4.2 Introducción al lenguaje C++

4.2.1 Historia

Dennis Ritchie desarrolló y llevó a cabo la primera implementación del lenguaje C en un DEC PDP-11 que utilizaba un sistema operativo UNIX. El lenguaje es el resultado de un proceso de desarrollo iniciado con un lenguaje anterior denominado BCPL.

Durante muchos años el estándar de C fue la versión que se proporcionaba con el sistema operativo UNIX System V. Esta versión se describía en el libro "The C Programming Language", escrito en 1978 por Brian Kernighan y Dennis Ritchie. Aunque fue ampliamente usado, la falta de un estándar provocaba numerosas discrepancias.

En verano de 1983, el ANSI estableció un comité para la estandarización del lenguaje C. El estándar fue adoptado en Diciembre de 1989.

El lenguaje C soporta el concepto de tipo de datos. Un tipo de datos define un conjunto de valores que se pueden almacenar en una variable junto con un conjunto de operaciones que se pueden realizar sobre esa variable.

C permite la manipulación directa de bits, bytes y punteros. C tiene el problema de que es difícil manipular programas con muchas líneas de código.

C++ es una versión extendida de C. Bjarne Stroustrup, de los laboratorios Bell, intentó resolver este problema desarrollando extensiones de C. Finalmente, en 1983, surge C++. C++ es un lenguaje orientado a objetos.

C++ ha sido revisado varias veces hasta que se creó la primera versión del estándar propuesto, en 1994.

C++ permite al programador la libertad y el control de C junto con la potencia de la programación orientada a objetos (POO).

4.2.2 La programación orientada a objetos en C++

La programación orientada a objetos es una forma de enfocar el trabajo de la programación. La programación orientada a objetos permite descomponer un problema en subgrupos de partes relacionadas entre sí. Y estos subgrupos se pueden traducir a unidades autocontenidas denominadas objetos.

Todos los lenguajes POO tienen tres cosas en común: encapsulación, herencia y polimorfismo.

La encapsulación es el mecanismo que agrupa el código y los datos y los protege frente a interferencias o fallos externos. Un objeto es una entidad lógica que encapsula los datos y el código que manipula dichos datos. Dentro de un objeto, el código y los datos pueden ser accesibles o no desde el exterior dependiendo de si son públicos o privados, respectivamente.

Un objeto no es más que un tipo de dato definido por el usuario. Cuando se define un objeto, ímplicitamente se define un nuevo tipo de dato.

El polimorfismo es el atributo que permite usar una interfaz con una clase general de acciones. Es tarea del compilador seleccionar la acción específica que se aplica en cada situación.

La herencia es el proceso que permite a un objeto adquirir las propiedades de otro.

Las clases están relacionadas con los objetos. Antes de crear un objeto en C++ se debe definir su formato general usando la clase (palabra reservada class). Una clase se parece a la estructura (struct) de C, pero incluye funciones además de datos. Por lo tanto, una clase define un nuevo tipo de dato que enlaza código y datos.

Las clases pueden tener partes privadas y públicas. Las partes públicas son accesibles desde cualquier parte del programa. Las partes privadas sólo son accesibles desde funciones que sean miembros de la clase.

Las clases son la forma en que C++ consigue la encapsulación.

Una forma con la que se consigue el polimorfismo es mediante la sobrecarga de funciones. En C++, varias funciones pueden tener el mismo nombre siempre y cuando sus parámetros sean diferentes.

Otra forma de conseguir polimorfismo es mediante la sobrecarga de operadores. Se pueden sobrecargar operadores de C++ definiendo lo que significan para una clase específica.

En C++ la herencia se consigue permitiendo que una clase incorpore a otra clase dentro de su declaración. Esto permite definir una jerarquía de clases, desde la más general a la más específica.

Se debe partir de una clase base, que incluye todas las características comunes. Las clases derivadas incluyen todas las características de la clase base genérica además de otras características específicas.

4.2.3 Características de C++

Esta introducción al lenguaje C++ no pretende ser un tutorial ni un manual de referencia de C++. Lo que se pretende es dar una visión general del lenguaje elegido para la implementación del modelo. Una descripción, aunque no fuese muy detallada, del lenguaje C++, necesitaría tal cantidad de espacio que haría imposible que fuese incluida en esta introducción.

Tras un breve introducción histórica a C++ y a la programación orientada a objetos en C++, en esta sección se reseñarán algunas de las características más importantes de C++ que aportan mejoras sobre el ya conocido C.

Contructores y Destructores

Es muy común que al crear un objeto haya que realizar algún tipo de inicialización. Por ello C++ incluye los constructores.

Un constructor es una función que es miembro de la clase (es un método) y que tiene el mismo nombre de la clase y no puede devolver valores.

La función constructor se ejecuta automáticamente cuando se crea el objeto.

Análogamente existe el destructor. El destructor se ejecuta cuando se destruye el objeto. Es muy útil, por ejemplo, si el objeto es una pila, o una cola, o cualquier tipo de lista, pues puede usarse para liberar la memoria de cada uno de los elementos que contiene. El destructor tiene el mismo nombre que el constructor pero precedido del símbolo ~.

El constructor puede recibir parámetros.

Ejemplo:

```
class fraccion
{
int numerador;
int denominador;
public:
fraccion (int , int );
...
};

fraccion::fraccion (int a, int b)
{
numerador=a;
denominador=b;
}
```

Un objeto de la clase fracción podría crearse de dos formas:

- *fraccion A;*

en este caso, los valores de las propiedades *numerador* y *denominador* no están definidos

- *fracccion A(1,2);*

en este caso, se crea el objeto de manera que la propiedad *numerador* vale 1 y *denominador* vale 2.

Funciones afines

Una función afín es aquella que puede acceder a los miembros privados de una clase sin ser miembro de dicha clase.

Para definir una función como afín a cierta clase debe incluirse su declaración, precedidad de la palabra friend, junto a las declaraciones de las funciones miembro de la clase.

Ejemplo:

```
class fraccion
{
int numerador;
int denominador;
public:
fraccion (int , int );
friend int devuelve_valor();
...
};

//Observese que no es int fraccion::devuelve_valor()
int devuelve_valor ()
{
if (denominador!=0)
return (numerador/denominador);
else
return -1;
}
```

Aunque en este caso quizás lo mejor hubiera sido definir *devuelve_valor* como una función pública de la clase, pero puede ser útil, por ejemplo, si se tienen que usar varios atributos (privados) de clases diferentes.

Argumentos de funciones por omisión

C++ permite asignar a un parámetro de una función un valor por omisión cuando ningún argumento se corresponde con los parámetros especificados en la llamada.

Ejemplo:

```
int devuelve (int a=1)
{
return a;
}
```

Una llamada *devuelve(3)* devolverá el valor 3, pero una llamada *devuelve()* devolverá 1.

Funciones inline

Una función en línea es una función en la que el código se expande en el punto donde se realiza la llamada, es decir, no se hace una llamada a una función. Es parecido a las macros de C pero con mayor flexibilidad.

Es muy útil para funciones pequeñas.

Ejemplo:

```
class fraccion
{
int numerador;
int denominador;
public:
void asigna  (int , int );
...
};

inline void fraccion::asigna (int a, int b)
{
numerador=a;
denominador=b;
}
```

Es equivalente a:

```
class fraccion
{
int numerador;
int denominador;
public:
void asigna  (int , int ){numerador=a;denominador=b;}
...
};
```

Sobrecarga de funciones

Como ya se ha mencionado, es uno de los mecanismos de C++ para ofrecer polimorfismo.

La sobrecarga de funciones se explica en la sección Creación de componentes en tiempo de ejecución, ya que se usa en *Crear_componente.*

Sobrecarga de operadores

El otro mecanismo usado para ofrecer polimorfismo. Se explica detalladamente en la sección del mismo nombre.

El puntero this

Cada vez que se llama a una función miembro, se le pasa a dicha función, de forma transparente al programador, un puntero el objeto que la llama. Este puntero está accesible mediante *this*.

Ejemplo:

```
class fraccion
{
```

```
int numerador;
int denominador;
public:
void asigna  (int , int );
...
};

void fraccion::asigna (int a, int b)
{
numerador=a;
denominador=b;
}
```

El código de la función asigna es equivalente a:

```
void fraccion::asigna (int a, int b)
{
this->numerador=a;
this->denominador=b;
}
```

Paso de parámetros por referencia

En C los parámetros se pasaban por valor. En C++, por defecto, ocurre lo mismo. Esto implica que para intercambiar el valor de dos enteros se debe hacer:

```
void intercambio (int *a, int *b)
{
int aux;
aux=*a;
*a=*b;
*b=aux;
}
```

La llamada sería algo así:

```
int x,y;
x=4;
y=5;
intercambio (&x, &y);
/* hay que pasarle la dirección del elemento, no el elemento*/
```

Usando los parámetros por referencia:

```
void intercambio (int &a, int &b)
{
// no se recibe el puntero, sino el parámetro por valor
int aux;
aux=a;
```

```
a=b;
b=aux;
}
```

Y la llamada sería algo así:

```
int x,y;
x=4;
y=5;
intercambio (x, y);
// se pasa el elemento por referencia en lugar de por valor, no se pasa el puntero
```

Funciones virtuales

Proporcionan polimorfismo en tiempo de ejecución. Un función virtual es una función que se declara como virtual en una clase base y se redefine en las clases derivadas. Son especiales porque cuando se llama a una de estas funciones a través de un puntero, se determina en tiempo de ejecución la función a la que se debe llamar teniendo en cuenta el tipo de objeto al que apunta el puntero.

Ejemplo:

```
class base
{
public:
virtual void quien_soy() {printf (“Soy la clase base\n”);}
};

class derivada: public base
{
public:
void quien_soy() {printf (“Soy la clase derivada\n”);}
}

class hija: public base
{
public:
void quien_soy {printf (“Soy la clase hija\n”);}
}
```

El programa:

```
int main()
{
base clase_base;
derivada clase_derivada;
hija clase_hija;
base * p;
```

```
p=&clase_base;
p->quien_soy();

p=&clase_derivada;
p->quien_soy();

p=&clase_hija;
p->quien_soy();

return 0;
}
```

Genera la siguiente salida:
Soy la clase base
Soy la clase derivada
Soy la clase hija

Las funciones virtuales puras obligan a que cada una de las clases derivadas redefina la función. En caso de no ser así, se genera un error.

Se definiría así:

```
class base
{
public:
virtual void quien_soy()=0;
};
```

Biblioteca de entrada/salida

C++ proporciona flujos de entrada/salida y funciones y operadores para operar con ellos, pero es un tema demasiado extenso para ser tratado aquí.

En cualquier caso, se pueden seguir usando las funciones de entrada/salida de C.

Asignación dinámica de memoria

Aunque se puede seguir usando *malloc*, *calloc*, *free* y las demás funciones relacionadas, C++ ofrece dos operadores para asignar y liberar memoria dinámica, *new* y *delete*, respectivamente.

El operador new reserva memoria y devuelve el puntero a la zona de memoria reservada. Para liberar dicha memoria se debe usar el operador delete.

Ejemplo:

```
fraccion *p; // suponiendo la clase fraccion definida
p=new fraccion;
// operaciones...
delete p;
```