

Metodologías y Técnicas de Programación II

Programación Orientada a Objeto (POO) C++

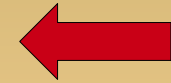
El proceso de compilación
Constantes

Estado del Programa

Introducción a la POO

Historia de la Programación
Conceptos de POO

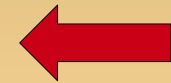
C++
Mi primera Clase



Repaso de Conceptos

Estándares de Programación
Punteros y Memoria

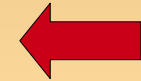
E/S
Control y Operadores



Clases y Objetos en C++

Uso y aplicación
Constructores
Constantes e "inline"

Funciones Amigas
Sobrecarga de Funciones



Sobrecarga

De Operadores

Creación Dinámica de Objetos

Herencia.

Tipos de Visibilidad

Herencia Múltiple

Polimorfismo

Funciones Virtuales

Polimorfismo y Sobrecarga.

Plantillas

Contenedores

Iteradores

9.1 Repaso de Sesiones Anteriores

Inicialización y Limpieza de Objetos

El constructor es una función sin tipo de retorno y con el mismo nombre que la estructura. **Alumno()**

Podemos usar inicializadores en los constructores

El destructor tiene la misma forma, salvo que el nombre va precedido el operador "~". **~Alumno()** Lo definimos cuando hay punteros

Sobrecarga de Funciones

La sobrecarga de funciones y los argumentos por defecto ayudan a escribir menos código y a hacerlo más legible.

```
Pareja(int n);  
Pareja(int a, int b);
```

```
Pareja(int a, int b=-1);
```

Recordemos que una función es un nombre que le ponemos a una zona de memoria en la que hay unas instrucciones para el procesador.

Una variable (objeto) también es un nombre de una zona de memoria en la que hay unos datos.

9.1 Repaso de Sesiones Anteriores

Uso de Sobrecarga y Argumentos por defecto

```
struct punto3D
{
    float x, y, z;
};

class punto
{
public:
    punto(float xi, float yi, float zi=0)
        : x(xi), y(yi), z(zi) {}
    punto(punto3D p)
        : x(p.x), y(p.y), z(p.z) {}
    void Asignar(float xi, float yi, float zi)
    {
        x = xi;
        y = yi;
        z = zi;
    }
};
```

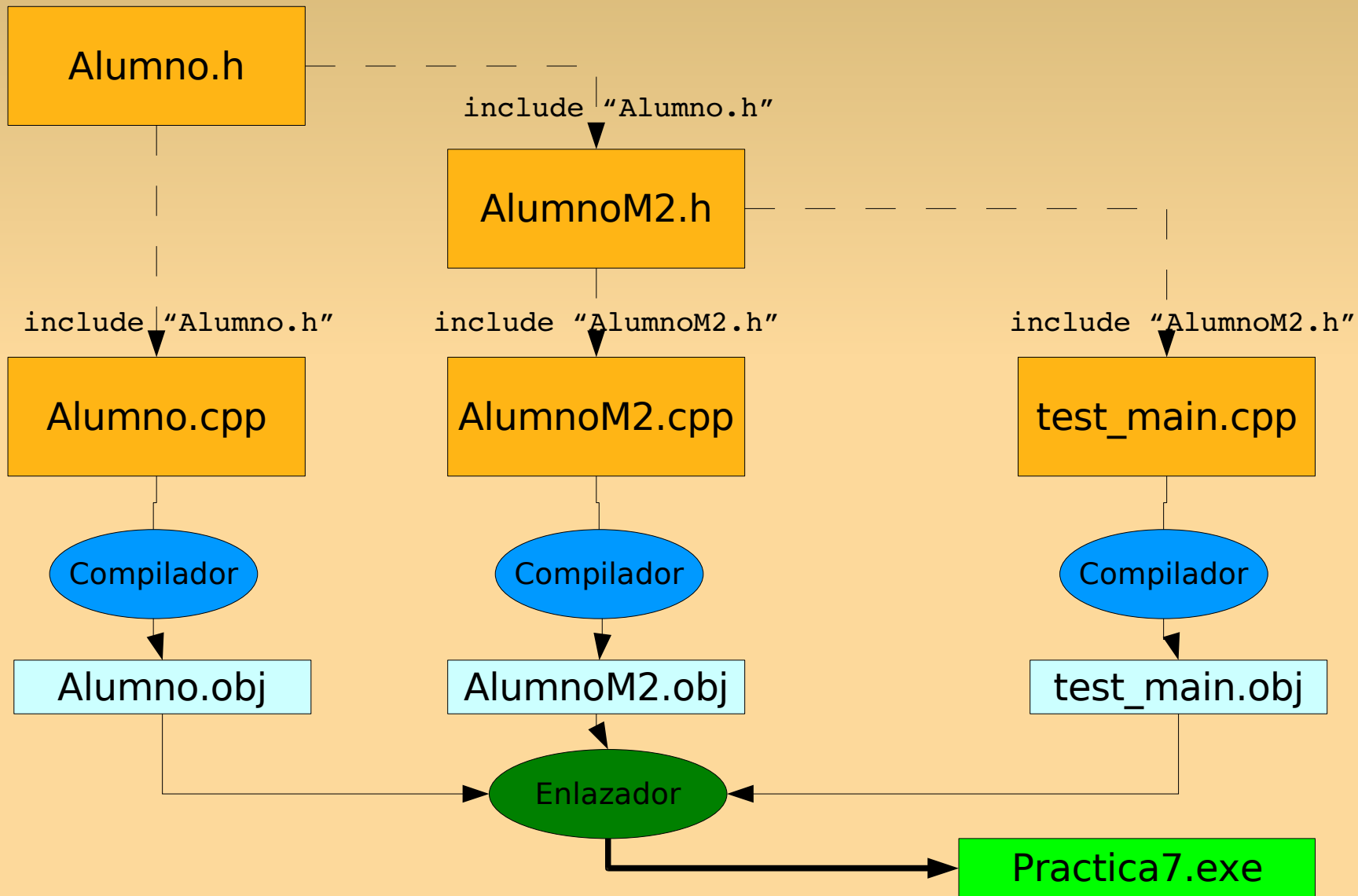
```
void Asignar(punto3D p)
{
    Asignar(p.x, p.y, p.z);
}

void Ver()
{
    cout <<"(" <<x<<"," << y
        <<"," <<z<< ")" << endl;
}

private:
    float x, y, z;
};
```

9.2 Compilación y Enlazado

Compilación y Enlazado por separado (repasso)



9.2 Constantes

Orígenes de *const*

La palabra reservada “const” se incorpora a C++ primero y posteriormente a C.

Se ha utilizado para varios propósitos a lo largo del tiempo, y esto puede generar cierta confusión al principio.

En esta sesión queremos aclarar:

- ¿Cuándo usar ***const***?
- ¿Por qué usar ***const***?
- ¿Cómo usar ***const***?

El motivo unicial para usar *const* fue para eliminar el uso de la directiva de compilación `#define` para sustitución de valores.

const se usa para punteros, argumentos de funciones, retornos de funciones, objetos y métodos.

9.2 Constantes

Sustitución de Valores

```
#define NOTA_ESPERADA 8
```

El preprocesador hace un reemplazamiento textual y no realiza ninguna comprobación de tipo. (Antes de compilar)

La sustitución de valores introduce pequeños problemas que pueden ser evitados usando valores const.

NOTA_ESPERADA no contiene información sobre el tipo que esperamos, y el compilador no puede comprobarla. Esto puede dar lugar a *errores difíciles de encontrar* (poco reutilizable).

const ayuda a eliminar estos problemas.

Podemos usar const con los tipos básicos (char, int, float y double) y sus variantes así como con clases como veremos.

```
const float nota_esperada = 8;  
  
const int cantidad = 19;  
Alumnos curso[cantidad];
```

9.2 Constantes

Constantes seguras

Además de para sustituir valores podemos usar *const* para asegurarnos de que una variable cuyo valor sabemos que no va a cambiar durante su vida en nuestro programa.

El compilador producirá un mensaje de error (o mensaje de ayuda).

```
const int i = 100;      // Constante Típica
const int j = i + 10;  // Valor constante calculado como expresión constante
long address = (long)&j; // Obliga al compilador a almacenar j.
char buf[j + 10];     // Pero j sigue siendo una expresión constante.

int main()
{
    cout << "Introduzca un caracter, por favor:";
    const char c = cin.get(); // Inicializamos de algo que no sabemos a priori
    const char c2 = c + 'a';  // Constante de expresión constante.
    cout << c2;
    // MAL:-> c2 << cin;
}
```

9.2 Constantes

Arrays (vectores)

Cuando aplicamos *const* a un array normalmente no nos ahorramos espacio en memoria.

“Conjunto de datos en memoria que no pueden modificarse”

```
const int i[] = { 1, 2, 3, 4 };  
//! float f[i[3]]; // iMAL!  
struct S { int i, j; };  
const S s[] = { { 1, 2 }, { 3, 4 } };  
//! double d[s[1].j]; // iMAL!  
  
int main() {}
```

9.2 Constantes

Punteros

Hay dos opciones cuando aplicamos *const* a un puntero:

- Queremos que sea constante lo que apunta el puntero
- Queremos que se constante la dirección del puntero.

Puntero a algo constante:

```
// "p es un puntero que apunta a:  
//      un entero constante.  
const int* p;  
const int *p;  
  
// Esta forma es igual, pero mejor  
// no la usamos. Confunde.  
// "p es un puntero a:  
//      un entero que es constante.  
int const* p;
```

¿Cómo será si lo que queremos es sea el puntero el constante.

9.2 Constantes

Punteros

Puntero constante:

Osea que su valor – que es una dirección - no puede cambiar.

```
// "w es un puntero constante que apunta a:  
//      un entero.  
int i=3;  
int* const w = &i;  
int * const w = &i;  
  
*w = 2; // ¿Está bien o está mal?  
w = &j; // ¿Y esto?
```

Podemos combinar punteros constantes que apuntan a elementos constantes:

```
int d = 1;  
const int* const x = &d; // (1)  
int const* const x2 = &d; // (2) Mejor usamos la forma (1)
```

9.2 Constantes

Argumentos de Funciones

En el caso de que pasemos los argumentos por valor (se hace copia de los datos) no tiene sentido para el cliente utilizar *const*. Dentro de la función tiene el significado: “El argumento no se puede cambiar”.

```
void funcion(const int i)
{
    i++; // Ilegal
}
```

```
int f3() { return 1; }
const int f4() { return 1; }
.....
    const int j = f3(); // Bien.
    int k = f4();      // Bien también
.....
```

Retorno por valor constante

En este caso se está diciendo que el valor de la variable original (en el ámbito de la función) no se modificará. Como lo está devolviendo por valor, es la copia lo que se retorna, de modo que el valor original nunca se podrá modificar.

```
const int g();
```

Devolver por valor como constante se vuelve importante cuando se trata con tipos definidos por el programador.

9.2 Constantes

Devolviendo valores constantes

Si devolvemos un tipo básico lo mejor es no utilizar const (no aporta mucho).

Si una función devuelve un objeto por valor como constante, el valor de retorno de la función no puede ser un recipiente.

```
class X
{
    int i;
public:
    X(int ii = 0);
    void modify();
};
X::X(int ii) { i = ii; }
void X::modify() { i++; }
X f5()
{
    return X();
}
```

```
const X f6()
{
    return X();
}
void f7(X& x) { // Pass by non-const
reference
    x.modify();
}
f5() = X(1); // BIEN
f5().modify(); // BIEN
// No compilan:
//! f7(f5());
// Causes compile-time errors:
//! f6() = X(1);
```

9.2 Constantes

Objetos Temporales

Como ya hemos visto en determinadas situaciones se crean objetos temporales que se construyen y se destruyen.

Estos objetos no se ven, los crea y los destruye el compilador según sus necesidades. Repasad en la sesión 7 el programa `cuantos.cpp`.

Los objetos temporales siempre se crean como constantes.

Hacer algo que cambie un temporal es casi seguro un error.

(*) Revisa el código anterior teniendo esto en cuenta...

```
f5() = x(1);  
f5().modify();
```

Estas sentencias son aceptables para el compilador pero son problemáticas. `f5()` devuelve algo de la clase `X`, pero en cuanto se evalúa la expresión el objeto devuelto desaparece.

Probablemente un código así será erróneo.

9.2 Constantes

Paso y retorno de direcciones

Si pasamos o retornamos una dirección (puntero o referencia), el programador cliente puede recoger y modificar el valor original.

```
void t(int*) {}
void u(const int* cip)
{
    //! *cip = 2; // ??
    int i = *cip; // ??
    //! int* ip2 = cip; // ??
}
const char* v()
{// La dirección de un arr. estático
  return "result of function v()";
}
const int* const w()
{
  static int i;
  return &i;
}
```

```
int main()
{
  int x = 0;
  int* ip = &x;
  const int* cip = &x;
  t(ip); // ??
  //! t(cip); // ??
  u(ip); // ?? u() es más general
  u(cip); // ?? que t() !!
  //! char* cp = v(); // ??
  const char* ccp = v(); // ??
  //! int* ip2 = w(); // ??
  const int* const ccip = w(); // ??
  const int* cip2 = w(); // ??
  //! *w() = 1; // ??
} //Mira página 219 de Pensar en C++
```

9.2 Constantes

Recomendación de paso de argumentos en C++

Antes de existir las referencias la única forma de:

- modificar algo dentro de una función era usar un puntero.
- proteger algo de ser tocado dentro era usar paso por valor (costoso).

En C++ lo mejor es utilizar una referencia constante en lugar de paso por valor:

- Para el programador cliente es lo mismo.
- No hay confusión con los punteros.
- Para el creador es más eficiente pasar una dirección que un objeto entero.

Además es posible pasarle un objeto temporal a una función que recibe una referencia constante. Con un puntero no podemos.

Veamos un ejemplo:

9.2 Constantes

Paso de Argumentos mediante referencias

```
class X {};  
X f() { return X(); }           // Retorno por valor  
void g1(X&) {}                  // Paso por referencia no-constante  
void g2(const X&) {}           // Paso por referencia constante  
int main()  
{  
    g1(f());    // ¿Bien o mal? Piensa cómo son los objetos temporales.  
    g2(f());    // ¿Bien o mal? ¿eh?  
}
```

f() retorna un objeto de la clase X por valor. Esto significa que cuando tome el valor de retorno y lo pase inmediatamente a otra función como en las llamadas a g1() y g2(), se crea un temporal y los temporales son siempre constantes.

Por eso, la llamada a g1() es un error pues g1() no acepta una referencia constante, mientras que la llamada a g2() si es correcta.

9.2 Constantes

Uso de *const* y clases

En la lista de inicialización del constructor.

La inicialización de la lista ocurre antes de ejecutar el cuerpo del constructor ({.....}).

Las constantes se deben inicializar en el punto en que se crean, y al llegar al cuerpo del constructor debe estar ya inicializada:

Ok, utilizamos la lista de inicialización del constructor.

```
class RecienNacido
{
    const int peso_;
public:
    RecienNacido(int peso);
    void imprimir();
};

RecienNacido::RecienNacido(int peso) : peso_(peso) {}
void RecienNacido::imprimir() { cout << peso_ << endl; }

RecienNacido a(1), b(2), c(3);
a.imprimir(), b.imprimir(), c.imprimir();
```

9.2 Constantes

Objetos y Métodos Constantes

Un objeto constante se define del mismo modo para un tipo definido por el usuario que para un tipo básico del lenguaje.

```
const int i = 1;  
const Alumno a(2);
```

Para que el compilador imponga que el objeto sea constante, debe asegurar que el objeto no tiene atributos que vayan a cambiar durante el tiempo de vida del objeto.

Si durante la definición de la función se modifica algún miembro o se llama algún método no constante, el compilador emitirá un mensaje de error.

Por eso, está garantizado que los miembros que declare *const* se comportarán del modo esperado.

Recordad que colocar *const* delante de la declaración del método indica que el valor de retorno es constante.

Hay que colocar el especificador *const* después de la lista de argumentos.

9.2 Constantes

Objetos y Métodos Constantes

```
class X
{
    int i_;
public:
    X(int i);
    int f() const;
};

X::X(int i) : i_(i) {}
int X::f() const { return i_; }
int main()
{
    X x1(10);
    const X x2(20);
    x1.f();
    x2.f();
}
```

La palabra `const` debe incluirse tanto en la declaración como en la definición del método.

Como `f()` es un método constante, si intenta modificar de alguna forma o llamar a otro método que no sea constante, el compilador informará de un error.

Un miembro constante puede llamarse tanto desde objetos constantes como desde no constantes de forma segura.

Un método que no modifica ningún atributo se debería escribir como constante y así se podría usar desde objetos constantes.

9.3 Resumen

Constantes

La palabra reservada para definir constantes es **const**

const nos permite definir como constantes:

- Variables.
- Objetos.
- Argumentos de funciones.
- Punteros.
- Métodos.

Elimina el uso de constantes simbólicas con 'define' para sustitución de valores por el preprocesador sin perder sus ventajas, añadiendo comprobación de tipos y haciendo nuestro código más seguro.

El uso de la llamada “constancia exacta” (const correctness) es decir, el uso de const en todo lugar donde sea posible, puede ser un salvavidas para muchos proyectos.