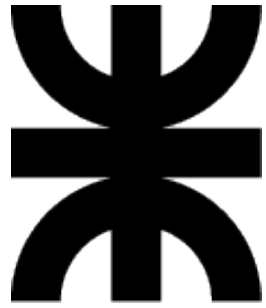
Informática II

Clases de almacenamiento, reglas de alcance, y calificadores de variables

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Clases de almacenamiento

- Identificadores: para nombres de variables (y funciones)

Clases de almacenamiento

- Identificadores: para nombres de variables (y funciones)
- Atributos de una variable: nombre, tipo (tamaño) y valor

Clases de almacenamiento

- Identificadores: para nombres de variables (y funciones)
- Atributos de una variable: nombre, tipo (tamaño) y valor

Cada identificador tiene otro atributo llamados clase de almacenamiento, el cual define:

Clases de almacenamiento

- Identificadores: para nombres de variables (y funciones)
- Atributos de una variable: nombre, tipo (tamaño) y valor

Cada identificador tiene otro atributo llamados clase de almacenamiento, el cual define:

1. Duración de almacenamiento: período durante el cual el identificador existe en memoria

Clases de almacenamiento

- Identificadores: para nombres de variables (y funciones)
- Atributos de una variable: nombre, tipo (tamaño) y valor

Cada identificador tiene otro atributo llamados clase de almacenamiento, el cual define:

1. Duración de almacenamiento: período durante el cual el identificador existe en memoria
2. Alcance: desde donde se puede referenciar al identificador (reglas de alcance)

Clases de almacenamiento

- Identificadores: para nombres de variables (y funciones)
- Atributos de una variable: nombre, tipo (tamaño) y valor

Cada identificador tiene otro atributo llamados clase de almacenamiento, el cual define:

1. Duración de almacenamiento: período durante el cual el identificador existe en memoria
2. Alcance: desde donde se puede referenciar al identificador (reglas de alcance)
3. Enlace/vinculación: para programas de varios archivos fuentes

Clases de almacenamiento

- Identificadores: para nombres de variables (y funciones)
- Atributos de una variable: nombre, tipo (tamaño) y valor

Cada identificador tiene otro atributo llamados clase de almacenamiento, el cual define:

1. Duración de almacenamiento: período durante el cual el identificador existe en memoria
2. Alcance: desde donde se puede referenciar al identificador (reglas de alcance)
3. Enlace/vinculación: para programas de varios archivos fuentes

C cuenta con 4 clases de almacenamiento

- `auto`
- `register`
- `static`
- `extern`

Clases de almacenamiento – Persistencia/duración

Persistencia automática vs. persistencia estática

Clases de almacenamiento – Persistencia/duración

Persistencia automática vs. persistencia estática

Persistencia automática: se crean cuando se entra en el ámbito de un bloque, existen solo en dicho bloque y se destruyen cuando se sale. (solo para variables)

Clases de almacenamiento – Persistencia/duración

Persistencia automática vs. persistencia estática

Persistencia automática: se crean cuando se entra en el ámbito de un bloque, existen solo en dicho bloque y se destruyen cuando se sale. (solo para variables)

Las palabras reservadas `auto` y `register` se usan en variables de persistencia automática.

Clases de almacenamiento – Persistencia/duración

Persistencia automática vs. persistencia estática

Persistencia automática: se crean cuando se entra en el ámbito de un bloque, existen solo en dicho bloque y se destruyen cuando se sale. (solo para variables)

Las palabras reservadas `auto` y `register` se usan en variables de persistencia automática.

Persistencia estática: existen a partir de que el programa inicia su ejecución. Esto no significa que puedan ser utilizados (alcance). (también para funciones)

Clases de almacenamiento – Persistencia/duración

Persistencia automática vs. persistencia estática

Persistencia automática: se crean cuando se entra en el ámbito de un bloque, existen solo en dicho bloque y se destruyen cuando se sale. (solo para variables)

Las palabras reservadas `auto` y `register` se usan en variables de persistencia automática.

Persistencia estática: existen a partir de que el programa inicia su ejecución. Esto no significa que puedan ser utilizados (alcance). (también para funciones)

Las palabras reservadas `static` y `extern` se usan en variables de persistencia estática.

Persistencia automática

- `auto`: Variables locales de una función, ya sean declaradas en la lista de parámetros o en el cuerpo de la función.

```
auto float x, y;
```

  Las variables locales tienen persistencia automática por defecto.

Persistencia automática

- `auto`: Variables locales de una función, ya sean declaradas en la lista de parámetros o en el cuerpo de la función.

```
auto float x, y;
```

  Las variables locales tienen persistencia automática por defecto.

- `register`: Le sugiere al compilador que la variable se guarde en los registros del microprocesador.

```
register int contador = 0;
```

  La palabra reservada `register` se puede usar solo en variables automáticas.
  (el compilador puede ignorar la sugerencia)

Persistencia estática

- `static`: Variables locales declaradas con el especificador de clase de almacenamiento `static`. Conocidas solo en la función donde son definidas, pero a diferencia de las variables automáticas, estas conservan su valor cuando se sale de la función.

```
static int contador = 1;
```

  Se inicializan a cero si no son inicializadas de forma explícita (o `NULL` para punteros).

Persistencia estática

- `static`: Variables locales declaradas con el especificador de clase de almacenamiento `static`. Conocidas solo en la función donde son definidas, pero a diferencia de las variables automáticas, estas conservan su valor cuando se sale de la función.

```
static int contador = 1;
```

  Se inicializan a cero si no son inicializadas de forma explícita (o `NULL` para punteros).

- `extern`: Se utiliza para programas de varios archivos fuentes. Por defecto, las variables globales y los nombres de funciones son de la clase de almacenamiento `extern`.

Más sobre `extern`

- *externo* en contraste a *interno* –en funciones–

Más sobre `extern`

- *externo* en contraste a *interno* –en funciones–
- Una variable es *externa* si se encuentra fuera de cualquier función

Más sobre `extern`

- *externo* en contraste a *interno* –en funciones–
- Una variable es *externa* si se encuentra fuera de cualquier función

Para el caso de variables `extern`:

Declaración: expone las propiedades de una variable (principalmente su tipo)

Definición: provoca que se reserve espacio para el almacenamiento

Alcance de variables

Alcance de un identificador: porción del programa donde puede ser referenciado.

Alcance de variables

Alcance de un identificador: porción del programa donde puede ser referenciado.

Ejemplo: variable local declarada en un bloque –puede ser referenciada en el bloque o bloques anidados–

Alcance de variables

Alcance de un identificador: porción del programa donde puede ser referenciado.

Ejemplo: variable local declarada en un bloque –puede ser referenciada en el bloque o bloques anidados–

Tipos de alcances

1. Alcance de archivo: identificador declarado fuera de cualquier función

Alcance de variables

Alcance de un identificador: porción del programa donde puede ser referenciado.

Ejemplo: variable local declarada en un bloque –puede ser referenciada en el bloque o bloques anidados–

Tipos de alcances

1. Alcance de archivo: identificador declarado fuera de cualquier función
2. Alcance de función: etiquetas (identificador seguido de `:`). p.e.: `start:` Etiquetas `case` en estructura `switch`

Alcance de variables

Alcance de un identificador: porción del programa donde puede ser referenciado.

Ejemplo: variable local declarada en un bloque –puede ser referenciada en el bloque o bloques anidados–

Tipos de alcances

1. Alcance de archivo: identificador declarado fuera de cualquier función
2. Alcance de función: etiquetas (identificador seguido de `:`). p.e.: `start:`
   Etiquetas `case` en estructura `switch`
3. Alcance de bloque: identificadores declarados dentro de un bloque
   Las variables locales `static` declaradas dentro de funciones tienen alcance de bloques aún cuando existen al momento de ejecutarse el programa

Alcance de variables

Alcance de un identificador: porción del programa donde puede ser referenciado.

Ejemplo: variable local declarada en un bloque –puede ser referenciada en el bloque o bloques anidados–

Tipos de alcances

1. Alcance de archivo: identificador declarado fuera de cualquier función
2. Alcance de función: etiquetas (identificador seguido de `:`). p.e.: `start:`
   Etiquetas `case` en estructura `switch`
3. Alcance de bloque: identificadores declarados dentro de un bloque
   Las variables locales `static` declaradas dentro de funciones tienen alcance de bloques aún cuando existen al momento de ejecutarse el programa
4. Alcance de prototipo de función: lista de parámetros en los prototipos de funciones

Alcance de variables – Ejemplos

El alcance de una variable o función *externa* va desde el punto en el que se declara hasta el fin del archivo.

Alcance de variables – Ejemplos

El alcance de una variable o función *externa* va desde el punto en el que se declara hasta el fin del archivo.

```
int main(void) { . . . }

int indice = 0;
double vector[MAXVAL];

void f1(double f) { . . . }
double f2(void) { . . .}
```

Alcance de variables – Ejemplos

El alcance de una variable o función *externa* va desde el punto en el que se declara hasta el fin del archivo.

```
int main(void) { . . . }

int indice = 0;
double vector[MAXVAL];

void f1(double f) { . . . }
double f2(void) { . . .}
```

- Las variables `indice` y `vector` se pueden utilizar en `f1()` y `f2()`, pero no en `main()`.
- Si se hace referencia a una variable *externa* antes de su definición, o si está definida en un archivo fuente diferente, es obligatorio una declaración `extern`.

Alcance de variables – Ejemplos

Si las líneas

```
int indice;
double vector[MAXVAL];
```

aparecen fuera de cualquier función, *definen* las variables externas `indice` y `vector`, y reserva espacio para su almacenamiento.

Alcance de variables – Ejemplos

Si las líneas

```
int indice;
double vector[MAXVAL];
```

aparecen fuera de cualquier función, *definen* las variables externas `indice` y `vector`, y reserva espacio para su almacenamiento.

Las líneas

```
extern int indice;
extern double vector[];
```

declaran para el resto del archivo que `indice` es un `int` y que `vector` es un arreglo `double` (cuyo tamaño se determina en algún otro lugar), pero no crea las variables ni les reserva espacio.

Alcance de variables – Ejemplos

Si las líneas

```
int indice;
double vector[MAXVAL];
```

aparecen fuera de cualquier función, *definen* las variables externas `indice` y `vector`, y reserva espacio para su almacenamiento.

Las líneas

```
extern int indice;
extern double vector[];
```

declaran para el resto del archivo que `indice` es un `int` y que `vector` es un arreglo `double` (cuyo tamaño se determina en algún otro lugar), pero no crea las variables ni les reserva espacio.

- Debe existir una única *definición* de una variable externa entre todos los archivos fuentes que forman un programa.
- Los demás archivos pueden hacer *declaraciones* `extern` de estas variables para tener acceso a ellas.

Alcance de variables – Ejemplos

Archivo 1

```
int indice = 0;
double vector[MAXVAL];
```

Archivo 2

```
extern int indice;
extern double vector;

void f1(double f) { . . . }

double f2(void) { . . . }
```

Alcance de variables – Ejemplos

Archivo 1

```
int indice = 0;
double vector[MAXVAL];
```

Archivo 2

```
extern int indice;
extern double vector;

void f1(double f) { . . . }

double f2(void) { . . . }
```

Ver ejemplo de *alcance* D&D 5.12.

Calificadores – `volatile` y `const`

`volatile`

- Le indica al compilador no optimizar lo relacionado a dichas variables
- Utilizado en la mayoría de los casos para el acceso al hardware
- Le indica que la variable no se guarde en cache

Calificadores – `volatile` y `const`

`volatile`

- Le indica al compilador no optimizar lo relacionado a dichas variables
- Utilizado en la mayoría de los casos para el acceso al hardware
- Le indica que la variable no se guarde en cache

Ejemplo:

```
salir = 0;
while(!salir)
{
  /* bucle corto completamente
   * visible al compilador */
}
```

Calificadores – `volatile` y `const`

`volatile`

- Le indica al compilador no optimizar lo relacionado a dichas variables
- Utilizado en la mayoría de los casos para el acceso al hardware
- Le indica que la variable no se guarde en cache

Ejemplo:

```
salir = 0;
while(!salir)
{
  /* bucle corto completamente
   * visible al compilador */
}
```

- El compilador determina que el cuerpo del bucle no modifica la variable `salir` y convierte el bucle en un bucle `while(1)`. Incluso si la variable de salida se establece en el controlador de señal para SIGINT y SIGTERM.
- Si la variable `salir` se declara `volatile`, el compilador se ve obligado a cargarla cada vez, ya que puede modificarse en otro lugar.

Calificadores – `volatile` y `const`

`const`

- Le informa al compilador que el valor de una variable no debe modificarse.

Calificadores – `volatile` y `const`

`const`

- Le informa al compilador que el valor de una variable no debe modificarse.
- No existía en la primeras versiones de C. Fue agregado en el ANSI C.

Calificadores – `volatile` y `const`

`const`

- Le informa al compilador que el valor de una variable no debe modificarse.
- No existía en la primeras versiones de C. Fue agregado en el ANSI C.
- Seis posibilidades del uso y no uso de `const` con parámetros de función.

Calificadores – `volatile` y `const`

`const`

- Le informa al compilador que el valor de una variable no debe modificarse.
- No existía en la primeras versiones de C. Fue agregado en el ANSI C.
- Seis posibilidades del uso y no uso de `const` con parámetros de función.
  - dos al pasar parámetros en llamada por valor, y

Calificadores – `volatile` y `const`

`const`

- Le informa al compilador que el valor de una variable no debe modificarse.
- No existía en la primeras versiones de C. Fue agregado en el ANSI C.
- Seis posibilidades del uso y no uso de `const` con parámetros de función.
  - dos al pasar parámetros en llamada por valor, y
  - cuatro al pasar parámetros en llamada por referencia (punteros).

Calificadores – `volatile` y `const`

`const`

- Le informa al compilador que el valor de una variable no debe modificarse.
- No existía en la primeras versiones de C. Fue agregado en el ANSI C.
- Seis posibilidades del uso y no uso de `const` con parámetros de función.
  - dos al pasar parámetros en llamada por valor, y
  - cuatro al pasar parámetros en llamada por referencia (punteros).

Ejemplo:

```
void imprimir_arreglo(const int datos[], const int tam)
{
. . .
}
```

Calificador `const` con punteros

Existen cuatro formas de pasar punteros a funciones:

Calificador `const` con punteros

Existen cuatro formas de pasar punteros a funciones:

1. Puntero no constante a datos no constantes, p.e.: `int *pi;` (notación) (no incluye `const`)

Calificador `const` con punteros

Existen cuatro formas de pasar punteros a funciones:

1. Puntero no constante a datos no constantes, p.e.: `int *pi;` (notación) (no incluye `const`)

2. Puntero no constante a datos constantes

Calificador `const` con punteros

Existen cuatro formas de pasar punteros a funciones:

1. Puntero no constante a datos no constantes, p.e.: `int *pi;` (notación) (no incluye `const`)

2. Puntero no constante a datos constantes

3. Puntero constante a datos no constantes. Inicializados al declararlos (nombre de arreglo)

Calificador `const` con punteros

Existen cuatro formas de pasar punteros a funciones:

1. Puntero no constante a datos no constantes, p.e.: `int *pi;` (notación) (no incluye `const`)

2. Puntero no constante a datos constantes

3. Puntero constante a datos no constantes. Inicializados al declararlos (nombre de arreglo)

4. Puntero constante a datos constantes

Calificador `const` con punteros –Ejemplos

Puntero a entero constante

```
int * const foo = &var;
```

Calificador `const` con punteros –Ejemplos

Puntero a entero constante

```
int * const foo = &var;
```

Puntero constante a entero

```
int const * foo = &var;
const int * foo = &var;
```

Calificador `const` con punteros –Ejemplos

Puntero a entero constante

```
int * const foo = &var;
```

Puntero constante a entero

```
int const * foo = &var;
const int * foo = &var;
```

Puntero constante a un entero constante

```
int const * const foo = &var;
const int * const foo = &var;
```