UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

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

2025-3

Sesión 10:

Estructura de Datos

Resumen del Curso

SEGUNDA PARTE:

1. Gestión Dinámica de Memoria

2. **Estructura:** Es un tipo de dato compuesto que permite almacenar un conjunto de datos de diferente tipo.

3. Archivos

4. Introducción a la POO.

Contenido

1. Estructuras

   Definición, declaración, inicialización, asignación, lectura, escritura

2. Arreglo de Estructuras

3. Anidamiento de una Estructura

4. Puntero a Estructura

5. Paso de Estructura como Parámetro de una función

6. Funciones y Estructuras

Introducción: Tipos de Datos en C++

- Los **tipos de datos** son fundamentales para la manipulación y almacenamiento de información.

- Se dividen en dos categorías principales:

  Tipos de datos integrados o primitivos (built-in data types)

  Tipos de datos definidos por el usuario (user-defined data types)

  Esta distinción es importante, ya que nos lleva al siguiente nivel de abstracción, como las estructuras (un tipo de dato definido por el usuario)

Tipos de datos integrados

Un tipo de dato se llama integrado si el compilador sabe:

1. Cómo **representar objetos del tipo**

2. Qué **operaciones** se pueden realizar en él sin que se lo indiquen las declaraciones proporcionadas por un programador en el código fuente.

Built-in types	
bool x	**x** is a Boolean (values **true** and **false**).
char x	**x** is a character (usually 8 bits).
short x	**x** is a short **int** (usually 16 bits).
int x	**x** is of the default integer type.
float x	**x** is a floating-point number (a "short double").
double x	**x** is a ("double-precision") floating-point number.
void* p	**p** is a pointer to raw memory (memory of unknown type).
T* p	**p** is a pointer to **T**.
T *const p	**p** is a constant (immutable) pointer to **T**.
T a[n]	**a** is an array of **n** **T**s.
T& r	**r** is a reference to **T**.
T f(arguments)	**f** is a function taking **arguments** and returning a **T**.
const T x	**x** is a constant (immutable) version of **T**.
long T x	**x** is a **long T**.
unsigned T x	**x** is an **unsigned T**.
signed T x	**x** is a **signed T**.

Tipos de datos definidos por el usuario

El **programador** necesita indicarle al compilador:

1. **Cómo representar objetos** de ese tipo en la memoria.

2. **Qué operaciones** se pueden realizar sobre él, ya que no están predefinidas en el lenguaje.

Este tipo de datos **permite** al programador **crear estructuras más complejas, que pueden representar objetos del mundo real** o abstracciones específicas que no están cubiertas por los tipos de datos integrados.

Tipos definidos por el usuario

Ejemplos típicos de datos definidos por el usuario:

1. Clases (class): Objetos que encapsulan datos y funciones.
2. Estructuras (struct): Agrupa diferentes tipos de datos.
3. Uniones (union): Permiten almacenar diferentes tipos de datos en la misma ubicación de memoria, uno a la vez.
4. Campos de bits, una variación struct y permite un fácil acceso a bits individuales
5. Enumeración (enum): Lista de constantes enteras con nombre.
6. Alias de tipo (typedef/using): Define un nuevo nombre para un tipo existente.

class vs struct

- En C++ una clase se compone de tipos integrados, de tipos definidos por el usuario y funciones. Las partes utilizadas para definir la clase se denominan **miembros**.

```
class X {
        public://especificador de acceso
        int m; //miembro de datos (atributos)
       private:
        int fm(int v) {// miembro de función (métodos)
             int c = v;
             return c;
       }
};
```

- En general los miembros pueden ser: (i) **miembros de datos**, que definen la representación de un objeto de la clase, (ii) **miembros de funciones**, que proporcionan operaciones sobre dichos objetos.

Estructuras: como caso particular de clases

- Los miembros de la clase **son privados por defecto**; es decir,

```
class X {
 int fm();// . . .
};
```

significa

```
class X {
 private:
  int fm();// . . .
};
```

- Un usuario no puede referirse directamente a un miembro privado.

```
X x; //x es una variable del tipo  X
int y = x.fm(); // error: mf es private (inaccessible)
```

- Una **estructura** **es una clase cuyos miembros son públicos por defecto**

es equivalente a

```
struct X {
int m; // . . .
};
```

```
class X {
public:
int m; // . . .
};
```

1. Estructura

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

- `struct` es una palabra reservada que indica que los elementos que vienen agrupados a continuación entre llaves componen una estructura.
- `nombre` es opcional, identifica el tipo de dato que se describe y del cual se podrán declarar variables.
- `myNum, myString` son los elementos (miembros) que componen la estructura de datos, deben estar precedidos por el tipo de dato.

Observación: **una estructura define un tipo de dato**, no una variable, lo que significa que no existe reserva de memoria cuando el compilador está analizando la estructura.

Definición, declaración, inicialización

- **Forma 1**: (nombradas) permite tratarla como un tipo de dato. **Podemos crear variables con esta estructura en cualquier parte del programa (instanciar).**

```
struct tarjetas {
  long int  num_tarjeta;
  string tipo_cuenta;
  char nombre [80];
  float  saldo;
};

tarjetas cli1, cli2;
```

- Como cualquier tipo de variables, pueden ser inicializadas al ser declaradas:

```
tarjetas cli1 = { 439334000526,"Ahorro", "Alberto García Pérez", 1950.33};
```

```cpp
#include <iostream>
#include <string>
using namespace std;
struct car{
    string marca;
    string modelo;
    int year;
  };
int main() {
  car myCar1;  //Creamos una estructura car y lo almacenamos en myCar1
  myCar1.marca = "BMW";
  myCar1.modelo = "X5";
  myCar1.year = 1999;

  car myCar2;  // Creamos otra estructura car  y lo almacenamos en myCar2
  myCar2.marca = "Ford";
  myCar2.modelo = "Mustang";
  myCar2.year = 1969;

  cout << myCar1.marca << " " << myCar1.modelo << " " << myCar1.year << "\n";
  cout << myCar2.marca << " " << myCar2.modelo << " " << myCar2.year << "\n";
  return 0;
}
```

- **Forma 2**: Incluir en la propia definición de la estructura aquellas variables que se van a emplear en el programa. El ámbito de estas variables será el mismo que el de la declaración del tipo de dato estructura.

```
struct  {
  long int  num_tarjeta;
  char tipo_cuenta;
  char nombre [80];
  float  saldo;
} cliente1, cliente2;
```

- En el caso que no se vayan a declarar más variables de este tipo de dato en otros lugares del programa, el nombre de la estructura es innecesario.

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
  struct {
    string marca;
    string modelo;
    int year;
  } myCar1, myCar2;

  myCar1.marca = "BMW";// inicializando la primera estructura
  myCar1.modelo = "X5";
  myCar1.year = 1999;

  myCar2.marca = "Ford";   // inicializando la segunda estructura
  myCar2.modelo = "Mustang";
  myCar2.year = 1969;

  // Mostrando los miembros de la estructura
  cout << myCar1.marca << " " << myCar1.modelo << " " << myCar1.year << "\n";
  cout << myCar2.marca << " " << myCar2.modelo << " " << myCar2.year << "\n";
  return 0;
}
```

Estructura: Acceso a sus miembros

- Accedemos a los miembros utilizando la notación **objeto.miembro**.
  Por ejemplo, para la estructura

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

```
// Asignación de valores a los miembros de la estructura
myStructure.myNum = 1;
myStructure.myString = "Hola Mundo!";

// Imprimiendo miembros de la estructura
cout << myStructure.myNum << "\n";
cout << myStructure.myString << "\n";
```

Estructura: lectura y escritura

```
struct Alumnos { //Estructura de alumnos
        char nombre[20];
        int edad;
};
int main(){
    Alumnos alumnos[3];  //Arreglo de 3 Alumnos

    for (int i = 0; i < 3; i++){//INGRESAR LOS DATOS DE LOS ALUMNOS
        cout << "Ingrese el nombre del alumno " << i+1 <<": ";
        cin.getline(alumnos[i].nombre,20);
        cout << "Introduce edad del alumno " << i+1 << ": ";
        cin >> alumnos[i].edad;
        cin.ignore();  //Limpiar el buffer de entrada
    }

    for (int i = 0; i < 3; i++){ //MOSTRAR LOS DATOS DE LOS ALUMNOS
        cout << "\nALUMNO: " << i+1 << ": ";
        cout << "Nombre: " << alumnos[i].nombre << endl; //Nombre del alumno
        cout << "Edad: " <<alumnos[i].edad << endl;; //Edad del  alumno
    }
}
```

2. Arreglos de Estructuras

1) **Arreglo como miembro de una estructura**:
   Una estructura puede contener un arreglo como miembro (arreglo dentro de una estructura) y se puede acceder a ella del mismo modo que accedemos a otros elementos de la estructura.

Ejercicio: Escribir un programa que muestre el registro de un estudiante con su número de lista, su sección y las 4 calificaciones obtenidas en el curso, almacenadas en un arreglo dentro de la estructura.

Ejemplo: Registro de estudiantes

```
// Definición
struct Alumno {
    int n_orden;
    char seccion;
    // Array dentro de la estructura
    float notas[4];
};
```

```
int main(){
    //Declaración e inicialización la estructura
    Alumno A = { 1,'A',{18.5,17,12.9,8.5 }};
    // Mostrando la estructura
    imprimir(A);
    return 0;
}
```

```
// Mostrando el contenido de la estructura
void imprimir(Alumno a1){
    cout << "Numero de orden : " << a1.n_orden << endl;
    cout << "Sección : " << a1.seccion << endl;
    cout << "Notas:\n";
    int n = sizeof(a1.notas) / sizeof(float);
    // Accediendo al contenido del arreglo
    for (int i = 0; i < n; i++) {
        cout << "Nota " << i + 1 << " : " << a1.notas[i] << endl;
    }
}
```

2) **Arreglo de estructuras**:
Cada elemento de un arreglo puede ser una estructura.

PARÁMETRO	ARREGLO DENTRO DE UNA ESTRUCTURA	ARREGLO DE ESTRUCTURAS
Idea básica	Una estructura contiene un arreglo como variable miembro.	Un arreglo en la que cada elemento es de tipo estructura.
Acceso	Se puede acceder usando el operador punto del mismo modo que accedemos a otros elementos de la estructura.	Se puede acceder indexando del mismo modo que accedemos a un arreglo.

Ejemplo: Registro de estudiantes

```
// Declaracion
struct Alumno{
    int n_orden;
    char seccion;
    float notas;
};
```

```
int main(){
    //inicializamos la structura
    Alumno A = { 1,'A',{18.5,17,12.9,8.5 }};
    // Mostrando la estrutura
    imprimir(A);
    return 0;
}
```

```
// Mostrando el contenido del arreglo de estructuras
void imprimir(Alumno registro_alumnos[] a1){
    cout << "Numero de orden : " << a1.n_orden << endl;
    cout << "Sección : " << a1.seccion << endl;
    cout << "Notas:\n";
    int n = sizeof(a1.notas) / sizeof(float);
    // Accediendo al contenido del array
    for (int i = 0; i < n; i++) {
        cout << "Nota " << i + 1 << " : " << a1.notas[i] << endl;
    }
}
```

3. Anidamiento de una estructura

Consiste en definir una estructura dentro de otra.

```
//Estructura interna
struct Empleado{
  int Empleado_id;
  char nombre[20];
  int salario;
};
//Estructura externa
struct Organizacion  {
  char nombre_org[20];
  char numero_org_[20];
  Empleado empleado;
};
```

```
// Estructura externa
struct Organisation  {
  char nombre_org[20];
  char numero_org[20];

  // Estructura interna
  struct Empleado{
    int empleado_id;
    char nombre[20];
    int salario;
  }; empleado;
};
```

Se puede anidar de dos formas: (i) **Estructura anidada separada**

(ii) Estructura anidada integrada

4. Puntero a estructura

- Es una variable que almacena la dirección de memoria donde se encuentra una instancia de una estructura. Esto permite:

  ✓ Acceso indirecto a los miembros de la estructura mediante el operador **->**
  Forma directa: desreferenciar el puntero (*), aplicar el operador punto (.)
  Forma indirecta: aplicar el operador flecha al puntero

  ✓ Manejo dinámico de estructuras (crear modificar liberar en tiempo de ejecución)

  ✓ Crear estructuras de datos complejas: (pila, cola, listas enlazadas, árboles, grafos, ...)

Ejemplo: puntero a estructura

```
struct Punto{
    int valor;
};

int main(){
    Punto *pdin = new Punto;
    pdin -> valor = 10; // (*pdin).valor = 10
    delete pdin;
    pdin = nullptr;
    return 0;
}
```

```
struct Estudiante{
    int no_orden;
    char nombre[30];
    char especialidad[40];
    int ingreso;
};

int main(){
    Estudiante s1;
    Estudiante* ptr = &s1;

    s1.no_orden = 27;
    strcpy(s1.nombre, "Raul");
    strcpy(s1.especialidad, "Ciencia de la Computación");
    s1.ingreso = 2019;

    cout<<"Numero de orden: " <<  (*ptr).no_orden<<endl;
    cout<<"Nombre: " << (*ptr).nombre<<endl;
    cout<<"Especialidad : " << (*ptr).especialidad;
    cout<<"Año de ingreso: " << (*ptr).ingreso;

    return 0;
}
```

```
struct Estudiante {
    int no_orden;
    char nombre[30];
    char especialidad[40];
    int ingreso;
};
Estudiante s, *ptr; // Def variable estructura y puntero
int main(){
    ptr = &s;
    //Solicitando ingresos
    cout <<"ingrese el número de orden del estudiante: "; cin>> ptr->no_orden;
    cout <<"ingrese el nombre del estudiante: "; cin>> ptr->nombre;
    cout <<"ingrese la especialidad del estudiante: "; cin>> ptr->especialidad;
    cout <<"Año de ingreso del estudiante: "; cin>> ptr->ingreso;

    //Mostrando detalles del estudiante
    cout <<"Numero de orden: " << ptr->no_orden;
    cout <<"Nombre: " << ptr->nombre;
    cout <<"Especialidad: " << ptr->especialidad;
    cout <<"Año de ingreso: " << ptr->ingreso;
    return 0;
}
```

5. Paso de Estructura como Parámetro de una función

- Las estructuras se pueden pasar a una función y devolverlas de la misma forma que cualquier otro tipo de variable: por valor, por referencia o por puntero.

- Desde la función principal main() o cualquier otra función, se puede proporcionar una estructura a cualquier otra función.

```
struct myStructure {
   int x;
   int y;
};
void swap(myStructure);
```

```
struct myStructure {
   int x;
   int y;
};
void swap(myStructure &);
```

```cpp
struct Persona {
    char nombre[50];
    int edad;
    float salario;
};

void mostrarData(Persona);   // Prototipo (declaración) de la función

int main() {
    Persona p;
    cout << "Ingrese sus nombres: "; cin.getline(p.nombre, 50);
    cout << "Ingrese su edad: "; cin >> p.edad;
    cout << "Ingrese su salario: "; cin >> p.salario;
    mostrarData(p); // Llamada a función con variable estructura como argumento
    return 0;
}
void mostrarData(Persona p) {
    cout << "\nMostrando Información." << endl;
    cout << "nombre: " << p.nombre << endl;
    cout <<"edad: " << p.edad << endl;
    cout << "salario: " << p.salario;
}
```

6. Estructuras y funciones

- **Observación**: siempre debe definir la estructura antes de las declaraciones de funciones; de lo contrario, obtendrá un error de compilación.

```
#include <iostream>
using namespace std;
struct Student{
    char stuName[30];
    int stuRollNo;
    int stuAge;
};


void printStudentInfo(Student);
```

```
int main(){
    Student s;
    cout<<"Enter Student Name: ";
    cin.getline(s.stuName, 30);
    cout<<"Enter Student Roll No: ";
    cin>>s.stuRollNo;
    cout<<"Enter Student Age: ";
    cin>>s.stuAge;
    printStudentInfo(s);
    return 0;
}

void printStudentInfo(Student s){
    cout<<"Student Record:"<<endl;
    cout<<"Name: "<<s.stuName<<endl;
    cout<<"Roll No: "<<s.stuRollNo<<endl;
    cout<<"Age: "<<s.stuAge;
}
```

Estructuras y funciones

Podemos retornar variables de tipo estructura mediante funciones:

```
#include <iostream>
using namespace std;
struct Student{
    char stuName[30];
    int stuRollNo;
    int stuAge;
};

Student getStudentInfo();

void printStudentInfo(Student);


int main(){
    Student s;
    s = getStudentInfo();
    printStudentInfo(s);
    return 0;
}
```

```
Student getStudentInfo(){
    Student s;
    cout<<"Enter Student Name: ";
    cin.getline(s.stuName, 30);
    cout<<"Enter Student Roll No: ";
    cin>>s.stuRollNo;
    cout<<"Enter Student Age: ";
    cin>>s.stuAge;
    return s;
}


void printStudentInfo(Student s){
    cout<<"Student Record:"<<endl;
    cout<<"Name: "<<s.stuName<<endl;
    cout<<"Roll No: "<<s.stuRollNo<<endl;
    cout<<"Age: "<<s.stuAge;
}
```