1

Conceptos Básicos de Programación

Un programa es la especificación de una tarea de computación. Un lenguaje de programación es una notación para escribir programas.

Contenido

Referencias Bibliográficas

SETHI, R. (1992). Lenguajes de Programación: Conceptos y Constructores. Addison-Wesley Iberoamericana. Wilmington, Delaware, USA.

Wikipedia (2006). La Enciclopedia Libre. Documento electrónico disponible en http://es.wikipedia.org

Programación

Se llama programación a la creación de un programa de computadora, un conjunto concreto de instrucciones que una computadora puede ejecutar. El programa se escribe en un lenguaje de programación, aunque también se pueda escribir directamente en lenguaje de máquina. Un programa se puede dividir en diversas partes, que pueden estar escritas en lenguajes distintos.

Software es el sustantivo que denomina a los programas y datos de computadora.

Programas y Algoritmos

Un algoritmo es una secuencia no ambigua, finita y ordenada de instrucciones que han de seguirse para resolver un problema. Un programa normalmente implementa (traduce a un lenguaje de programación concreto) un algoritmo. Puede haber programas que no se ajusten a un algoritmo (pueden no terminar nunca), en cuyo caso se denomina procedimiento a tal programa.

Los programas suelen subdividirse en partes menores (módulos), de modo que la complejidad algorítmica de cada una de las partes sea menor que la del programa completo, lo cual ayuda al desarrollo del programa.

Según Niklaus Wirth un programa está formado por algoritmos y estructura de datos.

Se han propuesto diversas técnicas de programación, cuyo objetivo es mejorar tanto el proceso de creación de software como su mantenimiento. Entre ellas se pueden mencionar las programaciones estructurada, modular y orientada a objetos.

Compilación

El programa escrito en un lenguaje de programación (comprensible por el ser humano, aunque se suelen corresponder con lenguajes formales descritos por gramáticas independientes del contexto) no es inmediatamente ejecutado en una computadora. La opción más común es compilar el programa, aunque también puede ser ejecutado mediante un intérprete informático

El código fuente del programa se debe someter a un proceso de transformación para convertirse en lenguaje máquina, interpretable por el procesador. A este proceso se le llama compilación.

Normalmente la creación de un programa ejecutable (un típico .exe para Microsoft Windows) conlleva dos pasos. El primer paso se llama compilación (propiamente dicho) y traduce el código fuente escrito en un lenguaje de programación almacenado en un archivo a código en bajo nivel, (normalmente en código objeto no directamente al lenguaje máquina). El segundo paso se llama enlazado (del inglés link o linker) se junta el código de bajo nivel generado de todos los ficheros que se han mandado compilar y se añade el código de las funciones que hay el las bibliotecas del compilador para que el ejecutable pueda comunicarse con el sistemas operativo y traduce el código objeto a código máquina.

Estos dos pasos se pueden mandar hacer por separado, almacenando el resultado de la fase de compilación en archivos objetos (un típico .obj para Microsoft Windows, .o para Unix), para enlazarlos posteriormente, o crear directamente el ejecutable con lo que la fase de compilación se almacena sólo temporalmente.

Un programa podría tener partes escritas en varios lenguajes (generalmente C, C++ y Asm), que se podrían compilar de forma independiente y enlazar juntas para formar un único ejecutable.

Pseudocódigo

Un pseudocódigo o falso lenguaje, es una serie de normas léxicas y gramaticales parecidas a la mayoría de los lenguajes de programación, pero sin llegar a la rigidez de sintaxis de estos ni a la fluidez del lenguaje coloquial. Esto permite codificar un programa con mayor agilidad que en cualquier lenguaje de programación, con la misma validez semántica, normalmente se utiliza en las fases de análisis o diseño de Software, o en el estudio de un algoritmo. Forma parte de las distintas herramientas de la ingeniería de software.

No hay ningún compilador o intérprete de pseudocódigo informático, y por tanto no puede ser ejecutado en un computador, pero las similitudes con la mayoría de los lenguajes informáticos lo hacen fácilmente convertible.

El pseudocódigo describe un algoritmo utilizando una mezcla de frases en lenguaje común, instrucciones de programación y palabras clave que definen las estructuras básicas. Su objetivo es permitir que el programador se centre en los aspectos lógicos de la solución, evitando las reglas de sintaxis de los lenguajes de programación convencionales.

No siendo el pseudocódigo un lenguaje formal, varían de un programador a otro, es decir, no hay una estructura semántica ni arquitectura estándar. Es una herramienta ágil para el estudio y diseño de aplicaciones, veamos un ejemplo, que podríamos definir como: lenguaje imperativo, de tercera generación, según el método de programación estructurada.

Paradigma de Programación

Un paradigma es una forma de representar y manipular el conocimiento. Representa un enfoque particular o filosofía para la construcción del software. No es mejor uno que otro sino que cada uno tiene ventajas y desventajas. También hay situaciones donde un paradigma resulta más apropiado que otro.

Algunos ejemplos de paradigmas de programación:

- El paradigma imperativo es considerado el más común y está representado, por ejemplo, por el C o por BASIC.
- El paradigma funcional está representado por la familia de lenguajes LISP, en particular Scheme.
- El paradigma lógico, un ejemplo es PROLOG.

- El paradigma orientado a objetos. Un lenguaje completamente orientado a objetos es Smalltalk.

Si bien puede seleccionarse la forma pura de estos paradigmas al momento de programar, en la práctica es habitual que se mezclen, dando lugar a la programación multiparadigma.

Programación Estructurada

La programación estructurada es una forma de escribir programas para computadoras de forma clara, para ello utiliza únicamente tres estructuras: secuencial, selectiva e iterativa; siendo innecesario y no permitiéndose el uso de la instrucción o instrucciones de transferencia incondicional (GOTO).

A finales de los años sesenta surgió una nueva forma de programar que no solamente daba lugar a programas fiables y eficientes, sino que además estaban escritos de manera que facilitaba su comprensión posterior.

Un famoso Teorema de Dijkstra, demostrado por Edsger Dijkstra en los años sesenta, comprueba que todo programa puede escribirse utilizando únicamente las tres instrucciones de control siguientes:

- Secuencial de instrucciones.
- Instrucción condicional.
- Iteración, o bucle de instrucciones.

Sólo con estas tres estructuras se puede hacer un programa informático, si bien los lenguajes de programación, y sus compiladores, tienen un repertorio de estructuras de control mayor.

Ventajas de la Programación Estructurada

Con la programación estructurada, elaborar programas de computador sigue siendo una labor que demanda esfuerzo, creatividad, habilidad y cuidado. Sin embargo, con este nuevo estilo se pueden obtener las siguientes ventajas:

1. Los programas son más fáciles de entender. Un programa estructurado puede ser leído en secuencia, de arriba hacia abajo, sin necesidad de estar saltando de un sitio a otro en la lógica, lo cual es típico de otros estilos de programación. La estructura del programa es más clara puesto que las instrucciones están más ligadas o relacionadas entre sí, por lo que es más fácil comprender lo que hace cada función.
2. Reducción del esfuerzo en las pruebas. El programa se puede tener listo para producción normal en un tiempo menor del tradicional; por otro lado, el seguimiento de las fallas se facilita debido a la lógica más visible, de tal forma que los errores se pueden detectar y corregir más fácilmente.
3. Reducción de los costos de mantenimiento.

4. Programas más sencillos y más rápidos.
5. Aumento de la productividad del programador.
6. Se facilita la utilización de las otras técnicas para el mejoramiento de la productividad en programación.
7. Los programas quedan mejor documentados internamente.

Inconvenientes de la Programación Estructurada

El principal inconveniente de este método de programación, es que se obtiene un único bloque de programa, que cuando se hace demasiado grande puede resultar problemático su manejo, esto se resuelve empleando la programación modular, definiendo módulos interdependientes programados y compilados por separado, cada uno de los cuales ha podido ser desarrollado con programación estructurada.

Un método un poco más sofisticado es la programación por capas, en la que los módulos tienen una estructura jerárquica muy definida y se denominan capas.

Programación por Capas

La programación por capas es un estilo de programación en la que el objetivo primordial es la separación de la lógica de negocios de la lógica de diseño. Un ejemplo básico de esto es separar la capa de datos de la capa de presentación al usuario.

La ventaja principal de este estilo, es que el desarrollo se puede llevar a cabo en varios niveles y en caso de algún cambio sólo se ataca al nivel requerido sin tener que revisar entre código mezclado. Un buen ejemplo de este método de programación seria: Modelo de interconexión de sistemas abiertos. Además permite distribuir el trabajo de creación de una aplicación por niveles, de este modo, cada grupo de trabajo está totalmente abstraído del resto de niveles, simplemente es necesario conocer la API que existe entre niveles.

En el diseño de sistemas informáticos actual se suele usar las arquitecturas multinivel o programación por capas. En dichas arquitecturas a cada nivel se le confía una misión simple, lo que permite el diseño de arquitecturas escalables (que pueden ampliarse con facilidad en caso de que las necesidades aumenten). El diseño más en boga actualmente es el diseño en tres niveles (o en tres capas).

Capas o Niveles

1. Capa de presentación: es la que ve el usuario, presenta el sistema al usuario, le comunica la información y captura la información del usuario dando un mínimo de proceso (realiza un filtrado previo para comprobar que no hay errores de formato). Esta capa se comunica únicamente con la capa de negocio.

2. Capa de negocio: es donde residen los programas que se ejecutan, recibiendo las peticiones del usuario y enviando las respuestas tras el proceso. Se denomina capa de negocio (e incluso de lógica del negocio) pues es aquí donde se establecen todas las reglas que deben cumplirse. Esta capa se comunica con la capa de presentación, para recibir las solicitudes y presentar los resultados, y con la capa de datos, para solicitar al gestor de base de datos para almacenar o recuperar datos de él.
3. Capa de datos: es donde residen los datos. Está formada por uno o más gestor de bases de datos que realiza todo el almacenamiento de datos, reciben solicitudes de almacenamiento o recuperación de información desde la capa de negocio.

Todas estas capas pueden residir en un único ordenador, si bien lo más usual es que haya una multitud de ordenadores donde reside la capa de presentación (son los clientes de la arquitectura cliente/servidor). Las capas de negocio y de datos pueden residir en el mismo ordenador, y si el crecimiento de las necesidades lo aconseja se pueden separar en dos o mas ordenadores. Así, si el tamaño o complejidad de la base de datos aumenta, se puede separar en varios ordenadores los cuales recibirán las peticiones del ordenador en que resida la capa de negocio.

Si por el contrario fuese la complejidad en la capa de negocio lo que obligase a la separación, esta capa de negocio podría residir en uno o más ordenadores que realizarían solicitudes a una única base de datos. En sistemas muy complejos se llega a tener una serie de ordenadores sobre los cuales corre la capa de datos, y otra serie de ordenadores sobre los cuales corre la base de datos.

Programación Modular

La modularidad es la propiedad de los programas de computación en la cual están compuestos de partes separadas llamadas módulos. Los programas que tienen muchas relaciones directas entre 2 partes del código al azar son menos modulares que programas donde esas relaciones ocurren principalmente en interfaces bien definidas para los módulos.

Programación Imperativa

La programación imperativa, en contraposición a la programación declarativa, es un paradigma de programación que describe la programación en términos del estado del programa y sentencias que cambian dicho estado. Los programas imperativos son un conjunto de instrucciones que le indican al computador cómo realizar una tarea.

La implementación de hardware de la mayoría de computadores es imperativa; prácticamente todo el hardware de los computadores está diseñado para ejecutar código de máquina, que es nativo al computador, escrito en una forma imperativa. Esto se debe a que el hardware de los computadores implementa el paradigma de las Máquinas de Turing. Desde esta perspectiva de bajo nivel, el estilo del programa está definido por los contenidos de la memoria, y las sentencias

son instrucciones en el lenguaje de máquina nativo del computador (por ejemplo el lenguaje ensamblador).

Los lenguajes imperativos de alto nivel usan variables y sentencias más complejas, pero aún siguen el mismo paradigma. Las recetas y las listas de revisión de procesos, a pesar de no ser programas de computadora, son también conceptos familiares similares en estilo a la programación imperativa; cada paso es una instrucción, y el mundo físico guarda el estado.

Los primeros lenguajes imperativos fueron los lenguajes de máquina de los computadores originales. En estos lenguajes, las instrucciones fueron muy simples, lo cual hizo la implementación de hardware fácil, pero obstruyendo la creación de programas complejos. El Fortran (*FORmula TRANslator*) cuyo desarrollo fue iniciado en 1954 por John Backus en IBM, fue el primer gran lenguaje de programación en superar los obstáculos presentados por el código de máquina en la creación de programas complejos.

La lista de lenguajes imperativos incluye al Basic, Pascal, C, C++, Java, C# y Perl.

Programación Funcional

La programación funcional es un paradigma de programación declarativa basado en la utilización de funciones matemáticas. Sus orígenes provienen del Cálculo Lambda, una teoría matemática elaborada por Alonzo Church como apoyo a sus estudios sobre computabilidad. Un lenguaje funcional es a grandes rasgos, un azúcar sintáctico del Cálculo Lambda.

El objetivo de la programación funcional es conseguir lenguajes expresivos y matemáticamente elegantes, en los que no sea necesario bajar al nivel de la máquina para describir el proceso llevado a cabo por el programa, y evitando el concepto de estado del cómputo. La secuencia de computaciones llevadas a cabo por el programa se regiría única y exclusivamente por la reescritura de definiciones más amplias a otras cada vez más concretas y definidas, usando lo que se denominan "definiciones dirigidas".

Los programas escritos en un lenguaje funcional están constituidos únicamente por definiciones de funciones, entendiendo éstas no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas, en las que se verifican ciertas propiedades como la transparencia referencial (el significado de una expresión depende únicamente del significado de sus subexpresiones), y por tanto, la carencia total de efectos laterales.

Otras características propias de estos lenguajes son la no existencia de asignaciones de variables y la falta de construcciones estructuradas como la secuencia o la iteración (lo que obliga en la práctica a que todas las repeticiones de instrucciones se lleven a cabo por medio de funciones recursivas).

Existen dos grandes categorías de lenguajes funcionales: los funcionales puros y los híbridos. La diferencia entre ambos estriba en que los lenguajes funcionales híbridos son menos dogmáticos que los puros, al admitir conceptos tomados de los lenguajes imperativos, como las secuencias de instrucciones o la asignación de variables. En contraste, los lenguajes funcionales puros tienen una mayor potencia expresiva, conservando a la vez su transparencia referencial, algo que no se cumple siempre con un lenguaje funcional híbrido.

Entre los lenguajes funcionales puros, cabe destacar a Haskell y Miranda. Los lenguajes funcionales híbridos más conocidos son Lisp, Scheme, Ocaml y Standard ML (estos dos últimos, descendientes del lenguaje ML).

Programación Lógica

La programación lógica consiste en la aplicación del corpus de conocimiento sobre lógica para el diseño de lenguajes de programación. La programación lógica comprende dos paradigmas de programación: la programación declarativa y la programación funcional. La programación declarativa gira en torno al concepto de predicado, o relación entre elementos. La programación funcional se basa en el concepto de función (que no es más que una evolución de los predicados), de corte más matemático.

Históricamente, los computadores se han programado utilizando lenguajes muy cercanos a las peculiaridades de la propia máquina: operaciones aritméticas simples, instrucciones de acceso a memoria, etc. Un programa escrito de esta manera puede ocultar totalmente su propósito a la comprensión de un ser humano, incluso uno entrenado. Hoy día, estos lenguajes pertenecientes al paradigma de la programación imperativa han evolucionado de manera que ya no son tan crípticos. Sin embargo, aún existen casos donde el uso de lenguajes imperativos es inviable debido a la complejidad del problema a resolver.

En cambio, la lógica matemática es la manera más sencilla, para el intelecto humano, de expresar formalmente problemas complejos y de resolverlos mediante la aplicación de reglas, hipótesis y teoremas. De ahí que el concepto de "programación lógica" resulte atractivo en diversos campos donde la programación tradicional es un fracaso.

La programación lógica encuentra su hábitat natural en aplicaciones de inteligencia artificial o relacionadas:

- Sistemas expertos, donde un sistema de información imita las recomendaciones de un experto sobre algún dominio de conocimiento.
- Demostración automática de teoremas, donde un programa genera nuevos teoremas sobre una teoría existente.
- Reconocimiento de lenguaje natural, donde un programa es capaz de comprender (con limitaciones) la información contenida en una expresión lingüística humana.

La programación lógica también se utiliza en aplicaciones más "mundanas" pero de manera muy limitada, ya que la programación tradicional es más adecuada a tareas de propósito general.

La mayoría de los lenguajes de programación lógica se basan en la teoría lógica de primer orden, aunque también incorporan algunos comportamientos de orden superior. En este sentido, destacan los lenguajes funcionales, ya que se basan en el cálculo lambda, que es la única teoría lógica de orden superior que es demostradamente computable.

El lenguaje de programación lógica por excelencia es el Prolog.

Programación Orientada a Objetos

La Programación Orientada a Objetos (POO u OOP según siglas en inglés) es un paradigma de programación que define los programas en términos de "clases de objetos", objetos que son entidades que combinan estado (es decir, datos), comportamiento (esto es, procedimientos o métodos) e identidad (propiedad del objeto que lo diferencia del resto). La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener y reutilizar.

De esta forma, un objeto contiene toda la información, (los denominados atributos) que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases (e incluso entre objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos). A su vez, dispone de mecanismos de interacción (los llamados métodos) que favorecen la comunicación entre objetos (de una misma clase o de distintas), y en consecuencia, el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separan (ni deben separarse) información (datos) y procesamiento (métodos).

Dada esta propiedad de conjunto de una clase de objetos, que al contar con una serie de atributos definitorios, requiere de unos métodos para poder tratarlos (lo que hace que ambos conceptos están íntimamente entrelazados), el programador debe pensar indistintamente en ambos términos, ya que no debe nunca separar o dar mayor importancia a los atributos en favor de los métodos, ni viceversa. Hacerlo puede llevar al programador a seguir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen esa información por otro (llegando a una programación estructurada camuflada en un lenguaje de programación orientado a objetos).

Esto difiere de los lenguajes imperativos tradicionales, en los que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. Los programadores de lenguajes imperativos escriben funciones y después les pasan datos. Los programadores que emplean lenguajes orientados a objetos definen objetos con datos y métodos y después envían mensajes a los objetos diciendo qué realicen esos métodos en sí mismos.

Algunas personas también distinguen la POO sin clases, la cual es llamada a veces programación basada en objetos.

Los conceptos de la programación orientada a objetos tienen origen en Simula 67, un lenguaje diseñado para hacer simulaciones, creado por Ole-Johan Dahl y Kristen Nygaard del Centro de Cómputo Noruego en Oslo. Según se informa, la historia es que trabajaban en simulaciones de naves, y fueron confundidos por la explosión combinatoria de cómo las diversas cualidades de diversas naves podían afectar unas a las otras. La idea ocurrió para agrupar los diversos tipos de naves en diversas clases de objetos, siendo responsable cada clase de objetos de definir sus propios datos y comportamiento. Fueron refinados más tarde en Smalltalk, que fue desarrollado en Simula en Xerox PARC (y cuya primera versión fue escrita sobre Basic) pero diseñado para ser

un sistema completamente dinámico en el cual los objetos se podrían crear y modificar "en marcha" en lugar de tener un sistema basado en programas estáticos.

La programación orientada a objetos tomó posición como la metodología de programación dominante a mediados de los años ochenta, en gran parte debido a la influencia de C++, una extensión del lenguaje de programación C. Su dominación fue consolidada gracias al auge de las Interfaces gráficas de usuario, para los cuales la programación orientada a objetos está particularmente bien adaptada. En este caso, se habla también de programación orientada a eventos.

Las características de orientación a objetos fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, BASIC, Lisp, Pascal, y otros. La adición de estas características a los lenguajes que no fueron diseñados inicialmente para ellas condujo a menudo a problemas de compatibilidad y a la capacidad de mantenimiento del código. Los lenguajes orientados a objetos "puros", por otra parte, carecían de las características de las cuales muchos programadores habían venido a depender. Para saltar este obstáculo, se hicieron muchas tentativas para crear nuevos lenguajes basados en métodos orientados a objetos, pero permitiendo algunas características imperativas de maneras "seguras". El Eiffel de Bertrand Meyer fue un temprano y moderadamente acertado lenguaje con esos objetivos pero ahora ha sido esencialmente reemplazado por Java, en gran parte debido a la aparición de Internet, y a la implementación de la máquina virtual de Java en la mayoría de navegadores.

Diferencias con la Programación Imperativa

Aunque la programación imperativa (a veces llamada procedural o procedimental) condujo a mejoras de la técnica de programación secuencial, tales como la programación estructurada y "refinamientos sucesivos", los métodos modernos de diseño de software orientado a objetos incluyen mejoras entre las que están el uso de los patrones de diseño, diseño por contrato, y lenguajes de modelado (ej: UML).

Las principales diferencias entre la programación imperativa y la orientada a objetos son:

- La programación orientada a objetos es más moderna, es una evolución de la programación imperativa que plasma en el diseño de una familia de lenguajes conceptos que existían previamente con algunos nuevos.
- La programación orientada a objetos se basa en lenguajes que soportan sintáctica y semánticamente la unión entre los tipos abstractos de datos y sus operaciones (a esta unión se la suele llamar clase).
- La programación orientada a objetos incorpora en su entorno de ejecución mecanismos tales como el polimorfismo y el envío de mensajes entre objetos.

Erróneamente se le adjudica a la programación imperativa clásica ciertos problemas como si fueran inherentes a la misma. Esos problemas fueron haciéndose cada vez más graves y antes de la programación orientada a objetos diversos autores encontraron soluciones basadas en

aplicar estrictas metodologías de trabajo. De esa época son los conceptos de cohesión y acoplamiento. De esos problemas se destacan los siguientes:

- Modelo mental anómalo. La imagen del mundo se apoya en los seres, a los que se asignan nombres sustantivos, mientras la programación clásica se basa en el comportamiento, representado usualmente por verbos.
- Es difícil modificar y extender los programas, pues suele haber datos compartidos por varios subprogramas, que introducen interacciones ocultas entre ellos.
- Es difícil mantener los programas. Casi todos los sistemas informáticos grandes tienen errores ocultos, que no surgen a la luz hasta después de muchas horas de funcionamiento.
- Es difícil reutilizar los programas. Es prácticamente imposible aprovechar en una aplicación nueva las subrutinas que se diseñaron para otra.
- Es compleja la coordinación y organización entre programadores para la creación de aplicaciones de media y gran envergadura.

En la programación orientada a objetos pura no deben utilizarse llamadas de subrutinas, únicamente mensajes. Por ello, a veces recibe el nombre de programación sin CALL, igual que la programación estructurada se llama también programación sin GOTO. Sin embargo, no todos los lenguajes orientados a objetos prohíben la instrucción CALL (o su equivalente), permitiendo realizar programación híbrida, imperativa y orientada a objetos a la vez.

La programación orientada a objetos es una nueva forma de programar que trata de encontrar solución a estos problemas. Introduce nuevos conceptos, que superan y amplían conceptos antiguos ya conocidos. Entre ellos destacan los siguientes:

- Objeto: entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad ("métodos"). Corresponden a los objetos reales del mundo que nos rodea, o a objetos internos del sistema (del programa).
- Clase: definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.
- Método: algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "mensaje". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.
- Evento: un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto). El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente.
- Mensaje: una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.
- Propiedad o atributo: contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera del objeto, y cuyo valor puede ser alterado por la ejecución de algún método.

- Estado interno: es una propiedad invisible de los objetos, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos).

En comparación con un lenguaje imperativo, una "variable" no es más que un contenedor interno del atributo del objeto o de un estado interno, así como la "función" es un procedimiento interno del método del objeto.

Características de la Programación Orientada a Objetos

Hay un cierto desacuerdo sobre exactamente qué características de un método de programación o lenguaje le definen como "orientado a objetos", pero hay un consenso general en que las características siguientes son las más importantes (para más información, seguir los enlaces respectivos):

- Abstracción: Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar cómo se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos y cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción.
- Encapsulamiento: también llamado "ocultación de la información". Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.
- Polimorfismo: comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama asignación tardía o asignación dinámica. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.
- Herencia: las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que

reimplementar su comportamiento. Esto suele hacerse habitualmente agrupando los objetos en clases y estas en árboles o enrejados que reflejan un comportamiento común. Cuando un objeto pertenece a más de una clase se dice que hay herencia múltiple; esta característica no está soportada por algunos lenguajes (como Java).

La lista de lenguajes orientados a objeto incluye al Ada, C++, C#, Visual Basic.net, Clarion, Delphi, Eiffel, Java, Lexico, Objective-C, Ocaml, Oz, PHP, PowerBuilder, Pitón, Ruby y Smalltalk.

Programación Orientada a Aspectos

La Programación Orientada a Aspectos (POA) es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la POA se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos y eliminando las dependencias inherentes entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables. Varias tecnologías con nombres diferentes se encaminan a la consecución de los mismos objetivos y así, el término POA es usado para referirse a varias tecnologías relacionadas como los métodos adaptativos, los filtros de composición, la programación orientada a sujetos o la separación multidimensional de competencias.

Lenguaje de Programación C

C es un lenguaje de programación creado en 1969 por Ken Thompson y Dennis M. Ritchie en los Laboratorios Bell como evolución del anterior lenguaje B, a su vez basado en BCPL. Al igual que B, es un lenguaje orientado a la implementación de Sistemas Operativos, concretamente Unix. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas, aunque también se utiliza para crear aplicaciones.

Se trata de un lenguaje fuertemente tipado de medio nivel pero con muchas características de bajo nivel. Dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.

La primera estandarización del lenguaje C fue en ANSI, con el estándar X3.159-1989. El lenguaje que define este estándar fue conocido vulgarmente como ANSI C. Posteriormente, en 1990, fue ratificado como estándar ISO (ISO/IEC 9899:1990). La adopción de este estándar es muy amplia por lo que, si los programas creados lo siguen, el código es portable entre plataformas y/o arquitecturas. En la práctica, los programadores suelen usar elementos no-portables dependientes del compilador o del sistema operativo.

Filosofía

C es un lenguaje de programación relativamente minimalista. Uno de los objetivos de diseño de este lenguaje fue que sólo fueran necesarias unas pocas instrucciones en lenguaje máquina para traducir cada elemento del lenguaje, sin que hiciera falta un soporte intenso en tiempo de ejecución. Es muy posible escribir C a bajo nivel de abstracción; de hecho, C se usó como intermediario entre diferentes lenguajes.

En parte a causa de ser de relativamente bajo nivel y de tener un conjunto de características modesto, se pueden desarrollar compiladores de C fácilmente. En consecuencia, el lenguaje C está disponible en un amplio abanico de plataformas (seguramente más que cualquier otro lenguaje). Además, a pesar de su naturaleza de bajo nivel, el lenguaje se desarrolló para incentivar la programación independiente de la máquina. Un programa escrito cumpliendo los estándares e intentando que sea portable puede compilarse en muchos computadores.

C se desarrolló originalmente (conjuntamente con el sistema operativo Unix, con el que ha estado asociado mucho tiempo) por programadores para programadores. Sin embargo, ha alcanzado una popularidad enorme, y se ha usado en contextos muy alejados de la programación de sistemas, para la que se diseñó originalmente.

C tiene las siguientes características de importancia:

- Un núcleo del lenguaje simple, con funcionalidades añadidas importantes, como funciones matemáticas y de manejo de ficheros, proporcionadas por bibliotecas.
- Es un lenguaje muy flexible que permite programar con múltiples estilos. Uno de los más empleados es el estructurado no llevado al extremo (permitiendo ciertas licencias rupturistas).
- Un sistema de tipos que impide operaciones sin sentido.
- Usa un lenguaje de preprocesado, el preprocesador de C, para tareas como definir macros e incluir múltiples ficheros de código fuente.
- Acceso a memoria de bajo nivel mediante el uso de punteros.
- Un conjunto reducido de palabras clave.
- Los parámetros se pasan por valor. El paso por referencia se puede simular pasando explícitamente el valor de los punteros.
- Punteros a funciones y variables estáticas, que permiten una forma rudimentaria de encapsulado y polimorfismo.
- Tipos de datos agregados (structs) que permiten que datos relacionados se combinen y se manipulen como un todo.

Algunas características de las que C carece que se encuentran en otros lenguajes:

- Recolección de basura.
- Soporte para programación orientada a objetos, aunque la implementación original de C++ fue un preprocesador que traducía código fuente de C++ a C.
- Encapsulamiento.

- Funciones anidadas, aunque GCC tiene esta característica como extensión.
- Polimorfismo en tiempo de código en forma de sobrecarga, sobrecarga de operadores y sólo dispone de un soporte rudimentario para la programación genérica.
- Soporte nativo para programación multihilo y redes de computadores.

Aunque la lista de las características útiles de las que carece C es larga, este factor ha sido importante para su aceptación, porque escribir rápidamente nuevos compiladores para nuevas plataformas, mantiene lo que realmente hace el programa bajo el control directo del programador, y permite implementar la solución más natural para cada plataforma. Esta es la causa de que a menudo C sea más eficiente que otros lenguajes. Típicamente, sólo la programación cuidadosa en lenguaje ensamblador produce un código más rápido, pues da control total sobre la máquina, aunque los avances en los compiladores de C y la complejidad creciente de los procesadores modernos han reducido gradualmente esta diferencia.

En algunos casos, una característica inexistente puede aproximarse. Por ejemplo, la implementación original de C++ consistía en un preprocesador que traducía código fuente C++ a C. La mayoría de las funciones orientadas a objetos incluyen un puntero especial, que normalmente recibe el nombre "this", que se refiere al objeto al que pertenece la función. Mediante el paso de este puntero como un argumento de función, esta funcionalidad puede desempeñarse en C. Por ejemplo, en C++ se puede escribir:

```
stack.push(val);
```

Mientras que en C, se podría escribir:

```
push(stack, val);
```

Donde el argumento stack es un puntero a una struct equivalente al puntero this de C++, que es un puntero a un objeto.

Historia

El desarrollo inicial de C se llevó a cabo en los Laboratorios Bell de AT&T entre 1969 y 1973; según Ritchie, el periodo más creativo tuvo lugar en 1972. Se le dio el nombre "C" porque muchas de sus características fueron tomadas de un lenguaje anterior llamado "B".

Hay muchas leyendas acerca del origen de C y el sistema operativo con el que está íntimamente relacionado, Unix. Algunas de ellas son:

- El desarrollo de C fue el resultado del deseo de los programadores de jugar con Space Travel. Habían estado jugando en el mainframe de su compañía, pero debido a su poca capacidad de proceso y al tener que soportar 100 usuarios, Thompson y Ritchie no tenían suficiente control sobre la nave para evitar colisiones con los asteroides. Por ese motivo decidieron portar el juego a un PDP-7 de la oficina que no se utilizaba; pero esa máquina no tenía

sistema operativo, así que decidieron escribir uno. Finalmente decidieron portar el sistema operativo del PDP-11 que había en su oficina, pero era muy costoso, pues todo el código estaba escrito en lenguaje ensamblador. Entonces decidieron usar un lenguaje de alto nivel y portable para que el sistema operativo se pudiera portar fácilmente de un ordenador a otro. Consideraron usar B, pero carecía de las funcionalidades necesarias para aprovechar algunas características avanzadas del PDP-11. Entonces empezaron a crear un nuevo lenguaje, C.

- La justificación para obtener el ordenador original que se usó para desarrollar Unix fue crear un sistema que automatizase el archivo de patentes. La versión original de Unix se desarrolló en lenguaje ensamblador. Más tarde, el lenguaje C se desarrolló para poder reescribir el sistema operativo.

En 1973, el lenguaje C se había vuelto tan potente que la mayor parte del kernel Unix, originalmente escrito en el lenguaje ensamblador PDP-11/20, fue reescrita en C. Este fue uno de los primeros núcleos de sistema operativo implementados en un lenguaje distinto al ensamblador. (Algunos casos anteriores son el sistema Multics, escrito en PL/I, y Master Control Program para el B5000 de Burroughs, escrito en Algol en 1961).

En 1978, Ritchie y Brian Kernighan publicaron la primera edición de El lenguaje de programación C. Este libro fue durante años la especificación informal del lenguaje. El lenguaje descrito en este libro recibe habitualmente el nombre de "el C de Kernighan y Ritchie" o simplemente "K&R C" (La segunda edición del libro cubre el estándar ANSI C, descrito más abajo.)

Kernighan y Ritchie introdujeron las siguientes características al lenguaje:

- El tipo de datos struct.
- El tipo de datos long int.
- El tipo de datos unsigned int.
- Los operadores =+ y =- fueron sustituidos por += y -= para eliminar la ambigüedad semántica de expresiones como i=-10, que se podría interpretar bien como i =- 10 o bien como i = -10.

El C de Kernighan y Ritchie es el subconjunto más básico del lenguaje que un compilador debe de soportar. Durante muchos años, incluso tras la introducción del ANSI C, fue considerado "el mínimo común denominador" en el que los programadores debían programar cuando deseaban que sus programas fueran transportables, pues no todos los compiladores soportaban completamente ANSI, y el código razonablemente bien escrito en K&R C es también código ANSI C válido.

En estas primeras versiones de C, las únicas funciones que necesitaban ser declaradas si se usaban antes de la definición de la función eran las que retornaban valores no enteros. Es decir, se presuponía que una función que se usaba sin declaración previa devolvería un entero.

Dado que el lenguaje C de K&R no incluía ninguna información sobre los argumentos de las funciones, no se realizaba comprobación de tipos en los parámetros de las funciones, aunque algunos compiladores lanzan mensajes de advertencia si se llamaba a una función con un número incorrecto de argumentos.

En los años siguientes a la publicación del C de Kernighan y Ritchie, se añadieron al lenguaje muchas características no oficiales, que estaba soportadas por los compiladores de AT&T, entre otros. Algunas de estas características eran:

- Funciones void y el tipo de datos void *.
- Funciones que retornaban tipos de datos struct o union (en lugar de punteros).
- Asignación de tipos de datos struct.
- Calificador const, que hace que un objeto sea de sólo lectura.
- Una librería estándar, que incorporaba la mayoría de las funcionalidades implementadas por varios desarrolladores de compiladores.
- Enumeraciones.

ANSI C e ISO C

A finales de la década de 1970, C empezó a sustituir a BASIC en como lenguaje de programación de microcomputadores predominante. Durante la década de 1980 se empezó a usar en los IBM PC, lo que incrementó su popularidad significativamente. Al mismo tiempo, Bjarne Stroustrup empezó a trabajar con algunos compañeros de Bell Labs para añadir funcionalidades de programación orientada a objetos a C. El lenguaje que crearon, llamado C++, es hoy en día el lenguaje de programación de aplicaciones más común en el sistema operativo Microsoft Windows; mientras que C sigue siendo más popular en el entorno Unix. Otro lenguaje que se desarrolló en esa época, Objective C, también añadió características de programación orientada a objetos a C. Aunque hoy en día no es tan popular como C++, se usa para desarrollar aplicaciones Cocoa para Mac OS X.

En 1983, el Instituto Nacional Estadounidense de Estándares organizó un comité, X3j11, para establecer una especificación estándar de C. Tras un proceso largo y arduo, se completó el estándar en 1989 y se ratificó como el "Lenguaje de Programación C" ANSI X3.159-1989. Esta versión del lenguaje se conoce a menudo como ANSI C, o a veces como C89 (para distinguirla de C99). En 1990, el estándar ANSI (con algunas modificaciones menores) fue adoptado por la Organización Internacional para la Estandarización (ISO) en el estándar ISO/IEC 9899:1990. Esta versión se conoce a veces como C90. No obstante, "C89" y "C90" se refieren en esencia el mismo lenguaje. Uno de los objetivos del proceso de estandarización del ANSI C fue producir una extensión al C de Kernighan y Ritchie, incorporando muchas funcionalidades no oficiales. Sin embargo, el comité e estandarización incluyó también muchas funcionalidades nuevas, como prototipos de función, y un preprocesador mejorado. También se cambió la sintaxis de la declaración de parámetros para hacerla semejante a la empleada habitualmente en C++:

```
int main (argc,argv)
     int argc;
     char **argv
{
…
}
```

ANSI C está soportado hoy en día por casi la totalidad de los compiladores. La mayoría del código C que se escribe actualmente está basado en ANSI C. Cualquier programa escrito sólo en C estándar sin código que dependa de un hardware determinado funciona correctamente en cualquier plataforma que disponga de una implementación de C compatible. Sin embargo, muchos programas han sido escritos de forma que sólo pueden compilarse en una cierta plataforma, o con un compilador concreto, debido a (i) la utilización de bibliotecas no estándar, como interfaces gráficos de usuario, (ii) algunos compiladores no cumplen, en el modo por defecto, las especificaciones del estándar ANSI C o su sucesor, o (iii) el código está escrito con dependencia de un tamaño determinado de ciertos tipos de datos, o de un determinado orden de los bits de la plataforma.

La macro __STDC__ puede usarse para dividir el código en secciones ANSI y K&R.

```
#if __STDC__
extern int getopt (int,char * const *,const char *);
#else
extern int getopt ();
#endif
```

C99

Tras el proceso de estandarización de ANSI, la especificación del lenguaje C permaneció relativamente estable durante algún tiempo, mientras que C++ siguió evolucionando. Sin embargo, el estándar continuó bajo revisión a finales de la década de 1990, lo que llevó a la publicación del estándar ISO 9899:1999 en 1999. Este estándar se denomina habitualmente "C99". Se adoptó como estándar ANSI en marzo de 2000. Las nuevas características de C99 incluyen:

- Funciones inline.
- Las variables pueden declararse en cualquier sitio (como en C++), en lugar de poder declararse sólo tras otra declaración o al comienzo de una declaración compuesta.
- Muchos tipos de datos, incluyendo long long int (para reducir el engorro de la transición de 32 bits a 64 bits), un tipo de datos booleano, y un tipo complex que representa números complejos.
- Arrays de longitud variable.
- Soporte para comentarios de una línea que empiecen con //, como en BCPL o en C++, característica para la que muchos compiladores habían dado soporte por su cuenta.
- Muchas funciones nuevas, como snprintf()
- Muchos headers nuevos, como stdint.h.

Una consideración importante es que hasta la publicación de este estándar, C había sido mayormente un subconjunto estricto del C++. Era muy sencillo "actualizar" un programa de C hacia C++ y mantener ese código compilable en ambos lenguajes. Sin embargo, el nuevo estándar agrega algunas características que C++ no admite, como por ejemplo los inicializadores estáticos de estructuras. También define al tipo "bool" de una manera que no es exactamente la del C++.

El compilador GCC, entre muchos otros, soportan hoy en día la mayoría de las nuevas características de C99. Sin embargo, este nuevo estándar ha tenido peor acogida entre algunos desarrolladores de compiladores, como Microsoft y Borland, que se han centrado en C++. Brandon Bray, de Microsoft, dijo a este respecto: "En general, hemos visto poca demanda de muchas características de C99. Algunas características tienen más demanda que otras, y consideraremos incluirlas en versiones futuras siempre que sean compatibles con C++."

Ventajas

- Lenguaje muy eficiente puesto que es posible utilizar sus características de bajo nivel para realizar implementaciones óptimas.
- A pesar de su bajo nivel es el lenguaje más portado en existencia, habiendo compiladores para casi todos los sistemas conocidos.
- Proporciona facilidades para realizar programas modulares y/o utilizar código o bibliotecas existentes.

Estructura de un Programa en Lenguaje C

Los programas en lenguaje C se construyen a partir de la definición de funciones. Existe una función principal (llamada main) que es la primera en ejecutarse. Las funciones están constituidas por un encabezado y un cuerpo.

En el encabezado se definen el nombre de la función y los argumentos que esta recibe. En el cuerpo de la función se declaran las variables locales y las sentencias que constituyen la función. Es posible declarar variables fuera de las funciones, en cuyo caso dichas variables serán de tipo global.

El programa más sencillo que se puede escribir en lenguaje C es el siguiente:

```
main ()
{
}
```

Aunque este programa no lleva a cabo ninguna tarea, compilará exitosamente y permitirá generar un archivo ejecutable.