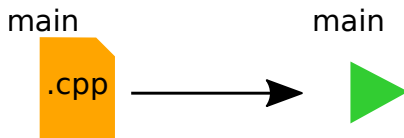


Bajo el capó

Algoritmos y Estructuras de Datos II

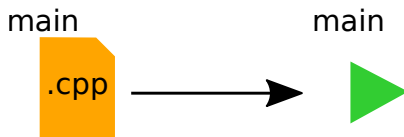
## ¿Qué pasa cuando compilo?

```
$> g++ main.cpp -o main
```



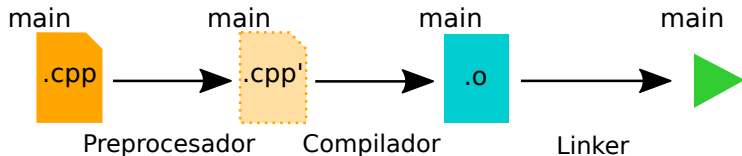
## ¿Qué pasa cuando compilo?

```
$> g++ main.cpp -o main
```

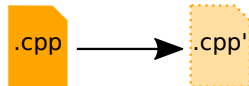


```
$> g++ -c main.cpp -o main.o
```

```
$> g++ main.o -o main
```



# The C++ Preprocessor<sup>1</sup>



- ▶ Directivas específicas para manipular el texto del código
- ▶ Directivas empiezan con #
- ▶ Ejemplos: #define, #include, #ifdef, #ifndef, #if, #endif

---

<sup>1</sup><http://en.cppreference.com/w/cpp/preprocessor>

```
#define DEBUG
```

```
int foo(int x) {  
    if (x % 2 == 0) {  
        return x / 2;  
        #ifdef DEBUG  
        cout << "x era par" << endl;  
        #endif  
    } else {  
        return x + 1;  
    }  
}
```

```
int foo(int x) {  
    if (x % 2 == 0) {  
        return x / 2;  
    } else {  
        return x + 1;  
    }  
}
```

```
int foo(int x) {  
    if (x % 2 == 0) {  
        return x / 2;  
        cout << "x era par" << endl;  
    } else {  
        return x + 1;  
    }  
}
```

a.cpp

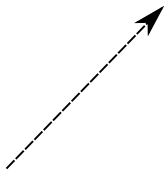
```
int foo(int x) {  
    return x + 5;  
}
```

b.cpp

```
#include "a.cpp"  
  
int bar(int y) {  
    return foo(y) + 4;  
}
```

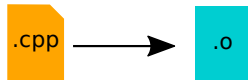
b.cpp'

```
int foo(int x) {  
    return x + 5;  
}  
  
int bar(int y) {  
    return foo(y) + 4;  
}
```



```
#include <vector>
```

# Compilación (esquemático)



\$> g++ -c bar.cpp -o bar.o

```
int external(int x);

int foo() {
    int a = 4;
    int b = a + 2;
    b++;
    b = external(b);
    return b;
}
```



```
int external(int x);

int foo() {
    01100101011111
    00110001011101
    10111000110010
    01  external(b);
}
```



```
int external(int x);

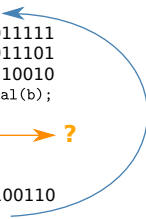
int foo() {
    01100101011111
    00110001011101
    10111000110010
    01  external(b);
}
```



```
int bar() {
    int x = 5;
    x + foo();
    return x;
}
```

```
int bar() {
    111001011100110
    110 foo();
    10001110100001
}
```

```
int bar() {
    111001011100110
    110 foo();
    10001110100001
}
```



# Compilación (esquemático)

## Compilación

\$> g++ -c bar.cpp -o bar.o

```
int external(int x);

int foo() {
    int a = 4;
    int b = a + 2;
    b++;
    b = external(b);
    return b;
}

int bar() {
    int x = 5;
    x + foo();
    return x;
}
```

```
int external(int x);

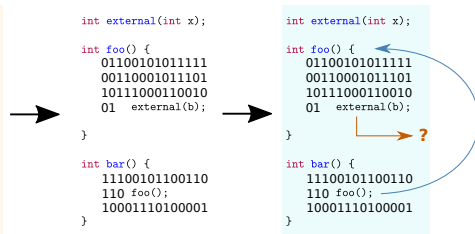
int foo() {
    01100101011111
    00110001011101
    10111000110010
    01  external(b);
}

int bar() {
    11100101100110
    110 foo();
    10001110100001
}
```

```
int external(int x);

int foo() {
    01100101011111
    00110001011101
    10111000110010
    01  external(b);
}

int bar() {
    11100101100110
    110 foo();
    10001110100001
}
```



\$> g++ -c ext.cpp -o ext.o

```
int external(int x) {
    return x + 10;
}
```

```
int external(int x) {
    00110001011101
}
```

\$> g++ -c main.cpp -o main.o

```
int bar();

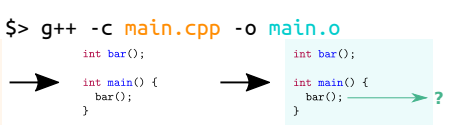
int main() {
    bar();
}
```

```
int bar();

int main() {
    bar();
}
```

```
int bar();

int main() {
    bar();
}
```



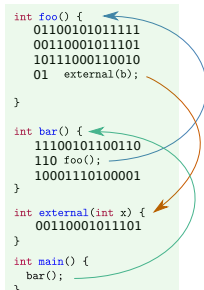
## Linkeo

```
int foo() {
    01100101011111
    00110001011101
    10111000110010
    01  external(b);
}

int bar() {
    11100101100110
    110 foo();
    10001110100001
}

int external(int x) {
    00110001011101
}

int main() {
    bar();
}
```



\$> g++ bar.o  
ext.o  
main.o  
-o exec



a.cpp

```
int foo(int x) {  
    return x + 5;  
}
```

b.cpp

```
#include "a.cpp"  
  
int bar(int y) {  
    return foo(y) + 4;  
}
```

```
#include "a.cpp"  
#include "b.cpp"  
  
int main() {  
    foo(5);  
    bar(7);  
    return 9;  
}
```

\$> g++ -c a.cpp -o a.o

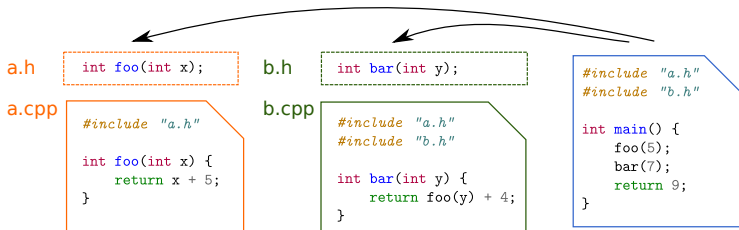
\$> g++ -c b.cpp -o b.o

\$> g++ -c a\_b\_main.cpp -o a\_b\_main.o

\$> g++ a.o b.o a\_b\_main.o -o a\_b\_main

```
In file included from b.cpp:1:0,
                  from a_b_main.cpp:2:
a.cpp: In function 'int foo(int)':
a.cpp:1:5: error: redefinition of 'int foo(int)'
    int foo(int x) {
        ^
```

```
In file included from a_b_main.cpp:1:0:
a.cpp:1:5: note: 'int foo(int)' previously defined here
    int foo(int x) {
        ^
```



```
$> g++ -c a.cpp -o a.o
```

```
$> g++ -c b.cpp -o b.o
```

```
$> g++ -c a_b_main.cpp -o a_b_main.o
```

```
$> g++ a.o b.o a_b_main.o -o a_b_main
```

## class.h

```
#ifndef CLASS_H
#define CLASS_H

#include <string>


using namespace std;

class MiClase {
public:
    MiClase();

    int  obs_1();
    string obs_2();

private:
    int val_1;
    string val_2;
};

#endif
```



## class.cpp

```
#include "class.h"
#include <string>

using namespace std;

MiClase::MiClase() {
    val_1 = 10;
    val_2 = "Hola";
}

int MiClase::obs_1() {
    return val_1 + 5;
}

string MiClase::obs_2() {
    if (val_1 + 5) {
        return val_2;
    } else {
        return val_2 + " mundo";
    }
}
```

## Member Classes



Algoritmos y Estructuras de Datos II

# Algobot

Cuando se entregan los TPs y talleres al Algobot por mail, es común que en la entrega falte el código adjunto o se envíe de forma incorrecta. Normalmente los alumnos se dan cuenta a tiempo y envían un nuevo mail con los archivos correctos.

El código del algobot debe, entonces, poder detectar mails de un mismo grupo para quedarse con el código de la última entrega que enviaron.

# Algobot

¿Qué queremos del Algobot? Pensemos la interfaz...

# Algobot

¿Qué queremos del Algobot? Pensemos la interfaz...

- ▶ Recibir entregas: libretas del grupo + código. Asumimos 2 integrantes.
- ▶ Saber si un grupo entregó
- ▶ Conocer la última versión del código de cualquiera de los integrantes.



```
class Algoritmo {
public:
    void entrega(string i1, string i2, string codigo);

    bool entrego(string i1, string i2) const;
    string codigo(string i) const;

private:
    vector<tuple<tuple<string, string>, string>>> _entregas;
};
```

```
class Algobot {  
public:  
    void entrega(string i1, string i2, string codigo);  
  
    bool entrego(string i1, string i2) const;  
    string codigo(string i) const;  
  
private:  
    vector<tuple<tuple<string, string>, string>>> _entregas;  
};
```

¿"007/04" es lo mismo que "7/4"?

```
class LU {  
    public:  
        LU(string);  
        int numero() const;  
        int anio() const;  
        bool operator==(LU o) const;  
}
```

```

class LU {
public:
    LU(string);
    int numero() const;
    int anio() const;
    bool operator==(LU o) const;
}

class Algobot {
public:
    void entrega(string i1, string i2, string codigo);
    bool entrego(string i1, string i2) const;
    string codigo(string i) const;

private:
    vector<tuple<tuple<LU, LU>, string>>> _entregas;
};

```

```

class LU {
public:
    LU(string);
    int numero() const;
    int anio() const;
    bool operator==(LU o) const;
}

```

```

class Algobot {
public:
    void entrega(string i1, string i2, string codigo);
    bool entrego(string i1, string i2) const;
    string codigo(string i) const;

private:
    vector<tuple<tuple<LU, LU>, string>>> _entregas;
};

```

vector<tuple<vector... ¿Confuso, no?

```
typedef Grupo tuple<LU, LU>;

class Algoritmo {
public:
    void entrega(string i1, string i2, string codigo);
    bool entregado(string i1, string i2) const;
    string codigo(string i) const;

private:

    struct Entrega {
        Grupo grupo;
        string codigo;
    }

    vector<Entrega> _entregas;
};
```

¿Porqué no `void` entrega(LU i1, LU i2, string codigo)?

¿Porqué no `void` entrega(LU i1, LU i2, string codigo)?

Es válido, pero la otra interfaz es más sencilla de usar.



```

void Algoritmo::entrega(string i1, string i2, string codigo) {
    Grupo grupo = make_tuple(LU(i1), LU(i2));
    for (int i = 0; i < _entregas.size(); i++) {
        if (_entregas[i].grupo == grupo) {
            _entregas[i].codigo = codigo;
            return;
        }
    }
    Entrega e;
    e.grupo = grupo;
    e.codigo = codigo; //TODO (march): ponerle un constructor a Entrega
    _entregas.push_back(e);
}

string Algoritmo::codigo(string s) const {
    LU lu(s);
    for (int i = 0; i < _entregas.size(); i++) {
        if (get<0>(_entregas[i].grupo) == lu ||
            get<1>(_entregas[i].grupo) == lu) {
            return _entregas[i].codigo;
        }
    }
}

bool Algoritmo::entrego(string i1, string i2) const { ... }

```

```

typedef Grupo tuple<LU, LU>;

class Algoritmo {
public:
    void entrega(string i1, string i2, string codigo);
    bool entregado(string i1, string i2) const;
    string codigo(string i) const;

private:

    struct Entrega {
        Entrega(Grupo g, string c);
        Grupo grupo;
        string codigo;
    }

    vector<Entrega> _entregas;
};

Algoritmo::Entrega::Entrega(Grupo g, string c)
    : grupo(g), codigo(c) {}

```

```
void Algoritmo::entrega(string i1, string i2, string codigo) {
    Grupo grupo = make_tuple(LU(i1), LU(i2));
    for (int i = 0; i < _entregas.size(); i++) {
        if (_entregas[i].grupo == grupo) {
            _entregas[i].codigo = codigo;
            return;
        }
    }
    Entrega e(grupo, codigo);
    _entregas.push_back(e);
}
```



```
string Algoritmo::codigo(string s) const {
    LU lu(s);
    for (int i = 0; i < _entregas.size(); i++) {
        if (get<0>(_entregas[i].grupo) == lu ||
            get<1>(_entregas[i].grupo) == lu) {
            return _entregas[i].codigo;
        }
    }
}
```

```
bool Algoritmo::entrego(string i1, string i2) const { ... }
```

```

void Algoritmo::entrega(string i1, string i2, string codigo) {
    Grupo grupo = make_tuple(LU(i1), LU(i2));
    for (int i = 0; i < _entregas.size(); i++) {
        if (_entregas[i].grupo == grupo) {
            _entregas[i].codigo = codigo;
            return;
        }
    }
    Entrega e(grupo, codigo);
    _entregas.push_back(e);
}

```

```

string Algoritmo::codigo(string s) const {
    LU lu(s);
    for (int i = 0; i < _entregas.size(); i++) {
        if (get<0>(_entregas[i].grupo) == lu ||
            get<1>(_entregas[i].grupo) == lu) {
            return _entregas[i].codigo;
        }
    }
}

```

```

bool Algoritmo::entrego(string i1, string i2) const { ... }

```

¿Que pasa si entrego primero con "123/09;231/15" y después con "231/15;123/09"?

```

class Algoritmo {
public:

    ...

private:
    class Grupo {
    public:
        Grupo(LU i1, LU i2);

        vector<LU> integrantes() const;
        bool esIntegrante(LU i) const;
        bool operator==(Grupo o) const;

    private:
        vector<LU> _integrantes;

    };

    struct Entrega { ... }

    vector<Entrega> _entregas;
};

```

```

Algobot::Grupo::Grupo(LU i1, LU i2) {
    _integrantes.push_back(i1);
    _integrantes.push_back(i2);
}

vector<LU> Algobot::Grupo::integrantes() const {
    return _integrantes;
}

bool Algobot::Grupo::esIntegrante(LU i) const {
    return (_integrantes[0] == i) or (_integrantes[1] == i);
}

bool Algobot::Grupo::operator==(Algobot::Grupo o) const {
    return o.esIntegrante(_integrantes[0]) &&
           o.esIntegrante(_integrantes[1]);
}

```

```

void Algoritmo::entrega(string i1, string i2, string codigo) {
    Grupo g(LU(i1), LU(i2));
    for (int i = 0; i < _entregas.size(); i++) {
        if (_entregas[i].grupo == g) {
            _entregas[i].codigo = codigo;
            return;
        }
    }
    Entrega e(g, codigo);
    _entregas.push_back(e);
}

string Algoritmo::codigo(string integrante) const {
    LU lu(integrante);
    for (int i = 0; i < _entregas.size(); i++) {
        if (_entregas[i].grupo.esIntegrante(lu)) {
            return _entregas[i].codigo;
        }
    }
}

```

# Testing

Algoritmos y Estructuras de Datos II



# Testing

Supongamos que programamos la clase Fecha y los métodos:

```
void Fecha::sumar_meses(int meses) { ... }  
void Fecha::sumar_dias(int dias) { ... }
```

¿Cómo verificamos si cumple con la especificación?

# Testing

¿Cómo verificamos si un programa cumple con la especificación?

## Problema difícil

- ▶ No es razonable probar con todas las entradas. Entre el año 1 y el 2100 hay  $\sim 750.000$  fechas posibles, y el parámetro `dias` es un `int` que típicamente podría tomar 4.294.967.296 valores posibles.
- ▶ Aun suponiendo que contáramos con una especificación formal, no hay un método automático para comprobar que un programa cumple con esa especificación (Tema de LyC).

# Testing

¿Cómo verificamos si un programa cumple con la especificación?

## Una solución (parcial)

**Testing:** probar que el programa funcione en varios casos.

*Program testing can be used to show the presence of bugs,  
but never to show their absence. —E. W. Dijkstra*

Es decir:

- ▶ El testing sirve para encontrar errores.  
*“no pasa los tests  $\Rightarrow$  hay errores”*
- ▶ El testing no sirve para asegurar que no hay errores.  
*“pasa los tests  $\nRightarrow$  no hay errores”*

Nota: el testing es **muy** útil, a pesar de no ser una técnica exacta.

# Testing

¿Cómo verificamos si un programa cumple con la especificación?

Opción 1: a ojo

```
int main() {  
    Fecha f(2000, 1, 1);  
    f.sumar_dias(10);  
    cout << f << endl; // 2000-01-11  
    f.sumar_meses(1);  
    cout << f << endl; // 2000-02-11  
    f.sumar_dias(20);  
    cout << f << endl; // 2000-03-03  
    f.sumar_meses(2);  
    cout << f << endl; // 2000-05-03  
    return 0;  
}
```

# Testing

## Opción 1: a ojo — desventajas

- ▶ Si tenemos 100 tests, tenemos que mirar 100 líneas y calcular a ojo si son correctas. Los humanos somos pésimos para esto.
- ▶ La condición que verificamos no queda documentada.
- ▶ Si cambiamos alguna parte del programa, corremos el riesgo de introducir un bug y tenemos que repetir este proceso.
- ▶ Los tests del ejemplo están “enredados”: si hay un error, puede ser difícil encontrar de dónde proviene.
- ▶ Idealmente, cada test debería comprobar una funcionalidad puntual para que sea más fácil encontrar la causa del bug.

# Testing

## Opción 2: testing con código

```
bool test_sumar_dia_sin_cambio_mes() {  
    Fecha f(2000, 4, 20);  
    f.sumar_dias(5);  
    if (f != Fecha(2000, 4, 25)) {  
        cout << "Error al sumar días (sin cambio de mes)." << endl;  
        return false;  
    }  
    return true;  
} ...  
bool test_sumar_dia() {  
    return test_sumar_dia_sin_cambio_mes() &&  
           test_sumar_dia_con_cambio_mes() && ...;  
} ...  
int main() {  
    if (test_sumar_dia() && test_sumar_mes() && ...) {  
        return 0;  
    } else { return -1; }  
}
```

# Testing

## Opción 3: usando un entorno de testing

Por ejemplo, con google-test:

```
#include "gtest/gtest.h"
#include "../src/Fecha.h"

TEST(sumar_dia, sin_cambio_mes) {
    Fecha f(2000, 4, 20);
    f.sumar_dias(5);
    ASSERT_EQ(f, Fecha(2000, 4, 25));
}

TEST(sumar_dia, con_cambio_mes) {
    Fecha f(2000, 4, 20);
    f.sumar_dias(19);
    ASSERT_EQ(f, Fecha(2000, 5, 9));
}

...
```

# Testing

## Opción 3: usando un entorno de testing

- ▶ El *framework* incorpora un `main` que corre todos los tests.
- ▶ Reporta cuáles fueron los tests que pasaron exitosamente y cuáles fallaron.
- ▶ Aserciones útiles en `google-test`:
  - ▶ `ASSERT_TRUE(x)`, `ASSERT_FALSE(x)`
  - ▶ `ASSERT_EQ(x, y)`, `ASSERT_NE(x, y)`, `ASSERT_LT(x, y)`,  
`ASSERT_LE(x, y)`, `ASSERT_GT(x, y)`, `ASSERT_GE(x, y)`
- ▶ Todos los lenguajes de programación populares cuentan con algún *framework* de testing.



## Algunas ideas

- ▶ Al menos un test por función / método
- ▶ Tests por comportamiento específico
- ▶ Al testear colecciones, pensar en los casos: vacío, un elemento, muchos elementos.

```
class Algoritmo {  
public:  
    void entrega(LU i1, LU i2, string codigo);  
    bool entregado(LU i1, LU i2) const;  
    string codigo(LU i) const;  
  
    ...  
};
```

- ▶ Entregaron 0, 1, 2 o más grupos.
- ▶ Código de integrantes de cada grupo.

Entregaron 0, 1, 2 o más grupos.

```
TEST(algobot, entrega) {  
    Algobot ab;  
    EXPECT_FALSE(ab.entrego(LU("001/01")));  
    EXPECT_FALSE(ab.entrego(LU("002/02")));  
  
    ab.entrega(LU("001/01"), LU("002/01"), "codigo_g1");  
    EXPECT_TRUE(ab.entrego(LU("001/01")));  
    EXPECT_FALSE(ab.entrego(LU("002/02")));  
  
    ab.entrega(LU("001/02"), LU("002/02"), "codigo_g2");  
    EXPECT_TRUE(ab.entrego(LU("001/01")));  
    EXPECT_TRUE(ab.entrego(LU("002/02")));  
}
```

Código de integrantes de cada grupo.

```
TEST(algobot, codigo) {  
    Algobot ab;  
    ab.entrega(LU("001/01"), LU("002/01"), "codigo_g1");  
    ab.entrega(LU("001/02"), LU("002/02"), "codigo_g2");  
    EXPECT_EQ(ab.codigo(LU("001/01")), "codigo_g1");  
    EXPECT_EQ(ab.codigo(LU("002/01")), "codigo_g1");  
    EXPECT_EQ(ab.codigo(LU("001/02")), "codigo_g2");  
    EXPECT_EQ(ab.codigo(LU("002/02")), "codigo_g2");  
}
```

## Comportamientos especiales

- ▶ Grupo entrega dos veces, se actualiza el código.
- ▶ Grupo entrega dos veces con integrantes en otro orden, igual se actualiza el código.

```
TEST(algobot, doble_entrega) {  
    Algobot ab;  
    ab.entrega(LU("001/01"), LU("002/01"), "codigo_v1");  
    EXPECT_EQ(ab.codigo(LU("001/01")), "codigo_v1");  
  
    ab.entrega(LU("001/01"), LU("002/01"), "codigo_v2");  
    EXPECT_EQ(ab.codigo(LU("001/01")), "codigo_v2");  
  
    ab.entrega(LU("002/01"), LU("001/01"), "codigo_v3");  
    EXPECT_EQ(ab.codigo(LU("001/01")), "codigo_v3");  
}
```