UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente:** Américo Chulluncuy Reynoso

2025-II

Sesión 11:

Gestión dinámica de memoria y estructuras

Resumen del Curso

SEGUNDA PARTE:

1. Gestión Dinámica de Memoria

2. **Estructuras (+ Gestión dinámica de memoria):** C++ permite definir punteros a estructuras, al igual que a cualquier tipo de variable, esto conlleva a la gestión dinámica de memoria con estructuras.

3. Archivos

4. Introducción a la POO.

Contenido

1. Punteros: errores frecuentes
2. Estructuras: repaso
3. Anidamiento de una Estructura
4. Puntero a Estructura, asignación dinámica

1. Punteros: errores frecuentes

- Cuando se utilizan punteros, es difícil encontrar los errores, ya que **no se generan mensajes de error** y el programa se comporta de manera impredecible.

Sugerencias:

- Evitar utilizar punteros que apunten a variables estáticas y en especial a las variables locales. Ejemplo
- Todo puntero ha de apuntar a una dirección de memoria reservada, o si no, asignarle el valor nullptr (buena práctica) Ejemplo
- No olvidar reservar memoria (cuando sea necesario) antes de utilizar el puntero.
- No confundir la asignación de punteros con la asignación de contenidos (p = q; con *p = *q).

2. Estructuras: Repaso

- `int a;` define una variable (un espacio de memoria) que solo podrá almacenar un valor.

- `int a[5];` define un conjunto de 5 variables todas identificadas por un mismo nombre. Todos los elementos del arreglo tienen el mismo tipo de dato (inconveniente)

- Una **Estructura** (registro) permite definir un conjunto de datos identificados con el mismo nombre donde cada elemento (miembro) del conjunto puede ser de diferente tipo.

- La evolución de las estructuras dio lugar a las clases definida en programación orientada a objetos (POO) e implementadas en C++

Estructura: Definición

```
struct nombre {
  int myNum;          // miembro (variable int)
  string myString; // miembro (variable string)
};
```

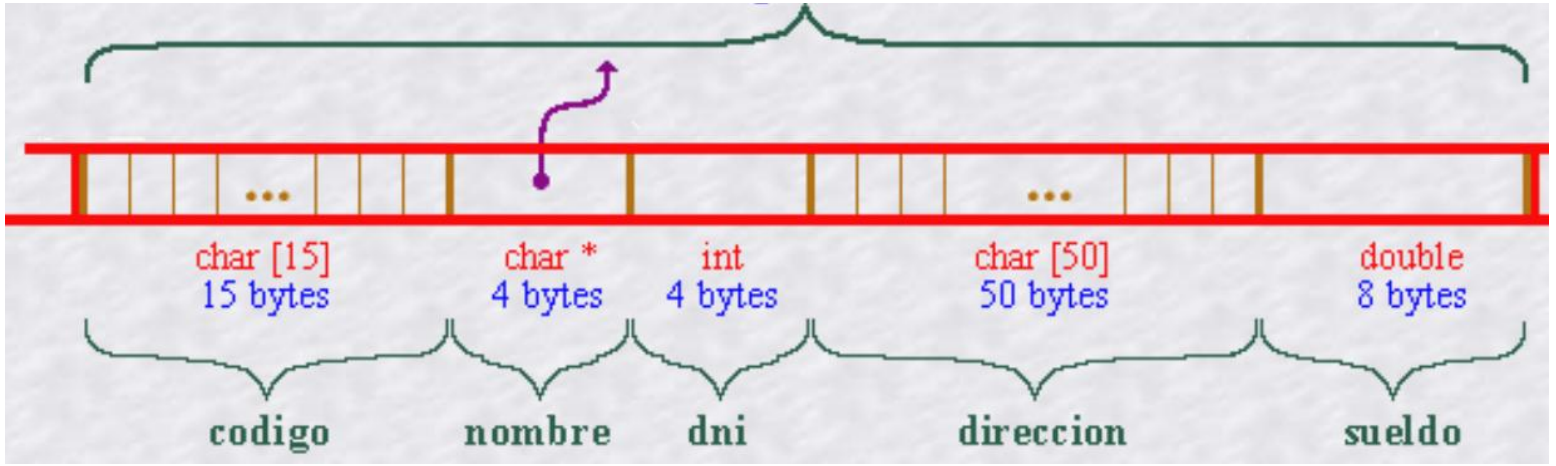
- A diferencia de los arreglos al definir una estructura **no estamos definiendo una variable directamente**, si no un "Tipo de dato". Esto quiere decir que el nombre dado a la estructura servirá para declarar variables de ese tipo

Estructura: declaración, representación interna

- Una variable de tipo struct se almacena en la pila, como se muestra a continuación:

```
struct TPersona {
        char codigo[15];
        char *nombre;
        int dni;
        char direccion[50];
        double sueldo;
};
```

TPersona empleado;



Estructura: Asignación de datos

- Una vez declarada la variable ésta se podrá manipular, asignando valores a sus campos o utilizando los valores de éstos en otras expresiones.

```
strcpy(empleado.codigo, "995-535262");
empleado.nombre = new char [30];
strcpy (empleado.nombre, "Juan Lopez");
empleado.dni = 82716612;  //...
```

- Otra forma de asignar valores a una variable de tipo struct es al momento de declararla, esta operación se hace de la siguiente manera:

```
TPersona empleado = {"995-535262", "Juan Lopez", 82716612, "Av. ABC 123", 12345.67};
```

- La manera en que se almacenan las variables de tipo struct permite manejar las variables como una unidad, y **a diferencia de los arreglos**, se podrán asignar los valores de todos los campos de una variable a otra en una sola operación.

Ejemplo: Asignación de datos

```
TPersona trabajador;
TPersona empleado = {"995-535262", "Juan Lopez", 82716612, "Av. ABC 123", 12345.67};
trabajador  = empleado; // pasamos el contenido de la variable empleado
                        // a la variable trabajador
```

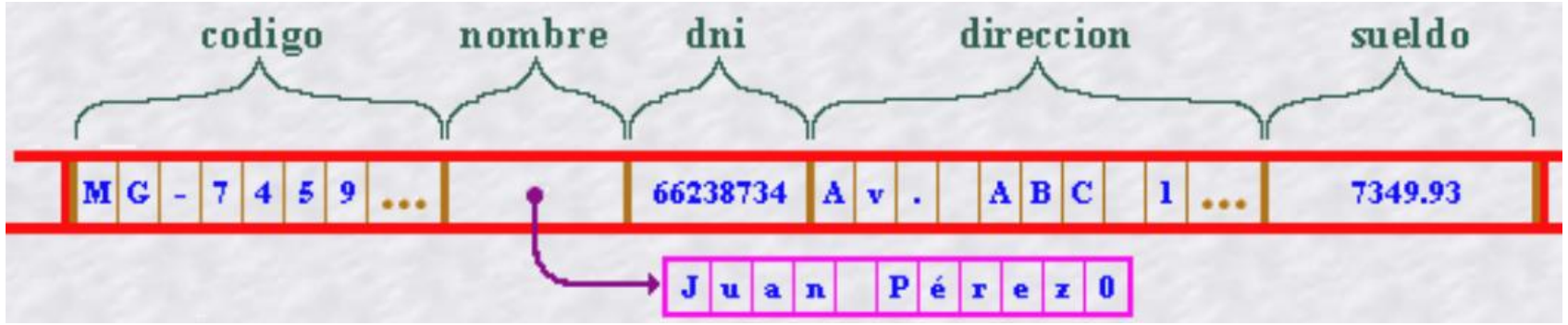
- Aunque la operación de asignación se puede dar entre variables de tipo struct, hay que tener cuidado cuando asignamos datos a sus miembros.

- Ejemplo: Si reasignamos el campo dni de trabajador, esto no afecta al valor de dni en empleado. Sin embargo, si reasignamos el campo nombre, el cambio sí afecta a ambos. Esto ocurre porque nombre es un puntero; al realizar trabajador = empleado, se copia la dirección de memoria a la que apunta nombre en empleado hacia nombre en trabajador, haciendo que ambos apunten al mismo lugar. Por lo tanto, cualquier modificación en el contenido de nombre en trabajador también será visible en empleado.

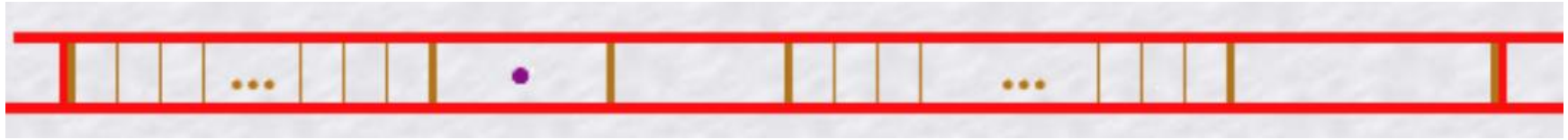
Estructura: Asignación de datos

Antes de la asignación:

`empleado`

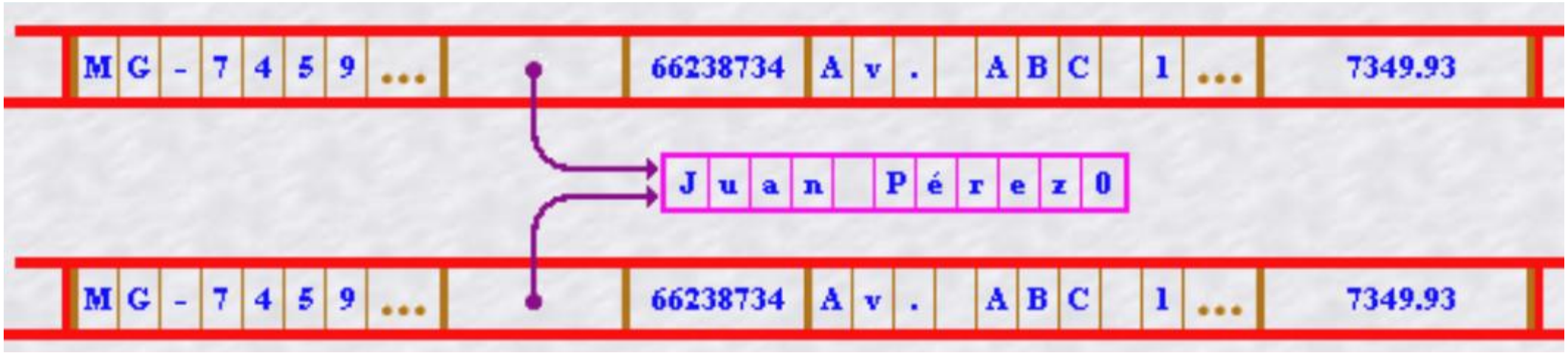


`trabajador`



Estructura: Asignación de datos

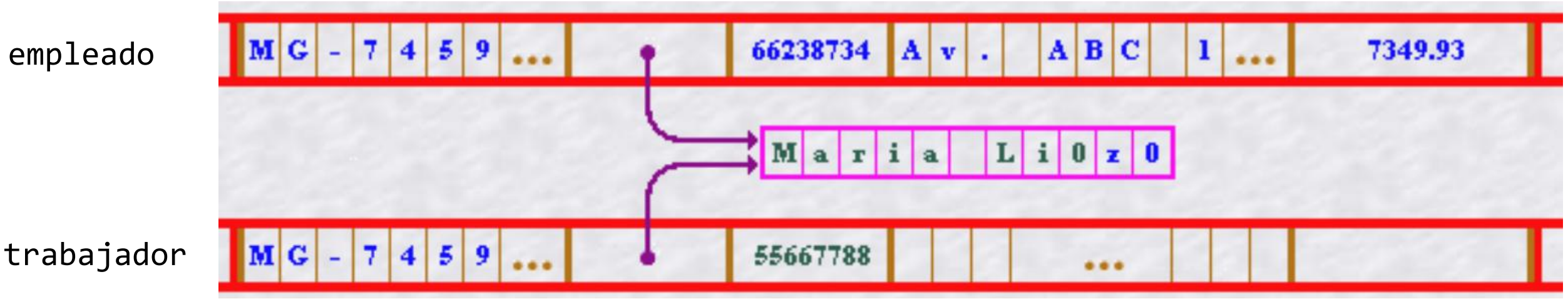
Luego de la asignación: `trabajador = empleado;`



Estructura: Asignación de datos

Luego de la asignación a sus miembros:

```
trabajador.dni = 55667788;
strcpy (trabajador.nombre, "Maria Li");
```



Advertencia: Si la estructura contiene punteros, una asignación directa entre estructuras copiará las direcciones de memoria y no los valores, por lo tanto, los cambios en un campo de la estructura afectarán a la otra.

Estructura: Asignación de datos

Otra forma de asignar datos a una variable de tipo struct es mediante funciones. **Una función puede devolver un dato de tipo struct**

El siguiente ejemplo muestra cómo:

- Definir una función que devuelve una estructura
- Utilizar punteros como miembros de una estructura
- Limpiar el buffer
- Asignación dinámica a miembros de una estructura

```cpp
#include <iostream>
#include <cstring>
using namespace std;

struct Persona {
    int edad;
    char* nombre;
    float altura;
}
```

```cpp
Persona crearPersona() {
    Persona p;
    cout << "Ingrese la edad: ";
    cin >> p.edad;
    cin.ignore();
    cout << "Ingrese el nombre: ";
    char buffer[100];
    cin.getline(buffer, 100);
    p.nombre = new char[strlen(buffer) + 1];
    strcpy(p.nombre, buffer);

    cout << "Ingrese la altura: ";
    cin >> p.altura;
    return p;
}
```

…continuación

```
int main() {

    Persona empleado = crearPersona();

    cout << "\nDatos del empleado:" << endl;
    cout << "Nombre: " << empleado.nombre << endl;
    cout << "Edad: " << empleado.edad << endl;
    cout << "Altura: " << empleado.altura << endl;

    delete[] empleado.nombre;

    return 0;
}
```

Comparación de datos entre estructuras

- A pesar de que podemos asignar las estructuras de datos directamente, **no se pueden comparar de forma directa**. Debido a la forma cómo se representa una estructura en memoria.

- Al comparar estructuras, se debe hacer esta operación miembro a miembro y no como una unidad.

3. Estructuras anidadas

Un tipo de dato struct se puede emplear como tipo de dato para definir uno o más miembros (campos) en otra estructura. El ejemplo siguiente muestra esta propiedad:

```
struct TFecha {
    int dd;    // día
    int mm;  // mes
    int aa;    // año
};
struct TEmpleado{
    int dni;
    char *nombre;
    char *direccion;
    TFecha fechaDeNacimiento; // miembro de tipo struct
    char *cargo;
    TFecha fechaDeIngreso; // miembro de tipo struct
};
```

Estructuras anidadas

Para manejar los campos en la estructura anidada se sigue la misma lógica que en el caso de una estructura simple. Considerando el ejemplo anterior:

```
int main() {
    TEmpleado empleado;
    empleado.fechaDeNacimiento.dd = 27;
    empleado.fechaDeNacimiento.mm = 3;
    empleado.fechaDeNacimiento.aa = 1989;
}
```

4. Punteros a estructuras, asignación dinámica

- Podemos definir un puntero a una estructura, como en cualquier variable.

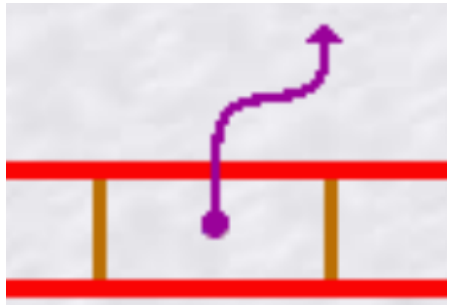
- El siguiente ejemplo muestra una estructura sencilla, y a partir de ella veremos en detalle el manejo de la variable referenciada.

```
struct TPersona {
        int dni;
        char *nombre;
        double sueldo;
};

int main() {
    TPersona *empleado; // declaración (puntero a variable struct)
    empleado = new Tpersona; // asignación de memoria
}                                //se crea la variable referenciada
```

Puntero a estructura: representación

- `TPersona *empleado;`
  crea el espacio en la pila solo para el puntero



`empleado`

- `empleado = new TPersona;`
  Esta instrucción crea un bloque de memoria en el montón (heap) suficiente para almacenar una estructura del tipo TPersona y asigna la dirección de este bloque al puntero empleado. De esta forma, empleado puede referenciar y manipular la estructura en la memoria dinámica.

`*empleado`

`empleado`

Acceso a los miembros de la variable referenciada

- Cuando se hace mención al miembro de una variable de tipo struct el compilador reconoce al nombre de la variable, al punto y al nombre del miembro como el identificador.

- **Error común** : escribir `*empleado.dni` es un error porque el campo dni no es un puntero, es una variable de tipo int. El operador . **tiene mayor precedencia** que el operador *

- La manera correcta de llegar al campo dni es: `(*empleado).dni` , de ese modo el operador * se aplica sobre el puntero y no sobre el campo.

- **Forma alternativa**: usar el operador flecla `->`, para llegar al campo de la variable referenciada. Ejemplo: `empleado -> dni`

```
#include <iostream>
#include <string>
using namespace std;
struct Empleado{
   string nombre;
   string n_telef;
};
void trabajadores(Empleado *);  // prototipo de funcion
void mostrar(Empleado *);   // prototipo de funcion
int main(){
   char key;
   Empleado *recPoint;
   cout << "Desea crear un nuevo registro? (y/n): ";
   key = cin.get();
   if (key == 'y') {
      key = cin.get();         //leer '\n' en la entrada almacenada en el búfer <>cin.ignore()
      recPoint = new Empleado; //ASIGNACIÓN DINÁMICA …liberar??
      trabajadores(recPoint);
      mostrar(recPoint);
   }
   else{
      cout << "\nNo se creo ningun registro.";
      return 0;
   }
```

…continuación

```
// Ingresar un nombre y un numero de telefono
void trabajadores(Empleado *record) {
    cout << "Ingrese un nombre: ";
    getline(cin,record->nombre);
    cout << "Ingrese el numedo de telefono: ";
    getline(cin,record->n_telef);
}


// Mostrar el contenido del registro
void mostrar(Empleado *cont){
    cout << "\nContenidos del registro creado:" << "\nnombre: " << cont->nombre
    << "\nNumero de Telefono: " << cont->n_telef << endl;
}
```

Estructuras autoreferenciadas

- Una estructura autoreferenciada es una variable de tipo struct en la que uno o más de sus miembros (campos) son punteros del mismo tipo que la estructura que lo define.

- Bajo este concepto, una variable definida así puede enlazarse (apuntar), a través de estos miembros, a otra u otras variables del mismo tipo, generando las Listas enlazadas (ligadas), Árboles, etc.

```
struct NodoLSL {
    int codigo;
    char *nombre;
    NodoLSL *siguiente;
};
```