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

En esta entrega vamos a especificar 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]`
  `devuelve`valorRetorno; `//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.

- **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 declaración de tipos 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])
        [devuelve valor;]
    }
```

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];

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 (un puntero) utilizamos el opetador @. 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;
@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 contará 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.

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++

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".

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;

También es posible declarar constantes. Para ellos se utiliza la palabra reservada `const`. Puede inicializarse en la declaración haciendo `const tipo` Nombre = valor o puede quedarse sin inicializar (basta omitir la parte = valor)

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(variable; expresión buleana para comprobar parada; operación) {

// lista de instrucciones

}

Variable puede ser una variable ya existente en el programa o puede utilizarse para inicializar una variable nueva de uso exclusivo para este bucle (generalmente entero i = 0). 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

}

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.

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 esta primera versión del lenguaje Ñ++ solamente se incluirá entrada y salida mediante la consola y el teclado del usuario.

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

Expresión puede ser una expresión tipada o puede ser una cadena de carácteres.
De forma análoga para leer la entrada por consola: `lee`(); Esta instrucción mostrará por consola la expresión que recibe como argumento y además devolverá el valor que el usuario introduzca mediante el teclado.

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>

6. Ejemplos de programas en Ñ++

6.1. Factorial de un número de forma recursiva

Ejemplo para comprobar como funciona la recursión.

```
entero factorialRec(entero n) {
    si (n == 0) {
        devuelve 1;
    }
    sino {
        devuelve n * factorialRec(n - 1);
    }

}

principal() {
    entero n = lee();
    imprime(factorialRec(n));
    devuelve;
}
```

6.2. Factorial de un número de forma iterativa

Versión iterativa del código anterior.

```
entero factorialIterativo(entero n) {
    entero resultado = 1;
    paraCada(entero i = 1; i <= n; i= i + 1) {
        resultado = resultado * i;
    }
    devuelve resultado;
}


principal() {
    entero n = lee();
    imprime(factorialIterativo(n));
    devuelve;
}
```

6.3. Días de la semana

Programa para comprobar el funcionamiento de la instrucción siwtch 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);
}
```

6.4. Registros, punteros, arrays y condicional de dos ramas.

En este programa, se define una estructura Persona que contiene un dni y una edad. Se utiliza un vector array de tipo Persona para almacenar tres objetos de tipo Persona.

En la función principal, se utiliza un bucle for para iterar sobre todos los objetos del vector personas. Se utiliza una condicional if con la instrucción else if para comparar las edades de las personas y asignar el puntero a la persona más vieja.

Finalmente, se muestra el nombre de la persona más vieja en la pantalla.

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

principal() {
    personas = lista <Persona>[3];
    personas[0].DNI = 1;
    personas[0].edad = 30;
    personas[1].DNI = 2;
    personas[1].edad = 25;
    personas[2].DNI = 3;
    personas[2].edad = 35;

    Persona* ptrPersona;

    paraCada(entero i = 0; i < 3; i= i+1) {
        si(i == 0) {
            ptrPersona = @personas[i];
        } sino {
            si(personas[i].edad > (*ptrPersona).edad) {
                ptrPersona = @personas[i];
            }
        }
    }
    imprime((*ptrPersona).DNI);
}
```