default:` *clause**`] }`. Compara el resultado de evaluar la expresión *expr* con cada *comp_expr*, en orden, donde *comp_expr* puede ser una constante o un rango de enteros consecutivos $[a, b]$ en el que puede encontrarse el valor, indicado por *a* `...` *b*. Entonces empieza a ejecutar las sentencias a partir de este caso (o a partir de `default` si está presente y la expresión no coincide con ningún caso), y hasta el final del `switch`.

`while` `while (`*cond*`)` *clause*. Evalúa la expresión *cond* y si el resultado es distinto de cero, se ejecuta *clause* y vuelve a evaular *cond*, en bucle.

`for` `for (`*init_expr*`;` *cond*`;` *update_expr*`)` *clause*. Equivale a *init_expr*`; while (`*cond*`) {` *clause update_expr*`; }`. Las tres expresiones entre los paréntesis se pueden omitir.

`do while` `do` *clause* `while (`*cond*`)`. Equivale a *clause* `while (`*cond*`)` *clause*.

`break;` Sale de un `while`, `do`, `for` o `switch`. Suele usarse con `switch`.

`continue;` Termina la iteración actual de un bucle para comenzar la siguiente (incluyendo evaluar *cond*, *update_expr*, etc.).

`return` `return` *[valor]*`;`. Termina la ejecución de la función y devuelve un valor, que se omite de la sentencia si la función no devuelve nada.

0.6. Funciones

Se definen con *tipo_devuelto nombre* `(`*tipo1 param1*`, ...) {` *cuerpo* `}`, donde el cuerpo está formado por sentencias y definiciones de variables (y puede que tipos) y debe terminar su ejecución por una sentencia `return`. Para **declarar** una función sin dar su definición (porque se defina más adelante o en otro módulo) sustituimos `{` *cuerpo* `}` por `;`. Si la función no devuelve nada, el tipo devuelto es `void`, y si no toma ningún parámetro, la lista de parámetros es `void`. También se puede omitir la lista de parámetros (poniendo sólo los paréntesis), pero esto indicaría que a la función se le puede pasar cualquier número de parámetros de cualquier tipo.

Las llamadas a la función se hacen dentro de una expresión con *nombre*`(`*val1*`, ...)`. Podemos declarar una función de forma similar a como se define pero sustituyendo `{` *cuerpo* `}` por `;`. De esta forma la función puede ser usada sin haber sido definida todavía, o si pertenece a otro módulo o a una biblioteca. Los procedimientos en C son funciones que no devuelven ningún valor.

Los parámetros se pasan por valor, es decir, se copia el **parámetro actual** (el que se indica en la llamada) al **parámetro formal** (el que se indica en la lista de parámetros y que luego

puede ser usado en el cuerpo). Por tanto una función no puede alterar el valor de las variables usadas como parámetros, y esto incluye los `struct`.

El paso de una tabla como parámetro a una función, no obstante, equivale al paso de un apuntador al primer elemento de este, y si por ejemplo la tabla es bidimensional, esto equivale a pasar un doble apuntador. Por otro lado, si una función desea devolver una tabla, deberá construirla, por ejemplo, en memoria dinámica (no puede construirla en una variable local por su extensión) y devolver un apuntador.

Todo programa en C debe tener una función `main`, que es invocada por el sistema operativo al comienzo de la ejecución del programa, y puede recibir argumentos de este. Su prototipo suele ser `int main(int argc, char **argv)`, donde `argc` es el número de argumentos (incluyendo el nombre de la aplicación), `argv` es una tabla de longitud `argc` de cadenas de caracteres que contiene los argumentos y el valor devuelto es 0 si no ha ocurrido ningún error.

0.7. El preprocesador

Es invocado por el compilador antes de la compilación en sí. Tiene su propio lenguaje independiente del C y todas sus órdenes empiezan por `#`. Podemos usarlo para:

- Incluir ficheros, con `#include <`*fichero*`>` para un fichero en una lista estándar de directorios, entre los que se encuentran las cabeceras de los archivos de la biblioteca estándar de C, o con `#include "`*fichero*`"`, que antes busca en el directorio actual y aquellos indicados por el usuario.
- Definir macros, con `#define` *nombre código_sustituido*, de forma que a partir de entonces cada aparición de *nombre* fuera de una cadena de caracteres es sustituida. También se puede usar `#define` *nombre*`(`*par1*`, `*par2*`, ...)` *código_sustituido* para definir una macro con parámetros, y entonces en el código sustituido se usan dichos parámetros.
- «Des-definir» macros, con `#undef` *nombre*.
- Insertar código de forma condicional. Los fragmentos de código condicional empiezan por `#if` *expr_del_preprocesador*, `#ifdef` *macro* o `#ifndef` *macro*, pueden contener una directiva `#else` en medio y terminan con `#endif`. Entonces, si, respectivamente, se cumple la condición, la macro está definida o la macro no está definida, se inserta el código hasta el `#else`, o hasta el `#endif` si no hay `#else`, y de lo contrario, si existe, se inserta el código entre el `#else` y el `#endif`.

0.8. La biblioteca estándar

0.8.1. Entrada y salida (<stdio.h>)

- `int printf(const char *fmt, ...)` imprime un texto en la salida estándar con formato indicado por `fmt`, parámetro al que le sigue un número variable de argumentos (puede ser 0) que se imprimen según lo indicado por los especificadores de conversión en `fmt`, que son: `%c` para un `char`, `%d` para un `int`, `%f` para un `float`, `%lf` para un `double`, `%s` para una cadena de caracteres (`char*`), etc.

- `int scanf(const char *fmt, ...)` lee de la entrada estándar y usa los mismos especificadores de convesión que `printf`, pero requiere que los parámetros se le pasen por referencia para poder modificarlos, salvo las cadenas de caracteres, que ya son de por sí apuntadores. Devuelve el número de datos de entrada asignados o `EOF` (macro definida como -1) si ocurre un error de entrada antes de realizar ninguna conversión.

- `int puts(const char *s)` imprime `s` por la salida estándar, acabado en salto de línea, devolviendo `EOF` si hay un error o un valor no negativo en caso contrario.

- `char *gets(char *s)` lee de la entrada estándar, guardando el resultado en la cadena apuntada por `s`, hasta encontrar un caracter de salto de línea, que descarta, escribiendo al final un caracter nulo en `s`, que debe tener suficiente espacio para almacenar la cadena.

De forma más general, el tipo `FILE` es un tipo opaco que representa un flujo (*stream*) o canal de comunicación, y que manejamos con un apuntador. Por lo general contiene un identificador, un indicador de posición, un apuntador a un búfer, un indicador de errores y un indicador de final de fichero. En C se definen como *streams* estándares la entrada estándar `stdin`, la salida estándar `stdout` y la salida estándar de errores `stderr`.

- `FILE *fopen(const char *name, const char *mode)` abre un fichero con un nombre dado y nos devuelve un manejador (apuntador a `FILE`), o `NULL` si no se ha podido abrir. El modo puede ser:

`r`	Lectura, el fichero debe existir.
`w`	Escritura, y si el fichero existe se sobrescribe.
`a`	Escritura, y si el fichero existe se escribe al final (el puntero para escritura se sitúa al final de lo que ya hay).
`r+`	Lectura y escritura, y el fichero debe existir.
`w+`	Lectura y escritura, y si el fichero existe se sobrescribe.
`a+`	Lectura y escritura, con el puntero para lectura y escritura al final.
`t`	Fichero de texto (se especifica detrás de una de las otras opciones, pero si se omite se entiende por defecto).
`b`	Fichero binario (se especifica detrás de una de las primeras seis opciones).

- `int fclose(FILE *f)` cierra el fichero y libera el manejador asociado, junto con todo lo necesario. Devuelve 0 si se ha cerrado con éxito o `EOF` en caso contrario.

- `int fprintf(FILE *f, const char *fmt, ...)` es similar a `printf`, pero escribe en un flujo indicado por `f`.

- `int fscanf(FILE *f, const char *fmt, ...)` es similar a `scanf`.

- `int fputs(char *s, FILE *f)` es similar a `puts`, pero no inserta el salto de línea al final.

- `char *fgets(char *s, int n, FILE *f)` es similar a `gets`, pero lee de un flujo indicado por `f` y hasta un máximo de `n` caracteres incluyendo el caracter nulo, que siempre guarda al final. Devuelve `s` si la operación tiene éxito o `NULL` si falla, en cuyo caso el contenido de `s` queda invariable si ha encontrado un final de fichero antes de leer ningún caracter, o indeterminado si ocurre un error de lectura durante el proceso.
- `int feof(FILE *f)` devuelve un valor distinto de 0 si se ha llegado al final del fichero y 0 en caso contrario.
- `void rewind(FILE *f)` mueve el indicador de posición de fichero al comienzo de este.
- `int fseek(FILE *f, long offset, int orig)` mueve el indicador de posición de fichero a una posición `offset` respecto a un origen `orig`, que puede ser:

  `SEEK_SET` El principio del fichero.

  `SEEK_CUR` La posición actual.

  `SEEK_END` El final del fichero.

0.8.2. Manipulación de cadenas de caracteres (<string.h>)

- `char *strcpy(char *dst, const char *src)` copia la cadena apuntada por `src`, incluyendo el caracter nulo, a la cadena apuntada por `dst`, que debe tener espacio suficiente.
- `char *strcat(char *dst, const char *src)` copia la cadena apuntada por `src`, incluyendo el caracter nulo, al final de `dst`, con el caracter inicial de `src` sobreescribiendo el caracter nulo de `dst`.
- `unsigned int strlen(const char *s)` devuelve el número de caracteres de `s`, sin contar el caracter nulo.
- `int strcmp(const char *s1, const char *s2)` devuelve un número mayor, igual o menor que cero según si `s1` es mayor, igual o menor que `s2` en orden lexicográfico.

0.8.3. Gestión de la memoria dinámica (<stdlib.h>)

- `void *malloc(unsigned int bytes)` asigna una zona de memoria dinámica de un tamaño dado en bytes (por lo que se suele usar con `sizeof`) y devuelve un puntero a dicha zona, o `NULL` si no hay espacio suficiente para ello.
- `void *calloc(unsigned int bytes)` actúa igual que `malloc` pero inicializa la memoria asignada a cero.
- `void *realloc(void *ptr, unsigned int bytes)` cambia el tamaño de un bloque asignado inicialmente por `malloc` o `calloc`, conservando el contenido de la memoria hasta el menor de los tamaños nuevo y viejo. Si `ptr` es `NULL`, la llamada equivale a `malloc(`*`bytes`*`)`, y si `bytes` es 0, equivale a `free(`*`ptr`*`)`. Como puede ser necesario cambiar la ubicación del contenido, liberando el bloque anterior y asignando uno nuevo en otro sitio, la función devuelve un puntero a la nueva ubicación del bloque.

- `void free(void *ptr)` libera un bloque de memoria dinámica asignado por `malloc`, `calloc` o `realloc`. Es importante liberar un bloque antes de dejarlo sin un apuntador que lo apunte, pues de lo contrario acumulamos **basura**. También es importante no liberar (con `realloc` o `free`) una zona de memoria que vaya a ser utilizada posteriormente en otra parte del código.

0.8.4. Gestión de errores

- La variable `extern int errno`, definida en `<errno.h>`, contiene el valor del último error producido, al menos por la biblioteca estándar de C. Esta biblioteca también define macros para los códigos de error que se pueden producir.
- `char *strerror(int n_error)`, definida en `<string.h>`, devuelve el mensaje de error asociado a un valor de `errno`.
- `void perror(const char *msg)`, definida en `<stdio.h>`, imprime por `stderr` el mensaje `msg` (salvo si es `NULL`) y, a continuación, el mensaje de error estándar asociado al valor de `errno`.

0.9. Programación modular

Es un paradigma de programación consistente en dividir el programa en **módulos** o **subprogramas** para hacerlo más manejable. Estos deben tener una tarea bien definida y normalmente requieren de otros para operar. La comunicación entre módulos se realiza mediante una **interfaz de comunicación** bien definida.

En C, cada módulo está definido en un fichero fuente con extensión `.c` que puede ser compilado por separado creando un fichero `.o`, y que lleva asociado un fichero cabecera con extensión `.h`, en el que ofrece una serie de funciones, tipos de datos, variables y macros.

Un fichero cabecera empieza, en general, por las líneas `#ifndef __`*`NOMBRE`*`_H` seguido de `#define __`*`NOMBRE`*`_H`, y termina por `#endif`, con el fin de evitar declaraciones múltiples. Dentro suele llevar lo siguiente:

- Las macros que se desean exportar.
- Para cada tipo que se desea exportar, un renombrado de la forma `typedef` *`tipo_original`* `*`*`tipo_exportado`*, que define un apuntador a un tipo incompleto.
- La declaración de cada función que se desea exportar.

Los módulos que desean usar un cierto módulo deben incluir su fichero de cabecera con `#include` en el fichero fuente, o en su propio fichero de cabecera si fuera necesario. Dado que un módulo no conoce el funcionamiento interno de las estructuras de otro, cada módulo debe exportar funciones para la creación y liberación de memoria, y para la manipulación y acceso a campos, de cada estructura que defina.

Capítulo 1

Abstracción de datos

La **abstracción** es un proceso mental consistente en simplificar un problema realzando los detalles relevantes mientras se ignoran los irrelevantes. En programación, consisten en enfatizar el «qué hace» sobre el «cómo lo hace», y existen tres formas fundamentales:

- **Abstracción de control:** Establece nuevos mecanismos de control sencillos ocultando los detalles de su implementación. Por ejemplo, `while` usa internamente instrucciones de salto y condicionales de más bajo nivel.
- **Abstracción funcional:** Abstrae un conjunto de operaciones como una única operación, separando el propósito de la implementación.
- **Abstracción de datos:** Permite mejorar la representación de los datos de un problema mediante **tipos de datos abstractos** o **TDAs**.

1.1. Tipos de datos abstractos

Un TDA es un tipo de datos caracterizado por un conjunto de operaciones o **interfaz pública**, que representa el comportamiento del tipo y se define mediante una **especificación**. Cumple las propiedades de **privacidad**, pues los usuarios del TDA no necesitan conocer la representación de los valores en memoria, y **protección**, pues los datos sólo pueden ser manipulados a través de las operaciones previstas.

Los tipos primitivos de un lenguaje de programación se consideran TDAs, pues cumplen estas dos propiedades. El uso de TDAs tiene las siguientes ventajas:

- **Facilidad de uso:** No es necesario conocer los detalles internos, sino sólo la **documentación**.
- **Desarrollo y mantenimiento:** Cualquier cambio al TDA que siga respetando la interfaz no afecta al resto del programa, y viceversa, lo que además facilita la localización de errores.
- **Reusabilidad:** El TDA puede usarse en varios programas.
- **Fiabilidad:** Es más fácil realizar pruebas sobre los módulos de forma independiente.

Podemos clasificar los TDAs en **simples**, si usan un espacio de almacenamiento constante, o **contenedores**, si este espacio varía. También podemos distinguir entre TDAs **mutables**, si cuentan con operaciones de modificación, o **inmutables**, si sus instancias no pueden modificarse una vez creadas.

1.2. Especificación

Existen dos tipos de especificaciones: **informales** y **formales**. Las informales usan lenguaje natural. No son breves, pero son sencillas de entender. Contienen dos partes:

1. **Definición:** Se define el nuevo TDA junto con los términos relacionados necesarios para comprender el resto de la especificación. Se define el dominio de valores del TDA, y se puede hablar también de su mutabilidad.
2. **Operaciones:** Se define la sintaxis y semántica de cada operación. Para la sintaxis, se incluye el nombre de la operación, los nombres y tipos de los parámetros y el tipo devuelto. Para la semántica se incluyen las precondiciones y efectos de la operación.

Las especificaciones formales permiten establecer un sistema de comunicación claro, simple y conciso, que permite la deducción formal de propiedades y la verificación formal de programas, si bien son más difíciles de leer y escribir. Constan de tres partes:

1. **Tipo:** Nombre del TDA.
2. **Sintaxis:** Forma de cada operación: nombre de la función (tipo de los argumentos) $\rightarrow$ tipo del resultado.
3. **Semántica:** Significado de las operaciones: nombre de la función (valores particulares) $\Longrightarrow$ expresión del resultado.

A continuación vemos cómo elegir el conjunto de operaciones de un TDA. Este debe ser suficiente, pero no necesariamente mínimo, pues puede ser conveniente añadir nuevas operaciones a las operaciones básicas si van a ser muy utilizadas, o si su eficiencia empeora si se implementa mediante operaciones básicas. No obstante, un TDA está sujeto a mantenimiento y es menos costoso añadir nuevas operaciones que eliminar las ya existentes, por lo que conviene no implementar operaciones de necesidad dudosa.

Normalmente se incluyen operaciones y funciones asociadas habitualmente al tipo de dato, entre las que puede haber operaciones de acceso y modificación (*getters* y *setters*) de campos de la estructura interna. Pueden incluirse también operaciones **estáticas**, que sirven para acceder a datos comunes a todas las instancias del TDA.

Dependiendo del lenguaje de programación pueden ser necesarias operaciones para, por ejemplo, liberación de memoria o gestión de errores, que no se incluyen en la especificación pero sí en la documentación. También se puede modificar la sintaxis de las operaciones para hacer eficiente su implementación, cambiando por ejemplo un paso por valor a uno por referencia.

En C es necesario diferenciar operaciones con el mismo nombre en distintos TDAs, como pueden ser la creación y liberación de una instancia. Para ello, una forma es anteponer el nombre del TDA al de la operación, que irá seguido del nombre del tipo de sus elementos si se trata de un TDA contenedor. El nombre de los tipos, por su parte, debe identificarlos de forma adecuada sin indicar la representación interna de estos.

1.3. Implementación

En la fase de implementación, se escribe el código que implementa el comportamiento especificado en un lenguaje de programación concreto, y al mismo tiempo se genera la documentación del software, normalmente mediante programas que generan documentación a partir de comentarios en el código. Para escribir el código, se define primero la representación y a continuación se implementan las operaciones.

1.3.1. Representación

Primero se establece un tipo *rep* (estructura, tabla, etc.) en el que se puedan representar todos los valores del TDA y operar con ellos de forma eficiente. Establecemos también una **función de abstracción** $f_{Abs} : X \subseteq rep \to \mathcal{A}$ suprayectiva pero no necesariamente inyectiva. No todos los posibles valores de *rep* corresponden a valores del TDA, sino que este debe cumplir un **invariante de la representación**, modelado como $f_{Inv} : rep \to$ boolean con $f_{Inv}(x) = \text{True} \iff x \in X$. Todos los valores construidos con las operaciones del TDA deben cumplir este invariante.

1.3.2. Ocultación en C

El **encapsulamiento** permite juntar los datos y código que los manipula manteniéndolos aislados de posibles usos indebidos, de forma que el acceso a estos se realiza de forma controlada a través de una interfaz (en inglés *interface*, en maquinavajense «interfeih») definida.

C no permite encapsulamiento, pero tiene ciertos mecanismos para ocultar información mediante programación modular, tipos incompletos o apuntadores a `void`. Los apuntadores a `void` también se usan para obtener **genericidad**, creando herramientas de propósito general para posteriormente especializarlas.

En particular, cada TDA se define en un módulo separado. Las funciones que crean nuevos valores devuelven un puntero al valor o `NULL` si ha habido un error, y cada TDA debe tener una función para liberar la memoria ocupada por una instancia.

C no implementa excepciones, por lo que para la gestión de errores se suele utilizar uno de estos mecanismos:

- En la cabecera se declara una variable de tipo `extern int` que almacena un código de error asociado al error, y una función que, dado un código de error, devuelve el mensaje correspondiente.
- En la cabecera se declara una función que devuelve el mensaje asociado al último error ocurrido.

Capítulo 2

Estructuras de datos enlazadas

Muchas veces necesitamos estructuras de datos cuyo tamaño puede ir cambiando con el tiempo. Podemos usar representaciones contiguas (tablas), pero no son eficientes en ciertos casos porque las inserciones y eliminaciones pueden suponer reubicaciones de elementos.

Una **estructura de datos lineal enlazada con simple enlace** es una sucesión finita de nodos formados por un elemento y un enlace al siguiente (o una marca de fin como `NULL` si es el último).

Una estructura de este tipo se gestiona a través de un apuntador al primer nodo, inicialmente con valor `NULL`. Podemos definir, por ejemplo, operaciones para:

- Inicializar y liberar una lista enlazada.
- Imprimir sus elementos, comprobar si contiene uno y cuál es su índice, obtener el apuntador al nodo que contiene un elemento.
- Insertar un elemento al principio, al final, en un cierto índice o detrás de otro dado por el apuntador al nodo. Insertar en una lista enlazada ordenada.
- Eliminar un elemento al principio, al final, en un cierto índice o detrás de otro dado por el apuntador al nodo. Eliminar la primera ocurrencia de un elemento si existe, con un caso especial por eficiencia para cuando la lista está ordenada.

Algunas de estas operaciones pueden requerir modificar el apuntador al primer elemento, y por tanto es necesario pasarles un apuntador a dicho apuntador (o que devuelvan el nuevo valor de este). Para estos podemos añadir al principio un **nodo de encabezamiento** que apunta al primer elemento «real», simplificando el código a cambio de ocupar más memoria. Por otro lado, para ciertas operaciones conviene tener apuntadores tanto al principio como al final de la estructura enlazada.

También existen **estructuras de datos lineales doblemente enlazadas**, en las que los nodos no apuntan sólo al siguiente elemento sino también al anterior, permitiendo operaciones como eliminar un elemento de la lista con un apuntador al nodo correspondiente en vez de al anterior, por ejemplo.

Capítulo 3

Contenedores fundamentales

3.1. Pilas

Una **pila** o secuencia **LIFO** (*Last In First Out*) es una secuencia mutable de cero o más elementos en el que las inserciones, accesos y supresiones se realizan siempre por el mismo extremo, llamado **tope**. Operaciones:

```
Stack create();
void push(Stack s, Element e);
void delete(Stack s);
Element pop(Stack s);
int isEmpty(Stack s);
```

Podemos implementar una pila mediante una tabla o una lista enlazada, aunque si la tabla es de tamaño fijo tenemos que implementar la operación `int isFull(Stack s);`.

3.2. Colas

Una **cola** o secuencia **FIFO** (*First In First Out*) es una secuencia mutable de cero o más elementos en el que las inserciones se realizan en un extremo llamado **posterior** o **final** y los accesos y supresiones por el otro, llamado **anterior** o **frente**. Operaciones:

```
Queue create();
void append(Queue q, Element e);
void delete(Queue q);
Element pop(Queue q);
int isEmpty(Queue q);
```

Podemos implementar una cola mediante una lista enlazada o una tabla de tamaño fijo circular, de forma que conforme se van sacando elementos al principio se pueden ir añadiendo más una vez se haya llegado al final de la tabla, si bien esto requiere implementar la operación `int isFull(Queue q);`.

3.3. Listas

Una **lista** es una secuencia mutable de cero o más elementos ordenados de acuerdo a su posición. Los accesos, inserciones y supresiones pueden realizarse en cualquier posición de la lista, y cada valor de esta tiene asociado un valor de tipo **posición**. Existe además una posición que indica el final de la lista y se usa para insertar elementos. Operaciones:

```
List create();
void insert(List l, Position p, Element e);
void delete(List l, Position p);
Element get(List l, Position p);
void set(List l, Position p, Element p);
unsigned length(List l);
Position begin(List l);
Position end(List l);
Position nth(List l, unsigned index);
Position next(List l, Position p);
Position prev(List l, Position p);
```

Podemos implementar una lista mediante una tabla, en cuyo caso las posiciones son enteros, o mediante una lista enlazada. Si hace con una lista enlazada con simple enlace, conviene guardar los apuntadores de inicio y de fin, así como la longitud, y en este caso las posiciones son apuntadores al nodo anterior al que contiene el elemento, pues esto es necesario para borrar un elemento por posición.

Las listas implementadas por estructuras enlazadas permiten insertar y borrar elementos en $O(1)$, en vez de en $O(n)$ como sucedería tablas, si bien requieren un tiempo $O(n)$ para acceder a los elementos por índice en vez de $O(1)$.

3.4. Conjuntos

Un **conjunto** es una colección mutable (en este caso finita) de elementos no repetidos y sin ordenación. Operaciones:

```
Set create();
void insert(Set c, Element e);
void delete(Set c, Element e);
int contains(Set c, Element e);
unsigned cardinal(Set c);
List toList(Set c);
```

Podemos implementarlo como una lista enlazada o como una tabla, y en el último caso podemos aprovechar que los elementos no están ordenados para «rellenar huecos» al eliminar elementos en $O(1)$.

Capítulo 4

Recursión

Un proceso es **recursivo** si se especifica basándose en su propia definición. Una función es recursiva si se llama a sí misma, y está bien construida si contiene al menos un **caso base** y al menos un **caso general** o **de recursión** en el que los parámetros están cada vez «más cerca» del caso base, garantizándose que este se alcanza. Tipos de recursividad:

- **Directa** o **simple**: La función contiene una llamada explícita a sí misma.
- **Múltiple**: Hay más de una llamada recursiva en el cuerpo de la función.
- **Indirecta** o **cruzada**: La función invoca a otra que a su vez acaba llamando a la primera.
- **De cola** o **de extremo final**: La llamada recursiva es la última instrucción que ejecuta la función.
- **Anidada**: En algún parámetro de la llamada recursiva hay otra llamada recursiva.

La **pila de llamadas** (*call stack*), **de ejecución**, **de control**, **de función** o **de tiempo de ejecución** es una estructura dinámica que almacena información sobre las subrutinas activas en un programa. Cada llamada añade un **marco de pila** (*stack frame*), al que el profesor llama **registro de activación**, donde se guardan variables locales, parámetros, dirección de retorno y valor devuelto. Las llamadas recursivas se gestionan igual que el resto, por lo que pueden producir un **desbordamiento de pila** si la recursión es muy profunda.

El **árbol de recursión** es una representación gráfica de un algoritmo recursivo, en el que cada llamada se representa con un nodo, etiquetado con los parámetros que recibe, y de este parte un nodo hijo por cada llamada que hace la función a sí misma, siendo el nodo raíz la llamada inicial y los nodos hoja ejecuciones de los casos base. Este árbol sirve para decidir si el enfoque recursivo es apropiado o es mejor buscar otro enfoque como el iterativo.

El mecanismo de **expansión de recurrencias** sirve para calcular el tiempo de ejecución de un algoritmo recursivo. Para ello, se crea una **ecuación de recurrencia**, una función que define el tiempo de ejecución en función de los parámetros y se contiene a sí misma en la definición, y por inducción se obtiene una **forma cerrada** para esta ecuación, que no dependa de sí misma.

La recursión por lo general es más ineficiente que el diseño iterativo, por lo que no se debe usar cuando la recursividad es por cola, el árbol de recursión es lineal o el árbol es ramificado pero con nodos repetidos.

Capítulo 5

Estructuras arborescentes

Un **árbol** es un grafo simple conexo no dirigido y acíclico. Trabajaremos con **árboles con raíz etiquetados** finitos, en los que a uno de los nodos lo denominamos raíz y todos los nodos almacenan un valor o un elemento.

Los nodos conectados al nodo raíz son **hijos** del nodo raíz, siendo este el **padre** de estos nodos, e inductivamente llamamos hijos de un nodo n a los nodos $\{m_i\}_{i\in I}$ que estén conectados a él y no son su padre, y decimos que n es el padre de los m_i. Llamamos **nodo hoja** a un nodo que no tiene hijos.

Un **camino** de n_1 a n_k es una sucesión de nodos $n_1, \ldots, n_k$ donde cada nodo n_i es el padre del n_{i+1} para $i = 1, \ldots, k-1$, y decimos entonces que el camino tiene **longitud** $k-1$. Dados dos nodos n y m, si existe un camino de n a m se dice que n es **ancestro** de m y m es **descendiente** de n. El **subárbol** de un árbol por un nodo es el árbol formado por este nodo, que será la nueva raíz, y todos sus descendientes, preservando las aristas entre estos nodos.

La **altura** de un nodo es el máximo de las longitudes de caminos desde este nodo hasta cualquier descendiente suyo, y la **altura del árbol** es la altura de su raíz. La **profundidad** de un nodo es la longitud del camino desde la raíz hasta el nodo. El **nivel** i de un árbol es el conjunto de todos los nodos a profundidad i.

En general representamos un árbol mediante un apuntador a su raíz, y representamos un nodo mediante una estructura con su elemento y, bien su lista de hijos (normalmente), o un apuntador hacia su primer hijo («hijo izquierdo») y el hermano siguiente (siguiente hijo del padre, «hermano derecho»), si bien esto es poco habitual.

Existen tres formas principales de recorrer un árbol:

- **Preorden**: En cada nodo, se considera primero el elemento del propio nodo y luego cada hijo en preorden.
- **Inorden**: En cada nodo, se considera primero el primer hijo en inorden, después el elemento del propio nodo y finalmente el resto de hijos en inorden. Esto es útil en **árboles binarios**, donde cada nodo tiene un «hijo izquierdo» y un «hijo derecho» (cada uno puede no existir).
- **Postorden**: En cada nodo, se considera primero cada uno de los hijos en postorden y finalmente el elemento del propio nodo.

Un árbol se dice que está **balanceado** si su altura es la mínima dado el máximo de hijos por nodo que puede tener. Esto es útil porque minimiza el tiempo de ejecución a la hora de operar con ellos. Algunos tipos de árbol:

- **Parcialmente ordenado**: Los descendientes de un nodo poseen un valor no mayor al del propio nodo.
- **Árbol binario de búsqueda**: Árbol binario en el que el hijo izquierdo de un nodo y sus descendientes tienen un valor menor o igual al del propio nodo, a su vez menor o igual al del hijo derecho y sus descendientes.
- **Árboles AVL**: Árbol binario auto-balanceable, que o bien es vacío o cumple que los subárboles por ambos hijos son AVL y la diferencia en la altura de ambos (valor absoluto) es no mayor que 1.
- **Árboles B**: Un árbol B de orden M es aquél en que: cada nodo tiene como máximo M hijos; todos los nodos salvo la raíz tienen un valor formado como mínimo por $\frac{M}{2}$ claves; la raíz tiene al menos 2 hijos si no es al mismo tiempo hoja; todos los nodos hoja aparecen al mismo nivel; un nodo no hoja con k hijos tiene un valor formado por $k-1$ claves, y dado un nodo no hoja con claves $r_1, \ldots, r_m$, las claves de sus nodos hijo (y descendientes respectivos) deben ser: menores que r_1 para el primer hijo, entre r_{i-1} y r_i para el nodo i-ésimo ($i = 2, \ldots, m$), o mayores que r_m para el último nodo.

Aplicaciones:

- Representación de datos jerárquicos, como sistemas de ficheros y directorios.
- Búsqueda (y otras operaciones) de forma eficiente en colecciones de datos.
- Árboles de decisión en inteligencia artificial.