Documentación del Lenguaje Ñ++

Resumen, especificación y funcionamiento del lenguaje Ñ++. Léxico, sintaxis y ejemplos Ñ++

Lucas Vukotic

Enrique Queipo de Llano

Alejandro Paz

Índice general

1. Introducción

Este documento resume la especificación e implementación de nuestro lenguaje Ñ++, así como algunos ejemplos típicos de programas escritos en él. Para desarrollarlo, nos hemos basado en una traducción al español del lenguaje C++.

2. Estructura de un programa

Como nos hemos basado en C++, nuestros programas por lo general tendrán en las primeras líneas declaraciones de `alias`, tipos `enumerados`, de tipos `registros`, de `funciones` y `clases`. Todo programa contará con una función principal llamada `principal` en la que comenzará la ejecución del programa. Los comentarios en Ñ++ comienzan con el operador `//` y terminan con un salto de línea. Para facilitar el uso de comentarios con más de una línea usamos la sintaxis `/**/` (el comentario se escribe entre los asteriscos). Mostramos a continución la sintaxis que nos permitirá declarar tanto tipos como funciones.

- **Alias**:

  `alias` nombre = tipo;

- **Enumerados**:

  `enum` NombreEnumerado {Nombre1, Nombre2,..., NombreN};

- **Registros**:

  ```
  estructura NombreReg {
     // lista de declaraciones
  };
  ```

- **Funciones**:

  ```
  tipoRetorno nombreFun(tipo1 [&] arg1, ..., tipoN [&] argN) {
     // lista de instrucciones [5]
     devuelvevalorRetorno;  //opcional y en cualquier parte de la función
  }
  ```

  El símbolo & es opcional y, si se incluye, se pasará la variable por referencia, en caso contrario se pasará por valor. La instrucción devuelve es opcional. Puede estar situada en cualquier parte de la función, aunque quede código que nunca se alcance. Para el tipo vacío, se puede poner simplemente devuelve para finalizar la ejecución de la función, aunque por motivos de seguridad, se recomienda añadirla. Las funciones pueden devolver los tipos `básicos`, pero **no** pueden devolver una `estructura` o una `lista`, estos casos se deben hacer con paso por referencia.

- **Función main**:

```
principal() {
   // lista de instrucciones [5]
   devuelve; // No obligatorio
}
```

3. Tipos

En Ñ++, el usuario debe declarar los tipos explícitamente. A lo largo de esta sección veremos cuál es la sintaxis para hacerlo así como los distintos tipos con los que cuenta el lenguaje. Es importante tener en cuenta que en Ñ++, después de toda instrucción se ha de incluir el símbolo terminal ;. Además, el nombre que se le pone a una variable al declararla debe estar formado por caracteres alfanuméricos y guiones bajos y además es obligatorio que comiencen con una letra ya sea minúscula o mayúscula.

3.1. Tipos básicos predefinidos

- `entero`: para representar enteros.
- `flotante`: para representar valores en punto flotante.
- `buleano`: para representar **verdad** o **mentira**.
- `vacio`: para representar el tipo vacío.

3.2. Tipos definidos por el usuario

Los usuarios de Ñ++ pueden crear sus propios tipos enumerados y sus propios registros. A continuación vemos cuál es la manera de hacerlo:

- `enum` representa un conjunto de valores constantes.
  `enum` NombreEnumerado = {Nombre1, Nombre2,..., NombreN};
  Por ejemplo:
  `enum` DiasSemana = {Lunes, Martes, Miercoles, Jueves, Viernes, Sábado, Domingo};
- `estructura` define una estructura de datos formada por campos, que son una o más variables de uno o distintos tipos.

```
estructura NombreReg {
   // lista de declaraciones
};
```

Por ejemplo:

```
estructura Tiempo {
    entero hora;
    entero minuto;
   flotante segundo;
};
```

- La sintaxis para las clases hace uso de la palabra reservada `clase` y es la siguiente:

```
[modificadorDeAcceso] clase NombreClase {
        // atributos de la clase (0 o más atributos)
        [modificadorDeAcceso] tipo nombreAtributo;
        // Constructores de la clase(0 o más constructores)
        [modificadorDeAcceso] NombreClase(0 o más argumentos)
        // métodos de la clase (0 o más métodos)
        [modificadorDeAcceso] tipoDevuelto nombreMetodo([lista parámetros])
    }
```

Por ejemplo:

```
publico clase Tiempo {
        privado entero segundos;
        privado entero minutos;
        privado entero horas;
        publico Tiempo(entero s, entero m, entero h) {
                segundos = s;
                minutos = m;
                horas = h;
           }
        publico entero aSegundos() {
                devuelve 3600*horas + 60*minutos + segundos;
        }
    }
```

No se permite herencia entre clases (por tanto no hay clases abstractas) y no existen interfaces.

Hay dos tipos de modificadores de acceso:

1. **publico:** son accesibles desde cualquier punto del programa
2. **privado:** solo puede acceder a ellos la clase que los declara.

3.3. Tipo array

Representa una lista de elementos del mismo tipo. Es importante recalcar que en Ñ++ los arrays son homogéneos, es decir, en un array no pueden existir dos elementos de tipos distintos.

Para declarar arrays, usamos la palabra reservada `lista`. Para declarar un array se sigue la sintaxis: `lista`< `tipo`>[N], donde N es el tamaño del array y debe ser un número entero. Se permiten arrays de varias dimensiones, esto es, arrays cuyos elementos a su vez son arrays. La sintaxis para este caso es la evidente, por ejemplo, para un array de 2 dimensiones (o aray de arrays), es decir, una matriz, formado por enteros y de tamaño NxM: `lista` <`lista` <`entero`>[M]>[N];

También es posible inicializar un array mediante los corchetes { y }. Siguiendo el ejemplo anterior:
`lista`<`lista`<`entero`>[2]>[2] = {{1,2},{3,4},{5,6}}

En Ñ++, los arrays son *estáticos*, lo que quiere decir que su tamaño va ligado a su tipo. Es por esto que el tamaño debe ir incluido siempre, incluso si el array es un parámetro de función.

3.4. Tipo puntero

Representa una dirección de memoria que almacena un valor. Se identifica por `tipo` *. Para acceder al valor almacenado en una dirección de memoria utilizamos el operador @ y para acceder a la dirección de memoria se usa de nuevo *. Se incluye la instrucción `nuevo` para reserva de memoria dinámica. Veamos un ejemplo en el que declaramos un puntero llamado n a un entero e inicializamos el valor al que apunta dicho puntero a 0 mediante el operador @ :

`entero`* n = `nuevo` `entero`;
@n = 0 ;

4. Expresiones, Operadores, Bloques y Clases.

4.1. Expresiones

Las expresiones en nuestro lenguaje pueden ser:

- Un **acceso** a:
  - Variables: con el nombre de la variable formado por caracteres alfanuméricos y guiones bajos. Tienen que comenzar por una letra y estar declaradas.
  - Punteros: Si queremos acceder al valor usamos @nombre y si queremos acceder a la dirección de memoria usamos *nombre.
  - Arrays: nombre[indice1][indice2]..[indiceN] para acceder al elmento del array que se encuentra en la posición marcada por los índices

- Estructuras: nombre.campo Por ejemplo: Tiempo contador; contador.hora = 12; contador.minuto = 15; contador.segundo = 59.9;
- Metodos de clases: Como las llamadas a función pero con un objeto de la clase. Por ejemplo si tuvéramos una clase Tiempo: Tiempo t; t.aSegundos();

Observación 4.1. Se permite el anidamiento de distintos tipos de accesos, esto es, en un registro puede haber un campo de tipo array, y por tanto se puede acceder a una posición concreta de dicho array mediante composición de expresiones de acceso.

Observación 4.2. El acceso más prioritario es el acceso a punteros, y es el único que permite asociación por paréntesis. Ejemplo:
@figura.circulo = 3;
@(figura.circulo) = 3;
El primer ejemplo es equivalente a (@figura).circulo = 3; pero no permitimos esta notación.

- Una **constante**:
  - de tipo `entero`: dígitos, por ejemplo: 20.
  - de tipo `flotante`: dígitos separados por punto (.), por ejemplo: 20.03.
  - de tipo `buleano`: verdad y mentira.
- Una **llamada a una función**: se escribe el nombre de la función seguido de los parámetros entre paréntesis, por ejemplo: `devuelveDia`(20, varDeEjemplo).
- Un `nuevo tipo`*: Para guardar memoria (dinámica) al iniciar los punteros, por ejemplo: `entero`* n = `nuevo entero`;

4.2. Operadores

En la siguiente tabla se muestra a modo resumen los operadores con los que cuenta Ñ++ así como sus respectivas prioridades y la forma en la que asocia cada uno de ellos. Las prioridades varían de 0 a 6 siendo 6 la prioriad más alta y 0 la más baja.

4.3. Bloques

Se permite el uso de bloques para localizar sentencias en el programa. Los bloques van encerrados entre las llaves { y }. Al entrar en un nuevo bloque se cambia automáticamente el ámbito del programa al bloque en el que se ha entrado. Se permiten bloques anidados, en tal caso, el ámbito del programa será el del bloque más "profundo". Dentro de los bloques (como veremos bloques incluirá a `while`, `if`, etc) las variables declaradas dentro del bloque ocultan a las que están "por encima".

Operador	Tipo	Prioridad	Asociatividad
oo	Binario infijo	0	Por la izquierda
yy	Binario infijo	1	Por la izquierda
==, !=	Binario infijo	2	Por la izquierda
<,>,<=,>=	Binario infijo	3	Por la izquierda
+,-	Binario infijo	4	Por la izquierda
*,/, %	Binario infijo	5	Por la izquierda
- , no	Unario prefijo	6	Sí

Cuadro 1: Operadores de Ñ++

5. Instrucciones

En esta sección se muestra la sintaxis de cada una de las instrucciones con las que contará el lenguaje Ñ++.

5.1. Declaración

Podemos declarar variables de dos formas:

- **No inicializadas**: `tipo` nombre;

  Ejemplo: `entero` numero;

- **Inicializadas**: `tipo` nombre = expresión;

  Ejemplo: `buleano` encontrado = verdad;

5.2. Asignación

Podemos asignar a las variables el valor de una expresión evaluable siempre y cuando los tipos de la expresión y de la variable concuerden. La sintaxis para esto es: var = expresion;

5.3. Condicional

Puede ser de una rama:

```
si(expresion) {
   // lista de instrucciones
}
```

o de dos ramas:

```
si(expresion) {
   // lista de instrucciones
} sino {
```

```
    // lista de instrucciones
}
```

5.4. Bucle While

```
mientras(expresion) {
    // lista de instrucciones
}
```

5.5. Bucle For

```
paraCada(declaracion; expresión; asignacion) {
    // lista de instrucciones
}
```

Puede utilizarse para inicializar una variable nueva de uso exclusivo para este bucle (generalmente entero i = 0). Si no se deseara, se puede dejar vacío: `paraCada`(;expresión;asignación). En general, tanto la condición de parada como la operación estarán estrechamente relacionadas con la variable del bucle. Se finaliza cuando la expresión buleana del bucle se evalue a verdad.

5.6. Switch

```
selecciona (variable) {
    caso valor1:
        instrucciones
    ⋮
    caso valorN:
        instrucciones
    porDefecto:
        instrucciones
}
```

Ñ++ cuenta con instrucción de parada: `¡para!`. Se podrá incluir de forma opcional después de las instrucciones que componen cada caso. El caso por defecto es también opcional. Nuestro `selecciona` sólamente admite variables, no expresiones.

5.7. Llamadas a función

Como vimos antes, podemos realizar llamadas a funciones como parte de una expresión.Además, Ñ++ también permite que la llamada a una función sea una instrucción en sí misma, sin importar el valor de retorno. En la instrucción `devuelve` de las funciones se permite cualquier expresión, incluida una llamada a otra función, siempre y cuando el tipo de la función y el del valor de retorno coincida.

`nombreFun`(param1, ... paramN);

5.7.1. Recursividad

En Ñ++ se permite la recursividad. La sintaxis para esta es la esperada, basta que una función se llame a si misma dentro de su propio cuerpo. No se verifican en tiempo de compilación ni de enlazado la condición de terminación.

5.8. Entrada y Salida

En el lenguaje Ñ++ solamente se incluirá entrada y salida de `entero` mediante la consola y el teclado del usuario.

Para mostrar por consola una expresión: `imprime`(expresión);

De forma análoga para leer la entrada por consola: `lee`(); Esta se utilizará exclusivamente como una expresión que asigne a una variable el valor leído por consola. `entero` n = `lee`();

Observación 5.1. Como limitación provocada por las operaciones input/output con las que contamos (importadas del profesor), es necesario remarcar varios apuntes:

- Para poder extender `imprime` también a elementos de tipo `buleano` y `enum`, si se imprimen estos por pantalla se mostrarán los valores 0 o 1 (si se trata de un `buleano`) o bien la posición que ocupa la constante a imrpimir dentro de su tipo `enum`.
- Para el correcto funcionamiento de la instrucción `lee`() hay que modificar previamente el archivo JavaScript **main.js**. En la función `start`() del mismo se encontrará en un comentario la instrucción `await readInput`(n);. Esta instrucción debe ser descomentada y n debe ser el número de instrucciones `lee`(). Al ejecutar, hay que introducir los n enteros por la consola para comenzar la ejecución del programa

5.9. Alias

Se permite el uso de una instrucción alias para renombrar ciertos tipos. La sintaxis es como sigue:

`alias` nombre = tipo;

Por ejemplo: `alias` listaEnteros = lista<entero>[10];

Se permiten anidamientos de `alias`, esto es, definir `alias` en función de otros `alias` que hayan sido previamente definidos.

5.10. Return

Para salir de una función devolviendo un valor o ninguno en las funciones `vacio`:

`devuelve` expresión;

`devuelve`;

La instrucción `devuelve` es **opcional** pero no puede haber más de una por función.

6. Gestión de errores léxicos y sintácticos

- **Errores léxicos**: Se detectan tokens no declarados y se cuentan como error, mostrando la fila y la columna donde se ha producido. Vamos a continuar analizando por si hubiese más.
- **Errores sintácticos**: Indicamos la fila y la columna del mismo y nos recuperamos de él para proseguir la compilación y detectar más. La recuperación de los errores se hace tras el punto y coma. Los principales errores sintácticos que detectamos son:
  - Cuando el programa no tiene función `principal`.
  - En los `enum` y `estructuras` controlamos cualquier tipo de error y nos recuperamos en en el siguiente ; (incluido que no se añada el ; al final de la declaración del mismo).
  - En las funciones controlamos los errores en la cabecera y el bloque de instrucciones.
  - En las instrucciones controlamos los ; finales y cualquier otro error que pueda haber en ellas. Por ejemplo, cuando faltan paréntesis, cuando hay tipos indefinidos o hay errores de construcción en las condiciones del `if`, `for`, `while` y `switch`.

Observación 6.1. La falta de llaves al crear bloques no se controla en funciones e instrucciones.

7. Vinculación y tipos

Una vez hecho el análisis léxico y sintáctico, pasamos a hacer el análisis semántico. Lo vamos a realizar en dos fases:

- **Vinculación**: Consiste en asociar un identificador al objeto que designa. Esto se lleva a cabo mediante un recorrido del AST con una pila de tablas que representa los ámbitos. Lo primero que se hace es la vinculación de todo lo que se define previo a la función `principal`. Esto es, se lleva a cabo la vinculación de `alias`, `estructuras`, `enums`, `funciones` y `clases`. Para las funciones, se recorren sus parámetros y luego sus instrucciones. *Sólo se alcanza esta fase si no ha habido fallos sintácticos.*

  Garantizamos lo siguiente:

  - Nos aseguramos de que las variables, y en general los accesos, que utiliza el programa se han definido antes de su uso y se vinculan a sus declaraciones.
  - Nos aseguramos de que los valores de retorno estén vinculados a sus funciones.
  - Nos aseguramos de que las llamadas a funciones estén vinculadas a la función a la que hacen referencia.

  Sobre variables tenemos en cuenta que:

- Las variables declaradas son visibles en su ámbito.
- Tomamos una variable declarada en un ámbito en el cual hay un bloque con otra declarada con el mismo nombre. En este caso, en el bloque será visible la más interna, es decir, la que se ha sobreescrito. Si esta última no existiese, en el bloque podríamos ver la externa.
- En un mismo bloque no se permite la declaración de dos variables con el mismo nombre.

- **Comprobación de tipos**: Nos aseguramos de que los tipos se usan de forma adecuada. De nuevo, comenzamos comprobando que la definición de tipos de todo lo que precede a `principal` sea correcta. *Sólo se alcanza esta fase si no ha habido fallos en vinculación.*

  La comprobación de tipos la hacemos de la siguiente manera:

  - Tenemos una serie de tipos `básicos` que se comparan de forma directa.
  - En los arrays comprobamos que las dimensiones y tipo de sus elementos coinciden.
  - Cada `estructuras` y cada `enum` forma su propio tipo.
  - Los punteros comprueban que el `tipo` del elemento que guardan es igual al `tipo` del puntero.
  - Contamos con tipos compuestos, es decir, tipos que contienen a otros tipos. Por ejemplo, si tenemos `lista`<`entero`>[3] el nodo vinculado a esa declaración tendrá tipo `lista`. A su vez, `lista` tendrá otro atributo tipo que, en este caso, será `entero`. Esta construcción se puede repetir tantas veces como se desee. Para la comprobación de tipos en estos casos, se desciende hasta el tipo básico y se comprueba la igualdad.

8. Generación de código

Sólo se alcanza esta fase si no ha habido fallos en el tipado. Antes de generar código realizamos dos cosas:

- Calcular la etiqueta de cada declaración, es decir, asignar a cada una un entero que representa el inicio de su posición de memoria relativa al bloque en el que se encuentra. Esto lo realizamos teniendo en cuenta el tamaño que ocupan.
- Calcular el tamaño máximo de memoria que cada elemento del AST.

En los dos pasos anteriores, para ahorrar memoria, tenemos en cuenta que el espacio que ocupan las variables declaradas dentro de un bloque se puede reutilizar cuando se sale de él. Además, en el tratamiento de bloques anidados, se considera la reserva de memoria para el bloque de tamaño máximo. Una vez finalizados esos pasos, generamos el código WebAssembly. Aceptamos programas constituidos por:

- Una única función main.
- Genera código para tipos enteros, buleanos, arrays, punteros, enums, structs.
- Generación de código para funciones.
- Puede contener todas las instrucciones y que hemos especificado con anterioridad.

Observación 8.1. Hemos conseguido implementar todo lo anterior. Por falta de tiempo, lo único que no hemos conseguido ha sido la generación del código de las clases.

9. Ejemplos de programas en Ñ++

Se incluyen una serie de ficheros de texto que comprueban distintas de las funcionalidades de Ñ++. Algunos se incluyen a continuación, pero presentamos un resumen de lo que se puede encontrar en cada uno.

- **alias.txt** Funcionalidad de los alias en Ñ++. Alias definidos en términos de otros alias. Funciones definidas con tipos que son alias.
- **arrays3d.txt** Inicialización y recorrido de arrays multidimensionales.
- **bloques.txt** Uso de bloques anidados y ocultación de variables.
- **combinacionTipos.txt** Combinación de tipos definidos por el usuario. Se tiene un array de una `estructura` cuyos campos son `enum`.
- **declaraciones.txt** Uso básico de declaraciones y tipos básicos.
- **erroresSintacticos.txt** Recuperación y print de distintos errores sintácticos en definiciones e instrucciones.
- **erroresTipado.txt** Detección de errores de tipado. Recuperación y muestra de los errores en asignaciones y expresiones.
- **erroresVinculacion.txt** Detección de errores de vinculacion. Recuperación y muestra de los errores.
- **factorialIterativo.txt** Ejemplo clásico de cálculo del factorial iterativo.
- **insercionBusquedaList.txt** Paso de array a funciones por referencia. Implementación de funciones buscar e insertar sobre listas.
- **llamadasFunciones.txt** Función recursiva y llamada a una función que recibe lo que devuelve otra.

- **memDinamica.txt** Se incluye a continuación. Uso de reserva de memoria dinámica para un puntero a una `estructura`. Paso de punteros a funciones.
- **parametrosValorRef.txt** Paso de parámetros a funciones por valor y por referencia.
- **pointer.txt** Intercambio de valores mediante punteros. Declaracion de punteros a partir de direcciones de variables.
- **punterosStructFuncs.txt** Uso de punteros y `structs` sin memoria dinámica
- **stdio.txt** Prueba de entrada/salida. Ver sección 5.8.
- **switch.txt** Prueba del switch con y sin breaks y con y sin casos por defecto.

9.1. Recursión y llamadas a funciones

```
entero cuadrado(entero n){
    devuelve n*n;
}

/* Funcion que calcula el factorial de un numero
utilizando una llamada recursiva*/
entero factorial(entero n) {
  entero sol;
  si (n == 0) {
    sol = 1;
  } sino {
    sol =  n * factorial(n - 1);
  }
  devuelve sol;
}

principal() {
  entero num = 5;
  imprime(cuadrado(factorial(num)));//14400 = 120^2

}
```

9.2. Días de la semana

Programa para comprobar el funcionamiento de la instrucción switch y de los tipos enumerados.

```
enum DiaSemana {
    Lunes,
    Martes,
    Miercoles,
    Jueves,
    Viernes,
    Sabado,
    Domingo,
    Error
};

DiaSemana AveriguaDia(entero n) {
    DiaSemana dia;
    selecciona(n) {
        caso 1: dia = Lunes; ¡para!;
        caso 2: dia = Martes; ¡para!;
        caso 3: dia = Miercoles; ¡para!;
        caso 4: dia = Jueves; ¡para!;
        caso 5: dia = Viernes; ¡para!;
        caso 6: dia = Sabado; ¡para!;
        caso 7: dia = Domingo; ¡para!;
        porDefecto: dia = Error;
        // en caso de un numero invalido, se devuelve el error por defecto
    }
    devuelve dia;
}

principal() {
    entero n = lee();
    DiaSemana dia = AveriguaDia(n);
    imprime(dia);
}
```

9.3. Manejo de punteros y structs

```
estructura Persona {
  entero dni;
  entero edad;
};

vacio cambiarEdad(Persona* p, entero nuevaEdad) {
  @p.edad = nuevaEdad;
  devuelve;
}

principal() {

  Persona* p1 = nuevo Persona;
  @p1.dni = 01234567;
  @p1.edad = 30;

  //Antes de cambiar la edad
  imprime(@p1.dni); // 0123...
  imprime(@p1.edad); // 30

  cambiarEdad(p1, 35);

  //Despues de cambiar la edad
  imprime(@p1.edad); // 35

}
```

9.4. Indicaciones sobre compilación y ejecución

Para compilar y ejecutar un ejemplo en concreto contamos con dos scripts de Shell. En primer lugar se debe ejecutar el script **compilarCodigo.sh** seguido del archivo de ejemplo (en dirección relativa, esto es, **ejemplos/archivo.txt**) o con la opción -t para compilar todos. Una vez compilado, el script **ejecutarCodigo.sh** ejecutará el código webassembly generado por la última ejecución satisfactoria de **compilarCodigo.sh**

También hemos incluido un último script **ejecutarEjemplos.sh** que compila y ejecuta cada uno de los ejemplos de la carpeta **ejemplos**, mostrando todos los pasos de la compilación y el output de la ejecución para cada uno de ellos.