
UTEC
UNIVERSIDAD DE INGENIERÍA
Y TECNOLOGÍA

# CS1103
# Programación Orientada a Objetos 2
# Unidad 1
# Tema: Conceptos Fundamentales de Programación

Rubén Rivas

# Conceptos Fundamentales de Programación

1. Sistema de datos compiladores y clasificación de los tipos
2. Clases, tipos de constructores
3. Sobrecarga de operadores y friend
4. Herencia y polimorfismo
5. Clase compuestas
6. Archivos

# Tipos de datos

¿Qué es el sistema de tipos?

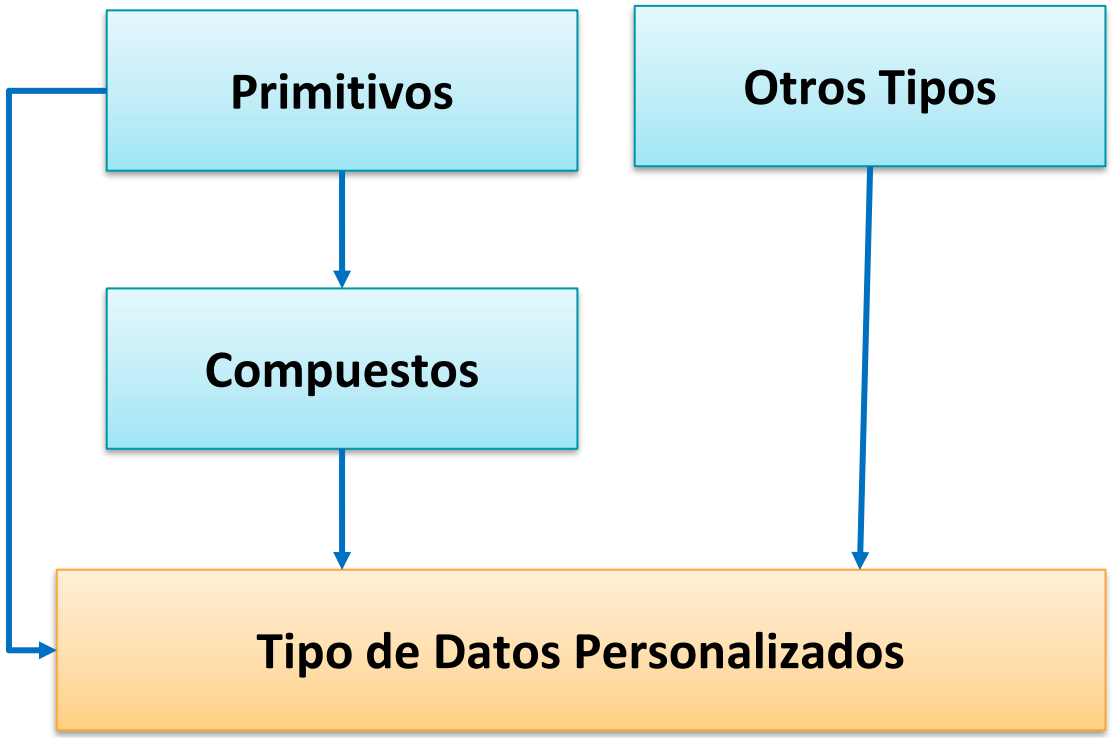
# Sistema de tipos

Es un **sistema lógico** compuesto por un **conjunto de reglas** que permite gestionar los Tipos de Datos (**TD**).

Su principales propósitos son:

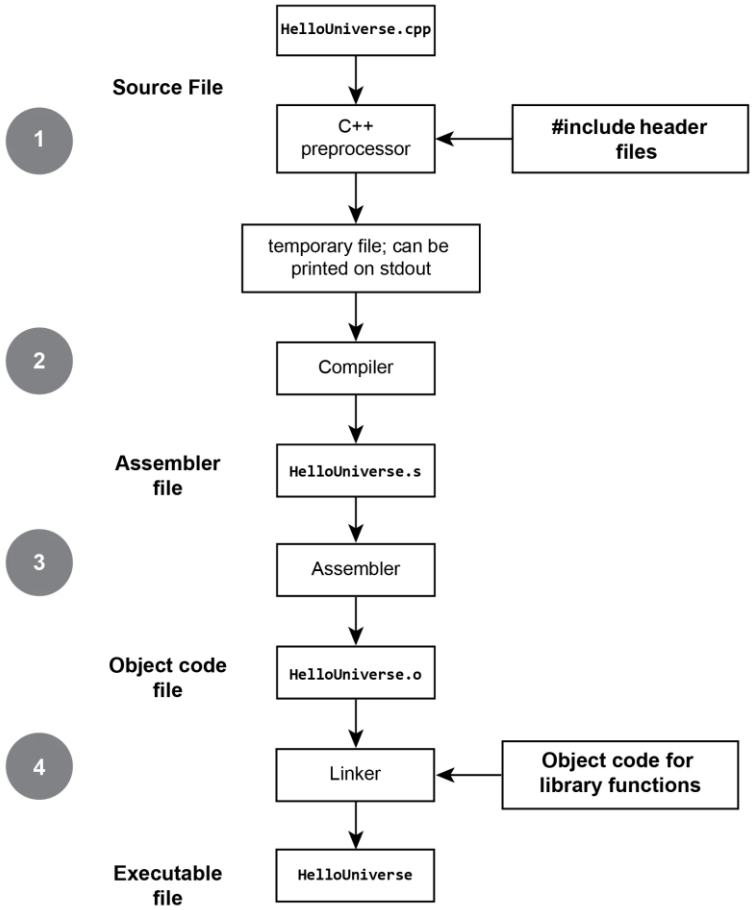
- Cumplimiento de las reglas gramaticales asociadas a los **TD**.
- Integridad de los datos y  detección temprana de errores.
- Desarrollar de nuevos tipos de datos y Mejora de Abstracción.

# Clasificación de los tipos de datos




- **Primitivos**, conocidos también como built-in, son los tipos básicos que provee el lenguaje: `int, float, bool, char`.
- **Compuestos**, derivados de los primitivos: `arreglos, registros, union, sets, clases, string, enumeraciones, vector, linked list`.
- **Otros tipos**, derivaciones de los tipos mencionados arriba: `punteros, referencias, funciones, procedimientos, lambdas.`
- **Tipos de Datos Personalizados**, tipos que no especifican su implementación: `stacks, queue, set, arboles, heap, hash, grafos y punteros inteligentes.`

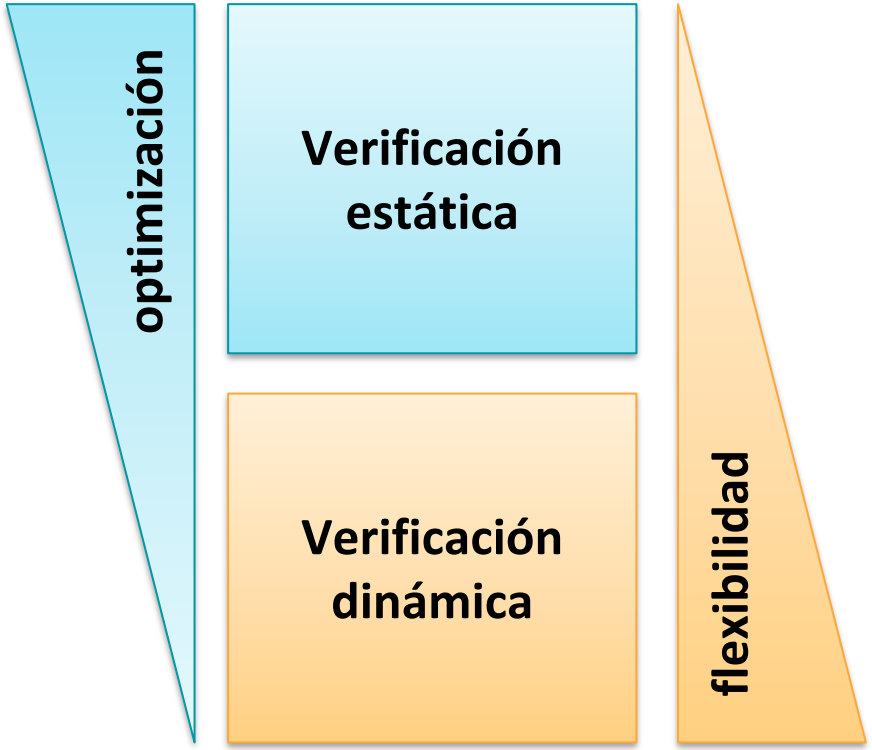
# Modelo de Compilación




1. Reemplaza en los archivos fuente (**`.cpp`**) las instrucciones **`#include <header>`** con el contenido de **`header`**.
2. Los archivos fuentes (**`.cpp`**) expandidos en **`1.`** son convertidos a archivos en lenguaje **ensamblador**.
3. Los **archivos ensamblador** son convertidos a **archivos binarios** (**`01001010`**).
4. Los **archivos binarios** de **`3.`** son enlazados junto a los archivos binarios de las librerías, identifica el **entry point** y generando **archivo ejecutable**.

***Fuente:*** *https://subscription.packtpub.com/book/programming/9781789801491/1/ch01lvl1sec03/the-c-compilation-model*

# Verificación Estática y Verificación Dinámica




Los Sistemas de tipos cuentan con 2 tipos de mecanismos de verificación que permiten una mejor protección de la integridad de los datos:

**Verificación estática**, analiza el código fuente (source code) al momento de **compilar**. Si el programa logra pasar esta verificación se considera que garantiza la consistencia del tipo.

**Verificación dinámica,** analiza el código al momento de **ejecutar**, verifica que al realizar una modificación del estado de una variable o atributo se garantice la consistencia de los datos, generalmente se origina por fuentes externas (I/O) y operaciones (side effects).

# Herramientas en C++

1. Tipos de datos primarios.
2. Tipos de datos compuestos.
3. Otros tipos.
4. Tipos Abstractos de Datos

# Tipos de datos primitivos

- `char`
- `bool`
- `int` (unsigned, long, long long)
- `float` (double, long double)
- `enum`

# Tipos de datos compuestos

- **cadena de caracteres** (`char*` y `const char*`)
- **arreglos** (built in)
- **class/struct**
- **templates (funciones y clases)**
- **contenedores de librería estándar** (`string, array, vector, list, set, map, etc.`)

# Otros tipos

- punteros
- referencias
- punteros inteligentes
- funciones, procedimientos y expresiones lambdas

# class y struct

Soporta las siguientes características principales:

- Abstracción, encapsulamiento, herencia y polimorfismo.
- Sobrecarga de operadores.
- 3 niveles de acceso: privado, público y protegido.
- 6 métodos especiales: 5 constructores y 1 destructor.

# Estructura de una clase

1. Constructores por default.
2. Constructores y operadores de copia, asignación y movimiento.
3. Constructores sintéticos.

**Además:**

1. Constructores delegados.
2. Constructores convertibles y por parámetros.

# Tipos de constructores

1. Constructores por default.
2. Constructores y operadores de copia, asignación y movimiento.
3. Constructores sintéticos.

**Además:**

1. Constructores delegados.
2. Constructores convertibles y por parámetros.

# Sobrecarga de operadores

1. Como función
2. Como método

# Relaciones friend

1. Entre clases
2. Entre una función y clase
3. Entre una clase y un método especifico de una clase

# Ejemplo

- Desarrollar una matriz polimórfica e implementar el siguiente código:

```
std::random_device rd;
utec::matrix m1(4, 5);
for (int i = 0; i < m1.row_size(); ++i) {
    for (int j = 0; j < m1.col_size(); ++j) {
        m1(i,j) = static_cast<int>(rd()) % 100;
    }
}
std::cout << m1;
utec::matrix m2 = m1;

for (int i = 0; i < m2.row_size(); ++i) {
    for (int j = 0; j < m2.col_size(); ++j) {
        m2(i,j) = static_cast<int>(rd()) % 100;
    }
}
std::cout << m1;
std::cout << m2;
```

# enum

Define un tipo de dato basado en valores discretos etiquetados que están restringidos a un rango de valores:

```
auto main() -> int {
    // definición de enum
    enum class Color { red, green = 20, blue };
    // uso del enum Color como tipo
    // inicializado con el valor blue
    Color r = Color::blue;
    switch(r) {
        case Color::red   : std::cout << "red\n";   break;
        case Color::green: std::cout << "green\n"; break;
        case Color::blue : std::cout << "blue\n";  break;
    }
    int n = static_cast<int>(r); // OK, n = 21
    return 0;
}
```

# namespaces

1. Define un nivel organización y encapsulamiento no protegido, mejora la composición y organización del código.

**Definición:**

```
namespace nombre {
          <variables>,
          <funciones>,
          <struct>,
          <class>, etc
}
```

**Importación:**

```
using namespace std;          // todo un namespace
using std::string;            // solo string en std

Using namespace ns;
Using ns::variables;
```

**Uso:**

```
std::string nombre;
std::cout << "Hola";
```

# Punteros

Un puntero en C/C++ se representa de la siguiente forma:

```
<Tipo de Dato>* <nombre del Puntero>;
```

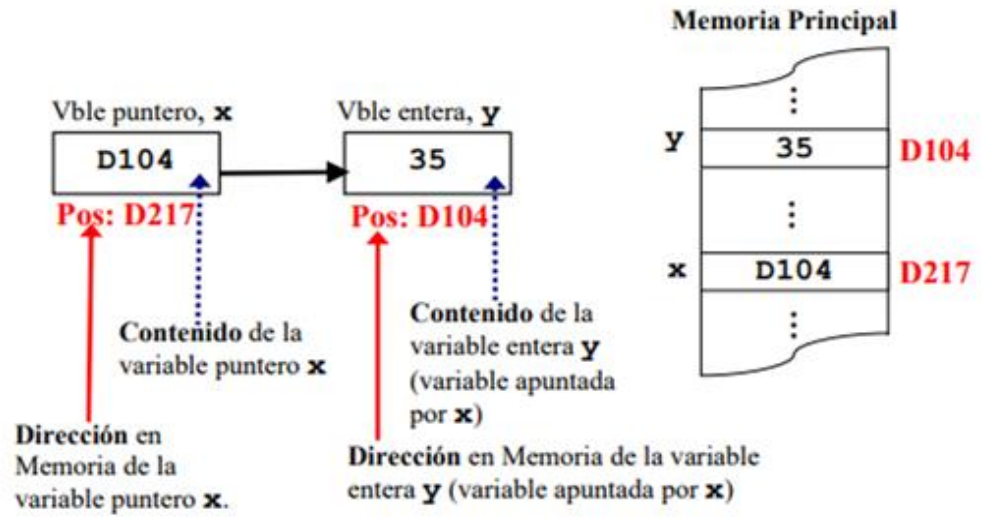
donde:

- **<Tipo de Dato>:** Representa uno de los tipo de dato válido en C/C++
- **<Nombre del Puntero>:** Expresión alfanumérica que sigue las mismas reglas de las variables en C/C++.

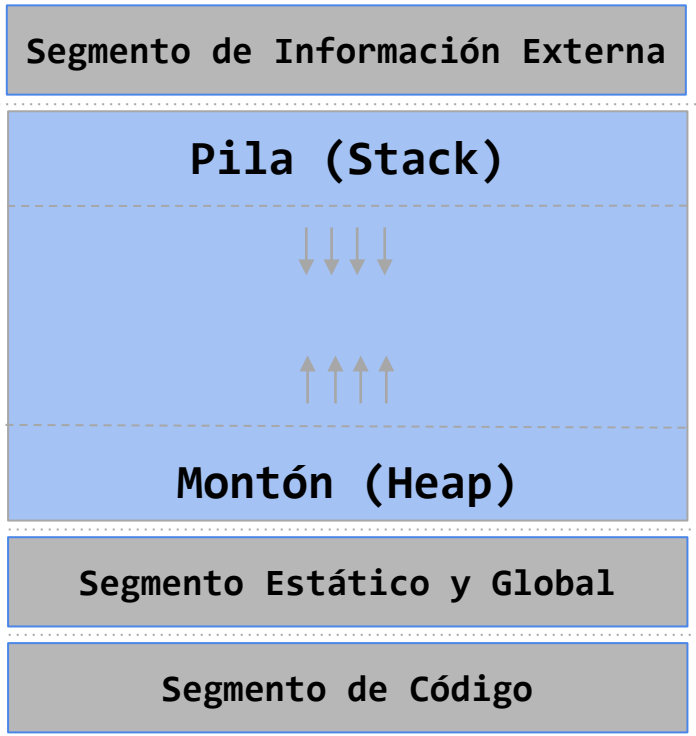
Ejemplo:

```
int* ptrEntero;
char* ptrCaracter;
```

```
int y = 35;
int* x = &y;
```



# Gestión de memoria - Stack / Heap




Argumento externos (argc, argv)

Memoria automática donde se asigna las variables estáticamente.

Memoria donde se asigna variables dinámicamente.

Memoria donde se asigna variables globales y tipo static.

Instrucciones del Programa.

# Gestión de memoria - Asignación/Libración en C++

- **Existen un operador que permiten asignar memoria dinámica:**
  - **new,** operador de uso genérico para asignación de memoria.

```
// ptr = new type; throw bad_alloc exception
double* ptr = new double;

// ptr = new (std::nothrow) type;
double* ptr = new (std::nothrow) double;
```

- **Existe sólo un operador (con sobrecarga) para la liberación:**
  - **delete,** para una colección de datos utiliza el modificador []

```
delete ptr;
// in case of ptr = new type[size]
delete [] ptr;
```

# Gestión de memoria - Asignación/Libración en C++

- **Existen un operador que permiten asignar memoria dinámica:**
  - **new,** operador de uso genérico para asignación de memoria.

```
// ptr = new type; throw bad_alloc exception
double* ptr = new double;

// ptr = new (std::nothrow) type;
double* ptr = new (std::nothrow) double;
```

- **Existe sólo un operador (con sobrecarga) para la liberación:**
  - **delete,** para una colección de datos utiliza el modificador []

```
delete ptr;
// in case of ptr = new type[size]
delete [] ptr;
```

# Archivos - Convenciones de Nombres de Archivo

En programación en C/C++ la extensión del nombre del archivo:

.cpp: C++ archivo fuente en sistemas Microsoft Windows.

.cc: C++ archivo fuente en sistemas Unix/Linux.

.h: C++ archivo de cabecera

.exe: archivo en formato ejecutable en sistemas Microsoft Windows.

.bin: archivo en formato binario. En sistemas Unix/Linux es por simple convención.

# Proceso de Compilación

El proceso de compilación incluye 4 etapas:

1. **Preprocesamiento**
2. **Generación de archivos assembler**
3. Compilación o generación de archivos binaries
4. Enlace o union de archivos binaries.

# Pasos para usar un Archivo

1. Abrir el Archivo
2. Usar (leer de, escribir a) el archivo
3. Cerrar el Archivo

# Objetos de Archivos Stream

- El uso de archivos requiere de objetos de archivo stream
- Existen tres tipos de objetos de archivo stream:

(1) `ifstream:` usado para leer
(2) `ofstream:` usado para escribir
(3) `fstream:` usado para ambos leer y escribir

# Nombre del Archivo

- El nombre del archivo puede ser el nombre de la ruta completa al archivo:

  `c:\datos\notas.dat` en Microsoft Windows

  `/usr/ubuntu/notas.dat` en linux

  esto le indica al compilador exactamente donde ubicarlo.

- El nombre del archivo puede ser también un nombre simple:

  `notas.dat`

  este debe estar en el mismo directorio del programa ejecutable, o en el directorio por defecto del compilador.

# ifstream: Abriendo un Archivo para Leer

- Crea un objeto **`ifstream`** en tu programa

```
ifstream archivoEntrada;
```

- Abre el archivo por pasar su nombre a la función miembro open del  objeto stream

```
archivoEntrada.open("notas.dat");
```

# ofstream: Abriendo un Archivo para Escribir

- Crea un objeto **`ofstream`** en tu programa

```
ofstream archivoSalida;
```

- Abre el archivo por pasar su nombre a la función miembro open del  objeto stream

```
archivoSalida.open("notas.dat");
```

# fstream: Archivo para Leer o Escribir

- El objeto **`fstream`** puede ser usado para Leer o Escribir

```
fstream archivoEntradaSalida;
```

- Para Leer se debe de especificar **`ios::in`** como el segundo argumento para abrir el archivo

```
archivoEntradaSalida.open("notas.dat",ios::in);
```

- Para Escribir se debe de especificar **`ios::out`** como el segundo argumento para abrir el archivo

```
archivoEntradaSalida.open("notas.dat",ios::out);
```

# Abriendo un Archivo para Leer y Escribir

- El objeto **`fstream`** puede ser usado para Leer y Escribir en el mismo tiempo

- Crea el objeto fstream y especifica ambos **`ios::in`** y **`ios::out`** como el segundo argumento para la función miembro open

```
fstream archivoEntradaSalida;
archivoEntradaSalida.open("notas.dat",
                          ios::in|ios::out);
```

# Abriendo Archivos con Constructores

- Incluir el **open**

```
fstream archivoEntrada("notas.dat",
ios::in);
```

# Modos de Abrir un archivo

- El modo de abrir un archivo especifica como el archivo es abierto y que se puede hacer con el archivo una vez abierto.

- **`ios::in`** y **`ios::out`** son ejemplos de modos de abrir un archivo, también llamado indicadores del modo de archivo

- Los modos de archivo pueden ser combinados y pasados como segundo argumento de la función miembro open

# Indicadores del Modo de Archivo

| | |
|---|---|
| **`ios::app`** | crea un nuevo archivo, o agrega al final de un archivo existente |
| **`ios::ate`** | va al final de un archivo existente; y puede escribir en cualquier parte del archivo |
| **`ios::binary`** | lee/escribe en modo binario (no en modo texto) |
| **`ios::in`** | abre para leer |
| **`ios::out`** | abre para escribir |

`app` seek to end before each write `ate` open and seek to end immediately after opening

With `ios::app` the write position in the file is "sticky" -- all writes are at the end, no matter where you seek.

# Modos por Defecto de Abrir un archivo

- **`ofstream`:**
  - abre solo para escribir
  - no puede ser leído el contenido del archivo
  - se crea el archivo si no existe
  - el contenido es borrado si existe el archivo

- **`ifstream`:**
  - abre solo para leer
  - no puede ser escrito el archivo
  - falla al abrir si el archivo no existe

# Detectando errores abriendo un Archivo

Dos métodos para detectar si falla al abrir un archivo

(1) Llamar a la función miembro **`fail()`** en el stream

```
archivoEntrada.open("notas.dat");
if (archivoEntrada.fail()) {
    cout << "No puede abrir el archivo";
    exit(1);
}
```

# Detectando errores abriendo un Archivo

(2) Verificando el estatus del stream usando el operador **!**

```
archivoEntrada.open("notas.dat");
if (archivoEntrada.is_open())
  { cout << "No se puede abrir el archivo";
    exit(1);
  }
```

# Usando fail() para detectar eof

Example de lectura de todos los enteros en un archivo

```
// intentando leer
int x;
archivoEntrada >> x;

while (!archivoEntrada.fail())
{  // Exitoso, no es un eof
   cout << x;
   // lee nuevamente otro entero
  archivoEntrada >> x;
}
```

# Usando >> para Detectar eof

- El operador de extracción retorna el mismo valor que será retornado por la siguiente llamada a fail:
  - (`archivoEntrada >> x`) es no cero
    si `>>` es exitoso
  - (`archivoEntrada >> x`) es cero
    si `>>` es fin de archivo

# Detectando el Final de un archivo

Leyendo todos los enteros en un archivo

```
int x;
while (archivoEntrada >> x)
{
    // la lectura fue exitosa
    cout >> x;
    // va al tope del bucle while e
    // intenta otra lectura
}
```

# Paradigmas de programación

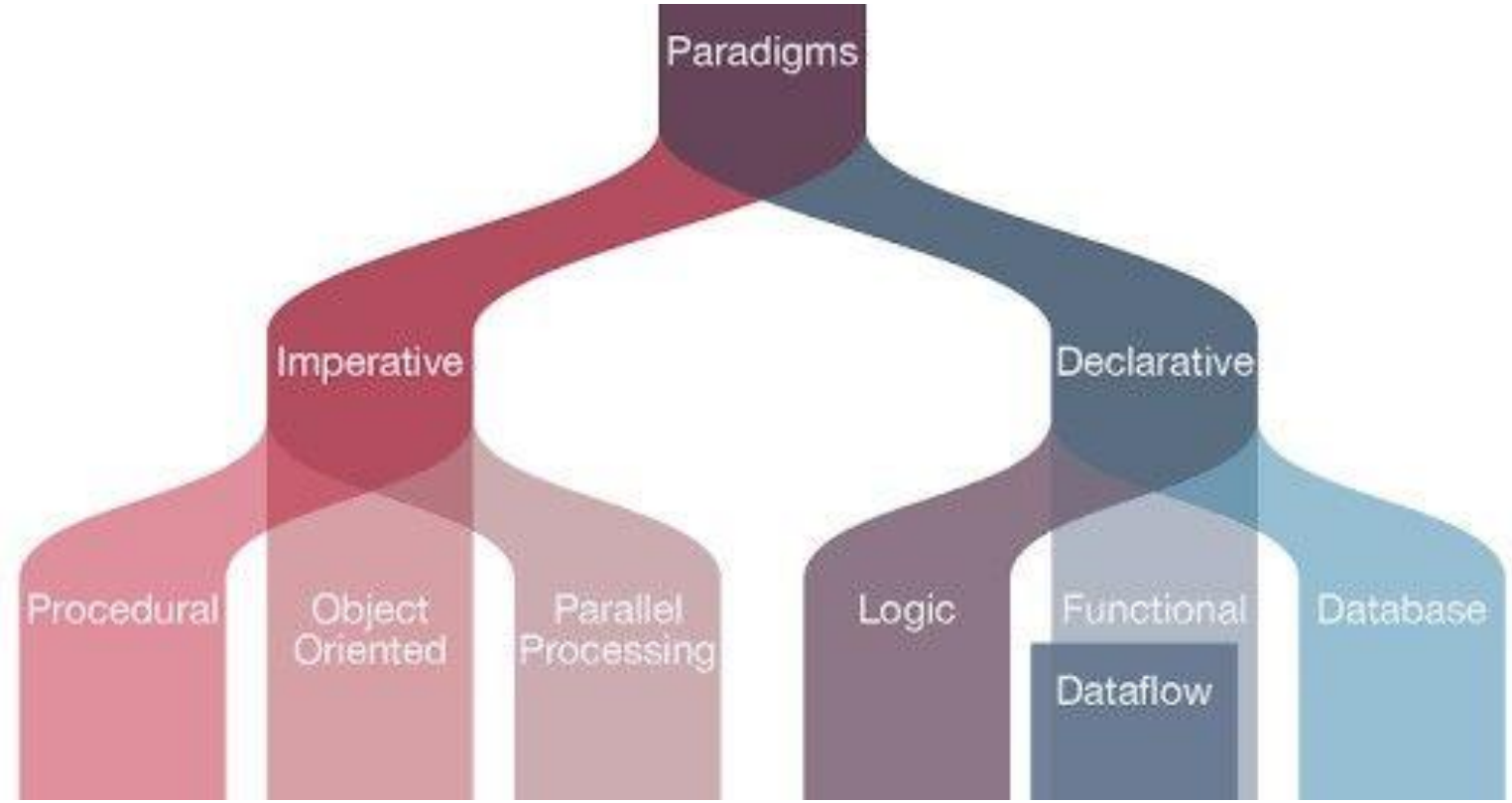
Un **paradigma** se define como:

- Un ejemplo que sirve de patrón o modelo.
- Un marco teórico y filosófico de una disciplina en el cual las teorías, leyes y generalizaciones y experimentos están definidos.

Un **paradigma de programación**:

- Es un patrón que sirve de escuela de pensamiento de programación.
- C++ es un lenguaje multiparadigma, en este curso se utilizará principalmente el paradigma Programación OO y el paradigma Programación Genérico.

# Paradigmas de programación



# Conclusiones

- En la programación se requieren de diversas herramientas para poder realizar representaciones abstractas a un problema.
- Los **datos** son clasificados en tipos y existen varias herramientas que ayudan a organizar y almacenarlos, C++ brinda varias herramientas que buscan aprovechar los recursos computacionales de forma eficiente.
- Los sistemas de tipos de datos de un lenguaje permiten reducir la cantidad de errores, extender los niveles de abstracción de un programa.
- Los paradigmas de programación son formas de abordar el modelamiento de la data y los algoritmos, C++ brinda algunos paradigmas que ayudan a tener una caja de herramientas amplia para la solución de diversos problemas.
- Los lenguajes de programación, no son un fin, son un medio.