Apuntes de Programación y estructuras de datos. Control de datos

Nikos Mylonakis, Fernando Orejas y Ana Cristina Zoltan

nicos@lsi.upc.edu

Dept. Llenguatges i Sistemes Informátics Universitat Politécnica de Catalunya

Barcelona

Contenido

- Relación identificadores con su entidad denotada
  - Ambito, visibilidad y vida de los identificadores
  - Lenguajes con estructura de bloques
  - Módulos
  - Lenguajes orientados a objetos
- Paso de parámetros

- Un identificador puede denotar constantes, tipos, variables, parámetros, subprogramas, módulos, etc
- En general un identificador puede denotar más de una entidad en un programa
- Esto es debido a que los identificadores no son visibles en todas las partes de un programa
- Un identificador es visible en una parte del programa si se puede utilizar en esa parte
- Ejemplo: Una variable es visible en un subprobgrama si está declarada como variable local o parámetro

- El ámbito (scope) de un identificador es la zona del programa donde es visible
- El ámbito puede ser estático o dinámico
- El ámbito estático se define en las partes del programa escrito
- El ámbito dinámico se define en el flujo de ejecución del programa y sus subprogramas

- La vida de un identificador es el intervalo de tiempo que va desde que nace hasta que desaparece
- La entidad denotada por un identificador puede no ser visible en una parte del programa pero estar viva
- Ejemplo: Subprogramas imbricados de Pascal con variables locales con el mismo nombre
- Dado un identificador determinar a qué entidad nos referimos depende de las reglas de ámbito y visibilidad del tipo de lenguaje

Lenguajes con estructura de bloques

- Ejemplos de estos lenguajes son Algol60, Pascal, Ada, C, etc
- Permiten definir subrprogramas y bloques anidados
- Regla de visibilidad estática: Toda entidad es visible en el bloque que se declara y en sus bloques internos
- En caso de ambigüedad se utiliza el identificador definido más cerca entre los bloques que contienen al bloque donde se utiliza el identificador

- Lo que hace el compilador de estos lenguajes dado una referencia a un identificador es buscarlo en la estructura de bloques del más anidado al más externo
- Los lenguajes que usan la regla de visibilidad anterior no tienen visibilidad dinámica

- Si la regla de visibilidad fuese dinámica las referencias se determinan en tiempo de ejecución
- La búsqueda de la declaración de la referencia se realiza por la secuencia de llamadas y no por la estructura estática de los bloques
- Ejemplos de lenguajes con reglas de visibilidad dinámica son Lisp y Java (éste último sólo para subprogramas y con limitaciones(methods) y no variables)

Módulos

- Los lenguajes de programación tipo Pascal permitían el uso de variables globales para pasar información entre bloques
- Esto se considera que dificulta la legibilidad
- Los lenguajes modulares evitaron esto
- Un lenguaje modular tiene estructura de bloques normalmente restringida y unidades llamadas módulos (Las clases de Java son módulos)
- Un módulo sirve para encapsular tipos y subprogramas

- Todo tipo o subprograma declarado en un módulo es visible dentro de éste
- Además un módulo puede importar o usar tipos y subprogramas de otros módulos
- Nada impide que dentro de un módulo haya estructura de bloques

- Los lenguajes con módulos también permiten restringir el acceso a ciertas declaraciones del módulo (en Java private)
- En caso de ambigüedad al referenciar a un identificador primero se intenta resolver entre los identificadores definidos en el módulo
- Si no existe declaración del identificador referenciado en el módulo se busca en los módulos importados.
- Si hay varias declaraciones en módulos diferentes se ha de referenciar precediendo el nombre del módulo (Ej: M.x) (En algunos LP es obligatorio)

Lenguajes orientados a objetos

- Los LOO tienen estructura de bloques, módulos (clases) y además herencia con subclases
- Nosotros utilizaremos una notación similar a la de Java o C++
- Veamos como ejemplo la definición de lista de enteros usando esta nueva notación

clase $Nodo$

$valor : entero;$

$sig :\uparrow Nodo;$

fclase

clase $Lista\_enteros$

$l :\uparrow Nodo$

accion $lista\_vacia()...$

accion $insertar\_entero(n : entero)...$

accion $borrar\_entero(n : entero)...$

accion $recorrido()$

$\vdots$

fclase

- Veamos ahora como funciona la herencia
- Para definir el tipo Cola_enteros lo podemos hacer como subclase de lista de enteros
-

$$\begin{array}{l}\textbf{clase}\ Cola\_enteros\\ \quad \textbf{es subclase de}\ Lista\_enteros\\ \quad cola : Lista\_enteros\\ \quad \textbf{accion}\ borrar\_entero(n : entero)...\\ \textbf{fclase}\end{array}$$

- Sean las variables $l : Lista\_enteros$ y $c : Cola\_enteros$
- Si hacemos $l.insertar\_entero(5)$ y $l.borrar\_entero(6)$ se ejecutan las acciones definidas en $Lista\_enteros$
- Si hacemos $c.insertar\_entero(5)$ se ejecuta la acción de $Lista\_entero$ y al hacer $c.borrar\_elemento(6)$ se ejecuta la acción de $Cola\_enteros$.

- Ejemplo 2: Supongamos que tenemos una empresa con 3 clases de trabajadores: directivos, ejecutivos y administrativos
- Queremos tener una base de datos de los trabajadores y calcular el salario y los impuestos de todos los trabajadores
- El salario de los directivos depende de los beneficios de la empresa, los ejecutivos tienen gratificaciones eventuales y los administrativos tienen sueldo fijo

- Para ello definiríamos los tipos Lista_Trabajadores, Trabajador y los tipos directivos, ejecutivos y administrativos como subclase de la clase trabajador
- Las subclases de trabajadores tendrían una operación específica para el cálculo del salario
- La acción Calcular_salarios recorrería la lista de trabajadores y para cada trabajador realizaría la llamada a la acción salarios

- En tiempo de compilación no sabemos qué acción se ejecutará.
- En tiempo de ejecución en función de la subclase del trabajador se ejecutará una acción salario u otra

Regla de visibilidad con subclases

- Si una operación f es redefinida en una subclase de una clase y es llamada por un objeto se utilizará la función f de la subclase más cercana al tipo del objeto en la jerarquía de subclases
- Si SC es subclase de C y $oc : C$, $osc : SC$, al realizar la asignación $oc := osc$ (la asignación $osc := oc$ no se puede realizar), esto tiene las siguientes implicaciones:

- oc continúa teniendo el tipo C.
- Sólo los atributos de la clase C se actualizan con los valores de osc en oc.
- También sólo podemos utilizar las operaciones de C con parámetro implícito oc y por asociación dinámica de los métodos redefinidos, se utilizarán las operaciones redefinidas de SC.

- Además existen mecanismos para ocultar las variables y los métodos definidos en una clase
- En Java podemos escribir

$$\begin{array}{l}\mathbf{class}\ A\{\\ \quad \mathbf{public} \mid \mathbf{protected} \mid \mathbf{private}\ int\ p;\\ \}\end{array}$$

- Si no ponemos nada int p lo puede ver todo el package
- Si ponemos public int p la variable es visible desde cualquier parte del programa
- Si ponemos protected int p la puede ver todo el package y cualquier subclase de A
- Si ponemos private int p sólo la puede ver A

- En C++ las declaraciones se dividen en public, private y protected
- Las declaraciones public son visibles por funciones miembros de cualquier clase.
- Las declaraciones protected son visibles por funciones miembros y amigas de la misma clase o de sus subclases
- Las declaraciones private son visibles por funciones miembros y amigas de la misma clase

La relación de subclase se define como $\mathbf{class}\, SC\ :\ \mathbf{public} \,|\, \mathbf{protected} \,|\, \mathbf{private}\, C, \ldots \{\ldots\}$ donde C es la clase de la cual hereda SC.

- Si no se especifica el tipo de acceso es como si fuera privado que definiremos a continuación.
- Acceso public: Todos los miembros public y protected de la clase C son heredados conservando su acceso, mientras que los miembros private no son accesibles.

- Acceso private:Todos los miembros public y protected de la clase C son heredados con acceso private mientras que los miembros private no son accesibles.
- Acceso protected: Todos los miembros public y protected de la clase C son heredados con acceso protected mientras que los miembros private no son accesibles.

Herencia múltiple

- En lenguajes como C++ y Lisp se permite la herencia múltiple
- Ejemplo:

$$\begin{array}{lll} \textbf{clase}\, C1 & \textbf{clase}\, C2 & \textbf{clase}\, C3 \\ \vdots & \vdots & \textbf{es subclase de}\, C1, C2 \\ \textbf{fclase} & \textbf{fclase} & \textbf{fclase} \end{array}$$

- La herencia múltiple tiene problemas adicionales. Ejemplo: Si en C3 se referencia f pero no esta declarada en C3 pero sí en C1 y C2 qué f escogemos. En Java no existe herencia múltiple por seguridad

Paso de parámetros

- Un subprograma contiene un identificador para ser llamado y parámetros formales
- La llamada de un subprograma contiene el identificador y parámetros reales, actuales o argumentos
- Mecanismo que permite pasar datos entre unidades de programas (subprogramas, módulos, procesos, etc)
- Los subprogramas pueden tener parámetros de entrada, salida y entrada/salida

- Los parámetros de entrada sirven para transferir información del subprograma principal al subprograma llamado
- Los parámetros de salida sirven para transferir información del subprograma llamado al subprograma principal después de ejecutar el subprograma llamado
- Los parámetros de entrada/salida sirven para transferir información en ambos sentidos

Implementación de paso de parámetros

- Recordamos que en subprogramas tenemos la definición ($\mathbf{accion}\, P(\ldots\, x : entero)$) y la llamada $P(exp)$
- Las diferentes implementaciones de pasos de parámetros difieren en lo que denota exp.
- Si exp denota el valor obtenido al evaluar la expresion el paso de parámetros es por **valor**. Si además se permite devolver resultados es por **valor-resultado**.
- Este mecanismo requiere copia de valores de parámetro real o argumento a parámetro formal y viceversa si es por valor-resultado. Ineficiente si los valores son muy grandes

- Cuando exp denota una dirección de memoria o referencia el paso de parámetros es por **referencia**
- En este caso el parámetro formal pasa a ser un alias de la dirección de memoria del parámetro real
- Necesariamente exp ha de ser una expresión que denota una variable

- Cuando exp denota la expresión sin evaluar el paso de parámetros es por **nombre** (anecdótico)
- Ejemplo

$Accion\ principal$
$\mathbf{var}\ i : entero$
$T : \mathbf{tabla}\ [1..100]\ \mathbf{de\ entero}$
$\mathbf{fvar}$
$inic(T[i])$

$\mathbf{accion}\ inic(x : entero)$
$\mathbf{Para}\ i := 1\ \mathbf{hasta}\ 100\ \mathbf{hacer}$
$\quad x := 0$
$\mathbf{fpara}$
$\mathbf{faccion}$

- El efecto de la ejecución de este programa es que toda la tabla se inicializa a 0

En diferentes LP el tipo de paso de parámetros que tenemos es el siguiente

- **Pascal:** Paso por valor sólo se indica el parámetro y el paso por referencia requiere la palabra clave **var** delante del parámetro
- **Ada:** Tiene tres clases diferenciadas por las palabras claves **in**, **out** e **inout**. Cada compilador las implementa como quiere

- **Fortran** Lo decide el compilador. Generalmente los tipos estructurados (ej. tablas) se pasan por referencia
- **Java** Siempre es por valor. Esto implica que los tipos básicos se pasan por valor y con el resto hay que tener en cuenta que lo que se realiza es una copia de la referencia que denota el parámetro real al parámetro formal
- **C++** Paso por valor y por referencia