Variables

Variable: Menoria principal que consiste en celdas identificadas cada una con una diorección.

Ambito de una variable (Alcance).
Es el conjunto de sentencias del programa en las que dicha variable es conocida y por lo tanto manipulable. Se dice que una variable es visible dentro de su ámbito e invisible fuera de él.

Tiempo de vida de una variable.
Es el intervalo de tiempo en el que un área de memoria está ligada a la variable. Esta área se utiliza para contener el valor de la variable. La acción de conseguir un áre de memoria para na variable se denimina asignación de memoria. En algunos lenguajes esta asignacion se lleva a cabo antes del tiempo de la ejecución del programa (aseganación estática). En otros se hace durante la ejecución (asiganación dinámica).

Valor de una variable. (R-valor)
El valor de una variable se representa en forma codificada en el área de memoria ligada a dicha variable. Esta represetnación codificada se interpreta de acuerdo con el tipo de la variable.

Tipo de una variable.
Se puede definir como una especificacion de la clase de valores que se pueden asociar a la variable junto con las operaciones que se pueden usar para crear, acceder y modificar tales valores.

Tipos de Variables

Estáticas: su lugar es reservado en tiempo de compilación. El tiempo de vida es todo el programa mientras que su alcance es la unidad donde fue definida. Están siempre almacenadas en memoria con el código fuente.
Dinámicas:son aquellas en las que el tiempo de vida está en manos del prorgamador; él es quien mediante instrucciones las aloca y desaloca.
Semidinámicas: su tamaño no se conoce en compilación. En ejecución se conoce cuando va a ocupar y se aloca en el registro de activación.
Automáticas: cada variable local de una rutina comienza a existir cuando se llama a una función y desaparece cuando la función termina; por esta razón estas variables reciben el nombre de automáticas. No conservan su valor entre dos llamados sucesivos. En tiempo de compilación se conoce su tamaño y se crean en el registro de activación pero no se alocan.

Ligadura de Variable

Ligadura de ámbito estático: define el ámbito de una variable en términos de la estructura léxica de un programa; es decir, cada referencia a una variable está estáticamente ligada a una declaración de variables concreta (implícita ó explícita).
Ligadura de ámbito dinámico: define el ámbito de una variable en términos de ejecución del programa. Normalmente, cada declaración de variable ejerce su influencia sobre todas las sentencias que se ejecutan a partir de dicha declaración hasta que se encuentra una nueva declaración para una variable con el mismo nombre.

Alocación de memoria: la acción de conseguir un área de memoria para una variable se denomina asignación de memoria. Esta a veces se lleva a cabo antes del tiempo de ejecución del programa (asignación estática), otras veces durante la ejecución (asignación dinámica) mediante una petición explícita ó bien automáticamente al ingresar el control de ejecución en el ámbito de la variable (asignación automática).
El tiempo de vida de una variable estática es mayor que el de una variable dinámica ó una automática. Las variables dinámicas tienen un tiempo de vida y un alcance equivalentes, si no hay un bloque interno a su alcance que contenga otra variable que la enmascare. Las variables automáticas nacen y mueren (se alocan en memoria y se desalocan) junto con el bloque en que están declaradas.

Se denomina **variable local** a aquella que se declara dentro de un bloque articular. Tiene alcance limitado y su tiempo de vida se define dentro del bloque; es decir, que cuando el bloque se activa se aloca en memoria y cuando se desactiva se desaloca.
Se denomina **variable global** a un bloque a aquella que se declara en un bloque más externo. Su alcance no está limitado y su tiempo de vida es todo el conjunto de bloques. Se conoce mientras alguno de ellos está activo.
En el lenguaje **Pascal**, una variable es global cuando se declara en la sección de variables del programa principal, y es local cuando se declara dentro de un módulo (procedimiento ó función).
En el lenguaje **C**, las variables locales son las automáticas y las globales son aquellas declaradas como externas.
En **Ada** las variables locales son las declaradas en procedimientos, funciones ó bloques y las variables globales son las declaradas en el procedimiento principal.
En Pascal el alcance tiene comienzo en la sentencia begin. En C y en Ada es a partir de la declaración.

Unidades de programas

Los lenguajes de programación hacen posible que un programa se componga de un cierto número de unidades. Las variables que se declaran dentro de una unidad se dicen que son locales a la misma. Una ***unidad*** se puede activar durante la ejecución.
La representación de una unidad de programa durante la ejecución recibe el nombre de activación de la unidad. Una ***activación de unidad*** se compone de un segmento de código y de un registro de activación.
El segmento de código, cuyo contenido es fijo, contiene las instrucciones de la unidad. El contenido del registro de activación es variable, este registro contiene toda la información necesaria para ejecutar la unidad.
Una unidad no es una parte de un programa completamente independiente ni autosuficiente.
Las unidades pueden activarse recursivamente, una unidad puede llamarse a sí misma bien directa o indirectamente a través de otra unidad. Se puede producir una nueva activación de una unidad, antes de que termine la activación anterior. Todas las activaciones de la misma unidad se componen del mismo segmento de código pero de diferentes registros de activación.
Al producirse la recursión, la ligadura entre un registro de activación y su segmento de código es necesariamente dinámica. Siempre que se activa una unidad se debe establecer una ligadura entre un registro de activación y su segmento de código, con el fin de producir una nueva activación de unidad.

Ambiente de referencia de una unidad

Ambiente local: variables locales, ligadas a los objetos almacenados en su registro de activación.

Ambiente no local: variables no locales, ligadas a objetos almacenados en los registros de activación de otras unidades.

Compartir datos entre unidades

Existen dos formas de compartir datos entre unidades:

- **Acceso al ambiente no local**
  - **Explícita:** áreas comunes. La desventaja que tiene es que obliga al programador especificar que es lo que se comparte.
  - **Implícita:** es automático.
    - **Cadena estática**
    - **Cadena dinámica**

En esta forma de compartir datos una desventaja es que no permite una buena moduilarización. Cuando más explícito es lo que se comparte más se acerca al principio del diseño de software; lo cual tiene como ventaja la,legibilidad y lamodificabilidad.

- **Parámetros**

Es la mejor manera de compartir datos. Se dice explícitamente que es lo que se comparte. Es la forma más abstracta, pues lo demás permanece oculto y lo que no se conoce no se puede destruir.

Heap – Utilidades

- En las estructuras dinámicas.
- Cuando la alocación y desalocación de las variables es impredecible.
- Para el caso de "datos dinámicos" se alocan explícitamente durante la ejecución mediante instrucciones de alocación (por ejemplo el new de los punteros). El tiempo de vida no depende de la sentencia de alocación, vivirá mientras este apuntada. El objeto apuntado se alocara en la heap.
- Para el caso de lenguajes dinámicos las ligaduras correspondientes se llevan a cabo en ejecución y no en compilación. Como el tamaño del descriptor puede cambiar debe almacenarse en la heap.

Arreglos dinámicos

Los arreglos dinámicos son variables semidinámicas.

Variables semidinámicas: estas variables se crean automáticamente al activarse la unidad, pero su tamaño puede depender de valores que sólo se conocen en el momento de ejecución al activarse la unidad. Tal es el caso de los arreglos dinámicos, cuyas ligaduras se dan a conocer durante la ejecución.

Sean:

[1:n] Int a;
[1:m] Int b;

Durante la traducción podemos reservar, en el registro de activación, espacio para los descriptores de las matrices dinámicas a y b. El descriptor incluye al menos una posición en la que se almacena un puntero para el área de almacenamiento de la matriz dinámica y otra para los limites inferior y superior de cada dimensión de la matriz. Dado que durante la traducción ya se conoce el numero de dimensión de la matriz (como en el ejemplo), se conocerá estáticamente el tamaño del descriptor. Todos los accesos a variables semidinámicas se traducirán como referencias indirectas a través del puntero en el descriptor, cuyo desplazamiento se determina estáticamente. La asignación del registro de activación para por varias etapas durante la ejecución:

- Se asigna el espacio necesario para las variables semiestáticas y para los descriptores de las variables semidinámicas. Cuando aparece la declaración de una variable semidinámica, se establecen los valores de las entradas de dimensión de los descriptores.
- Se evalúa el tamaño real de la variable semidinámica y se expande el registro de activación para incluir la variable.
- El puntero en el descriptor se coloca apuntando al área que se acaba de asignar ()en la misma pila.

Tipo de datos

Un tipo de datos es un conjunto de valores y un conjunto de operaciones para ser manipulados.

Se caracterizan por:

- un rango de valores posibles.
- Un conjunto de operaciones realizables.
- Su representación interna.

Los tipos se pueden dividir:

- Según su tipo
  - Predefinidos: definidos por el lenguaje.
  - Definido por el usuario.
    - Básico o compuestos.
    - TAD.
- Según su dominio
  - Elemental: el valor es un solo elemento.
  - Compuesto: cada elemento del dominio tiene un conjunto de valores.

Tipos Predefinidos

Características

Un tipo predefinido pueden ser vistos como un mecanismo para clasificar los datos manipulados.por un programa Por ejemplo, boolenos, caracteres, etc. Designar un tipo para un datos indica que operaciones se le pueden realizar legalmente.

Ventajas

Las ventanas de introducir tipos predefinidos son:

- *Ocultamiento de la representación*: no se tiene acceso a la cadena de bit que representa a un valor de un cierto tipo. Solo se puede cambiar la cadena mediante las acciones legales sobre ese tipo y lo que resulta es un nuevo valor del tipo predefinido. Esto incrementa la legibilidad del programa y permite la portabilidad mediante la abstracción de los detalles de la implementación.
- *Uso correcto de variables*: el tipo de cada variable es conocido en tiempo de compilación, se evitan errores de operaciones ilegales sobre la misma en tiempo de compilación.
- *Resolución de sobrecarga de operadores:* si el tipo de cada variable es conocido en tiempo de compilación, el binding entre un operador sobrecargado y su correspondiente operación puede ser establecido en tiempo de compilación.
- *Control de Precisión:* el programador puede indicarle al compilador que aloque memoria necesaria para almacenar un dato con una precisión dada.

En general los tipos predefinidos son elementales (sus valores son atómicos), pero hay excepciones como por ejemplo ADA tiene los strings como tipos.predefinidos y sin embargo son arreglos de caracteres.

Tipos definidos por el usuario y tipos abstractos de datos

Los dos mayores beneficios de introducir tipos en un lenguaje son la clasificación y la protección. Los tipos permiten que los datos estén organizados como una colección de distintas categorías y que estén protegidas de manipulaciones indesesdas especificando qué operaciones son legales para objetos de un tipo dado y ocultando la representación. La clasificación es lograda definiendo una estructura de datos como un tipo; para proveer protección, un nuevo tipo debe ser definido en términos de una estructura oculta y un conjunto de operaciones permitidas para manipularlos. Esto último es lo que se denomina un **tipo abstracto de datos**.

Sistema de tipos

Un sistema de tipos es un conjunto de reglas usadas por el lenguaje para estructurar y organizar su colección de tipos. Las operaciones definidas para un tipo son la única forma de manipular sus instancias, ellas protegen a los objetos de datos de sus ilegales.
Si dice que un programa es seguro en cuanto a los tipos si garantiza no tener errores de tipos.

Chequeo de programa estático vs dinámico

Los errores pueden ser clasificados en dos categorías:
- Errores de lenguaje: son los sintácticos y semánticos.
- Errores de aplicación: son desviaciones en el comportamiento del programa con respecto a las especificaciones.

El chequeo dinámico requiere que el programa sea ejecutado sobre datos de entrada; el estático no lo requiere, siendo este último preferible por dos razones: los errores en tiempo de ejecución dependen de la entrada, sin asegurar que el error sea revelado y el chequeo dinámico aumenta el tiempo de ejecución.

Tipado fuerte y chequeo de tipos

La finalidad de un sistema de tipos es prevenir la escritura de programas inseguros en tipos. Un sistema de tipos es fuerte si garantiza que los programas escritos siguiendo las restricciones del sistema de tipos no generan errores de tipo. Un lenguaje con un sistema de tipos fuerte se dice que es fuertemente tipado.
Los lenguajes tipados estáticamente requieren que el tipo de cada expresión sea conocido en tiempo de compilación. Un lenguaje tipado estáticamente es un lenguaje fuertemente tipado, pero no siempre ocurre al revés.

Compatibilidad de tipos

Un sistema de tipos estricto debe requerir que si una operación espera un operando de tipo T, ésta puede ser invocada sólo con un parámetro de tipo T. Los lenguajes sin embargo frecuentemente permiten más flexibilidad definiendo condiciones bajo las cuales un operando de otro tipo, digamos Q, es también aceptable sin violar la seguridad de tipos, es decir cuando es compatible (equivalente).

- Compatibilidad por nombre: se da cuando un nombre de tipo es compatible sólo por el mismo. Es preferible pues previene que dos tipos sean considerados compatibles sólo por tener representaciones idénticas.
- Compatibilidad estructural: se dan cuando los tipos tienen la misma estructura y cuando los nombres y tipos de sus campos coinciden y ocurren en el mismo orden.

Pascal posee por nombre.
C adopta compatibilidad estructural para todos tipos excepto las estructuras, para las cuales se requiere compatibilidad por nombre.
ADA adopta compatibilidad por nombre, y además los subtipos son compatibles con el tipo base.

Tipos y subtipos

Un **subtipo** es un subconjunto de los valores de su conjunto padre, donde las operaciones definidas por el tipo son heredadas por el subtipo.
Pascal fue el primero en introducir el concepto de subtipo como subrango de cualquier tipo ordinal discreto. Diferentes subtipos de un tipo dado son considerados compatibles entre ellos y con su tipo padre, sin embargo en alguna ocasiones pueden causar errores en tiempo de ejecución.
ADA provee un constructor explícito para subtipo. Los subtipos de ADA no definen nuevos tipos, sino que los valores de todos los subtipos de un cierto tipo T son de tipo T.

Tipos genéricos

En caso de definir rutinas genéricas parametrizadas y TADs genéricos los lenguajes como ADA y C++ realizan el chequeo de tipos a través de la instanciación en tiempo de compilación, ligando los parámetros de tipo genérico a un tipo concreto.

Tad

Un ***Tipo de Datos*** se compone de:
- Un conjunto de elementos con ciertas características comunes.
- Un conjunto de operaciones para manipular dichos elementos.

Un tipo de datos es abstracto (**TAD**) cuando se determinan las *operaciones* que manipularán los elementos sin decir cual será la forma exacta de éstos o aquellos. Disponer de un TAD nos proporciona encapsulamiento, reusabilidad y abstracción.

Que debe permitir un lenguaje para definir Tads

- *Encapsulamiento*: es decir que el usuario del nuevo tipo no pueda manipular los objetos de datos del tipo, excepto por el uso de las operaciones definidas.
- *Separación de declaración e implementación*
- *Ocultamiento de datos.*

Parametrizacion de Tad

Los Tad se pueden parametrizar. Un ejemplo son los tad genericos de Ada.

ADA

Mecanismo para Tad
El mecanismo que brinda ADA para definir TADs es a través de **packages**. Mediante los packages se logra separar la definición del tipo y sus operaciones de la implementación de dichas operaciones, ya que se hacen en archivos separados. Estos package constan de dos partes: El cuerpo y la especificación.

Sintaxis

Establece reglas que definen como deben combinares las componentes básicas para formar sentencias y programas.
La sintaxis es un conjunto de reglas que definen la forma del lenguaje; definen como deben ser formadas las sentencias como una secuencia de componentes básicos llamados ‘words’. Usando esta regla podemos decir cuando una sentencia es legal o no. La sintaxis no nos dice nada del contenido (significado) de una sentencia.
Las ‘words’ nos son elementales, son construidas de caracteres pertenecientes a un alfabeto.

Por eso, la sintaxis de un lenguaje es construida por dos conjuntos de reglas: reglas léxicas y

reglas sintácticas. Las reglas léxicas especifican el conjunto de caracteres que constituyen el

alfabeto del lenguaje y como estos caracteres pueden ser combinados para formar 'words' válidos.

Características

- Ayuda al programador a escribir programas sintácticamente correctos.
- Establece reglas que sirven para que el programador se comunique con el procesador.
- Favorece a:
    - Legibilidad
    - Verificabilidad
    - Traducción
    - Entrar en la ambigüedad

Elementos

- Alfabetos o conjuntos de caracteres, teniendo en cuenta que el orden de los caracteres es lo que se utiliza en las comparaciones.
- Identificadores: cadena de letras y dígitos, que deben comenzar con una letra.
- Operadores: uniformidad de las operaciones aritméticas. (estándares)
- Comentarios.
- Palabra clave y palabra reservada
    - *Palabra clave*: tiene un significado dentro de un contexto, puede ser usada por el programador como identificador de otra entidad.
    - *Palabra reservada*: palabra clave que no puede ser usada por el programador como identificador de otra entidad.

Formas de definirla

Se necesita una descripción finita para definir un conjunto finito. Tenemos tres formas para definir la sintaxis:

- Lenguaje natural: método no formal.
- BNF: método formal.
- Diagramas sintácticos (= BNF pero más intuitivo)

Semántica

Conjunto de reglas para dar significado a los programas sintácticamente válidos. Este significado puede ser expresado mediante el mapeo de cada construcción del lenguaje en un dominio cuya semántica es conocida. Para proveer una descripción formal completa de la semántica del lenguaje, los programas sintácticamente validos son mapeados en dominios matemáticos. Hay otra manera de expresar la semántica de un lenguaje, llamada semántica operacional, que se basa en mostrar los cambios de estado en una maquina abstracta que ejecuta las construcciones del lenguaje.
La semántica define el significado de los programas sintácticamente válidos en el lenguaje. No todos los programas sintácticamente validos tienen significado, por esto la semántica separa los programas que tienen significado de los que no lo tienen, estas reglas definen la **semántica estática** (se chequea antes de la ejecución); al contrario de la **semántica dinámica** que describe el efecto de usar diferentes construcciones del lenguaje. En esos casos los programas pueden ser ejecutados solo si son correctos para la sintaxis y para la semántica estática.

Semántica axiomática: ve al programa como una máquina de estados. Las construcciones del lenguaje de programación son formalizadas describiendo como su ejecución causa un cambio de estado. Un estado es descrito por un predicado lógico de primer orden que define las propiedades de los valores de las variables de programas en este estado. El significado de la construcción del lenguaje es definido por una regla que relaciona el estado anterior y el

posterior a la ejecución. Establece precondiciones y postcondiciones para la ejecución de cada sentencia.

Semántica denotacional: asocia cada sentencia del lenguaje con una función que va desde el estado del programa antes de la ejecución al estado después de la ejecución. El estado es representado por una función que va desde el conjunto de nombres de variables a los valores.

La diferencias entre estas semánticas está en la forma de definir el estado (predicados contra funciones).

PUNTEROS

Especificación

Un tipo de datos apuntador define una clase de objetos de datos cuyos valores son las ubicaciones de otros objetos de datos. Un único objeto de dato de tipo apuntador, se podría tratar de dos maneras.

1. los apuntadores solo pueden hacer referencia a objetos de un solo tipo. Los apuntadores (ubicaciones) que son permisibles como valor del objeto de datos apuntador pueden estar restringidos a apuntar solo hacia objetos de datos del mismo tipo. Este es el enfoque que se emplea en C, Pascal y Ada donde se utilizan declaraciones de tipo y verificación estática de tipos.
   Para declarar una variable apuntador en C que puede señalar hacia cualquier objeto de datos de tipo lista usamos: Lista*P; (el * designa el tipo de P como tipo apuntador. Se debe definir por separado la estructura de tipo Lista, por ejemplo; Struct Lista{int ValorLista, SigElemento;};
2. los apuntadores pueden hacer referencia a objetos de datos de cualquier tipo. Una alternativa es permitir q un objeto de datos apuntador señale a objetos de datos de tipos variables en diferentes momentos durante la ejecución del programa, este es el enfoque que se usa en lenguajes como SmallTalk, donde los objetos de datos tienen descriptores de tipo durante la ejecución y se efectúa verificación dinámica de tipos.

La operación de creación asigna almacenamiento para un objeto de datos de tamaña fijo, y también crea un apuntador al nuevo objeto de datos que se puede guardar en un objeto de datos apuntador. En Pascal y Ada esta operación se llama *new*. En C, la función se llama *malloc* (memory-allocator). C++ simplifica C, restaurando la función *new* como asignador.
Si tomamos el ejemplo anterior y queremos en un subprograma crear datos del tipo lista, se ejecutaría P = malloc(sizeof(Lista)) , esto significa “Crear un bloque de almacenamiento para usarse como un objeto de tipo lista y guardar su valor en P”.

La operación de selección permite seguir un valor de apuntador para alcanzar el objeto de datos designado. Puesto que los apuntadores son objetos de datos ordinarios, el objeto de datos apuntador mismo también se puede seleccionar usando solo el mecanismo ordinario para selección. En C, la operación de selección que sigue a un apuntador a su objeto designado se escribe *. El operador * toma el R-valor del apuntador y lo convierte en un valor. (direcciones absolutas: un valor de apuntador se puede representar como la direcciones de memoria real del bloque de almacenamiento para el objeto de datos.)

Implementación

Un objeto de datos apuntador se representa como una localidad de almacenamiento que contiene la dirección de otra localidad de almacenamiento. La dirección es la dirección base del bloque de almacenamiento que representa el objeto de datos al que el apuntador apunta. Se usan dos representaciones de almacenamiento importantes para valores de apuntador:

1. direcciones absolutas: un valor de apuntador se puede representar como la direcciones de memoria real de bloque de almacenamiento para el objeto de datos.

(En este caso se puede asignar almacenamiento durante la operación de creación. La selección es eficiente porque el valor de apuntador mismo proporciona acceso directo al objeto de datos usando la operación de acceso a memoria. La desventaja es que la gestión de almacenamiento es mas difícil. Por q ningún objeto de datos se puede mover de la memoria si existe un apuntador a el guardado en otra parte; a menos que se cambie la dirección del apuntador mismo para reflejar la nueva posición del objeto de datos. La recuperación del

almacenamiento para objetos de datos que se han convertido en basura también es difícil, debido a que cada objeto de datos de esta clase se recupera en forma individual)

2. direcciones relativas: un valor de apuntador se puede representar como un desplazamiento respecto a la dirección base de algún bloque de almacenamiento de montículo mas grande dentro del cual el objeto de datos está asignado.

(El uso de direcciones relativas como apuntadores requiere la asignación inicial de un bloque de almacenamiento dentro del cual tiene lugar la subsiguiente asignación de objetos de datos por parte de "new". Puede haber un área por cada tipo de objeto de datos por asignar (así "new" puede asignar almacenamiento a bloques de tamaño fijo), o una sola área para todos los objetos de datos.
La selección es mas costosa que con direcciones absolutas, debido a que se debe sumar el desplazamiento a la dirección base del área para obtener la dirección absoluta. La ventaja está en poder mover el bloque del área como un todo en cualquier momento, sin invalidar ninguno de los apuntadores. Una ventaja adicional es que el bloque completo se puede tratar como un objeto de datos que se crea al entrar al subprograma, usarlo, y borrarlo al salir. No es necesario recuperar almacenamiento de objetos individuales dentro del área; se puede permitir que se convierta a basura por que el área completa se recupera como un todo)

El problema principal de la implementación de apuntadores, es la asignación de almacenamiento asociada con la operación de creación. Como esta operación se puede usar para crear objetos de datos de distintos tamaños en momentos arbitrarios durante la ejecución del programa, requiere un sistema subyacente de gestión de almacenamiento.
Los objetos de datos apuntador, generan mucha basura si se pierden todos los apuntadores a un objeto de datos creado.
Se debe tener en cuenta si se los puede destruir y así recuperar el almacenamiento para volver a usarse.

Inseguridades al usar punteros

- Algunos lenguajes requieren que los punteros sean tipados, lo que permite que el compilador haga un chequeo de tipos sobre el uso de los punteros y los elementos apuntados. Otros lenguajes tratan los punteros como objetos sin tipo, permitiendo que puedan direccionar cualquier posición de memoria.
- Alguno lenguajes soportan aritmética de punteros, permitiendo, por ejemplo, sumarle cantidades, lo que hace que apunten tantos elementos adelante como se indica, en el caso de C, como los arreglos son manejados como punteros, hacer a[i] = a*(a + i)
- El R-valor de un puntero es la dirección. Si este objeto no esta alocado el puntero no referencia nada valido, lo que es una seria inseguridad. Esto se puede producir con un puntero global que apunta a una variable local y la unidad don esta definida esta variable termina su ejecución, el puntero queda referenciando cualquier cosa. Para solucionar esto algunos lenguajes requieren que en una asignación, el alcance del objeto apuntado sea al menos tal largo como el del puntero (se chequea en ejecución).
- En otros lenguajes surge el problema de la liberación del espacio alocado, esto es, como recuperar el espacio en la heap que esta desreferenciado. Algunos lenguajes tienen recuperación de espacio automática.
- Los lenguajes que permiten que un puntero sea componente de una unión pueden causar otras inseguridades, si permiten la modificación independiente de los campos.
- Otros lenguajes directamente no soportan los punteros para evitar los problemas asociados.

Problemas derivados del uso de punteros

- referencias colgadas: puede ocurrir que un puntero referencia a una posición de memoria que ya no contiene información válida. Esto puede hacerse en pascal usando new y dispose. La implementación correcta de la solución a este problema sería llevar cuenta de los objetos dinámicamente creados y cada puntero que hace referencia a ellos, para luego actualizar todas las referencias. No existe una implementación en pascal que lo haga.
- objetos perdidos: puede ocurrir que queden espacio en memoria alocados que ya no son accesibles desde el programa, perdiendo la información guardada y además desaprovechando memoria. Esto puede ocurrir fácilmente en pascal, ya que solo es necesario efectuar dos new consecutivos sobre un puntero. Observar que hasta cierto

punto esto es imposible de solucionar, ya que el compilador no puede saber si los datos son útiles. Existe la alternativa del dispose, pero esto requiere una elaboración cuidadosa de los programas (cada vez que reutilizamos un puntero, deberíamos verificar que la memoria alocada con anterioridad no tiene referencias y luego decidir que hacer) una alternativa seria la implementación de un garbage collector.

Descripción y comparación de dispose y garbage collector
La cantidad de memoria libre asignada durante la ejecución de un programa puede ser muy grande. No obstante, tan pronto con una zona de la memoria libre deja de ser referenciada, se debería devolver y posteriormente asignarse a nuevas variables de memoria libre.
Para hacer esto posible, muchas implementaciones de pascal, suministran el procedimiento standard *Dispose*, que libera explícitamente memoria, pero el programador puede pedir la liberación de una variable de memoria libre cuando todavía hay punteros a ella. Por lo tanto se pueden crear referencias sueltas, a menos que se compruebe en tiempo de ejecución la utilización incorrecta del dispose. Tanto Algol 68 como Simula 67 evitan este problema al no permitir la liberación explicita de variables de memoria libre. En cambio, delegan explícitamente en un recolector de residuos (garbage cllector) que reclama automáticamente la memoria libre no autorizada.

Necesidad de usar punteros
Permite la construcción de cualquier estructura para vincular entre si los objetos de datos componentes según se desee.

Prejuicios: genera lenguajes menos comprensibles e inseguros por las siguientes razones

- puede conducir a violación de tipos (por caso 2 de las especificaciones) en algunos lenguajes como PL1. Solución → Tratar de no usarlo, según las especificaciones del lenguaje.
- Referencias colgadas. Solución → Dispose o Garbage collector.
- Algunos lenguajes no proveen inicialización por defecto de los punteros (provocando acceso incontrolado a la memoria). Solución → recordar inicializarlos a mano

Pasaje de Parámetros

Ventajas del uso de parámetros

- Más flexible.
- Diferentes datos en cada llamada.
- Legibilidad.
- Modificabilidad.

Lista de Parámetros
Parámetros Reales: Parámetros que se codifican en la invocación del subprograma.
Parámetros Formales: Parámetros declarados en la especificación del subprograma.
Contiene los nombres y los datos compartidos.

Evaluación de los parámetros Reales y ligadura de los parámetros Formales
Evaluación: En el momento de la invocación primero se evalúan los parámetros reales y luego se hacen las ligaduras antes de transferir el control a la unidad llamada.
Ligadura: * Posicional: Posición que ocupa en la lista.
* Palabra clave o nombre: se corresponden por el nombre.

Clases de Parámetros

Semánticamente solo pueden ser:

- IN: Parámetro formal recibe el dato desde el parámetro formal.
- OUT: Parámetro formal envía el dato al parámetro real.
- IN/OUT: Parámetro formal recibe el dato del parámetro real y el parámetro formal envía el dato al parámetro real.

PARAMETRO IN

- **Por Valor:**

  El valor del parámetro real se usa para inicializar el correspondiente parámetro real al invocar la unidad.
  Se transfiere el dato real.
  En este caso el parámetro formal actúa como una variable local de la unidad llamada.
  Desventaja: Consume tiempo y almacenamiento.
  Ventaja: Protege los datos de la unidad llamadora.

- **Por Valor Constante:**

  No indica si realiza o no la copia, lo que establece es que la implementación debe verificar que el

  parámetro real no sea modificado.

  Desventaja: Requiere realizar mas trabajo para implementar los controles.
  Ventaja: Protege los datos de la unidad llamadora.

PARAMETRO OUT

- **Por Resultado:**

  El valor del parámetro formal se copia al parámetro real al terminar de ejecutar la unidad llamada.

  El parámetro formal es una variable local, sin valor inicial.
  Desventaja: Consume tiempo y espacio. Se debe tener encuentra el momento en que se evalúa el parámetro real.
  Ventaja: Protege los datos de la unidad llamadora.

- **Por Resultado de Funciones:**

  El resultado de una función puede devolver con el return o en el nombre de la función.

PARAMETRO IN/OUT

- **Por valor- resultado:**

Copia a la entrada y a la salida de la activación de la unidad llamadora.

El parámetro formal es una variable local que recibe una copia a la entrada del contenido del parámetro real y a la salida el parámetro real recibe una copia de lo que tiene el parámetro formal.
Cada referencia al parámetro formal es una referencia al parámetro local.
Tiene la ventaja y desventaja de ambos.

- **Por Referencia:**

Se transfiere la dirección del parámetro real al parámetro formal.

El parámetro formal será una variable local a la unidad llamadora que contiene la dirección en el ambiente no local.
Cada referencia al parámetro formal será a un ambiente no local. El parámetro formal es compartido por la unidad llamada.
Desventaja: El acceso del dato es mas lento por la indirección.
Se pueden generar alias.
Se puede modificar el parámetro inadvertidamente.
Ventaja: Eficiencia en tiempo y espacio.

- **Por Nombre:**

El parámetro formal es sustituido textualmente por el parámetro real.
La ligadura de valor se difiere hasta el momento en que se lo utiliza.
Si el dato a compartir es:
» Un único valor se comporta exactamente igual que el pasaje por referencia.

» Si es una constante es equivalente a por valor.

» Si es un elemento de un arreglo puede cambiar el suscripto entre las distintas referencias

» Si es una expresión se evalúa cada vez.

Para implementarlo se utilizan los thunks. Cada aparición del parámetro formal se reemplaza en el cuerpo de la unidad llamado por una invocación a un thunks que en el momento de la ejecución activara al procedimiento que evaluará el parámetro real en el ambiente apropiado.

Es un método más flexible pero debe evaluarse cada vez que se lo usa.

Es difícil de implementar y genera soluciones confusas para el lector y el escritor.

Ejemplos sobre pasaje de parámetros

Ejemplo donde se pone de manifiesto la diferencia entre el pasaje por referencia y por nombre.

```
Procedure inter (x, y: integer)
        Var
                Temp: integer;
Begin
        temp:=x;
        x:= y
        y:=temp;
End;
```

Si en inter los parámetros se pasan **por referencia** se realiza un intercambio de variables correctamente ya que se intercambian los valores de las referencias de x e y.

En cambio si los parámetros son pasados **por nombre** ante la invocación a inter (i, a(i)) la ejecución quedaría:

```
        Temp:=i;
1)      i := a(i);
2)      a(i) ;= temp;
```

pero aquí a(i) de 1) no será el mismo que a(i) de 2), pues como el valor de i ya fue modificado queda la posición del arreglo sin modificar y se modifica otra. Luego entonces no se intercambian los valores. Esto se da pues los parámetros pasados son un índice y un elemento de un arreglo.

Ejemplo de pasaje de parámetro por valor-resultado.

```
proc (f1, f2)
{
        f1 := f1 + 1;
        f2 := f2 +11;
}
```

- **Con igual efecto que por referencia.**

  Call proc (a,b)

  En este caso ya sea **por valor-resultado** o **por referencia** a a se le va a sumar 1 y a b se le sumará 11. En el pasaje por **valor-resultado** los valores se modificarán al terminar el procedimiento y en el caso del pasaje **por referencia** se modifica en el momento ya que se pasan las direcciones de las variables.

- **Con distinto efecto que por referencia.**

  Call proc(a,a)

  Si el pasaje es **por valor-resultado**. En este caso si a= 0, entonces al entrar a proc en f1 y en f2 se copiarán los valores de a. Luego f1 = 1 y f2 = 12. Al salir de proc a tendrá el valor de 1 u 11 (ver en que orden se copian los parámetros).

  Si el pasaje es **por referencia** a al terminar el procedimiento tendría el valor 12.

Diferencias y puntos de contacto entre el pasaje de parámetros por nombre y el de procedimientos.

Diferencias

1) Momento de ligadura: en el pasaje de parámetro por nombre la ligadura se difiere hasta el momento en que se la utiliza. En el pasaje por procedimiento, la ligadura se realiza en el momento de la invocación.
2) Modo: el pasaje de parámetro por nombre es modo IN/OUT. En parámetro rutina depende del tipo de la rutina (procedimiento o función) y de los tipos de parámetros que estos tienen.
3) Utilización de procedimientos adicionales: los parámetros por nombre utilizan los thunks y los parámetros rutina no.
4) Información requerida: en el pasaje por nombre se pasa la dirección de memoria del parámetro. En cambio en el caso de parámetros rutina se pasan los parámetros de la rutina, el ambiente de referencia y el encabezamiento de la rutina y una referencia al segmento de código del parámetro real.

Puntos de contacto

En ambos casos se necesitan referencias a otras zonas de memoria, debido al uso de thunks y por las referencias al registro de activación y al segmento de código. Los dos realizan una llamada a un subprograma.

Comparación entre el llamado a una rutina con una referencia a un elemento del un arreglo.

Supongamos que tenemos la siguiente sintaxis: A(i)

- Si interpretamos A(i) como la llamada a una rutina, i representaría el parámetro de la misma y A el nombre de la rutina.
- También se puede tomar como que A(i) es la referencia del elemento en la posición i del arreglo A.

El problema de utilizar indistintamente paréntesis para los parámetros de una rutina y la posición de un elemento de un arreglo hace que tengamos una misma sintaxis con diferente semántica y esto atento contra la simplicidad de los lenguajes.
Por esta razón algunos lenguajes utilizan el uso de [] para referirse a posiciones de arreglos y () para los parámetros de los módulos.

Estructuras de control

Son el medio por el cual los programadores pueden determinar el flujo de ejecución entre los componentes de un programa
Hay dos tipos de niveles:

- A NIVEL DE SENTENCIAS:
  Cuando el flujo de control se pasa entre unidades.
  Intervienen los pasajes de parámetros.

- A NIVEL DE UNIDAD
  - SECUENCIA
  - SELECCIÓN
  - ITERACION

Secuencia
Es el flujo de control más simple.
Indica la ejecución de una sentencia a continuación de otra.

Delimitadores

- ⇨ Pascal tiene como delimitador al punto y coma (;).
- ⇨ Smalltalk tiene por delimitados al punto (.).
- ⇨ Fortran permite solo una instrucción por línea. En consecuencia el delimitador seria el retorno de carro o enter.

Sentencias compuestas

La sentencia compuesta es una abstracción en una sentencia de una colección de sentencias. Generalmente están compuestas por un par de delimitadores y una secuencia, y no todos los lenguajes la implementan de la misma manera.

- ⇨ Pascal, Algol : <instrucción _ compuesta>= Begin <secuencia -instrucción> end
- ⇨ C, C++: <instrucción _ compuesta>= {secuencias _ instrucción}
- ⇨ Ada (bloque): <instrucción _ compuesta> = declare <sec_declaracion> Begin <secuencia_instrucción> end;

Atributos de un buen lenguaje:

Los lenguajes de programación son herramientas que nos permiten producir software de calidad; para ello, deben cumplir las siguientes características:

- Fáciles de escribir
- Legibles
- Confiables
- Mantenibles
- Eficientes

Faciles de Escribir

➔ **Simplicidad:** el lenguaje proporciona un marco conceptual para pensar algoritmos y expresar dichos algoritmos con el nivel de detalle adecuado. El lenguaje debe ser una ayuda al programador (incluso antes de comenzar a codificar) proporcionando un conjunto de conceptos claro, simple y unificado. Para ello, es deseable contar con un número mínimo de conceptos distintos cuyas reglas de combinación sean tan sencillas y regulares como sea posible.
La sintaxis de un lenguaje afecta la facilidad con la que un programa se puede escribir, poner a prueba y mas tarde entender y modificar. La legibilidad es muy importante, muchos programas contienen construcciones sintácticas que favorecen una lectura errónea, al hacer que dos enunciados casi idénticos signifiquen cosas radicalmente distintas. Es decir, las diferencias semánticas deberán reflejarse en la sintaxis del lenguaje.

➔ **Expresividad:** El programador debe poder expresar sus intenciones. En ocasiones, demasiada expresividad puede implicar falta de seguridad. De hecho, algunos sistemas limitan la expresividad para mejorar la fiabilidad de los programas (por ejemplo, la aritmética de punteros no es permitida en algunos lenguajes).

➔ **Ortogonalidad:** Dos características de un lenguaje son ortogonales si pueden ser comprendidas y combinadas de forma independiente. Cuando las características del lenguaje son ortogonales, el lenguaje es más sencillo de comprender, porque hay menos situaciones

excepcionales a memorizar. La ortogonalidad ofrece la posibilidad de combinar características de todas las formas posibles (sin excepciones). La falta de ortogonalidad puede suponer la enumeración de situaciones excepcionales o la aparición de incoherencias. Un ejemplo de falta de ortogonalidad es la limitación que impone Pascal para que una función devuelva determinados tipos de valores.

➔ **Abstracción**: El lenguaje debe evitar forzar a los programadores a tener que enunciar algo más de una vez. El lenguaje debe permitir al programador la identificación de patrones repetitivos y automatizar tareas mecánicas, tediosas o susceptibles de cometer errores. Ejemplos de técnicas de abstracción son los procedimientos y funciones, la genericidad, los lenguajes de patrones de diseño, etc. La modularización es un concepto clave para manejar la complejidad.

La expresividad y la ortogonalidad trabajan en forma ordenada y obtienen software autodocumentado.
La abstracción permite descomponer gradualmente el problema.
Todo junto permite que sea mas fácil escribir en ese lenguaje.

Legibles

Cuanto mas fácil de escribir un programa, mas legible. La legibilidad es un factor de influencia en la facilidad de modificación y mantenimiento de un programa.
➔ **Sintaxis (forma):** debe proveer la posibilidad de incluir comentarios; tener en cuenta que la falta de precisión sintáctica confunde el significado. No debe permitir ambigüedades entre la semántica y la sintaxis. Por ejemplo: A(1) llama a la rutina A con el parámetro 1 o es la primer posición del arreglo A?
➔ **Semántica:** Hay aspectos semánticos que influyen en la legibilidad. Ejemplo: Alias (pasar un parámetro por referencia, nombres diferentes para un mismo lugar de memoria), efectos laterales (y + f(X) + y es distinto de 2Y + f(X)) y estructuras de datos (el tipo boolean en algunas lados se representa false se representa con 0 y en otros con 'False', algo similar sucede para true).
➔ **Definiciones:** cuanto mas formal y precisa sea la definición del lenguaje mejor.

Lenguajes que exijan definición explicita de variables , sino surgen errores en el código (escribir mal los nombres es de los problemas mas comunes). Aislar estructuras de control es la mejor opción para no tener ambigüedades (ejemplo: el ";" en pascal). Un código bien indentado es mas legible que uno que no lo esta.

Confiables

➔ **Chequeo de tipos:** tanto estático y dinámico.
➔ **Excepciones:** Los lenguajes robustos, permiten el manejo de eventos indeseados, estos deben poder identificarse y se debe poder programar una rutina apropiada para responder a tal evento.

Mantenibles

➔ Las consideraciones económicas han reducido la posibilidad de desechar el software existente. Este tiene que se modificado para satisfacer los nuevos requerimientos.

Eficientes

➔ El programador debe poder expresar algoritmos suficientemente eficientes o el lenguaje debe incorporar técnicas de optimización de los programas escritos en él. El significado de eficiencia ha cambiado, ya no es mas medida en función de la velocidad de ejecución y el espacio, sino también por la productividad, la reusabilidad y su implementación.

Programación Orientada a Objetos

La tecnología orientada a objetos se define como una metodología de diseño de software que modela las características de objetos reales o abstractos por medio del uso de clases y objetos. Hoy en día, la orientación a objetos es fundamental en el desarrollo de software, sin embargo, esta tecnología no es nueva, sus orígenes se remontan a la década de los años sesenta. De hecho Simula, uno de los lenguajes de programación orientados a objetos más antiguos, fue desarrollado en 1967.

Bases sobre las cuales se fundamenta la programación orientada a objetos.

Los elementos más importantes que deben tener los objetos de software, para cumplir con el paradigma de orientación a objetos son:

- Abstracción.
- Modularidad.
- Encapsulamiento.
- Herencia.
- Polimorfismo.

Abstracción.
Por medio de la abstracción definimos las características esenciales de un objeto del mundo real, los atributos y comportamientos que lo definen como tal, para después modelarlo en un objeto de software.
En el proceso de abstracción no debemos preocuparnos por la implementación de cada método o atributo, solamente debemos definirlo de forma general.
Visualizar las entidades que deseamos trasladar a nuestros programas, en términos abstractos, resulta de gran utilidad para un diseño optimo de nuestro software, ya que nos permite comprender más fácilmente la programación requerida.
En la tecnología orientada a objetos la herramienta principal para soportar la abstracción es la clase. Esta clase es un molde o modelo en donde se especifican las características que definen a un objeto de manera general, a partir de una clase pedemos definir objetos particulares (instancias).

Modularidad.
La modularidad, nos permite poder modificar las características de la clase que definen a un objeto, de forma independiente de las demás clases en la aplicación. En otras palabras, si nuestra aplicación puede dividirse en módulos separados, normalmente clases, y estos módulos pueden compilarse y modificarse sin afectar a los demás, entonces dicha aplicación ha sido implementada en un lenguaje de programación que soporta la modularidad. La tecnología orientada a objetos nos brinda esta propiedad para hacer uso de ella en el software que desarrollemos.

Encapsulamiento u Ocultamiento.
El encapsulamiento es la propiedad de la orientación a objetos que nos permite asegurar que la información de un objeto le es desconocida a los demás objetos en la aplicación. Esto se debe principalmente a que no necesitamos, dentro de la programación orientada a objetos, saber como esta instrumentado un objeto para que este interactúe con los demás objetos.
En otras palabras, con el encapsulamiento ganamos modularidad, y además protegemos a los objetos de ser manipulados de forma inadecuada por objetos externos.

Herencia
La tecnología orientada a objetos nos permite definir jerarquías entre clases y jerarquías entre objetos. Las dos jerarquías más importantes que existen son la jerarquía “es un” que precisa la generalización y especificación entre clases y la jerarquía “es parte de” en la cual se delimita la agregación de objetos.

Comúnmente, a la jerarquía “es un” se le conoce como herencia. La herencia simple es la propiedad que nos permite definir una clase nueva en términos de una clase ya existente. Existirán casos en los cuales se necesite definir una clase a partir de dos o más clases preexistentes, en este caso estaremos hablando de herencia múltiple.
En cuanto a la jerarquía de agregación, también conocida como inclusión, podemos decir que se trata del agrupamiento lógico de objetos relacionados entre si dentro de una clase.
Estricta: una subclase no puede redefinir en forma contradictoria atributos definidos en la super-clase.
No estricta: si una sub-clase no quiere heredar algún atributo de la super-clase, puede redefinir ese atributo.

Polimorfismo
El polimorfismo es la propiedad por la cual una entidad puede tomar diferentes formas. Generalmente está entidad es una clase, y la forma en que se consigue que tome diferentes formas es por medio de nombrar a los métodos de dicha clase con un mismo nombre pero con diferentes implementaciones.

Programación funcional

La programación funcional es aquélla en la que en lugar de construir el programa utilizando instrucciones se construye utilizando funciones.
El esquema de un programa en este tipo de lenguajes se puede expresar de la siguiente forma:
PROGRAMA = FUNCIONES + ESTRUCTURAS DE DATOS
Una función es una regla de asociación o correspondencia entre los elementos de un conjunto, que se llama el dominio, y los elementos de otro conjunto, que se llama el rango. La función siempre devuelve un valor.
Un lenguaje funcional tiene las siguientes componentes:
- Un conjunto de funciones primitivas.
- Un conjunto de formas funcionales.
- La operación aplicación.
- Un conjunto de objetos o datos.

La **recursividad** consiste en el proceso de resolver un problema reduciéndolo a uno o más subproblemas que son idénticos en su estructura al problema original y más simples de resolver. Es una manera elegante, intuitiva y concisa de plantear una solución, y es uno de los pilares de la programación funcional. Pero no quiere decir que sea un sistema eficiente. En realidad, la recursividad es cara y exige un mayor procesamiento.
Otro pilar de la programación funcional es la composición funcional en la cuál, el resultado del cálculo de una función es la entrada a la siguiente, y así sucesivamente hasta que la composición resuelve el problema.
Los lenguajes de la programación funcional necesitan de un proceso de traducción para que sean comprensibles para la computadora, y deterministas, el resultado nunca cambia si las entradas de información tampoco lo hacen.
El campo científico es el mejor ámbito para el uso de estos lenguajes debido a su alta componente matemática. Además, su facilidad de búsqueda les hace ser los más utilizados en Inteligencia Artificial.

Características principales

1) Transparencia referencial: el valor de las expresiones depende solo de los elementos que la constituyen.
2) Tipos: un tipo es un conjunto de valores con propiedades comunes. Cada tipo tiene asociadas ciertas operaciones que no son significativas para otros tipos. A toda expresión bien formada se le puede asignar un tipo, este tipo se puede deducir de las partes constituyentes de la expresión (mediante inferencia de tipos, chequeo de tipos, según sistema de tipado fuerte).
   Tipos polimorficos: permite que el tipo de una funcion pueda ser instanciado de diferentes maneras en diferentes usos.

3) Alto orden: las funciones son valores, al igual que los números, las tuplas, etc. Una función de alto orden es aquella que recibe otra función como argumento o la retorna como resultado.
4) Currificación: correspondencia entre cada función de múltiples parámetros y una de alto orden que retorna una función intermedia que completa el trabajo.

Ventajas

Usualmente mayor eficiencia. Mejores condiciones de terminación. No hay necesidad de estructuras intermedias al componer funciones. Manipulación de estructuras de datos infinitas. Manipulación de computaciones infinitas.

Desventajas

Es difícil calcular el coste de ejecución pues depende del contexto en el que se usa.

- Las funciones se definen en un script. Los script son definiciones que dan los usuarios y esas definiciones generalmente son funciones.
- Las variables en los lenguajes funcionales se ven como en matemáticas, no hay asignación, las variables no varían. Una maquina funcional actúa como una calculadora. Las variables solo pueden tener asignado un valor a lo largo de la ejecución del programa, lo cual implica que no puede existir asignación destructiva. Debido a esto cobro especial importancia el uso de anidamiento y la recursividad. Los lenguajes funcionales no cuentan con variables globales.