# C+−

Juan Diego Barrado Daganzo, Javier Saras González y Daniel González Arbelo
4º de Carrera

20 de mayo de 2024*

## Índice



*Este documento se actualiza, para consultar las últimas versiones entrar en el enlace https://github.com/JuanDiegoBarrado/PracticaPL

# 1. Especificaciones técnicas del lenguaje

## 1.1. Identificadores y ámbitos de definición

El lenguaje posee las siguientes características:

- **Declaración de variables**: se pueden declarar variables sencillas de los tipos definidos y variables *array* de estos tipos, de cualquier dimensión.
- **Bloques anidados**: se permiten las anidaciones en condicionales, bucles, funciones, etc. Si dos variables tienen el mismo nombre, la más profunda (en la anidación) tapa a la más externa.
- **Funciones**: se permite la creación de funciones. El paso por valor y por referencia de cualquier tipo a las funciones está garantizado.
- **Punteros**: para cada tipo se puede declarar un puntero a una variable de ese tipo, mediante la asignación de su dirección de memoria a la variable puntero.
- **Registros y clases**: se incluyen dos tipos adicionales: los registros como "saco de datos" —sin métodos— y las clases, tanto con datos como con métodos de función.
- **Declaración de constantes**: se incluye la posibilidad de declarar constantes por parte del usuario.

## 1.2. Tipos

El lenguaje C+−es un lenguaje fuertemente tipado, donde la declaración de tipos ha de hacerse de manera explícita y de forma previa al uso del identificador, es decir, que para poder usar una variable tengo que haberla declarado antes.

El lenguaje consta de los siguientes tipos predefinidos:

- Enteros
- Booleanos
- Clases
- Registros
- Arrays
- Funciones
- Punteros

sobre los que se habla más en profundidad en Tipado. Asimismo, existe la posibilidad de declarar tipos por parte del usuario a través de la instrucción `difain`. Los detalles, de nuevo, se exponen de forma más precisa en su correspondiente apartado.

### 1.2.1. Enteros y booleanos

Los tipos básicos del lenguaje son los enteros y los booleanos. A continuación, se expone la sintaxis de declaración de ambos y las operaciones permitidas para cada uno de ellos. El identificador `var` corresponderá al identificador asignado a la variable declarada con estos tipos.

- **Enteros**: `int var;`

  Operaciones habilitadas para el tipo:

  - Suma: `a + b`
  - Resta: `a - b`
  - Multiplicación: `a * b`
  - División: `a / b`
  - Potencia: `a^b`
  - Paréntesis: `()`
  - Menor: `a < b`
  - Mayor: `a > b`
  - Igual: `a == b`
  - Menor o igual: `a <= b`
  - Mayor o igual: `a >= b`
  - Distinto: `a != b`

  Los literales permitidos para la asignación de un valor a las variables de tipo entero son decimales, binarios y hexadecimales, que serán convertidos internamente todos a números decimales.

- **Booleanos**: `bul var;`

  Operaciones habilitadas para el tipo:

  - *Y* lógico: `a an b`
  - *O* lógico: `a or b`
  - *No* lógico: `!a`

  Las palabras reservadas para definir los dos literales booleanos que pueden asignarse a variables booleanas son `tru` y `fols`, indicando respectivamente verdadero y falso.

### 1.2.2. Clases y registros

Las clases y registros son tipos no básicos compuestos por atributos (registros) y por atributos y métodos (clases). Además, cada elemento de la clase o del registro podrá tener una visibilidad a elegir entre `public` y `praivet`, denotando el primero que el campo es accesible por llamadas desde fuera del tipo y el segundo, que solo es accesible a través de llamadas internas del tipo. Por defecto, las clases tendrán todos sus campos privados y los registros públicos.

- **Clases**: `clas var {...};`

  Opciones habilitadas para el tipo:

  - Acceso a atributos: `var.atributo`
  - Acceso a métodos: `var.metodo()`
  - Constructor: `var(args)`

- **Registros**: `estrut var {...};`

  Opciones habilitadas para el tipo:

  - Acceso a atributos: `var.atributo`
  - Constructor: `var(args)`

  En ambos casos, el prefijo `dis` en las referencias a atributos, es decir, `dis.campo` permitirá distinguir entre el campo con dicho nombre dentro del tipo o la variable local que pudiese tener el mismo nombre.

### 1.2.3. Arrays

Todos los tipos descritos en esta subsección pueden formar un array multidimensional. La declaración de arrays es estáctica, esto es, que el tamaño se debe conocer en tiempo de compilación.

- **Array**: `Tipo[DIMENSION] var;`
  Opciones habilitadas para el tipo:
  - Operador de acceso: `var[INDEX]`

### 1.2.4. Funciones

Las funciones, que a priori podría parecer que no son un tipo, se han declarado también como uno para poder hacer expresiones lamdba que luego poder pasar como argumento a otras funciones o utilizar en la ejecución del programa.

- **Funciones**: `func foo(Tipo arg1, ...) : TipoRetorno {...; return var;};`

En caso de que la función no devuelva ningún valor —lo que se suele llamar *procedimiento*—, la sintaxis de declaración se cambiaría por

- **Funciones**: `func foo(Tipo arg1, ...) {...; return;};`

La instrucción `return` final es obligatoria y solo está permitido que haya una y, además, que sea la instrucción final del cuerpo de la función.

En cuanto a los parámetros de la función, existe tanto la posibilidad de pasar los parámetros por valor como por referencia, siendo la primera la opción por defecto. Para indicar explícitamente que quiere pasar un parámetro por referencia, es necesaria añadir el caracter "&" al final del tipo del argumento.

Los arrays también pueden ser argumentos de una función y, a diferencia de los tipos primitivos, su paso por defecto es por referencia. Cabe destacar que existe la posibilidad de pasar como argumento un array de dimensión variable. En tal caso se escribirá similar a la siguiente sentencia:

```
func foo(int[] array) : TipoRetorno {...}
```

### 1.2.5. Punteros

Cualquiera de los tipos anteriores, incluyendo los propios arrays, son susceptibles de ser apuntados por un puntero específico. La declaración de punteros es como la declaración del tipo al que se pretende apuntar, pero añadiendo el caracter ˜ al final del tipo.

- **Puntero**[1]: `Tipo˜ var`
  Opciones habilitadas para el tipo:
  - Asociación a memoria dinámica: `var := niu Tipo`
  - Acceso al dato: `˜var`

El valor guardado en el puntero es la dirección de memoria del objeto al que apunta, pudiendo ser esta una dirección de la pila o del heap en función del caso.

[1] A modo de aclaración, la afirmación "La declaración de punteros es como la declaración del tipo al que se pretende apuntar, pero añadiendo el caracter ˜ al final del tipo" se mantiene exactamente igual para cuando el tipo no es un tipo simple. Por ejemplo, la declaración de un puntero a un array de cualquier tipo, se haría como `Tipo[DIMENSION]˜ var;`.

### 1.2.6. Tipos definidos por el usuario y constantes

Además de los tipos declarados por nosotros, permitimos la definición de tipos por parte del usuario a través de la siguiente instrucción:

- **Definición de tipos de usuario**: `taipdef nuevoTipo expresionDeTipos`.

Sin embargo, la manera de entender este tipo declarado por el usuario no es como un nuevo tipo sino como un alias de la expresión de tipos a la que se asignó.

La declaración de constantes literales está permitida e implementada a través de la siguiente instrucción:

- **Declaración de constantes**: `difain NOMBRE valor`

Nótese que la expresión anterior no indica de manera explícita el tipo al que pertenece la constante, sino que se conoce de manera implícita a través del valor asignado al identificador de la constante.

## 1.3. Instrucciones del lenguaje

A continuación se presenta el repertorio de instrucciones del lenguaje. Préstese atención a aquellas que terminan con el caracter ";" para delimitar su final:

- **Instrucción de asignación**: `:=`

```
int var = 3;
```

- **Instrucciones condicionales**: `if-els`, `suich-queis`

```
if (var > 3) {
    ...
}
els {
    ...
}
```

```
suich (var) {
    queis(val1):
        ...
        breic;
    queis(val2):
        ...
        breic;
    difolt:
        ...
        breic;
}
```

- **Instrucción de bucle**: `while`, `for`

```
guail (var > 0) {
    ...
}
```

```
for (int i = 0; i < n; i = i + 1) {
    ...
}
```

Se incluyen además las instrucciones `breic` y `continiu`.

- **Instrucciones de entrada salida**: `cein()` y `ceaut()`

```
cein(var);
ceaut(var);
```

- **Instrucción de retorno de función**: `return`

```
func foo(Tipo arg) : TipoRetorno {
    TipoRetorno var;
    ...
    return var;
}
```

## 1.4. Elementos de la estructura del código

El código comenzará con la definición —separada o entremezclada— de los tipos compuestos (clases, registros, funciones, etc.), de los tipos de usuario y de las constantes del programa. Finalmente, comenzará la función principal del programa y que sirve como punto de entrada, que es la que denotamos por `func mein() : int {...; return 0;}`.

Es importante destacar que, a diferencia de otros lenguajes como Python, donde la ejecución del código no requiere de una función "especial" que sirva como punto de entrada para el programa, en nuestro lenguaje es obligatorio que el código del programa esté dentro de esta función final y que fuera solo se permita la declaración de tipos y constantes que hemos explicado previamente. Esto quiere decir que cualquier variable global o expresión fuera del `mein` que no se ajuste a estas restricciones será entendida como un error sintáctico.

Por otro lado, se incluye la posibilidad de escribir comentarios en el código, esto es, líneas que en el momento de compilación serán ignoradas. Se permiten tanto comentarios de una línea, encabezados por "//" y terminados en un salto de línea, como comentarios multilínea, encabezados por "/*" y terminados con "*/".

A continuación, un ejemplo de código que se ajusta a las restricciones estipuladas:

```
func mein() : int {
    // La ejecucion del programa empieza aqui
    int var = 3;
    /*
    La llamada a la funcion siguiente calcula el
    factorial de la variable var.
    Almacenamos el valor en la variable res para
    mostrarla por pantalla a continuacion.
    */
    int res = factorial(var);
    ceaut(res);
    return 0;
}
```

## 2. Especificación léxica del lenguaje

Durante la explicación previa de las especificaciones del lenguaje, ya se han presentado algunas de las palabras reservadas o grafías destinadas a realizar determinadas operaciones. A continuación, se explica de forma más detallada los elementos del lenguaje y los símbolos reservados para cada fin.

- Los nombres de los identificadores de las variables han de ser expresiones alfanuméricas que no comiencen por números y que posiblemente tengan el caracter "_".
- Los literales permitidos para los enteros son decimales (p. ej. `var = 10`), binarios (p.ej. `var = 0b1001`) y hexadecimales (p. ej. `var = 0x1F2BC`).
- Los literales permitidos para los booleanos son las palabras reservadas `tru` y `fols`.
- Los espacios en blanco, tabuladores y saltos de línea, se eliminan internamente durante el reconocimiento léxico.
- Los comentarios monolínea comienzan por // y los multilínea /**/. Estos elementos se eliminan durante el reconocimiento léxico.
- Las palabras reservadas para los tipos definidos y para sus operadores son las especificadas en los correspondientes apartados de Tipos.
- Los delimitadores de los cuerpos de funciones e instrucciones `guail`, `if`, etc. son las llaves {}.
- El delimitador de final de instrucción o bloque en aquellas que lo requieren es el ;.
- La palabra `niu` es la reservada para la reserva de memoria del heap.
- Las palabras `taipdef` y `difain` son las palabras reservadas para la definición de tipos y constantes.
- Las palabras y símbolos reservados para el repertorio de instrucciones son los declarados en la sección Instrucciones del lenguaje.
- La palabra reservada para la función principal del programa es `mein`.

Las expresiones regulares que definen los elementos del léxico aquí descrito pueden consultarse en el documento *lexicon.l*.

## 3. Semántica

A continuación, se muestra la semántica del lenguaje asociada a la construcción sintáctica que la genera. Por espacio, no se incluyen todas las definiciones semánticas, sino solo aquellas más relevantes. Para consultar el listado completo, se puede revisar el documento *Tiny.cup*.

| Constructora | Prog : Declaraciones × FuncionPrincipal → Programa |
|---|---|
| Descripción | Construye un programa a partir de una serie de definiciones y una función principal `mein`. |
| Sintaxis | `Definiciones && mein` |

| Constructora | classDef : String × CuerpoClase → Class |
|---|---|
| Descripción | Construye una definición de clase dado un nombre y un cuerpo de clase. |
| Sintaxis | `clas id { cuerpoClase }` |

| Constructora | structDef : String × CuerpoStruct → Struct |
|---|---|
| Descripción | Construye una definición de registro dado un nombre y un cuerpo de registro. |
| Sintaxis | `estrut id { cuerpoStruct }` |

| Constructora | funcDef : String × Tipo × CuerpoFuncion → Struct |
|---|---|
| Descripción | Construye una definición de función con tipo de retorno dado un nombre y un cuerpo de función. |
| Sintaxis | `func id { cuerpoStruct }` |

| Constructora | typeDef : String × String × String → Tipo |
|---|---|
| Descripción | Construye una definición de tipo a modo de alias. |
| Sintaxis | `taipdef previousType newType;` |

| Constructora | decVar : Tipo × String → Declaracion |
|---|---|
| Descripción | Construye una declaración de variable del tipo especificado. |
| Sintaxis | `Tipo id;` |

| Constructora | Asign Ins : Id × Expresión → Instrucción |
|---|---|
| Descripción | Construye la instrucción de asignación |
| Sintaxis | `Id = Expresión;` |

| Constructora | If Ins : Expresión × Bloque → Instrucción |
|---|---|
| Descripción | Construye la instrucción condicional `If`. |
| Sintaxis | `If (Expresión)` {`Bloque`} |

| Constructora | While Ins : Expresión × Bloque → Instrucción |
|---|---|
| Descripción | Construye la instrucción iterativa `While`. |
| Sintaxis | `guail (Expresión)` {`Bloque`} |

| Constructora | Asign Ins : Id × Expresión → Instrucción |
|---|---|
| Descripción | Construye la instrucción de asignación |
| Sintaxis | `Id = Expresión;` |

| Constructora | New Ins : Id × Expresión → Instrucción |
|---|---|
| Descripción | Construye la instrucción de asignación de memoria dinámica `New` |
| Sintaxis | `Id = niu Expresión;` |

## 3.1. Binding

El binding es la asociación entre un identificador y el objeto que designa (variables, constantes, funciones...). Distinguimos entre los identificadores introducidos con su definición (en una declaración de una variable, una constante, una definición de función o alias de tipos...) frente a las apariciones de uso (el resto de apariciones).

La forma de relacionar las apariciones de uso de un identificador con su aparición de definición la hacemos recorriendo el AST conectando cada nodo de uso con una referencia al de definición que le corresponde apoyándonos en una pila de tablas de símbolos auxiliares que va guardando las definiciones de los símbolos teniendo en cuenta los niveles de anidamiento del programa (cada tabla asociada a un ámbito).

En cuanto a los tipos definidos por el usuario (typedef, class y struct), los vamos asociando a una tabla que guarda el nombre (String) y la definición raíz (Nodo del AST de la definición). Aplicamos una política de palabras reservadas con las definiciones que se van declarando para evitar que se usen esos mismos nombres de tipos para declarar nuevos structs o clases. En cuanto a los alias de tipos, estas palabras reservadas únicamente impiden que se usen como alias de otro tipo ya creado. A su vez, aprovechamos la estructura de datos que tenemos para que los alias se asocien con el nodo definición original del tipo (y para también poder aprovecharnos de esta construcción con los tipos primitivos, los metemos primero en la tabla de definiciones como si todo programa tuviese esos tipos básicos "declarados por el usuarioïmplícitos)

Tenemos una política de declaración previa para cualquier uso de identificadores: para poder utilizarlos, previamente tienen que estar declarados. En el caso de funciones recursivas, al estar dentro de la definición de la función ya cuenta como que la función esta declarada.

C+/- admite sobrecarga por argumentos, mismo nombre de función y distinto tipo de argumentos (sin tener en cuenta el tipo de retorno). Por eso durante el binding, asociamos cada uso de identificador de función con una lista de posibles funciones y se determinará en tipado la función concreta.

Tenemos visibilidad de declaraciones de variables según el ámbito en el que se estén: el vínculo más interno oculta al más externo.

## 3.2. Tipado

Como se mencionó en el apartado 1.2, este lenguaje es fuertemente tipado. Así pues, el tipo de todos los elementos del código se conocen en tiempo de compilación. Como se explicó en aquel apartado, permitimos la declaración de constantes con la cláusula `difain`. Estas constantes también tienen un tipo que se conoce en el momento de compilación: no es un simple alias, sino que en caso de no poderse tipar como uno de los tipos permitidos en el lenguaje, el compilador se interrumpirá y avisará al usuario.

En general, la compatibilidad entre tipos distintos no existe; no se podrán hacer asignaciones entre tipos distintos o trabajar con ellos de forma indistinguible.

No obstante, hay una excepción a este caso, que son los punteros y los arrays. Estos dos últimos tipos sí son compatibles, siempre y cuando lo sean sus tipos internos. Así pues, al declarar una variable de tipo `int[5]` y otra de tipo `int~`, se podrá hacer una asignación entre ellas. No obstante, no se podría si la de tipo array fuera de tipo `bul[5]`, por ejemplo.

Con esta decisión de diseño surge otra necesidad: los arrays no sirven de nada si no se puede iterar sobre ellos. Por tanto, se debe habilitar un método para "iterar sobre punteros" cuando sabemos que apuntan a un array. Por ello, el lenguaje hace una sobrecarga de los operadores binarios `+` y `-`. De esta forma, podremos iterar sobre un puntero haciendo una instrucción como `ptr = ptr +/- 1`, que proporciona un comportamiento similar al de un iterador como el que proporcionan lenguajes de más alto nivel. Hay que tener en cuenta que esta es la única salvedad en cuando a dichos operadores: se permite operar un puntero o array con un entero (y, en el caso de la resta, el entero debe ser el operando derecho). No se pueden utilizar estos operadores para otros casos. Cabe mencionar que, para avanzar o retroceder con un puntero, siempre se hará `+ 1` o `- 1`, independientemente del tamaño del tipo interno.

Mostramos a continuación un ejemplo de lo mencionado:

Listing 1: Ejemplo de uso de arrays y punteros de formas equivalentes.

```
1 func mein(): int {
2     int array[5];
3     for (int i = 0; i < 5; i = i + 1) {
4         array[i] = i;
5     }
6     int~ ptr = array;
```

```
7       for (int i = 0; i < 5; i = i + 1) {
8           ceaut(array[i]);
9           ceaut(~ptr);
10          ptr = ptr + 1;
11      }
12
13      return 0;
14  }
```

En el ejemplo anterior se crea una lista de 5 enteros que contiene en cada posición la propia posición como valor. Tras esto, se declara un puntero que "apunta" a la misma lista, y finalmente se itera con otro bucle imprimiendo el valor de cada posición del array junto con el valor del puntero. El resultado de la ejecución será la secuencia `0011223344`, como es de esperar, pues lo que estamos haciendo es básicamente imprimir cada valor del array dos veces seguidas.

### 3.2.1. Tipos en funciones

Por defecto, en las llamadas a funciones los parámetros se pasan por valor. No obstante, se permite el paso de parámetros por referencia añadiendo el operador `&` al argumento en la declaración de la función. Internamente, el funcionamiento se corresponde con el paso de un puntero en vez del valor. Siendo más claros, cuando se pasa un parámetro por referencia a una función, internamente se está pasando un puntero a la variable en cuestión, pero dentro del cuerpo de la función se interpretará como lo que es, un puntero, y las operaciones se ejecutaran en consecuencia. De esta forma, permitimos al usuario que use y modifique el valor real de los parámetros, pero no se le pide que maneje los datos de forma distinta a como lo haría normalmente.

A todo lo mencionado anteriormente tenemos que especificar un caso particular, y es el de los arrays. En el caso de los arrays no se permite el paso por referencia explícito, sino que es así por defecto. Es decir, los arrays no se pasan por copia. Esto se hacer por dos motivos: el primero es que resulta muy caro en general. Implica un gasto en tiempo y memoria innecesario, y con haciéndolo de esta manera no se impide al usuario hacer nada, puede hacer copias dentro de la función. El segundo es que aporta homogeneidad al paso de arrays dinámicos. En nuestro lenguaje se permite que los arrays no tengan tamaño definido como parámetro de funciones, y el manejo de estos es distinto al de los arrays normales. En particular, lo más razonable en este caso es que las modificaciones en el array dentro de la función se traduzcan en cambios fuera de esta, y forzar que este sea el comportamiento por defecto hace que todo sea más homogéneo y entendible.

Por último, de forma similar a forzar que el paso de arrays a funciones sea por referencia, entre otras cosas, por el coste que supone, se restringe también el tipo de retorno de las funciones, de forma que se permite como tipo de retorno cualquiera excepto los arrays. Si se quiere hacer algo similar a devolver un array como retorno de una función, se deberá crear antes de la llamada a la función y pasarla por referencia a la función. De esta forma, además, no se restringe el tamaño del array "de retorno" de la función, bastando con declarar el argumento como array dinámico, es decir, sin tamaño definido.

# A. Ejemplos de programas habituales

A continuación, exponemos una serie de ejemplos sencillos de cómo se escribirían ciertos programas en nuestro lenguaje. No obstante, en la carpeta de *test* del proyecto hay gran variedad de códigos de ejemplo para probar.

Listing 2: Ejemplo de programa típico en $C+-$.

```
func mein() : int {
    int a;
    int b;
    cein(a);
    cein(b);
    int c = a + b;
    ceaut(c);
    return 0;
}
```

Listing 3: Ejemplo de programa iterativo en $C+-$.

```
func maximo(int lista[], int tam) : int {
    int maximo = -1;
    int i = 0;
    guail(i < tam) {
        if(lista[i] > maximo) {
            maximo = lista[i];
        }
        i = i + 1;
    }
    return maximo;
}
```

Listing 4: Ejemplo de arrays multidimensionales en $C+-$.

```
difain tam 5;

func multiplicarMatrices(int A[][], int B[][]) : int~~ {
    int~~ res;
    int C[5][5];
    res = C;
    int i = 0;
    int j;
    guail(i < tam) {
        j = 0;
        guail(j < tam) {
            C[i][j] = 0;
            j = j + 1;
        }
        i = i + 1;
    }
    int k;
    i = 0;
    guail(i < tam) {
        j = 0;
        guail(j < tam) {
            k = 0;
            guail(k < tam) {
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
                k = k + 1;
            }
            j = j + 1;
        }
        i = i + 1;
```

```
    }
    return res;
}
```

Listing 5: Ejemplo de programa recursivo y sentencia `if-els` en $C+-$.

```
func factorial(int num) : int {
    int res;
    if(num == 0){
        res = 1;
    }
    els {
        res = num * factorial(num - 1);
    }
    return res;
}
```

Listing 6: Ejemplo de uso de registros y sentencia `suich` en $C+-$.

```
estrut Alumno {
    int DNI;
    int nota;

    Alumno(int DNI, int nota) {
        dis.DNI = DNI;
        dis.nota = nota;
    }
}

func aprobado(Alumno a) : bul {
    bul res;
    suich (a.nota) {
        queis(0):
        queis(1):
        queis(2):
        queis(3):
        queis(4):
            res = fols;
            breic;
        queis(5):
        queis(6):
        queis(7):
        queis(8):
        queis(9):
        queis(10):
        difolt:
            res = tru;
            breic;
    }
    return res;
}

func nota(Alumno~ a) : int {
    return (~a).nota;
}

func mein():int {
    Alumno daniel = Alumno(123456789, 6);
    Alumno~ danielptr = &daniel;
    ceaut(nota(danielptr));
    ceaut(aprobado(daniel));

    return 0;
```

```
45 }
```