Programación Orientada a Objetos en C++

Instituto Balseiro

Paradigmas de Programación

- Un paradigma de programación es una colección de modelos conceptuales que juntos modelan el proceso de diseño y determinan, al final, la estructura de un programa.

- Un lenguaje dice soportar un paradigma si provee facilidades que hacen conveniente (razonablemente fácil, seguro y eficiente) usar ese estilo de programación.

Programación Procedural

> "*Decida qué procedimientos quiere y seleccione los mejores algoritmos para hacerlos*"

- El foco está en el procesamiento (algoritmo).
- Los lenguajes soportan este paradigma proveyendo facilidades para pasar argumentos y retornar valores desde funciones.

Programación Procedural

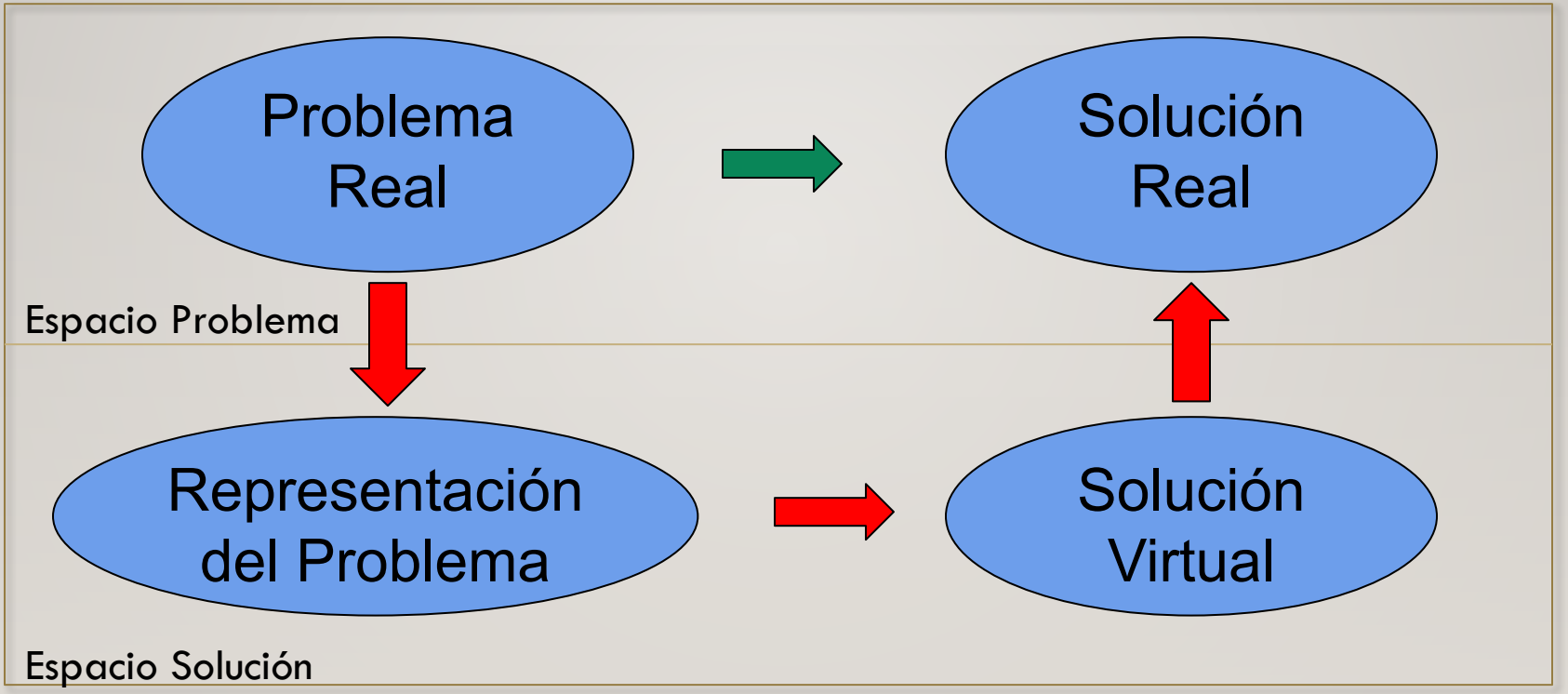
```
double sqrt(double value)
{
    // code for calculating a squared root
    // ...
}


void f()
{
    double root2 = sqrt(2);
    // ...
}
```

Programación Modular

- El énfasis se movió desde el diseño de procedimientos hacia la organización de los datos.
- Un conjunto de procedimientos relacionados con los datos que ellos manipulan define un "módulo".
- **Paradigma de programación modular:**

  "*Decida qué módulos quiere y divida el programa para que los datos estén ocultos entre los módulos*"

Proceso de Abstracción



Proceso de Abstracción

- Todos los lenguajes de programación proveen abstracciones.
  - Assembler: abstrae la máquina.
  - C, Fortran, Basic: abstraen el assembler

    Gran avance, pero todavía uno debe pensar en función de la estructura de la máquina y no del problema.
- La complejidad de problemas que uno es capaz de resolver está directamente relacionada con el tipo y calidad de abstracciones.

Proceso de Abstracción

- Una alternativa al modelado de la máquina es el modelado del problema.
  - Reducción del “gap semántico” (distancia entre espacio problema y espacio solución).

Programación Orientada a Objetos

- Permite que se describa el problema en términos de la entidades que están presentes en el problema más que en términos de la computadora donde éste será resuelto.

- Elementos en el espacio problema y su representación en el espacio solución → **objetos**

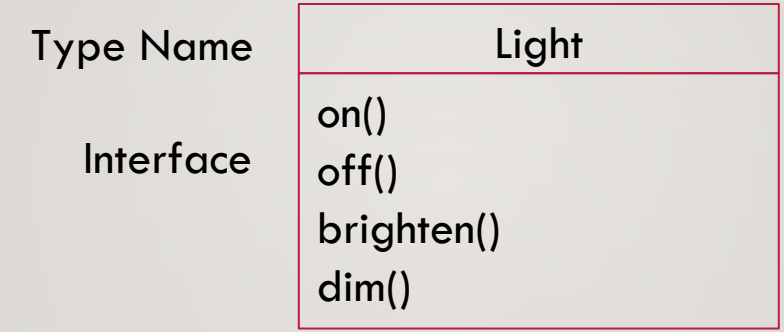
Programación Orientada a Objetos

- Todos son objetos → **variables elaboradas:** almacenan datos, pero se pueden hacer requerimiento para que realice operaciones sobre sí mismo.
- Un programa es una colección de objetos interactuando a través de mensajes.
- Cada objeto tiene su propia memoria hecha de otros objetos.
- Cada objeto tiene un tipo.
- Todos los objetos de un tipo particular pueden recibir los mismos tipos de mensajes.

Objetos con Interface

- Objetos que son idénticos excepto por su estado durante la ejecución de un programa son denominados **clases** de objetos.
- Crear clases es un concepto fundamental en O.O.P.
- Una clase describe a un conjunto de objetos con los mismos atributos (datos) y comportamiento (funcionalidad) → **tipo abstracto de datos**.
- La funcionalidad de un objeto se logra haciendo un requerimiento al objeto.
- Lista de requerimientos posibles → **interface**

Objetos con Interface




```
Ligth lt;
lt.on();
```

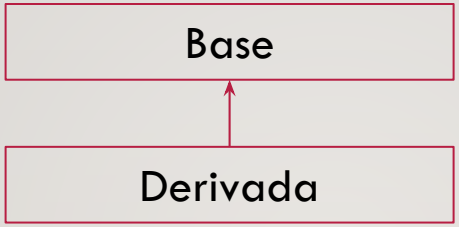
Implementación Oculta

- Separación entre creadores de clases y programadores clientes de esas clases.
- El objetivo de un programador cliente es disponer de una colección de clases para usar en el desarrollo rápido de aplicaciones.
- El objetivo de un creador de clases es definir clases que solo expongan lo que es necesario al cliente.

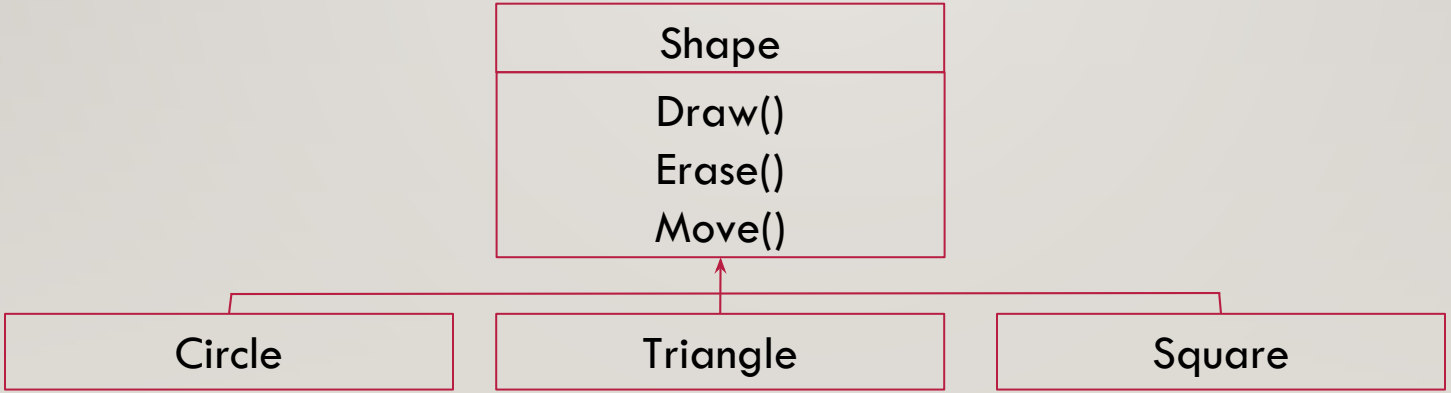
Reuso de Implementación

- Una clase creada y probada debería (idealmente) representar una unidad útil de código.
  - Usar objetos de esa clase directamente
  - Utilizar objetos de esa clase dentro de nuevas clases (objetos miembros). Esto se denomina composición o agregación. Establece la relación “tiene un”

Herencia: Reuso de Interface

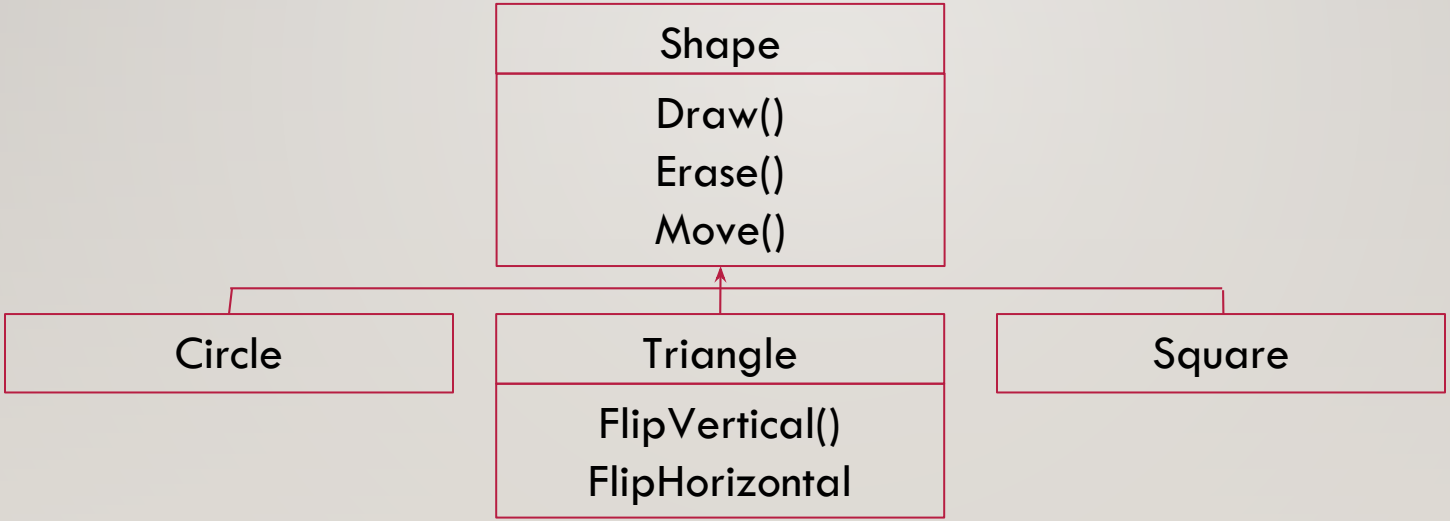


- Establece la relación “es un”



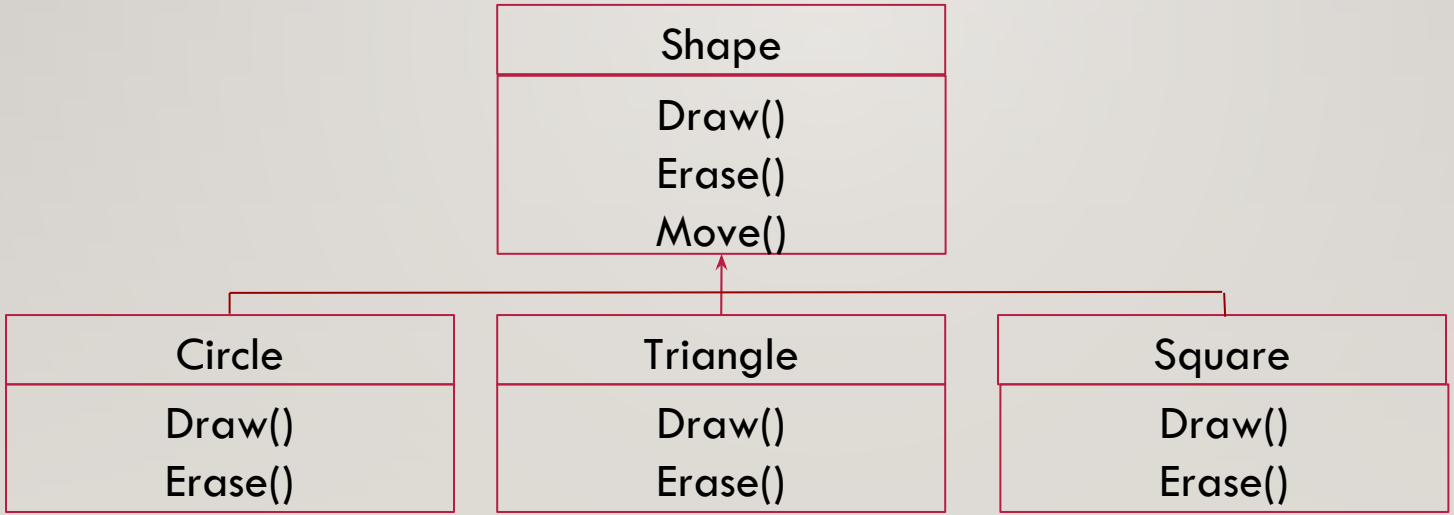
Herencia: Reuso de Interface

- Expansión de la interface.

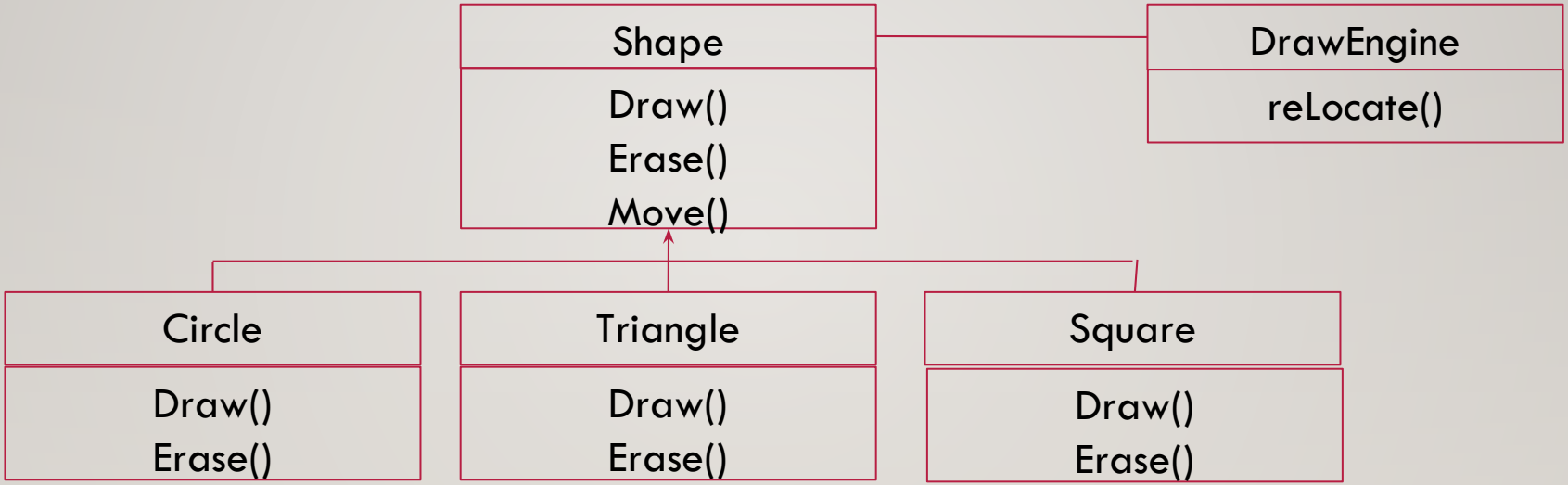


Herencia: Reuso de Interface

- Cambio de comportamiento (“overriding”).



Polimorfismo



- Early binding
- Late binding

C++ Highlights

- Lenguaje de programación de propósito general
- Un mejor C
- Soporta abstracción de datos
- Soporta programación orientada a objetos
- Soporta programación genérica

C++ Overview

- Variables

```
int inch;     // Declaración de variable de tipo entero
```

- Funciones

```
void f() {                          // Función que no retorna un valor
    double root2 = sqrt(2);         // Asignación con retorno de función
    // …                            // Comentario
}
```

- Tipos

```
bool      // booleano, valores: true y false
char      // caracter: 'a', 'z', 'F' y '9'
int       // entero: 1, 42 y 1216
double    // punto flotante de doble precisión: 3.14
```

C++ Overview

- Operadores aritméticos

```
+      // más, unario y binario
-      // menos, unario y binario
*      // multiplicación, binario; contenido de, binario
/      // división
%      // resto entero
```

- Operadores de comparación

```
==     // igual
!=     // distinto
<      // menor
>      // mayor
<=     // menor o igual
>=     // mayor o igual
```

C++ Overview

```
bool accept() {
     cout << "Quiere proceder? (Y/n) ?\n";     // output
     char answer = 0;
     cin >> answer;                            // input
     if( answer == 'y' ) return true;
         return false;
}

bool accept2() {
     cout << "Quiere proceder? (Y/n) ?\n";     // output
     char answer = 0;
     cin >> answer;
     switch( answer ) {
         case 'y': return true;
         default:  cout << "voy a tomar eso como un no.\n";
         case 'n': return false;
     }
}
```

C++ Overview

```
bool accept3() {
    int tries = 1;
    while( tries < 4 ) {
        cout << "Quiere proceder? (s/n) ?\n";
        char answer = 0;
        cin >> answer;
        switch( answer ) {
            case 'y': return true;
            case 'n': return false;
            default:  cout << "perdon, no entiendo\n";
                      tries = tries + 1;                    // tries++
        }
    }
    cout << "Quiere proceder? (s/n) ?\n";
    return false;
}
```

C++ Overview

- Punteros y arrays

```
char v[10];     // array de 10 caracters, v[0]...v[9]
char *p;        // puntero a char
p = &v[3]       // & unario: operador dirección de
                  // p apunta al cuarto elemento de v

void f() {
    int v1[10];
    int v2[10];
    // ...
    for( int i = 0; i < 10; ++i )
        v1[i] = v2[i];
}
```

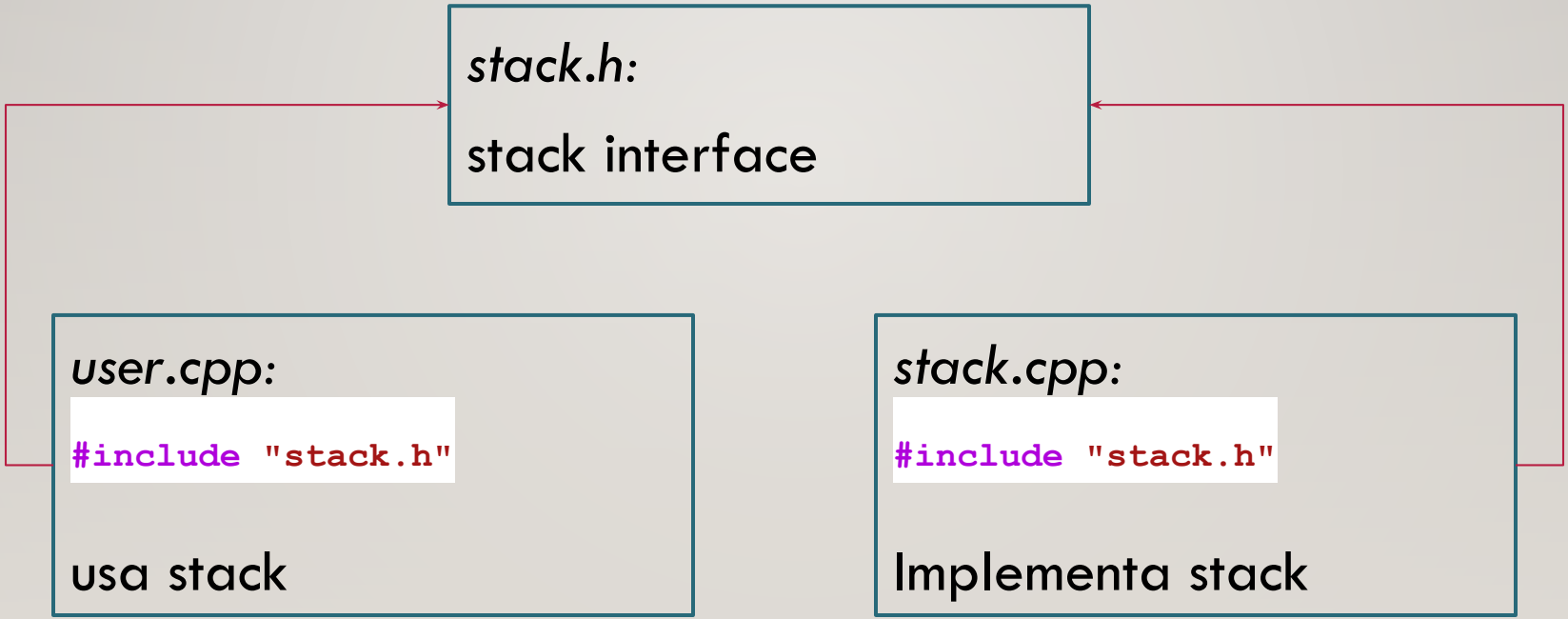
C++ Overview

- Namespaces, módulos

```
namespace Stack {                          // interface
    void push(char);                       // declaración
    char pop();
}

void f() {
    Stack::push('c');
    if( Stack::pop() != 'c' )
        error("imposible");
}

namespace Stack {                          // implementación
    const int max_size = 200;
    char v[max_size];
    int top = 0;
    void push(char c) { /* ... */ }        // definición
    char pop()        { /* ... */ }
}
```

C++ Overview

- Compilación separada, precompilador



C++ Overview

- Excepciones, error handling

```
namespace Stack {
    void push(char);
    char pop();
    struct Overflow { }; // tipo que representa excepciones
                         // por overflow
}

void Stack::push(char c) {
    if( top == max_size )
        throw Overflow{};
    // push c
    // ...
}
```

C++ Overview

- Excepciones, error handling

```
void f()
{
    // …

    try {                       // bloque con excepciones handleadas
        while( true )
            Stack::push('c');

    } catch( Stack::Overflow ) {
        // oops: stack overflow
    }
    // ...
}
```

C++ Overview

- Data abstraction, declaración

```
namespace Stack {
    struct Rep;                     // declaración de tipo de datos
    typedef Rep& stack;             // alias

    stack create();                 // crea un nuevo stack
    void destroy(stack s);          // destruye s

    void push(stack s, char c);     // push de c en s
    char pop(stack s);              // pop de s
}
```

C++ Overview

- Data abstraction, uso

```
void f() {
    Stack::stack s1 = Stack::create();     // crea un nuevo stack
    Stack::stack s2 = Stack::create();     // crea otro nuevo stack

    Stack::push(s1, 'c');
    Stack::push(s2, 'k');

    if( Stack::pop(s1) != 'c' ) throw Bad_pop{};
    if( Stack::pop(s2) != 'k' ) throw Bad_pop{};

    Stack::destroy(s1);
    Stack::destroy(s2);
}
```

C++ Overview

- Data abstraction, definición

```
namespace Stack {
    const int max_size = 200;
    struct Rep {
        char v[max_size];
        int top;
    };

    const int max = 16;
    Rep stacks[max];
    bool used[max];
};
stack Stack::create() {
    // buscar una Rep no usada en stacks, la marcar usada en used,
    // inicializarla y retornar una referencia a ella
}
```

C++ Overview

- User defined type

```
class Complex {
    double re, im;
public:
    Complex(double r, double i) { re = r; im = i; }
    Complex(double r) { re = r; im = 0 }
    Complex() { re = im = 0; }

    friend Complex operator+(Complex, Complex);
    friend Complex operator-(Complex, Complex);
    friend Complex operator-(Complex);
    friend Complex operator*(Complex, Complex);
    friend Complex operator/(Complex, Complex);

    friend Complex operator==(Complex);
    friend Complex operator!=(Complex);
};
```

C++ Overview

- Definición de operador para Complex

```
Complex operator+(Complex a, Complex b) {
    return Complex( a.re + b.re, a.im + b.im);
}
```

- Uso

```
void f(Complex z) {
    Complex a = 2.3;
    Complex b = 1 / a;
    Complex c = a + b * Complex(1, 2.3);
    // ...
    if( c != b )
        c = - ( b / a ) + 2 * b;
    // ...
}
```

C++ Overview

- User defined type, concrete type

```
class Stack {
    char *v;
    int top;
    int max_size;
public:
    class Underflow { };
    class Overflow { };
    class Bad_size { };

    Stack(int s);                    // constructor
    ~Stack();                        // destructor

    void push(char c);
    char pop();
};
```

C++ Overview

```
Stack::Stack(int s) {
    top = 0;
    if( 10000 < s ) throw Bad_size{};
    max_size = s;
    v = new char[s];
}
Stack::~Stack() {
    delete [] v;
}

void Stack::push(char c) {
    if( top == max_size ) throw Overflow{};
    v[top++] = c;
}
char Stack::pop() {
    if( top == 0 ) throw Underflow{};
    return v[--top];
}
```

C++ Overview

- Uso

```
Stack s_var1(10);

void f(Stack &s_ref, int i)
{
    Stack s_var2(i);
    Stack *s_ptr = new Stack(20);

    s_var1.push('a');
    s_var2.push('b');
    s_ref.push('c');
    s_ptr->push('d');
    // ...
}
```

C++ Overview

- User defined type, abstract type

```
class Stack {                                   // interface abstracta
public:
    class Underflow { };
    class Overflow { };

    virtual void push(char c) = 0;              // método virtual puro
    virtual char pop() = 0;                     // método virtual puro
};

void f(Stack &s_ref) {
    s_ref.push('c');
    if( s_ref.pop() != 'c' )
        throw bad_stack{};
}
```

C++ Overview

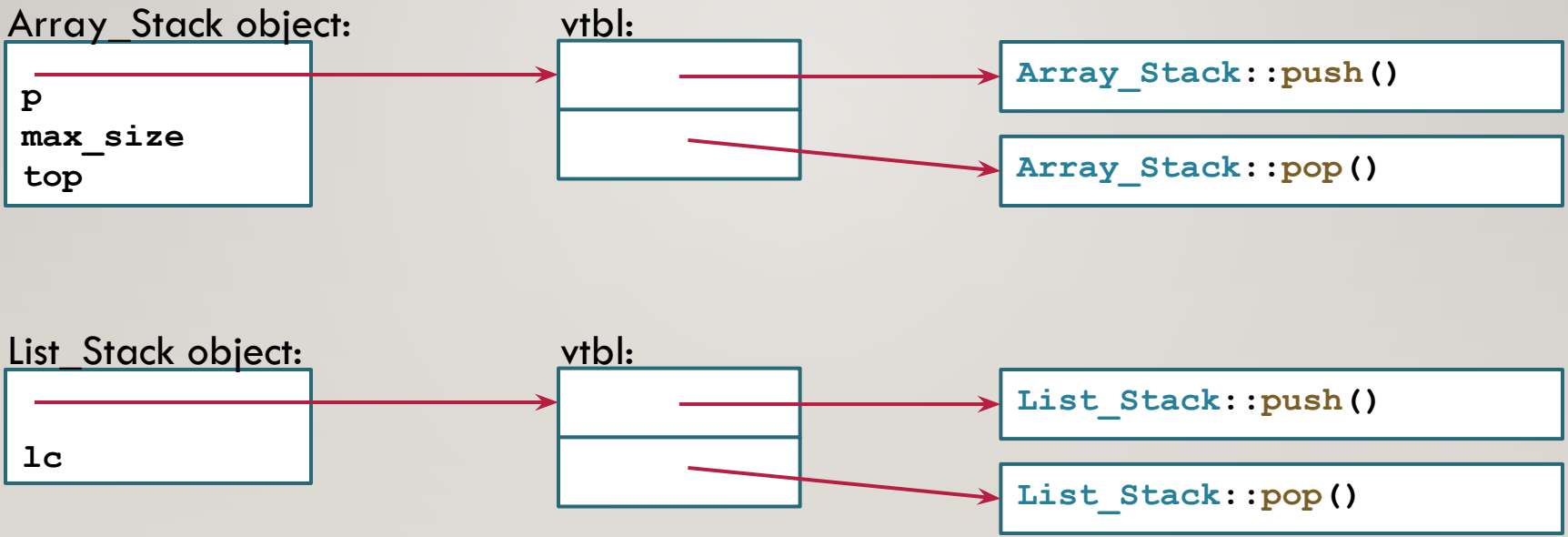
```
class Array_Stack : public Stack {  // Array_Stack implementa Stack
    char *p;
    int top;
    int max_size;
public:
    Array_Stack(int s);
    ~Array_Stack();

    void push(char c);
    char pop();
};

void g() {
    Array_Stack as(200);
    f(as);
}
```

C++ Overview

```
class List_Stack : public Stack {     // List_Stack implementa Stack
    list<char> lc;
public:
    List_Stack() { }

    void push(char c) {
        lc.push_front(c);
    }
    char pop() {
        char x = lc.front();
        lc.pop_front();
        return x;
    }
};

void h() {
    List_Stack ls;
    f(ls);
}
```

C++ Overview

- Funciones virtuales



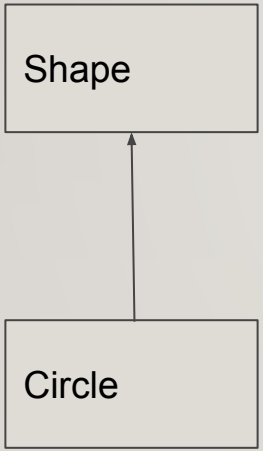
C++ Overview

- Jerarquía de clases

```
class Shape {
    Point center;
    Color color;
public:
    Point where() { return center; }
    void move(Point to) { center = to; /* ... */ draw(); }

    virtual void draw() = 0;
    virtual void rotate(int angle) = 0;
};

void rotate_all(vector<Shape *> &v, int angle) {
    for( int i = 0; i < v.size(); ++i )
        v[i]->rotate(angle);
}
```

C++ Overview

- Jerarquía de clases




```
class Circle : public Shape {
    int radius;
public:
    void draw() { /* ... */ };
    void rotate(int) { };
};
```

C++ Overview

- Programación genérica, containers

```
template<class T> class Stack {
    T *v;
    int max_size;
    int top;
public:
    class Underflow { };
    class Overflow { };

    Stack(int s);
    ~Stack();

    void push(T);
    T pop();
};
```

C++ Overview

- Programación genérica, containers

```
template<class T> void Stack<T>::push(T c)
{
    if( top == max_size )
        throw Overflow{};
    v[top++] = c;
}

template<class T> T Stack<T>::pop()
{
    if( top == 0 )
        throw Underflow{};
    return v[--top];
}
```

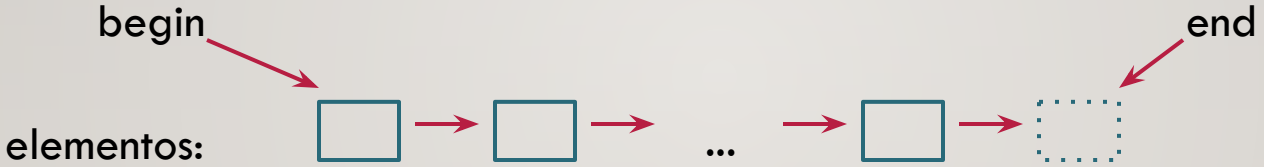
C++ Overview

- Programación genérica, containers, uso

```
Stack<char> sc;              // Stack de caracteres
Stack<Complex> scplx;        // Stack de números complejos
Stack<list<int>> sli;        // Stack de lista de enteros

void f() {
     sc.push('c');
     if( sc.pop() != 'c' )
          throw Bad_pop{};

     scplx.push(Complex(1,2));
     if( scplx.pop() != Complex(1,2) )
          throw Bad_pop{};
}
```

C++ Overview

- Programación genérica, algoritmos
- Concepto de secuencia para algoritmos no numéricos



- Algoritmos genéricos como `sort()`, `copy()`, `search()`, etc. se aplican a secuencias en vez de a containers específicos.
- Concepto de iterador de secuencia

C++ Overview

- Programación genérica, algoritmos

```
template<class In, class Out> void copy(In from, In too_far, Out to)
{
    while( from != too_far ) {
        *to = *from;
        ++to;
        ++from;
    }
}

char vc1[200];
char vc2[500];

void f() {
    copy(&vc1[0], &vc1[200], &vc2[0]);
}
```

C++ Overview

- Programación genérica, algoritmos, uso

```
char vc1[200];
char vc2[500];

void f()
{
    copy(&vc1[0], &vc1[200], &vc2[0]);
}

Complex ac[200];

void g(vector<Complex> &vc, list<Complex> &lc)
{
    copy(&ac[0], &ac[200], lc.begin());
    copy(lc.begin(), lc.end(), vc.begin());
}
```

C++ Overview — Standard Library Tour

- En general, los programas no están escritos solamente utilizando un lenguaje de programación desnudo.
- Conjunto de bibliotecas asociadas al lenguaje.
- C++ standard library:
  - STL Containers: vector, list, map, queue, stack, etc.
  - STL Algorithms: for_each, find, copy, transform, sort, etc.
  - Strings library: string
  - IOStream library: istream, fstream, stringstream, etc.
  - C Library
  - Numéricas, localización, excepciones, límites, etc.

C++ Overview — Standard Library Tour

- hello world

```
#include <iostream>

int main() {
    std::cout << "Hello, world!\n";
    return 0;
}
```

- Standard library `std` namespace

```
#include <string>
#include <list>

std::string s = "hello";
std::list<std::string> slogans;

using namespace std;
string s2 = "bye";
```

C++ Overview — Standard Library Tour

- Output

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

void f() {
    cout << 10;
}

void g(int i) {
    cout << "El valor de i es: ";
    cout << i;
    cout << '\n';
}

void h(int i) {
    cout << "El valor de i es: " << i << endl;
}
```

C++ Overview — Standard Library Tour

- Strings

```
#include <iostream>
#include <string>

using namespace std;
string s1 = "Hello";
string s2 = "world";

void f() {
    string s3 = s1 + ", " + s2 + '!';
    cout << s3 << endl;
}

void g(string &s1, string &s2) {
    s1 = s1 + '\n';
    s2 += "\n";
}
```

C++ Overview — Standard Library Tour

- Strings

```
string incantation;
void respond(const string &answer)
{
    if( answer == incantation ) {  /* perform magic */ }
    else if( answer == "yes")  {  /* ... */ }
}

string name = "Niels Stroustrup";
void g()
{
    string s = name.substr(6,10);     // s = "Stroustrup"
    name.replace(0, 5, "Nicholas");   // name = "Nicholas Stroustrup"
}
```

C++ Overview — Standard Library Tour

- Input

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

void f()
{
    int i;
    cin >> i;

    double d;
    cin >> d;

    string str;
    cout << "Ingrese su nombre: ";
    cin >> str;
    cout << "Hello, " << str << '!' << endl;
}
```

C++ Overview — Standard Library Tour

- Vector

```
struct Entry {
    string name;
    int number;
};

vector<Entry> phone_book(1000);
void print_entry(int i) {
    cout << phone_book[i].name << " " << phone_book[i].number << endl;
}
void add_entries(int n) {
    phone_book.resize(phone_book.size() + n);
}
void f(vector<Entry> &v) {
    vector<Entry> v2 = phone_book;
    v = v2;
}
```

C++ Overview — Standard Library Tour

- List

```
list<Entry> phone_book;

void print_entry(const string &s) {
    typedef list<Entry>::const_iterator LI;
    for( LI i = phone_book.begin(); i != phone_book.end(); ++i ) {
        Entry &e = *i;
        if( s == e.name )
            cout << e.name << " " << e.number << endl;
    }
}

void add_entries(const Entry &e, list<Entry>::iterator i) {
    phone_book.push_front(e);          // agrega al comienzo
    phone_book.push_back(e);           // agrega al final
    phone_book.insert(i, e);           // agrega antes de 'i'
}
```

C++ Overview — Standard Library Tour

- Map

```
#include <map>

using namespace std;

map<string, int> phone_book;

void print_entry(const string &s) {
    if( int i = phone_book[s] )
        cout << s << " " << i << endl;
}
```

C++ Overview — Standard Library Tour

- Algoritmos

```
void f(vector<Entry> &ve, list<Entry> &le) {
    sort(ve.begin(), ve.end());
    unique_copy(ve.begin(), ve.end(), le.begin());
}

void g(vector<Entry> &ve, list<Entry> &le) {
    sort(ve.begin(), ve.end());
    unique_copy(ve.begin(), ve.end(), back_inserter(le));  // append
}

void h(vector<Entry> &ve, list<Entry> &le) {
    copy(ve.begin(), ve.end(), le);        // error: le no es iterador
    copy(ve.begin(), ve.end(), le.end()); // mal: escribe pasando el final
    copy(ve.begin(), ve.end(), le.begin());  // sobreescribe
}
```

C++ Overview — Standard Library Tour

- Iteradores

```
int count(const string &s, char c) {
    string::const_iterator i = find(s.begin(), s.end(), c);
    int n = 0;
    while( i != s.end() ) {
        ++n;
        i = find(i + 1, s.end(), c);
    }
    return n;
}

void f() {
    string m = "Mary had a little lamb";
    int a_count = count(m, 'a');
}
```

C++ Overview — Standard Library Tour

- Iteradores

```
template<class C, class T> int count(const C &v, T val) {
    typename C::const_iterator i = find(v.begin(), v.end(), val);
    int n = 0;
    while( i != v.end() ) {
        ++n; ++i;
        i = find(i, v.end(), val);
    }
    return n;
}

void g(list<Complex> &lc, vector<string> &vc, string s) {
    int i1 = count(lc, Complex(1, 3));
    int i2 = count(vc, "Juan");
    int i3 = count(s, 'x');
}
```

Sugerencias

- **NO ENTRAR EN PÁNICO!**
- **No es necesario conocer todos los detalles de C++ para hacer buenos programas.**
- **Poner el foco en las técnicas de programación, no en las características del lenguaje.**