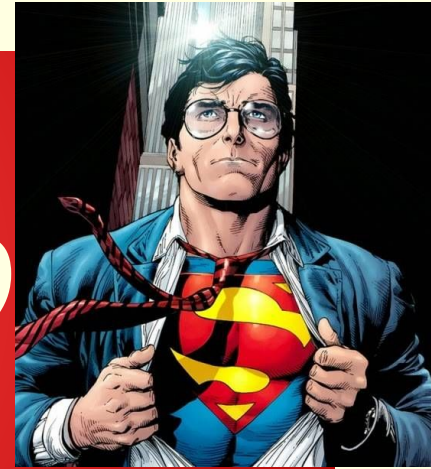


# POLIMORFISMO



Por: Yolanda Martínez Treviño  
Ma. Guadalupe Roque Díaz de León

# The address-of operator (&) The indirection operator (\*)

The indirection operator (\*) (also called dereference operator) allows us to access the value at a particular address:

```
#include <iostream>
// https://www.learncpp.com/cpp-tutorial/introduction-to-pointers/

int main(){
    int x{5};
    std::cout << x << '\n'; // print the value of variable x
    std::cout << &x << '\n'; // print the memory address of variable x
    std::cout << *(&x) << '\n'; // print the value at the memory
    address of variable x (parenthesis not required, but make it easier
    to read)

    return 0;
}
```

Note: Although the address-of operator looks just like the bitwise-and operator, you can distinguish them because the address-of operator is unary, whereas the bitwise-and operator is binary.

Note: Although the indirection operator looks just like the multiplication operator, you can distinguish them because the indirection operator is unary, whereas the multiplication operator is binary.

# Apuntadores == Pointers - Declaring a pointer

With the address-of operator and indirection operators now added to our toolkits, we can now talk about **pointers**. A **pointer** is a variable that holds a *memory address* as its value.

Pointers are typically seen as one of the most confusing parts of the C++ language, but they're surprisingly simple when explained properly.

Pointer variables are declared just like normal variables, only with an asterisk between the data type and the variable name. **Note that this asterisk is *not* an indirection.** It is part of the pointer declaration syntax.

```
int *iPtr{};    // a pointer to an integer value
double *dPtr{}; // a pointer to a double value
```

```
int* iPtr2{};    // also valid syntax
int * iPtr3{};   // also valid syntax (but don't do this, it looks like
multiplication)
```

```
// When declaring multiple variables in one line, the asterisk has to appear
// once for every variable.
int *iPtr4{}, *iPtr5{}; // declare two pointers to integer variables (not
recommended)
```

# Assigning a value to a pointer

Since pointers only hold addresses, when we assign a value to a pointer, that value has to be an address. One of the most common things to do with pointers is have them hold the address of a different variable.

To get the address of a variable, we use the address-of operator:

```
int v{ 5 };  
int *ptr{ &v }; // initialize ptr with address of variable v
```

Conceptually, you can think of the above snippet like this:



This is where pointers get their name from -- ptr is holding the address of variable value, so we say that ptr is "pointing to" v.

```
#include <iostream>

int main()
{
    int v{ 5 };
    int *ptr{ &v }; // initialize ptr with address of variable v

    std::cout << &v << '\n'; // print the address of variable v
    std::cout << ptr << '\n'; // print the address that ptr is holding

    return 0;
}
```

**The type of the pointer has to match the type of the variable being pointed to**

---

```
int value{ 5 };  
std::cout << &value; // prints address of value  
std::cout << value; // prints contents of value  
  
int *ptr{ &value }; // ptr points to value  
std::cout << ptr; // prints address held in ptr, which is &value  
std::cout << *ptr; // Indirection through ptr (get the value that ptr is  
pointing to)
```

# Member selection with pointers and references

<https://www.learncpp.com/cpp-tutorial/member-selection-with-pointers-and-references/>

It is common to have either a pointer or a reference to a **struct** (or **class**). As you learned previously, you can select the member of a struct using the **member selection operator** (.): **However, with a pointer, you need to use the arrow operator -> para tener acceso a los métodos de la clase dentro del contexto de clases:**

```
class Person{
    int age {};
    double weight{};
    string str();
};

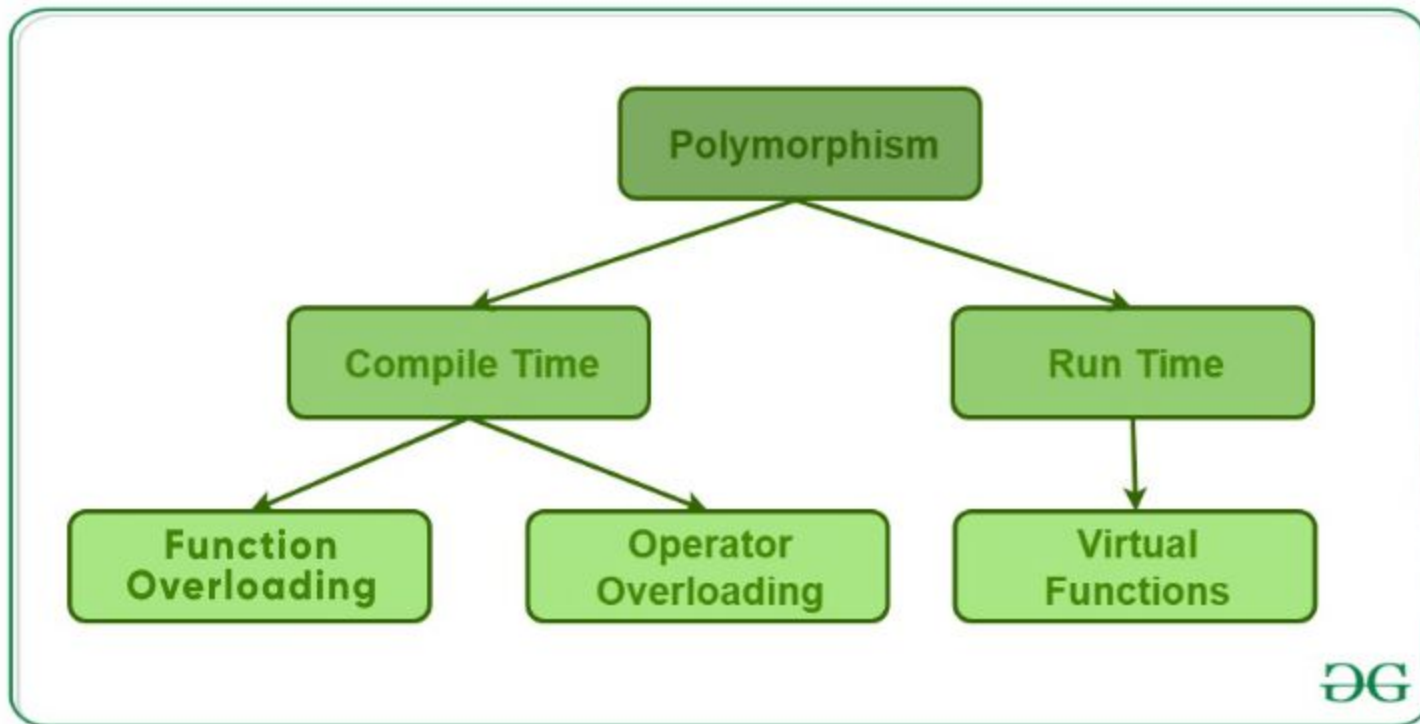
Person person{};

// Member selection using pointer to struct
Person *ptr{ &person };
ptr->age = 5;
cout << ptr->str( ) << '\n';
```

when doing member access through a pointer, always use the -> operator instead of the . operator.

**Rule:** When using a pointer to access the value of a member, use operator-> instead of operator. (the . operator)

# Polimorfismo





# Polimorfismo

---

- Llegamos ahora a los conceptos más sutiles de la programación orientada a objetos.
- La **virtualización** de funciones permite implementar una de las propiedades más potentes de POO: **el polimorfismo**. Pero vayamos con calma...
- En una **clase derivada** se puede definir una función que ya existía en la **clase base**, esto se conoce como "**overriding**".
- La definición de la función en la clase derivada **oculta** la definición previa en la clase base.
- En caso necesario, es posible usar la función oculta de la clase base mediante su nombre completo:
  - **<clase\_base>::<método>;**

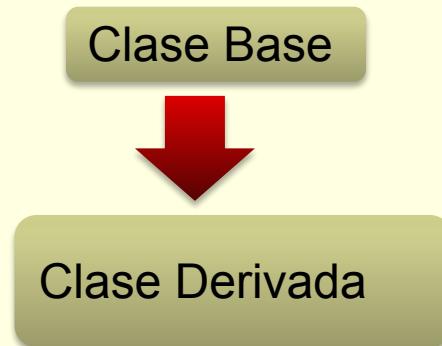
# Poliformismo

---

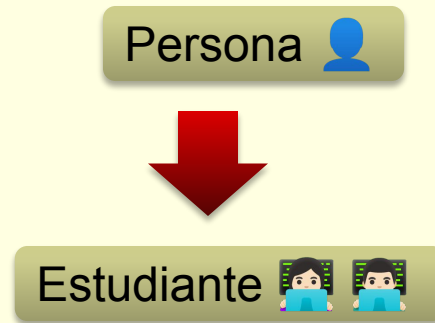
- En lo que concierne a clases, el polimorfismo en C++, llega a su máxima expresión cuando se usa junto con apuntadores.
- C++ permite acceder a objetos de una **clase derivada** usando **un puntero a la clase base**. En esa capacidad es posible el **polimorfismo**.
- Pero 😬, sólo podremos acceder a datos y funciones que existan en la **clase base**, los datos y funciones propias de los objetos de **clases derivadas serán inaccesibles**. 🔒

# Apuntador de la clase base apuntando a objeto de la clase derivada.

- Es posible tener variables de tipo **apuntador** para almacenar objetos.
  - Para declarar un **apuntador** se utiliza el siguiente formato:  
tipo \*nombreVariableApuntador;
  - Una variable que puede guardar una dirección de memoria se llama **apuntador**.
- Un objeto de la clase derivada **es un** objeto de la clase base (pero con más atributos y métodos)
- Una variable **apuntador** de la **clase base** puede apuntar 🖱 a un objeto de la clase derivada 🖱

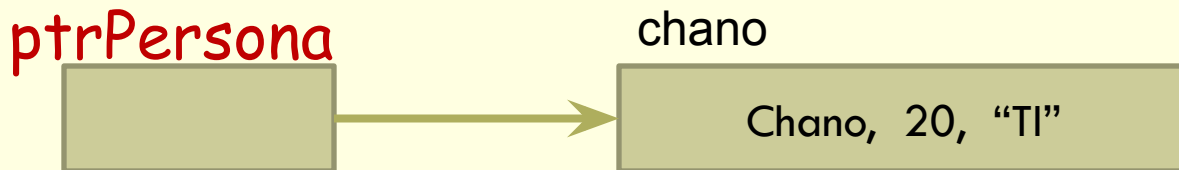
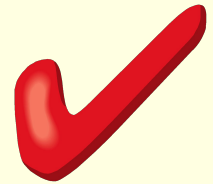


# Apuntador de la clase base apuntando a objeto de la clase derivada.



Ej:

```
Persona *ptrPersona;  
Estudiante chano("Chano", 20, "TI");  
ptrPersona = &chano;
```



# Variable apuntador - guarda 1 dirección de memoria: Sintaxis - `tipo *ptrNombre;`

```
Persona *ptrPersona, chanoPersona("Chano", 80);
```

```
Estudiante *ptrEstudiante;
```

```
Estudiante chaveloEstudiante("Chavelo", 90, "Dr.");
```

El operador `&` se utiliza para obtener la dirección de una variable, sintaxis `&variable`:

```
ptrPersona = &chanoPersona;
```

```
ptrEstudiante = &chaveloEstudiante;
```

¿Qué almacenan? almacenan la dirección de memoria

```
ptrPersona(chanoPersona):0x7ffd9368b108
```

```
ptrEstudiante(chaveloEstudiante):0x7ffd9368b080
```

# Variable apuntador - guarda 1 dirección de memoria: Sintaxis - **tipo** \***ptrNombre**;

---

```
cout << ptrPersona <<
```

```
ptrPersona->str()          Chavelo, 90
```

```
ptrEstudiante->str()      EstudianteChavelo, 90, Dr.
```

```
PtrPersona(chaveloEstudiante):0x7ffd9368b080
```

```
PtrPersona(chaveloEstudiante)->str():Chavelo, 90
```

# Ejemplo: Clase Estudiante

```
#include <iostream>
#include <string>
using namespace std;
#include "Persona.h"
class Estudiante : public Persona
{
public:
    Estudiante();
    Estudiante(std::string, int,
std::string);
    std::string getCarrera();
    void setCarrera(std::string);
    std::string str();
private:
    std::string carrera;
};
```

```
Estudiante::Estudiante(string nom, int ed, string
    ca) : Persona(nom, ed){
    carrera = ca;
}

string Estudiante::getCarrera(){
    return carrera;
}

void Estudiante::setCarrera(string ca){
    carrera = ca;
}

string Estudiante::str(){
    return "\nNombre: " + nombre + "\nEdad: " +
        to_string(edad) + "\nCarrera: " + carrera;
}
```

[https://repl.it/@roque\\_itesm\\_mx/Poliformismo](https://repl.it/@roque_itesm_mx/Poliformismo)

# Ejemplo

```
#include "Estudiante.h"  
#include "Medico.h"  
int main() {
```

```
// Una variable que puede guardar una dirección de memoria se  
// llama apuntador. Sintaxis: tipo *ptrNombre;
```

```
Persona *ptrPersona, chanoPersona("Chano", 80);  
Estudiante *ptrEstudiante, chaveloEstudiante("Chavelo", 90, "Dr.");
```

```
// El operador & se utiliza para obtener la dirección de una  
// variable, usa la siguiente sintaxis:
```

```
ptrPersona = &chanoPersona;  
ptrEstudiante = &chaveloEstudiante;  
cout << "\nPtrPersona(chanoPersona)->str():" << ptrPersona->str() << endl;  
cout << "\nPtrEstudiante(chaveloEstudiante)->str():" << ptrEstudiante->str() << endl;
```







```
// Ahora el apuntador persona se le asigna la dirección de un objeto estudiante  
// sin embargo el objeto al almacenarlo en un apuntador de la clase Base - este  
// objeto se comporta como si fuera de la clase Base-  
// La funcionalidad depende del tipo de apuntador, no del tipo del objeto -  
// Por lo tanto solo se podrán llamar a los métodos de la clase Persona
```

```
ptrPersona = &chaveloEstudiante;  
cout << "\nPtrPersona(chaveloEstudiante):" << ptrPersona << endl;  
cout << "\nPtrPersona(chaveloEstudiante)->str():" << ptrPersona->str() << endl;
```

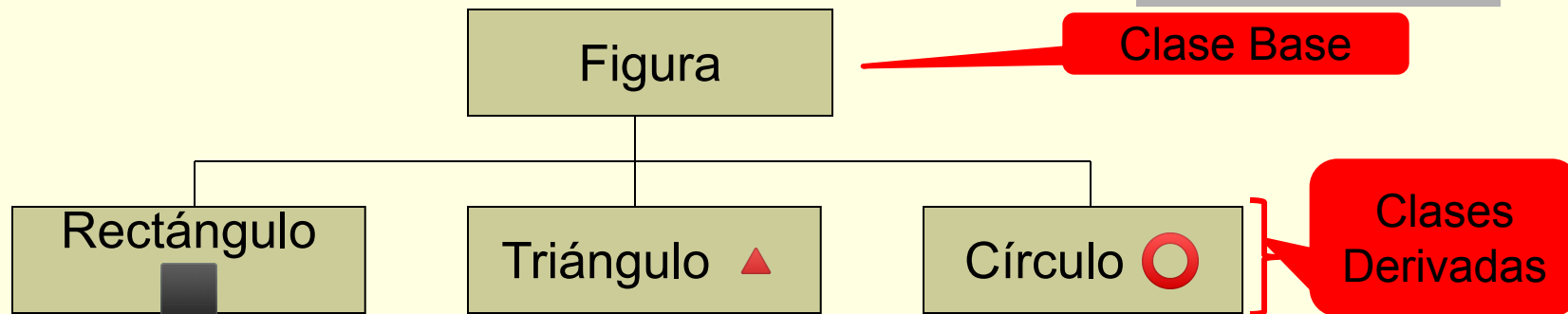
👁️ que el objeto chaveloEstudiante guardado en el apuntador de tipo Persona ptrPersona ejecutó el método str() de la clase Persona






# En resumen








- Al almacenar un objeto de la **clase derivada** en un apuntador a la **clase base**, el objeto se comporta **como si fuera de la clase base**, por eso se ejecutó el método **str( )** de la clase **Persona**.
-  Persona
-   
- Medico(is-a) Estudiante (is-a)
- Es decir, **la funcionalidad depende del tipo del apuntador, no del tipo del objeto.**
- Entonces como el apuntador es a un objeto de la clase **Persona** , solo se puede hacer referencia a los datos y llamar a los métodos que se heredaron de la clase **Persona** .

# Funciones Virtuales



- Supón que tenemos una aplicación que permite dibujar figuras en la pantalla, las figuras pueden ser Rectángulos , Círculos , Triángulos , etc.
- En la aplicación tenemos un arreglo de Figuras, y para **dibujar** solo se tuviera que mandar llamar al método **dibuja()** de cada uno de los objetos almacenados en el arreglo. 🤖
- Sería útil que al llamar al método **dibuja()** de cada elemento del arreglo se ejecutara el **dibuja()** del objeto correspondiente 🤖, no el **dibuja()** de la clase **Figura**.

# Funciones Virtuales

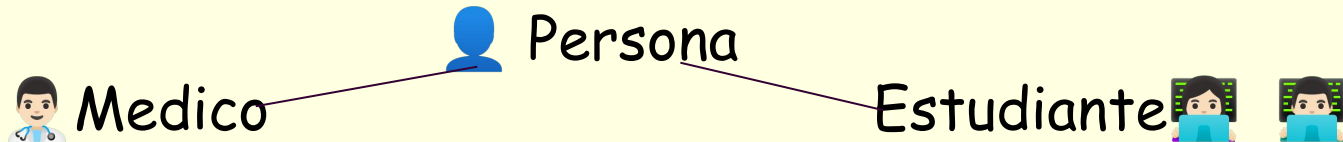
- Pero de acuerdo con lo que vimos en el ejemplo de la clase **Persona**  y **Estudiante**  :
  - Si tenemos un arreglo de objetos de tipo **Figura**
  - Solo se podrá tratar a dichos objetos como **Figura**, no como **Rectángulo** , **Círculo** , etc;
  - Por lo que **no** es posible que cada **objeto** del arreglo llame  a su método **dibuja( )** que le corresponde, sino que siempre se llama  al método **dibuja( )** de la clase **Figura**.

¿Cómo resolver esto? 

Para eso existen las **funciones virtuales**  

# Funciones Virtuales

- Al usar una **función virtual**, el tipo del objeto apuntado, **es el que determina cuál versión de la función va a utilizar** - la de la clase base o la de la clase derivada.



- La sintaxis de función es **virtual**, añadir la palabra "virtual" en la función dentro de la declaración de la clase base.

Por ejemplo:

```
virtual string str( );
```

# Ejemplo Funciones Virtuales

Veamos el ejemplo de la clase **Persona** y **Estudiante**, se cambió solamente la declaración del método **str( )** de la clase **Persona** de esta forma: 🙌

**virtual string str();**

[https://repl.it/@roque\\_itesm\\_mx/Funciones Virtuales](https://repl.it/@roque_itesm_mx/Funciones_Virtuales)

# Otro ejemplo: Funciones virtuales

```
int main() {  
    Persona *ptrPersona, chanoPersona("Chano", 80), chonitaPersona;  
    Estudiante *ptrEstudiante, chaveloEstudiante("Chavelo", 90, "Dr.");  
    Medico gattelMedico("Hugo Lopez",81,"Infectologo",0), manuelMedico;  
  
    // Arreglo de apuntadores de tipo Persona 👤  
    // El operador & se utiliza para obtener la dirección de un objeto  
    Persona *arrPersonas[ ]={&chanoPersona, &chaveloEstudiante, &chonitaPersona,  
        &gattelMedico, &manuelMedico};  
  
    ptrPersona = &chaveloEstudiante;  
    cout << "\nPtraPersona(chaveloEstudiante)->str()" << ptrPersona->str() << endl << endl;  
  
    // Arreglo de apuntadores de tipo Persona 👤, 🧑, 🧑, 🧑, 🧑  
    // Ahora si se ejecuta el método correspondiente a la clase del objeto  
    for (int iR = 0; iR < 5; iR++)  
        cout << arrPersonas[iR]->str() << endl << endl;  
    return 0;  
}
```

Ahora Si se ejecuta el método str() de la clase que corresponde con el tipo del objeto, no con el tipo del apuntador.

# Funciones Virtuales

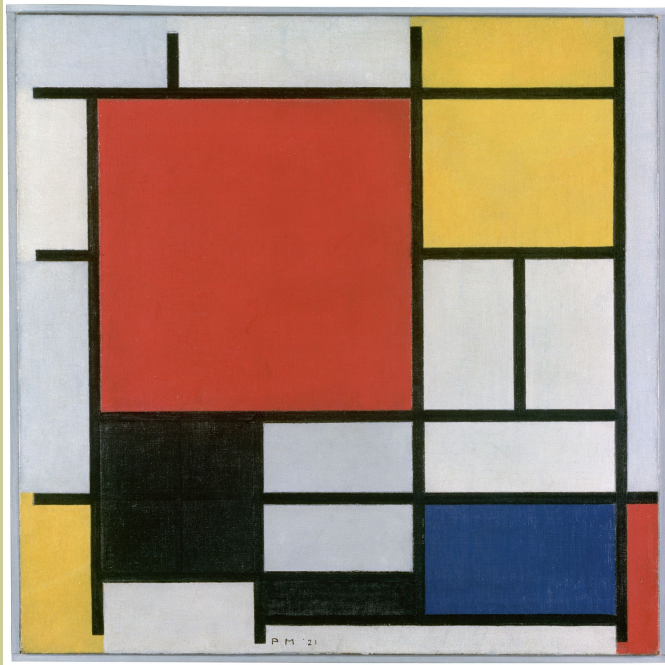
- 🙄 los ejemplos, al declarar una **función virtual**, la función a la que se llama depende del **tipo del objeto** y no del **tipo del apuntador**.
- Para utilizar una función virtual de manera que cada objeto llame a la función que le corresponda, **solamente se puede hacer con variables de tipo apuntador a objetos**.

# Polimorfismo

---

El término **polimorfismo** se refiere al hecho de que se llama al mismo método(nombre) para diferentes objetos y cada objeto ejecuta el método que le corresponde





## Piet Mondrian [1872-1944]


Composición con gran área roja, amarilla, negra, gris y azul.

<http://www.narodnimuzej.rs/?lang=en>

Clases Abstractas - tienen  
función virtual pura o función  
abstracta.

# Clases Abstractas

26

- Hasta ahora, **todas las funciones virtuales** que hemos escrito tienen un cuerpo - una codificación -.
- C++ permite crear un tipo especial de función virtual llamada **función virtual pura** o **función abstracta** que **no tiene cuerpo, no tienen una codificación en .cpp!** 
- Una **función virtual pura** actúa como una **función prototipo obligatoria** que debe ser redefinida(**overriding**) por las clases derivadas.
- Para crear una **función virtual pura**, solo se le asigna el valor 0 en la sección de declaración dentro del archivo **.h o .hpp**

■ Sintaxis:

**virtual void dibuja() = 0 ;**

← **Función Virtual Pura**

# Clases Abstractas

27



■ Sintaxis: **virtual void dibuja() = 0;**

- Una **Clase Abstracta** es una clase para la que no se van a definir objetos; porque como veremos más adelante **una clase abstracta está incompleta**.
- Entonces **las clases derivadas deben definir las “piezas faltantes”** para ser **clases concretas** de las que sí se pueden definir objetos.

```
class Alimento{
public:
    string ingrediente1;
    string ingrediente2;
    int precio;
    Alimento();
    // Funcion virtual pura - funcion abstracta
    virtual void dibuja() = 0;
};
```

Función Virtual Pura - Abstracta



```
class Pastel : public Alimento{
private:
    string relleno;
public:
    Pastel();
    void dibuja();
};
```


```
class Pizza: public Alimento{
private:
    string ingredienteExtra;
public:
    Pizza();
};
```



# Clases Abstractas

29

- Hay un par de cosas a tener en cuenta:
  - 1o. La función **dibuja ()** es ahora una **función virtual pura - abstracta**. Esto significa que **Alimento** ahora es una **Clase base abstracta** y no se puede crear una instancia.
  - 2o Debido a que nuestra clase **Pizza** es derivada de **Alimento**, pero si no definimos void dibuja() , **Pizza**  también será una clase abstracta . Ahora cuando intentamos compilar este código:

El compilador nos dará una advertencia porque **Pizza**  es una **Clase abstracta** y no podemos crear instancias de clases abstractas.

# Clases Abstractas

30

Una **función virtual pura** es útil cuando tenemos una función que queremos poner en la clase base, pero solo las clases derivadas saben lo que debería devolver o realizar 🧐.

Una **función virtual pura** hace que la **clase base no se pueda instanciar**, y las clases derivadas se ven obligadas a definir estas funciones antes de que se puedan instanciar.

✅ Esto ayuda a garantizar que las clases derivadas no olviden 😞 redefinir las funciones que la clase base esperaba que hicieran.




# Clases Abstractas

31

- Cuando añadimos una función virtual pura a nuestra clase, estamos diciendo, "**depende de las clases derivadas implementar esta función**".
- La declaración de una **función virtual pura** tiene dos consecuencias principales:
  - 1o. Cualquier clase con una o más **funciones virtuales puras** se convierte en una **Clase base abstracta**, lo que significa que **no se puede instanciar**.
  - 2o. **Todas las clases derivadas deben definir un cuerpo para esta función**, o de lo contrario **✗** esa clase derivada también se considerará una **Clase abstracta**.

# Clases Abstractas

32

- El ejemplo más común para explicar el concepto de clase abstracta es el ejemplo de la clase base **Alimento** y las clases derivadas **Pastel** , **Pizza** , **Corundas** , etc.
- La clase base tiene datos comunes para todos los Alimentos: ingrediente1, ingrediente2, precio, etc.
- Pero la clase base **no puede** implementar el método **dibuja()**, porque no es posible, **¿Cómo se dibuja un Alimento?**
- Cuando se declaran las clases derivadas, cada clase define los atributos y métodos específicos para ese tipo específico de Alimento.
- Y ahora sí, la clase **Pizza** debe tener un método **dibuja( )**, para dibujar una pizza.



# Clases Abstractas

33

- Para declarar una **clase Abstracta** se debe declarar una o más de sus **funciones virtuales** como **pure** de la siguiente manera:

**virtual void dibuja() = 0;**

- Y no se codifica la implementación del método `dibuja()`.
- La razón es para “**obligar**” a que las clases derivadas implementen dicho método.

# Headers de las Clases

```
class Alimento{
public:
    string ingrediente1;
    string ingrediente2;
    int precio;
    Alimento();
    // Funcion virtual pura - funcion abstracta
    virtual void dibuja() = 0;
};
```

← Función Virtual Pura - Abstracta

```
class Pastel : public Alimento{
private:
    string relleno;
public:
    Pastel();
    void dibuja();
};
```

```
class Pizza: public Alimento{
private:
    string ingredienteExtra;
public:
    Pizza();
    void dibuja();
};
```



# Implementación de las clases.

---

```
Alimento::Alimento(){  
    ingrediente1 = "Queso";  
    ingrediente2 = "Salami";  
    precio = 200;  
}
```

```
Pastel :: Pastel():Alimento(){  
    relleno = "Mermelada de Fresas de Irapuato";  
}  
  
void Pastel :: dibuja(){  
    cout << "🍓🍰🍓" << endl;  
}
```

```
Pizza :: Pizza(): Alimento(){  
    ingredienteExtra = "Trufa";  
}  
  
void Pizza :: dibuja(){  
    cout << "🍕🍕🍕🍕" << endl;  
}
```

```
int main(int argc, const char * argv[]) {  
    Pastel pastelitoBimbo, oreo, pastelMango, pastelFresa;  
    Pizza pizza, pizza2, pizza3;  
    // Una clase abstracta no se puede instanciar  
    Alimento comida; ❌
```

Una clase Abstracta no  
se puede instanciar



❗ Variable type 'Alimento' is an abstract class

```
    Alimento *ptrArrAlimentos[] = {  
        &pastelitoBimbo, &pizza, &pastelMango, &pastelFresa, &pizza3, &oreo,  
        &pizza2, &pizza };  
  
    Alimento *ptrArrAlimentos2[] = {  
        new Pastel(), new Pizza(), new Pastel(), new Pizza(), new Pizza(), new  
        Pizza(), new Pizza(), new Pizza()};  
  
    // arreglo de apuntadores de la clase Base, al momento de ejecutarse  
    // se ejecuta el metodo correspondiente al tipo del objeto apuntado  
    // no al tipo del apuntador  
    cout << "**** Alimentos Grupo2 ****" << endl;  
    for(int iR = 0; iR < 8; iR++)  
        ptrArrAlimentos[iR]->dibuja();  
  
    cout << "**** Alimentos Grupo3 ****" << endl;  
    for(int iR = 0; iR < 8; iR++)  
        ptrArrAlimentos2[iR]->dibuja();  
    return 0;  
}
```