

# Clases

## Algoritmos y Estructuras de Datos II

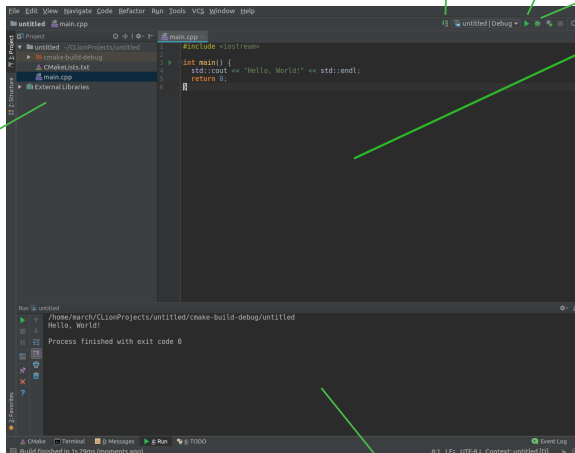
# Ambiente de desarrollo

## CLion

Compilar Ejecutar Debug

Editar

Lista de Archivos



Output ejecución

## CLion

Bajar trial en: <https://www.jetbrains.com/clion/download/>

Crear una cuenta de estudiante con cuenta @dc.uba.ar

Licenciarlo con código de activación una vez loggeado con la cuenta en [www.jetbrains.com](http://www.jetbrains.com)

## Alternativa

- ▶ Editar código: atom, sublime, vim
- ▶ Compilar: CMake + Makefile
- ▶ Ejecutar: consola
- ▶ Debuggear: gdb (valgrind)

# Scopes

- ▶ Los scopes definen el alcance de una declaración. Esto es, donde son visibles.
- ▶ En C++ los scopes se definen por un par de llaves.
- ▶ Los scopes con nombre se pueden acceder mediante ::

```

namespace NS {
    int ns1 = 1;
    int ns2 = ns1 + 1;
}

int global1 = 2;
// int global2 = ns1 + 1;
int global2 = NS::ns2 = 1;

int foo() {
    int foo1 = global1 + 1;
    {
        int foo2 = 2;
    }
    // int foo3 = foo2 + 1;  error: 'foo2' was not declared in this scope
}

```

# Clases

Las clases en C++ es la herramienta que tenemos para encapsular información y comportamiento.

## Ejemplo: Horario

Un horario en un calendario debe tener hora, minutos y segundos. La hora debe ser un número entre 0 y 23 y los minutos y segundos entre 0 y 59.

¿Representación?

# Clases

Las clases en C++ es la herramienta que tenemos para encapsular información y comportamiento.

## Ejemplo: Horario

Un horario en un calendario debe tener hora, minutos y segundos. La hora debe ser un número entre 0 y 23 y los minutos y segundos entre 0 y 59.

¿Representación?

```
int hora; int min; int seg;
```

# Classes

```
int segsEntre(int h1, int m1, int s1,  
              int h2, int m2, int s2);  
  
int segsEntre(int horario1[3], int horario2[3]);
```



# Classes

```
struct Horario {  
    int hora;  
    int min;  
    int seg;  
};
```

```
int segsEntre(Horario h1,  
              Horario h2);
```

```
int main() {  
    Horario h1;  
    h1.hora = 10;  
    h1.min = 14;  
    h1.segs = 32;  
  
    Horario h2;  
    h2.hora = 15;  
    h2.min = 0  
    h2.segs = 22;  
  
    segsEntre(h1, h2);  
}
```

```
Horario masTarde(Horario horarios[]);
```

## Clases: Inicialización

```
int main() {  
    Horario h1;  
    h1.hora = 15;  
    h1.min = 30;  
    h1.segs = 27;  
    cout << h1.hora << endl; // 15  
  
    Horario h2 = h1;  
    cout << h2.min << endl; // 30  
  
    Horario h3;  
    cout << h3.min << endl; // ??  
}
```

## Clases: Inicialización

```
Horario init_horario(int hora, int min, int segs) {  
    Horario h;  
    h.hora = hora;  
    h.min = min;  
    h.segs = segs;  
    return h;  
}  
  
int main() {  
    Horario h1 = init_horario(15, 30, 27);  
    cout << h1.hora << endl; // 15  
  
    Horario h3;  
    cout << h3.min << endl; // ??  
}
```

# Clases: Constructor

- ▶ Los constructores son funciones especiales para inicializar un tipo nuevo.
- ▶ Se escriben con el nombre del tipo.
- ▶ Cuando se define un constructor desaparece el constructor por defecto (sin parámetros).

```
struct Horario {  
    Horario (int _hora, int _min, int _secs);  
  
    int hora, min, secs;  
}
```

```
Horario::Horario(int _hora, int _min, int _secs) {  
    hora = _hora % 24;  
    min = _min % 60;  
    secs = _secs % 60;  
}
```

# Classes: Constructor

```
int main() {  
    Horario h1 = Horario(15, 30, 27);  
    cout << h1.hora << endl; // 15  
  
    Horario h2(16, 44, 7);  
    cout << h2.min << endl; // 44  
  
    // Horario h3; // error: no matching function for  
    ↪ call to 'Horario::Horario()'  
    // cout << h3.min << endl; // ??  
}
```

# Clases: Visibilidad

```
int main() {  
    Horario h1 = Horario(30, 30, 27);  
    cout << h1.hora << endl; // 6  
  
    cout << h1.hora = 30;  
    cout << h1.min << endl; // 30  
}
```

# Clases: Visibilidad

Podemos utilizar definir que partes de nuestro tipo son visibles desde afuera usando `private` y `public`. En tipos declarados con `class`, se asume por defecto `private`.  
En la materia vamos a usar `class` en lugar de `struct`.

# Clases: Visibilidad

```
class Horario {  
    private:  
        int hora, min, segs;  
  
    public:  
        Horario(int _hora, int _min, int _segs);  
  
}
```

```
Horario::Horario(int _hora, int _min, int _segs) {  
    hora = _hora % 24;  
    min = _min % 60;  
    segs = _segs % 60;  
}
```

```
int main() {  
    Horario h1 = Horario(30, 30, 27);
```

```
    // h1.hora = 30; // error: 'int Horario::hora' is private  
}
```



# Clases: Métodos

Necesitamos una forma controlada de exportar información interna del tipo. Los métodos nos permiten agregar comportamiento.

Los métodos son funciones internas del tipo que tienen acceso a la parte privada de la instancia mismo.

```
class Horario {  
    private:  
        int hora, min, segs;  
  
        int dame_hora();  
        int dame_min();  
        int dame_segs();  
  
    public:  
        Horario(int _hora, int _min, int _segs) { ... }  
};  
  
int Horario::dame_min() {  
    return min;  
}  
  
int Horario::dame_hora() {  
    return hora;  
}  
  
int Horario::dame_segs() {  
    return segs;  
}
```

## Clases: Métodos

```
int main() {  
    Horario h1 = Horario(30, 30, 27);  
    Horario h2 = Horario(22, 13, 5);  
  
    cout << h1.dame_segs() << endl; // 27  
  
    cout << h2.dame_segs() << endl; // 5  
}
```

# Clases: Métodos

Los métodos también pueden modificar el estado interno de una instancia.

Supongamos que queremos usar el Horario para un timer que baja de a segundos.

# Clases: Métodos

```
class Horario {  
    private:  
        int hora, min, segs;  
  
    public:  
        Horario(int _hora, int_min, int _segs);  
        int dame_hora();  
        int dame_min();  
        int dame_segs();  
  
        void restar_seg();  
};
```

## Clases: Métodos

```
void Horario::restar_seg() {  
    if (segs > 0) {  
        segs--;  
    } else if (segs == 0 and (min > 0 or hora > 0)) {  
        segs = 59;  
        if (min == 0 and hora > 0) {  
            min = 59;  
            hora--;  
        } else {  
            min--;  
        }  
    }  
}
```

## Clases: Métodos

```
int main() {  
    Horario h1(0, 0, 30);  
    Horario h2(0, 1, 50);  
  
    cout << h1.dame_segs() << endl; // 30  
    cout << h2.dame_segs() << endl; // 50  
    h1.restar_seg();  
    cout << h1.dame_segs() << endl; // 29  
    h1.restar_seg();  
    cout << h1.dame_segs() << endl; // 28  
  
    h2.restar_seg();  
    cout << h2.dame_segs() << endl; // 49  
    cout << h1.dame_segs() << endl; // 28  
}
```

# Listas de inicialización

Vamos a crear una clase `Recordatorio` que represente un recordatorio para el día de hoy. El mismo debe sonar en un cierto horario y debe tener un mensaje.



# Lista de inicialización

```
class Recordatorio {  
private:  
    string _mensaje;  
    Horario _horario;  
  
public:  
    Recordatorio(string mensaje, Horario horario);  
  
    Horario horario();  
    string mensaje();  
}  
  
Recordatorio::Recordatorio(string mensaje, Horario horario) {  
    _mensaje = mensaje;  
    _horario = horario;  
}  
  
Horario Recordatorio::horario() {  
    return _horario;  
}  
  
string Recordatorio::mensaje() {  
    return _mensaje;  
}
```

# Listas de inicialización

```
error: no matching function for call to 'Horario::Horario()'
in Recordatorio::Recordatorio(string mensaje, Horario horario)
```

```
Recordatorio::Recordatorio(string mensaje, Horario horario) {
    _mensaje = mensaje;
    _horario = horario;
}
```

# Listas de inicialización

La lista de inicialización permite definir con qué constructor se inicializa cada variable interna de la instancia de clase.

```
Recordatorio::Recordatorio(string mensaje, Horario horario) :  
    _mensaje(mensaje), _horario(horario) {  
    // Más código del constructor, de ser necesario.  
}
```

# Const

`const` es una palabra reservada del lenguaje que altera un tipo para convertirlo en una versión no modificable.

```
int main() {  
    const int x = 5;  
    y = x + 3;  
    y--;  
    // x--; // error: decrement of read-only variable 'x'  
}
```

# Const

`const` tiene muchos usos que vamos a ver incrementalmente. Otro uso es al final de una declaración de un método de una clase. Esto indica que el código de adentro de la clase no puede modificar las variables de la misma.

```
class Horario {  
    public:  
        int dame_hora() const;  
        ...  
  
        void restar_seg() const;  
  
    private:  
        ...  
};
```

```
int Horario::dame_hora() const {  
    return hora;  
}
```

```
/* error: decrement of member 'Horario::secs' in read-only object  
void Horario::restar_seg() const {  
    secs--;  
}  
*/
```

# Const

**Nota:** si el método es declarado `const` debe ser definido `const`.

```
class Horario {  
    public:  
        int dame_hora() const;  
        int dame_min();  
        ...  
  
    private:  
        ...  
};
```

*// error: prototype for 'int Horario::dame\_hora()' does not match any in  
↪ class 'Horario'*

```
int Horario::dame_hora() {  
    return hora;  
}
```

*// error: prototype for 'int Horario::dame\_min() const' does not match  
↪ any in class 'Horario'*

```
int Horario::dame_min() const {  
    return min;  
}
```

# Const

De hecho, esto es factible. Pero lo vemos otro día...

```
class Horario {  
    public:  
        Horario restar_seg() const;  
        void restar_seg();  
        ...  
  
    private:  
        ...  
};  
  
void Horario::restar_seg() {  
    ...  
}  
  
Horario Horario::restar_seg() const {  
    ...  
}
```

# Sobrecarga de Operadores

C++ nos permite *sobrecargar* operadores (eg.: -, +, <, >). Por ejemplo, podemos querer hacer las siguientes operaciones entre Horarios.

```
int main() {  
    Horario h1(2, 15, 26);  
    Horario h2(1, 25, 20);  
  
    // Saber si dos horarios son el mismo  
    h1 == h2;  
    // Saber si una hora pasa antes que la otra  
    h1 < h2;  
    // Conocer el tiempo entre dos horarios  
    Horario d = h1 - h2;  
}
```



# Sobrecarga de Operadores

```
class Horario {  
    public:  
        Horario(int _hora, int _min, int _secs);  
  
        ...  
  
        bool operator<(Horario o) const;  
        bool operator==(Horario o) const;  
  
    private:  
        ...  
};
```

# Sobrecarga de Operadores

```
bool Horario::operator==(Horario o) const {  
    return (hora == o.dame_hora() and  
            min == o.dame_min() and  
            secs == o.dame_segs());  
}
```

```
bool Horario::operator<(Horario o) const { ... }  
  
Horario operator-(Horario h1, Horario h2) {  
    ...  
}
```