ESPE

UNIVERSIDAD DE LAS FUERZAS ARMADAS

INNOVACIÓN PARA LA EXCELENCIA

Fundamentos de Programación

TypeDef y Plantillas

ESTRUCTURA DE DATOS

Ing. Washington Loza H. Mgs.

Departamento de Ciencias de la Computación

UNIDAD 1: INTRODUCCIÓN Y FUNDAMENTOS DE LAS ESTRUCTURAS DE DATOS

- Tipos de Datos Abstractos (TDA).
  - TypeDef y Plantillas
  - Sobrecarga de Operadores
  - Memoria Estática y Dinámica
- Recursividad
  - Caso Base y Recursivo
  - Principios y Tipos de Recursividad
  - Programa Recursivos
- Listas
  - Operaciones Básicas
  - Aplicaciones de Listas
- Pilas
- Colas

ESPE

UNIVERSIDAD DE LAS FUERZAS ARMADAS

INNOVACIÓN PARA LA EXCELENCIA

Uso de “Alias” o defincionesde TYPEDEF en C++

ESPE
UNIVERSIDAD DE LAS FUERZAS ARMADAS
INNOVACIÓN PARA LA EXCELENCIA

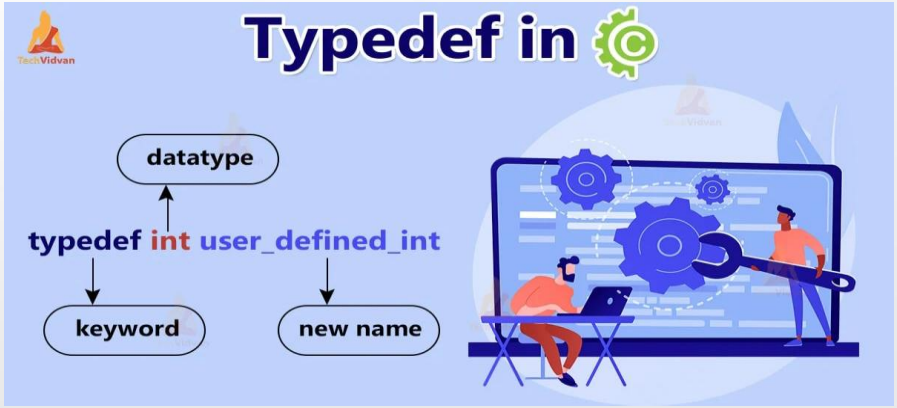
Uso de Typedef en C++

Generalidades

En la ingeniería de software, una de las metas fundamentales es lograr que los programas sean **claros, modulares, reutilizables y fáciles de mantener**. En el lenguaje C++, dos herramientas poderosas permiten alcanzar estos objetivos: **typedef** y las **plantillas (templates)**.

Según Joyanes Aguilar (2013), *"la programación moderna se basa en la abstracción de los datos y en la encapsulación de sus operaciones, de modo que el usuario manipule los datos a través de un contrato y no de su implementación"*.

Bajo este principio, **`typedef` y las plantillas** se convierten en los cimientos de la programación genérica orientada a objetos.



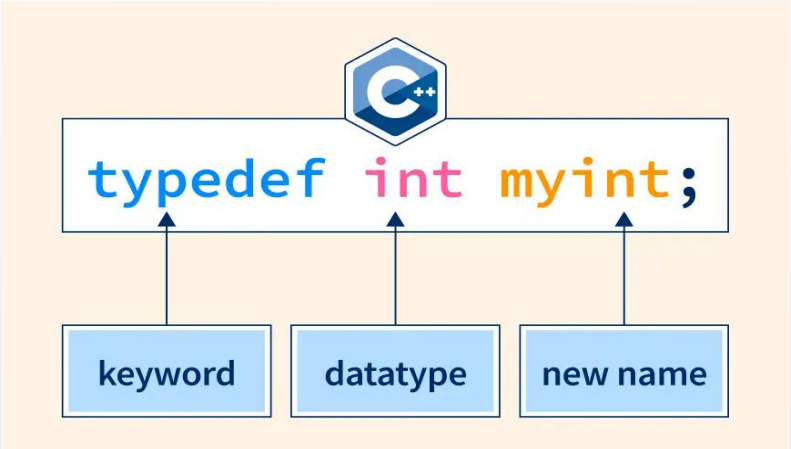
Uso de Typedef en C++

Concepto y Finalidad

El uso de **typedef** se orienta a mejorar la legibilidad del código mediante la creación de **nombres simbólicos o alias** que sustituyen tipos de datos complejos o de difícil interpretación.

Typedef es una palabra clave que se utiliza para **asignar un alias o nombre alternativo** a un tipo de dato existente. Este alias no crea un nuevo tipo de dato, sino que ofrece una manera más semántica y clara de referirse a un tipo ya conocido por el compilador.

Su objetivo principal es **aumentar la legibilidad**, **facilitar la comprensión del código** y **reducir el riesgo de errores**, especialmente cuando se trabaja con estructuras complejas, punteros o tipos genéricos.



Uso de Typedef en C++

Qué es y por qué usarlo?

Typedef no crea un tipo nuevo; **renombra** un tipo existente con un alias **semántico**. Esto:

- Reduce **ambigüedad** (se entiende la intención del dato).
- Centraliza **cambios** (p. ej., float→double en un único lugar).
- Estandariza el **vocabulario del dominio** (código, precio, stock).

Es por ello que se conoce como una técnica para **reducir complejidad declarativa** y **mejorar coherencia tipológica**.

Beneficios:

1. **Claridad semántica**: los nombres describen el propósito del dato.
2. **Estandarización**: todos los módulos del programa usan los mismos tipos definidos.
3. **Mantenibilidad**: si un tipo cambia (por ejemplo, de float a double), solo se modifica una línea.
4. **Reutilización en múltiples TDAs**: se pueden usar los mismos alias en diferentes estructuras.
5. **Documentación integrada en el código**: los alias comunican el significado del dominio sin necesidad de comentarios extensos.

Uso de Typedef en C++

Ejemplo:

Por ejemplo, en un sistema de inventario, escribir **unsigned int** para representar una cantidad o un código puede resultar poco expresivo. Con **typedef**, se puede dar un significado claro al tipo:

- **typedef** unsigned int CodigoProducto;
- **typedef** string NombreProducto;
- **typedef** float Precio;
- **typedef** unsigned int Cantidad;

De esta manera, se logra que el código exprese **intención** y **significado de negocio**. Esto tiene un valor enorme en ingeniería, pues transforma un código técnico en un modelo legible del mundo real, permitiendo al lector comprender fácilmente qué representa cada dato.

Uso de Typedef en C++

Ejemplo:

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

// Definición de alias con typedef
typedef unsigned int CodigoProducto;
typedef string NombreProducto;
typedef string Categoria;
typedef float Precio;
typedef unsigned int Cantidad;

// Definición del TDA Producto
struct Producto {
    CodigoProducto codigo;
    NombreProducto nombre;
    Categoria categoria;
    Precio precio;
    Cantidad stock;
};
```

```cpp
// Función para mostrar información de un producto
void mostrarProducto(const Producto& p) {
    cout << "Codigo: " << p.codigo << endl;
    cout << "Nombre: " << p.nombre << endl;
    cout << "Categoria: " << p.categoria << endl;
    cout << "Precio: $" << p.precio << endl;
    cout << "Stock: " << p.stock << " unidades" << endl;
}

int main() {
    Producto prod1 = {1001, "Laptop Lenovo", "Computadoras", 749.99, 25};
    mostrarProducto(prod1);
    return 0;
}
```

ESPE

UNIVERSIDAD DE LAS FUERZAS ARMADAS

INNOVACIÓN PARA LA EXCELENCIA

Plantillas de Funciones y Clases en C++

ESPE
UNIVERSIDAD DE LAS FUERZAS ARMADAS
INNOVACIÓN PARA LA EXCELENCIA

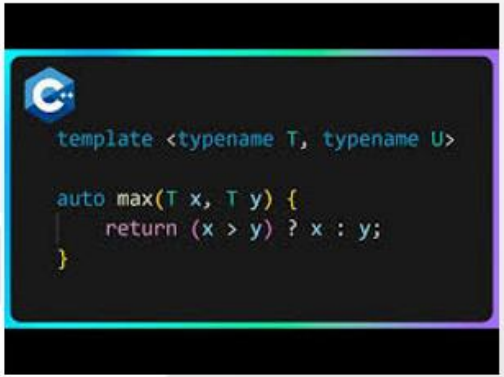
Uso de Plantillas en C++

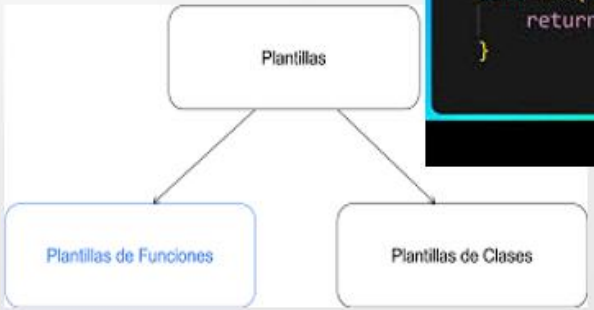
Generalidades

Las **plantillas** permiten **definir funciones y clases genéricas** que pueden operar con distintos tipos de datos sin reescribir el código, gracias a la capacidad de adaptación que ofrece la programación genérica.

Una **plantilla** es un **molde** de **función** o **clase** parametrizado por **tipo(s)**. El compilador genera (instancia) versiones concretas cuando se usa con tipos específicos.
*Joyanes lo presenta como la evolución natural de la **abstracción** hacia **algoritmos independientes del tipo** (2013).*

Una **plantilla** es una construcción que **genera un tipo o función** normal en tiempo de compilación **en función de los argumentos que proporciona el usuario** para los parámetros de la plantilla.





Plantillas de Funciones

Por qué se necesita plantillas?

Cuando diseñamos un **Tipo Abstracto de Datos (TDA)**, solemos usar tipos concretos: por ejemplo, `int` para edades, `float` para promedios o `string` para nombres.

Sin embargo, llega un momento en que queremos **reutilizar un mismo algoritmo o estructura** con distintos tipos de datos sin tener que escribir el mismo código varias veces.

```
int minInt(int a, int b);
float minFloat(float a, float b);
double minDouble(double a, double b);
```

- Cada función hace lo mismo: devuelve el menor de dos valores, solo cambia el tipo.
- Aquí es donde entra la **programación genérica** y las **plantillas (templates)**.

Uso de Plantillas en C++

Ventajas Claves:

1. **Reutilización**: el mismo código sirve para cualquier tipo.
2. **Reducción de errores**: se evita repetir la lógica para cada tipo de dato.
3. **Seguridad de tipos**: el compilador valida los tipos en tiempo de compilación.
4. **Eficiencia**: el código especializado se genera automáticamente durante la compilación.
5. **Flexibilidad**: se pueden construir estructuras que acepten diferentes tipos sin modificar su lógica base.

La sintaxis básica es:

```
template < typename T >
```

donde **T** representa un tipo genérico que será reemplazado por un tipo real durante la compilación.

Uso de Plantillas en C++

Ejemplo:

```
3   template <typename T>
4   T mayor(T a, T b) { return (a > b) ? a : b; }
5
6   int main() {
7       float precio1 = 15.75f, precio2 = 20.50f;
8       unsigned int s1 = 30, s2 = 45;
9
10      std::cout << "Precio mayor: " << mayor(precio1, precio2) << "\n";
11      std::cout << "Stock mayor: "  << mayor(s1, s2)           << "\n";
12  }
13
```

Este código implementa una **función plantilla** llamada **mayor**, que permite comparar dos valores de cualquier tipo y devolver el más grande.

La declaración **template <typename T>** indica que la función es genérica, por lo que el compilador **genera versiones** específicas **según el tipo de dato usado** (por ejemplo, float o unsigned int).

En el main, se comparan precios y cantidades usando la misma función, demostrando que **una única plantilla puede adaptarse a distintos tipos sin duplicar código**.

Uso de Plantillas en C++

Ejemplo:

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

// Plantilla genérica para obtener el valor mayor
template <typename T>
T mayor(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    float precio1 = 15.75;
    float precio2 = 20.50;
    unsigned int stock1 = 30;
    unsigned int stock2 = 45;

    cout << "Precio mayor: " << mayor(precio1, precio2) << endl;
    cout << "Stock mayor: " << mayor(stock1, stock2) << endl;

    return 0;
}
```

Uso de Plantillas en C++

Plantillas de Clases:

Las **plantillas de clases** extienden la idea de las funciones genéricas. Permiten definir **estructuras completas (TDAs)** que pueden operar con diferentes tipos de datos. Esto es especialmente útil en el diseño de **sistemas modulares** como inventarios, registros o catálogos.

En C++, las clases incluidas las **clases plantilla,** se dividen en **zonas de acceso** que determinan **quién puede ver o modificar sus atributos y métodos**. Estas **zonas** son fundamentales para mantener el principio de **encapsulamiento**, una de las bases de los **Tipos Abstractos de Datos (TDA)**.

```
1   template <typename T> // o template <class T>
2   class MiPlantilla {
3   private:
4       // Miembros privados que solo son accesibles dentro de la clase
5       T datoPrivado;
6
7   public:
8       // Miembros públicos que son accesibles desde fuera de la clase
9       MiPlantilla(T val) {
10          datoPrivado = val;
11      }
12
13      T obtenerDato() {
14          return datoPrivado;
15      }
16
17      void establecerDato(T val) {
18          datoPrivado = val;
19      }
20  };
```

Uso de Plantillas en C++

Miembros Privados de las Clases

Los miembros **privados** son aquellos **accesibles solo dentro de la propia clase** o por otras funciones o clases que hayan sido declaradas como **amigas (friend)**.

Estos elementos representan el **estado interno** del objeto y deben **protegerse del acceso directo** para evitar modificaciones indebidas o inconsistentes.

Por tanto, la sección private sirve para **encapsular** los datos, asegurando que el usuario solo interactúe con el objeto mediante operaciones controladas (métodos públicos).

Los miembros privados **no se pueden usar directamente** desde el `main()` ni desde otras clases, a menos que se accedan a través de **métodos públicos**.

```
1   template <typename T>
2   class NombreClase {
3   private:
4       // Atributos protegidos (estado interno)
5       T atributo1;
6       int atributo2;
7
8       // Métodos privados (funciones auxiliares)
9       void metodoInterno();
10  };
```

Uso de Plantillas en C++

Miembros Públicos de las Clases

Los miembros **públicos** son aquellos que pueden **accederse libremente** desde cualquier parte del programa.

Constituyen la **interfaz del TDA**, es decir, el conjunto de operaciones que el usuario puede ejecutar sobre los datos encapsulados.

Su objetivo es permitir que el usuario **interactúe de forma segura** con los datos internos sin violar las reglas de consistencia establecidas en la parte privada.

Los métodos y atributos definidos como **public** se pueden **invocar directamente** desde el **main()** o desde otras clases

```
1   template <typename T>
2   class NombreClase {
3   public:
4       // Constructor y destructor
5       NombreClase(T valor);
6       ~NombreClase();
7
8       // Métodos accesibles desde fuera
9       void insertar(T dato);
10      void mostrar() const;
11      bool estaVacio() const;
12  };
```

Uso de Plantillas en C++

Comparativo de Clases

Tipo de miembro	Accesibilidad	Rol principal	Ejemplo
Privado (private)	Solo desde dentro de la clase	Encapsula y protege los datos internos	precio, stock, validarPrecio()
Público (public)	Desde cualquier parte del programa	Define la interfaz del TDA	vender(), mostrar(), sinStock()

Uso de Plantillas en C++

Conclusión de Clases

El equilibrio entre **miembros privados y públicos** es esencial para el diseño correcto de una **clase plantilla (TDA)**.

- Los **privados** garantizan **seguridad y coherencia interna**, evitando modificaciones indebidas.
- Los **públicos** ofrecen una **interfaz clara y segura** para que otros módulos o usuarios interactúen con el objeto.

En conjunto, ambos refuerzan los pilares de la **programación orientada a objetos y la abstracción de datos**:

- **Ocultamiento:** protege los datos internos evitando accesos directos desde fuera de la clase.,
- **Modularidad:** divide el programa en partes independientes y fáciles de integrar.,
- **Reutilización:** permite usar el mismo código en diferentes contextos sin reescribirlo**;** y,
- **Mantenibilidad:** facilita actualizar o mejorar el código sin afectar su funcionamiento..

Uso de Plantillas en C++

Ejemplo:

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

// Alias con typedef
typedef string NombreProducto;
typedef string Categoria;
typedef float Precio;
typedef unsigned int Cantidad;

// Clase plantilla con miembros públicos (interfaz)
template <typename ID>
class Producto {
private:
    // Atributos protegidos (no accesibles desde fuera)
    ID codigo;
    NombreProducto nombre;
    Categoria categoria;
    Precio precio;
    Cantidad stock;

```

Uso de Plantillas en C++

Ejemplo:

```
21
22  public:
23      // ?? Constructor público
24      Producto(ID c, NombreProducto n, Categoria cat, Precio p, Cantidad s)
25          : codigo(c), nombre(n), categoria(cat), precio(p), stock(s) {}
26
27      // ?? Métodos públicos (interfaz del TDA)
28      void mostrar() const {
29          cout << "Codigo: " << codigo
30               << " | Nombre: " << nombre
31               << " | Categoria: " << categoria
32               << " | Precio: $" << precio
33               << " | Stock: " << stock << endl;
34      }
35
36      void vender(Cantidad cantidad) {
37          if (cantidad <= stock)
38              stock -= cantidad;
39          else
40              cout << "Error: stock insuficiente.\n";
41      }
42
43      void reponer(Cantidad cantidad) {
44          stock += cantidad;
45      }
46
47      bool sinStock() const {
48          return stock == 0;
49      }
50  };
```

Uso de Plantillas en C++

Ejemplo:

```
51
52  int main() {
53      Producto<int> p1(1002, "Teclado Mecanico", "Perifericos", 79.99f, 30);
54
55      // Uso de métodos públicos
56      p1.mostrar();
57      p1.vender(5);
58      p1.mostrar();
59
60      if (p1.sinStock())
61          cout << "El producto esta agotado.\n";
62      else
63          cout << "Aun hay unidades disponibles.\n";
64
65      return 0;
66  }
67
```

TypeDef y Plantillas

Actividad en clase 2.

1. ¿Cuál es la función principal del uso de typedef dentro de un TDA y cómo contribuye a la claridad semántica y mantenibilidad del código en C++?

2. Explique cómo las plantillas permiten la reutilización del código en C++ y describa un ejemplo en el que una función genérica sea más eficiente que una implementación tradicional.

3. ¿Por qué es importante combinar typedef y plantillas en el diseño de un TDA como “Producto”? Mencione los beneficios técnicos y de diseño que aporta esta integración.

4. En una clase plantilla, ¿qué diferencias existen entre definir miembros private y public, y cómo se relaciona esto con los principios de encapsulamiento y ocultamiento?

5. Describa cómo las plantillas de clases facilitan la creación de estructuras genéricas, indicando un ejemplo práctico de su uso en la gestión de inventarios.