PROYECTO PROCESADORES DE LENGUAJES

Especificación del Lenguaje Basic C++

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN
UNIVERSIDAD COMPLUTENSE DE MADRID

Doble Grado en Ingeniería Informática y Matemáticas

Curso 2024-2025

Integrantes:
Diego Martínez López
Damián Benasco Sánchez
Virginia Chacón Perez

Índice

1 Introducción

En este documento especificamos un lenguaje de programación que hemos llamado Basic C++ y que va a ser el proyecto de la asignatura. Los programas de este lenguaje partirán de una función `main`, que servirá como punto de comienzo del programa, y que no recibirá atributos y tendrá un valor de retorno vacío.

Los comentarios del código del programa irán precedidos de // y cada línea estará separada por ;.

2 Tipos

Nuestro lenguaje tendrá los siguientes tipos de datos:

2.1 Primitivos

- Enteros: Son el conjunto de números enteros, pertenecientes al conjunto **Z**. Los vamos a identificar con la palabra `int`. Pueden ir precedidos de tantos 0 como desee el usuario, pero esto no afecta al valor final de la variable.
- Booleanos: Son el conjunto de valores lógicos que se evalúan a `true` o `false`. Se identificarán mediante las palabras reservadas true y false. Su identificador será el tipo bool.

2.2 No primitivos

- Arrays: Lista de elementos de un mismo tipo, que podrá ser cualquiera del tipo que permita nuestro lenguaje. Tendrán un tamaño fijo predefinido, que será un entero no negativo. Se identificarán con la expresión:

  ```
  tipo nombre_array [tam_array];
  ```

  Permitiremos arrays de otras dimensiones, construyendo arrays anidados:

  ```
  tipo nombre_array[tam1][tam2]..[tamn];
  ```

  donde tam_i es el tamaño de esa dimensión.

- Tipos Estructurados: Colección de elementos de diferentes tipos, que el usuario decide. Estos elementos se llaman campos del struct, y el usuario podrá elegir cualquiera de los tipos disponibles en el lenguaje. Para la declaración de un tipo estructurado, se usará el identificador `struct` seguido del nombre que el usuario quiera darle al tipo estructurado. Cabe destacar que todos los campos del `struct` son públicos, es decir, podrán ser accedidos desde fuera de ellos. La sintaxis para definir cada campo es la misma que la declaración de variables, que veremos más adelante. Así, tendríamos:

  ```
  struct nombre{
      tipo1 nombre1;
      tipo2 nombre2;
  };
  ```

- Clases: Colección de elementos, que pueden ser tanto tipos disponibles en el lenguaje, incluyendo otras clases, como funciones que tienen acceso a los elementos de una misma clase. Una clase es como un tipo estructurado que además puede incluir métodos. Los atributos de la clase solo podrán ser accesibles desde la propia clase. Para declararla usaremos la palabra reservada `class`:

  ```
  class nombre{
      tipo1 nombre1;
      tipo2 nombre2;
      tipo_ret metodo(tipo_p parametro);
  };
  ```

  Para desambiguar el nombre de parámetros o de variables locales, que pueden coincidir con el nombre de variables de la clase, usaremos la palabra reservada `this`. De esta manera, si un identificador está precedido por `this->` hará referencia a un atributo de la clase. Si no se especifica nada, si hubiera algún tipo de ambigüedad, se referirá al parámetro. Ejemplo:

```
// Constructor de la clase Persona
Persona(string nombre) {
    this->nombre = nombre; // Desambiguar usando this->
}
```

- Punteros: almacenan la dirección de memoria de la variable a la que apuntan. Su declaración consistirá en el tipo al que apuntan seguido de un @:

  ```
  tipo @ nombrePuntero
  ```

- Definición de tipos de usuario (typedef): sirven para que el usuario defina tipos a partir de los ya existentes, renombrando un tipo por un alias elegido por el usuario. La sintaxis que seguiremos será con la palabra reservada type:

  ```
  type tipo alias_usuario;
  ```

  Por último, cabe destacar que llevaremos a cabo la comprobación de tipos de manera estática, asegurando que los errores de tipo se detecten en tiempo de compilación. Esto garantizará que las variables y expresiones tengan un tipo bien definido, evitando conversiones implícitas peligrosas y verificando la correcta compatibilidad en asignaciones, operaciones y llamadas a funciones. Además, los operadores y expresiones que estarán incluidos en el lenguaje se especifican a continuación, así como los ámbitos de definición (como bloques anidados) de cada tipo.

2.3 Expresiones y operadores

En primer lugar, en nuestro lenguaje los tipos anteriores serán instanciados como variables siguiendo la sintaxis ya vista en el punto anterior. Estas se podrán combinar mediante operadores y funciones para dar lugar a expresiones. Veamos los operadores soportados por Basic C++, pues las funciones las veremos en el apartado siguiente.

2.3.1 Operadores y expresiones aritméticas

- **Suma**: utilizamos el operador binario `+` para expresar la suma de dos enteros:

  `expresión_aritmética + expresión_aritmética`

- **Resta**: utilizamos el operador binario `-` para expresar la resta de dos enteros:

  `expresión_aritmética - expresión_aritmética`

- **Opuesto**: utilizamos el operador unario `-` para representar el opuesto de un entero:

  `-expresión_aritmética`

- **Positivo**: operador unario `+` para indicar que un entero es positivo:

  `+expresión_aritmética`

- **Multiplicación**: utilizamos el operador binario `*` para expresar la multiplicación de dos enteros:

  `expresión_aritmética * expresión_aritmética`

- **División**: utilizamos el operador binario `/` para expresar la división del primer entero entre el segundo:

  `expresión_aritmética / expresión_aritmética`

- **Módulo**: utilizamos el operador binario `%` para expresar el módulo del primer entero entre el segundo:

  `expresión_aritmética % expresión_aritmética`

2.3.2 Operadores relacionales

Ahora veamos otros operadores infijos que usan expresiones aritméticas pero que se evalúan a cierto o falso:

- **Menor** `<`: utilizamos el operador binario ¡ para indicar si el valor de la expresión de la izquierda es menor que el valor de la expresión derecha:

  `expresión_aritmética < expresión_aritmética`

- **Mayor**`>`:utilizamos el operador binario ¿ para indicar si el valor de la expresión de la izquierda es mayor que el valor de la expresión derecha:

  `expresión_aritmética > expresión_aritmética`

- **Menor o igual** `<=`: utilizamos el operador binario ¡= para indicar si el valor de la expresión de la izquierda es menor o igual que el valor de la expresión derecha:

  `expresión_aritmética <= expresión_aritmética`

- **Mayor o igual** `>=`: utilizamos el operador binario ¿= para indicar si el valor de la expresión de la izquierda es mayor o igual que el valor de la expresión derecha:

  `expresión_aritmética >= expresión_aritmética`

- **Igual** `==`: utilizamos el operador binario == para indicar si el valor de la expresión de la izquierda es igual que el valor de la expresión derecha:

  `expresión_aritmética == expresión_aritmética`

- **Distinto de** `!=`: utilizamos el operador binario != para indicar si el valor de la expresión de la izquierda es distinto que el valor de la expresión derecha:

  `expresión_aritmética != expresión_aritmética`

2.3.3 Operadores y expresiones booleanas

- **And** `&&`: devuelve cierto si ambas expresiones booleanas son ciertas.

  `expresion_booleana && expresion_booleana`

- **Or** `||`: devuelve cierto si al menos una de las expresiones booleanas es cierta.

  `expresion_booleana || expresion_booleana`

- **Not** `!`: devuelve la negación de la expresión booleana.

  `!expresion_booleana`

2.3.4 Operadores de acceso

- **Índice** `[]`: para acceder a la posición i-ésima de un array utilizamos el operador de índice.

  `nombre_array[indice]`

- **Dirección** `&`: para acceder a la dirección de memoria de una variable.

  `&nombre_variable`

- **Contenido de puntero** `@`: para acceder al valor apuntado por un puntero.

  `@nombre_puntero`

- **Acceso a estructuras y clases** `.`: para acceder a los campos de una estructura o los métodos de una clase.

  ```
  nombre_struct.nombre_campo
  nombre_clase.nombre_funcion(p1, ..., pn)
  ```

2.3.5 Prioridad de operadores

A continuación, se muestra la jerarquía de operadores en Basic C++, donde una prioridad más baja indica una mayor precedencia en la evaluación de expresiones. La **asociatividad** indica si los operadores se evalúan de izquierda a derecha o de derecha a izquierda.

Prioridad	Operador	Tipo	Asociatividad
1 (más alta)	`@` (contenido de puntero)	Acceso	Derecha a izquierda
2	`[]` (índice en array)	Acceso	Izquierda a derecha
3	`.` (acceso a struct/clase)	Acceso	Izquierda a derecha
4	`&` (dirección de memoria)	Acceso	Derecha a izquierda
5	`!` (negación lógica)	Lógico	Derecha a izquierda
6	`+`, `-` (unarios)	Aritmético	Derecha a izquierda
7	`*`, `/`, `%`	Aritmético	Izquierda a derecha
8	`+`, `-` (binarios)	Aritmético	Izquierda a derecha
9	`<`, `>`, `<=`, `>=`	Relacional	Izquierda a derecha
10	`==`, `!=`	Relacional	Izquierda a derecha
11	`&&`	Lógico	Izquierda a derecha
12	`||`	Lógico	Izquierda a derecha

Como observación, cabe destacar que el usuario podrá establecer prioridades mediante el uso de paréntesis.

3 Instrucciones

3.1 Asignación

La instrucción de asignación permite almacenar valores de cualquier tipo en variables del mismo tipo. La sintaxis que vamos a usar en nuestro lenguaje consiste en una variable y una expresión del tipo de la variable separadas por un símbolo `=`:

```
variable = expresion;
```

donde el tipo de la expresión es el mismo que el de la variable a la que se asigna. En el caso de los **arrays**, tanto la variable como la expresión deben tener el mismo tipo interno y el mismo tamaño.

Ejemplo:

```
a = 5;              // Asigna 5 a la variable 'a'
b = 10;             // Asigna 10 a la variable 'b'
c = a + b;          // Asigna la suma de 'a' y 'b' a 'c'

arr[0] = 5;         // Asigna 5 al primer elemento del array 'arr'
arr[1] = 10;        // Asigna 10 al segundo elemento del array 'arr'

ptr = &a;         // Asigna la direccion de memoria de 'a' al puntero 'ptr'
@ptr = 20;        // Asigna 20 a la variable a la que apunta 'ptr' (es decir, a 'a')

ptrArr = &arr[0]; // Asigna la direccion del primer elemento del array 'arr'
                  // al puntero 'ptrArr'
@ptrArr = 15;     // Asigna 15 al primer elemento del array a traves del puntero
```

3.2 Declaraciones

3.2.1 Declaraciones de variables

Para instanciar una variable de un tipo cualquiera que no sea un array, utilizamos la sintaxis de tipo seguida del nombre de la variable, en caracteres alfanuméricos. El identificador no puede coincidir con el identificador de otra variable declarada en el mismo ámbito, que veremos en el punto 5. La sintaxis es la siguiente:

```
tipo id_variable;
```

En el caso de arrays, la sintaxis es la misma, pero indicando además el tamaño de cada una de sus dimensiones entre corchetes:

```
tipo id_variable[t1]...[tn];
```

Además, permitimos opcionalmente inicializar la variable en la propia declaración, mediante el operador de asignación descrito anteriormente:

```
tipo id_variable = expresion;
```

Ejemplo:

```
int a = 5;              // Declara la variable entera 'a' y le asigna un 5
```

3.2.2 Declaraciones de constantes

En nuestro lenguaje permitiremos la declaración de variables cuyo valor permanecerá constante durante todo el programa. No se podrá asignar un nuevo valor a una variable declarada como constante en ningún lugar del código, excepto en su declaración. La sintaxis será la misma que la declaración de una variable, pero precedida de la palabra reservada `const`:

```
const tipo id_variable = expresion;
```

Es obligatorio que la declaración de una constante incluya una asignación.

Ejemplo:

```
const int MAX_VALOR = 100;    // Declaracion de una constante entera
const bool ESTADO_ACTIVO = true;    // Declaracion de una constante booleana
```

3.2.3 Declaraciones de funciones

Permiten crear funciones que pueden ser invocadas desde el programa principal o desde otras funciones. Se permite definir únicamente la cabecera de la función o bien la función con su implementación.

1. **Declaraciones de cabeceras**: Se escribe el valor de retorno de la función seguido del identificador y de los argumentos entre paréntesis, terminando en un punto y coma:

   ```
   tipo id_funcion (tipo1 nombre1, ..., tipon nombren);
   ```

2. **Declaración e implementación**: Se sigue la misma sintaxis que en la declaración de cabecera, pero en lugar de terminar con punto y coma, se introduce el cuerpo de la función entre llaves:

   ```
   tipo id_funcion (tipo1 nombre1, ..., tipon nombren) {
       cuerpo
   }
   ```

En la sección de ámbitos de definición veremos que el ámbito de los atributos de la función se restringe a su cuerpo de definición.

Ejemplo:

```
int sumar(int a, int b);    // Devuelve un entero y recibe dos enteros
bool esPar(int numero){
    return numero%2 == 0;
}     // Devuelve un booleano y recibe un entero
```

3.2.4 Observaciones

1. **Función principal**: Cada programa debe tener una única función `main` de tipo `void`, que será el punto de inicio del programa. No es posible sobrecargar esta función.

2. **Cabeceras sin implementación**: Una cabecera declarada pero no implementada no podrá usarse. Si se implementa una cabecera, la implementación debe coincidir exactamente con la declaración en cuanto a tipo de retorno, nombre y tipos de los parámetros.

3. **Sobrecarga de funciones**: Se permite la sobrecarga de funciones con el mismo nombre, siempre que sus listas de parámetros sean distintas. Al invocar la función, el compilador seleccionará aquella cuya lista de parámetros coincida con los argumentos proporcionados.

4. **Paso de parámetros por valor y por referencia**: Se permite el paso de parámetros tanto por valor como por referencia. Si un parámetro se pasa por **referencia**, se antepone un `&` al tipo en la declaración y cualquier cambio del parámetro dentro de la función persistirá fuera de ella. Si no se incluye `&`, el parámetro se pasa por **valor**, es decir, se crea una copia del argumento, y cualquier modificación dentro de la función no afectará a la variable original.

```
void funcion(int &parametro_ref, int parametro_valor);
```

5. **Instrucción de retorno**: Indica el fin de la ejecución de la función. Si la función es de tipo `void`, simplemente finaliza con la instrucción `return;` Sin embargo, si la función devuelve un valor, la instrucción `return` debe ir acompañada de una expresión del mismo tipo de retorno de la función.

```
void funcionVoid() {
    return;
}

int funcionEntero() {
    return 5;
}
```

6. **Arrays como parámetros**: Si una función tiene un array como parámetro, no es necesario especificar su tamaño en la declaración.

```
void funcion(int array[]);
```

Ejemplos:

3.2.5 Observaciones

```
// Cabeceras sin implementacion: Declaracion e implementacion deben
// coincidir exactamente.
void miFuncion(int parametro); // Declaracion

void miFuncion(int parametro) { // Implementacion
    // Cuerpo de la funcion
}

// Sobrecarga de funciones: Permite definir funciones con el mismo nombre pero
// parametros distintos.

// Multiplicacion de un entero y un entero
int multiplicar(int a, int b) {
    return a * b;
}

// Multiplicacion de un entero y un puntero
int multiplicar(int a, int@ b) {
    return a * (@b);
```

```
}

// Paso de parametros por valor y por referencia:
void cambiarValor(int &parametro_ref, int parametro_valor) {
    parametro_ref = 10; // Modifica el valor fuera de la funcion
    parametro_valor = 20; // No afecta al valor original
}
```

3.3 Condicionales

En este lenguaje utilizaremos condicionales para permitir tomar decisiones en función de la evaluación de ciertas expresiones booleanas.

3.3.1 Instrucción If Else

Utilizaremos la palabra reservada if seguida de una expresión booleana entre paréntesis, y posteriormente, una apertura y cierre de llaves que corresponda al cuerpo del if. Además, cabe destacar que en nuestro lenguaje SIEMPRE va a haber apertura y cierre de llaves, nunca permitiendo su omisión, a diferencia de otros lenguajes conocidos. Así, la sintaxis sería:

```
if (ExpresionBooleana) {cuerpo}
```

Respecto del else, análogamente, reservaremos esta palabra para identificarlo e irá seguido de una llave de apertura y cierre entre las que estará su cuerpo.

```
else {cuerpo}
```

3.3.2 Instrucción Switch

Implementaremos una instrucción con salto de ramas de tipo case la cual comenzará por la palabra reservada switch. Le seguirá un paréntesis de apertura y clausura que contendrán una expresión aritmética que determinará qué salto es el que se tomará. Seguirán una apertura y cierre de llaves entre las cuáles definiremos los diferentes casos de salto con la palabra reservada case seguida de una expresión aritmética que deberá ser constante y, finalmente, :. Tras evaluar la Expresión Aritmética del switch, se tomará el caso con mismo valor que el resultado de dicha expresión aritmética.

En caso de que varias expresiones tengan el mismo valor, se tomará la primera de ellas y, una vez tomado el salto, se ejecutará el cuerpo de dicho caso así como todos los que se encuentren por debajo de él. Para evitar esto, tendremos la expresión break que explicaremos más adelante.

Podremos añadir un caso por defecto que estará identificado por la palabra default al cual se saltará si ninguno de los otros casos se cumple.

```
switch(ExpresionAritmetica){
    case ExpresionAritmeticaCte1:
            cuerpo caso1
    case ExpresionAritmeticaCte2:
            cuerpo caso2
    etc

    case ExpresionAritmeticaCteN:
        cuerpo casoN
    default:
        cuerpo default
}
```

Ejemplos:

```
// Ejemplo de if-else
if (x > 5) {
    // Si x es mayor que 5, se ejecuta este bloque de codigo
```

```
    y = 10;
} else {
    // Si x no es mayor que 5, se ejecuta este bloque de codigo
    y = 0;
}

// Ejemplo de switch-case
switch(x) {
    case 1:
        // Si x es igual a 1, se ejecuta este bloque
        y = 10;
        break;
    case 2:
        // Si x es igual a 2, se ejecuta este bloque
        y = 20;
        break;
    default:
        // Si x no es ni 1 ni 2, se ejecuta este bloque
        y = 0;
}
```

3.4 Bucles

3.4.1 Bucle While

Tratará de un bucle que comenzará por la palabra reservada while e irá seguida de un paréntesis de apertura y uno de cierre entre los cuáles habrá una Expresión Booleana. Este bucle iterará hasta que dicha Expresión Booleana sea evaluada a False, y esta comprobación se hará al inicio de cada iteración. Tras estos paréntesis, habrán una llave de apertura y de cierre y entre ellas el cuerpo del propio while.

```
while(ExpresionBooleana) {cuerpo}
```

3.4.2 Bucle For

La expresión comenzará por la palabra reservada for seguida de un paréntesis de apertura y cierre donde deberemos tener la siguiente estructura dentro de ellos, primero la declaración de la variable de iteración (obligaremos a que sea de tipo entero) seguido de la condición de parada y terminando por la modificación que recibirá la variable de iteración tras cada vuelta. Tras esto irán una llave de apertura y cierre entre las que estará el cuerpo del bucle.

```
for(DeclaracionVariableEntera;ExpresionBooleana;ModificacionVariable){}
```

Ejemplos:

```
// Ejemplo de bucle while
int x = 0;
while(x < 10) {
    // Se ejecutara mientras x sea menor que 10
    x = x + 1;
}

// Ejemplo de bucle for
for(int i = 0; i < 10; i = i + 1) {
    // Este bucle se ejecutara 10 veces
    x = x + i;
}
```

3.5 Instrucción Break

La instrucción break se podrá utilizar tanto dentro de bucles como dentro de un caso de una instrucción switch. Su sintaxis será palabra reservada break seguida de ;

Su función será la de finalizar el bucle o caso en el que estemos. Destacamos que solo "saldremos" del bucle o case más interno en caso de tener varios anidados.

3.6 Instrucción new

La instrucción new es una nueva instrucción que se utiliza para la reserva de una nueva región de memoria dinámica. Se utilizará para asignarle a un puntero una región de memoria dinámica, del mismo tamaño que el tipo, mediante la inferencia de tipos. Un ejemplo es:

```
int @ p = new int;  // Reserva memoria para un entero
```

3.7 Llamadas a funciones

Para llamar o invocar a una función, se debe escribir el nombre de la función, seguido de una lista de parámetros entre paréntesis separados por comas, que representan los atributos de la función. Como todas las instrucciones, debe acabar en ;.

```
nombre_funcion(p1,...,pn);
```

Además, se deben añadir tantos parámetros como argumentos hay en la función, Si al declarar la función se establece un parámetro por valor, al llamar a la función se le puede pasar cualquier variable de ese tipo, pero si se establece por referencia, debe ser un designador con el mismo tipo. Además, si el tipo de retorno de la función no es void, la invocación de la función debe aparecer en un lugar donde el valor devuelto se utilice.

3.8 Instrucciones de E/S

Para que el usuario final pueda interactuar con el programa a través de la consola, introducimos las instrucciones de entrada y salida. La instrucción para escribir por pantalla será cout. La sintaxis será la palabra cout seguida por una expresión aritmética o booleana entre paréntesis, terminando la instrucción en ';'.

```
cout(Expresion_Aritmetica/Expresion_booleana)
```

Además, tenemos dos posibilidades, si se trata de una expresión aritmética, el valor devuelto será el resultado de evaluar dicha expresión, mientras que si se trata de una expresión booleana, el valor devuelto tras evaluar la expresión, será true por lo que se imprimirá un '1' o false, por lo que se imprimirá un '0'.

Por otro lado, la expresión de lectura de una entrada será true. De nuevo, la sintaxis será parecida a la del cout, pero esta vez será la palabra true seguida de una variable entre paréntesis. La variable podrá ser de tipo entero o booleano, en función del valor introducido por el usuario, mediante inferencia de tipos el compilador será capaz de saber el tipo de la misma. En el caso de ser booleana, el usuario deberá teclear 'true' para asignar un valor a true y 'false' para asignar un valor a false.

```
cin(variable)
```

Ejemplos:

```
// Instruccion break en un bucle
int i = 0;
while(i < 10) {
    if(i == 5) {
        break;  // Salimos del bucle cuando i es igual a 5
    }
    i = i + 1;
}
```

```
// Instruccion break en un switch
switch(x) {
    case 1:
        // Accion para el caso 1
        break;  // Salimos del case 1
    case 2:
        // Accion para el caso 2
        break;  // Salimos del case 2
    default:
        // Accion para el caso default
}

// Instruccion new para reserva de memoria dinamica
int @p = new int;  // Reserva memoria para un entero

// Llamada a una funcion
int suma(int a, int b) {
   return a + b;
}

int resultado = suma(5, 10);  // Llamada a la funcion suma con parametros 5 y 10

// Instrucciones de E/S
cout(5 + 10);  // Imprime 15 en pantalla

int numero;
cin(numero);  // Lee un valor entero del usuario y lo almacena en numero

bool esVerdadero;
cin(esVerdadero);  // Lee un booleano del usuario y lo almacena en esVerdadero
```

4 Ámbitos de definiciones

En nuestro lenguaje, tenemos instrucciones declarativas que abarcan ciertos ámbitos, es decir, una declaración de, por ejemplo, una variable, no es considerada en todas las partes del código. Los ámbitos estarán anidados unos dentro de otros, y una declaración será únicamente visible en el ámbito que ha sido declarada y en todos ámbitos que están contenidos en él. Así, las definiciones globales estarán definidas en un ámbito global, que será el propio fichero y así seguiremos restringiendo el ámbito que abarca una declaración.

Otros ámbitos son las siguientes regiones:

- Parámetros y cuerpo de una función: los parámetros de una función serán únicamente visibles en el cuerpo de la función, así como todo tipo de variable declarada en él.
- Cuerpo de un bucle: las declaraciones dentro de un cuerpo de un bucle, tanto de un for como de un while será únicamente visible en el cuerpo del bucle, es decir los cuerpos de los bucles forman un nuevo ámbito. Además en el caso de los bucles for las definiciones en las condiciones del bucle pertenecen también al ámbito del cuerpo del bucle y son visibles en éste.
- Cuerpo del if: el cuerpo del if forma un nuevo ámbito de definición con lo que las declaraciones dentro de éste serán únicamente visibles dentro del cuerpo.
- cuerpo de los else: forman un nuevo ámbito de definicion y funcionan al igual que lo explicado con el cuerpo del if.
- Casos de los switch: cada caso del switch forma un nuevo ámbito de definición, con lo que las definiciones en uno de sus casos no serán visibles en el resto de los casos, ni fuera del switch.

- Cuerpo de una clase o un struct: al igual que los casos anteriores el cuerpo de una clase o de un struct forma un nuevo ámbito de definición.

Observación: en un mismo ámbito, no podrán coexistir dos declaraciones con el mismo identificador, pero sí en ámbitos distintos, con lo que se tomará la declaración más próxima al ámbito actual. Además, en el caso de los métodos de una clase, si un atributo tiene el mismo identificador que uno de los atributos del método, si no se pone el identificador `this->` antes de la variable se referirá al atributo del método, mientras que si se pone se referirá a la variable global de la clase.

Ejemplos:

```
// Ambito global
int a = 10;  // Declaracion global
int b = 20;  // Declaracion global

// Funcion con ambito de parametros y cuerpo
int suma(int x, int y) {  // x y y son visibles solo en el cuerpo de la funcion
    int z = x + y;  // z es visible solo en el cuerpo de la funcion
    return z;
}

// Cuerpo de un bucle (while)
int i = 0;
while(i < 5) {  // El cuerpo del while tiene su propio ambito
    int x = i * 2;  // x es visible solo dentro del bucle
    i++;
}

// Cuerpo de un bucle (for)
for(int i = 0; i < 5; i++) {  // La variable i tiene su ambito dentro del for
    int x = i * 2;  // x es visible solo dentro del bucle
}

// Instruccion if
if(a > b) {  // El cuerpo del if tiene su propio ambito
    int x = a + b;  // x es visible solo dentro del if
}

// Instruccion else
if(a < b) {
  // El cuerpo del else tiene su propio ambito
    int y = b - a;  // y es visible solo dentro del else
} else {
    int z = a * b;  // z es visible solo dentro del else
}

// Instruccion switch
switch(a) {
  case 1:
    int caseVar = 100;  // caseVar es visible solo en el case 1
    break;
    case 2:
        int caseVar = 200;  // caseVar es visible solo en el case 2
        break;
    default:
        int caseVar = 300;  // caseVar es visible solo en el default
        break;
}
```

```
// Cuerpo de una clase
class MiClase {
    int atributoClase;  // Visible en todos los metodos de la clase

    void metodo() {
        int atributoMetodo = 10;  // atributoMetodo es visible solo dentro
                                  // del metodo
        this->atributoClase = atributoMetodo;  // referencia a atributo global
                                               // de la clase
    }
}
```

5 Gestión de Errores

Necesitaremos una gestión de errores en nuestro lenguaje, con el objetivo de indicar la presencia de errores y facilitar su corrección. Todos los mensajes de error tendrán la misma estructura:

```
ERROR "TIPO DE ERROR" (fila n, columna m): "Breve explicacion del error"
```

5.1 Errores Léxicos

Estos ocurren durante la conversión de caracteres a tokens. Algunos ejemplos son el reconocimiento de caracteres inválidos ($, €, etc.) o identificadores no válidos (456numero).

Ejemplo:

```
void main(){
    int n;
    $;
    bool 123true;
}
```

```
ERROR LEXICO (fila 3, columna 5): Caracter no valido: $
ERROR LEXICO (fila 4, columna 10): Identifador no valido: 123true
Deteniendo Compilacion
```

Como podemos observar, aunque reconozcamos un error que nos obligue a detener la compilación, mostraremos el resto de errores que reconozcamos.

5.2 Errores Sintácticos

A diferencia de los anteriores, estos tratan de violaciones de la grámatica del lenguaje. Pueden suceder por estructuras incorrectas o uso incorrecto de palabras reservadas, entre otros. En el caso de que hubiera varios errores sintácticos, el compilador seguirá con el fin de reconocer todos los errores posibles. En general, se detectarán errores a nivel de instrucción, pero para ello haremos un análisis más exhaustivo en las condiciones de los if y while y en las cabeceras de los for y switch.

Ejemplo:

```
void main(){
    int n
    if (n == 4){

    return;
}
```

```
ERROR SINTACTICO (fila 2, columna 10): Elemento inesperado "if"
ERROR SINTACTICO (fila 6, columna 1): Elemento Inesperado "}"
    (Error en el cuerpo del if)
Deteniendo Compilacion
```

5.3 Errores Semánticos

Por último, trataremos los errores semánticos. Estos son los relacionados con el significado del código y el cumplimiento de las reglas del lenguaje de programación. Algunos ejemplos pueden ser errores de tipado o de ámbito de variables.

Ejemplo:

```
bool or(bool a, bool b){
    return a || b;
}
void main(){
    bool a = 5;
    or(2, false);

    return;
}
```

```
ERROR SEMANTICO (fila 5):
    No se puede asignar un int a un bool
ERROR SEMANTICO (fila 6):
    No se ha encontrado una implementacion correcta para la funcion
Deteniendo Compilacion
```