# Objetos y Clases

Instituto Balseiro

# Definición de objeto

## «Datos con actitud»

- Las estructuras pasan de ser una mera aglomeración de datos a ser utilizadas como conceptos.
- Las estructuras tienen datos y comportamiento.
- Los objetos en C++ son variables. Una región de memoria para guardar datos y funciones que los manipulan.
- Los objetos tiene un identificador único. Su dirección de memoria.

# Abstract data type / User data type

- La posibilidad de empaquetar datos con funciones permite crear nuevos tipos de datos.
- Llamado también encapsulación.
- Permiten abstraer conceptos del problema del espacio real al espacio virtual.
- Llamado a métodos: `object.memberFunction(args);`
- Enviar un mensaje a un objeto.

# Programación orientada a objetos

## Envío de mensajes a objetos

- Un programa es un conjunto de objetos que se envían mensajes entre ellos.
- El truco de diseño está en darse cuenta cuales objetos y mensajes son los necesarios.
- Una vez identificados, implementarlos en C++ es fácil.

# Estructuras anidadas

```
struct Stack {
    struct Link {
        void *data;
        Link *next;
        void init(void *, Link *);
    };

    Link *head;

    void init();
    void push(void *);
    void *peek();
    void *pop();
    void cleanup();
};
```

```
void Stack::Link::init(void *dat,
                       Link *nxt)
{
    data = dat;
    next = nxt;
}

void Stack::init() {
    head = nullptr;
}

void Stack::push(void *dat) {
  Link *newLink = new Link;
  newLink->init(dat, head);
  head = newLink;
}
```

# Límites y fronteras en los objetos

- Diferenciación entre quien implementa un objeto y el usuario que lo utiliza.
- Control de acceso:
  - Mantener al usuario lejos de lo que no tiene que tocar. Lo ayuda a ignorar detalles innecesarios o peligrosos.
  - Permitir al implementador cambiar los detalles sin molestar al usuario.
- Definición de interface.

# Control de acceso en C++

- Keywords para poner límites en estructuras:
  **`public`**, **`private`** y **`protected`**.

```
struct A {
    int i;
    float f;
    void func();
};
void A::func() { }

struct B {
    public:
        int i;
        float f;
        void func();
};
void B::func() { }
```

```
struct C {
    private:
        int i;
        float f;
    public:
        void func();
};
void C::func() { i = 0; }

void f() {
    C c;
    C.func();
    C.i = 4;        // ERROR
}
```

# Control de acceso en C++

- `friend` permite a funciones, métodos y otros objetos acceder a miembros privados. Puristas abstenerse.

```
struct X {
    private:
        int i;
        float f;
        int g();

    public:
        void func();

        friend void h(X *, int);
        friend struct Z;
        friend Y::f(X *);
};
```

```
void h(X *px, int i) {
    px->i = i;
}

struct Z {
    private:
        int a;
    public:
        void init(X *);
};

void Z::init(X *px) {
    a = px->g();
}
```

# Clase

- Keyword **`class`**. Idéntico a **`struct`**, pero con acceso **`private`** por default.

```
struct A {
    private:
        int i, j, k;
    public:
        int f();
        void g();
};
```

```
class B {
        int i, j, k;
    public:
        int f();
        void g();
};
```

- Concepto fundamental de OOP en C++.

```
class X {
    public:
        void interface_method();
    private:
        void private_function();
        int internal_representation;
};
```

# Clase

- Se vio a la clase como una forma de tomar los componentes desparramados de una biblioteca típica de C y encapsularlos dentro de una estructura, un tipo abstracto de datos: `class`.
- Control de acceso es la forma de ocultar la implementación.
- Encapsulación y control de acceso juntos proveen una mejora significativa en facilitar el uso de una biblioteca.

# Inicialización y cleanup

- En C++, los conceptos de inicialización y cleanup son fundamentales para facilitar el uso de bibliotecas y eliminar bugs que ocurren cuando quienes programan se olvidan de realizar estas acciones.
- `Stack` y `Stash` tenían una función `initialize()`, pero dejan la responsabilidad de llamarlas al programador.
- El diseñador puede garantizar la inicialización de todas las instancias de una clase proveyendo una función especial denominada *constructor*.

# Inicialización

- Si una clase tiene constructor, el compilador automáticamente lo llama en el punto de creación del objeto, antes que se pueda realizar otra acción sobre el mismo.

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

```
void f() {
    X a;
    // ...
}
```

- El compilador hace una llamada a `X::X()` y como en cualquier función miembro pasa un primer argumento oculto: `this`

# Inicialización

- Un constructor puede tener argumentos.

```
Tree t(12);
```

- Si `Tree(int)` es el único constructor de la clase, entonces el compilador no permitirá que se puedan crear objetos de otra manera.
- Los constructores eliminan una gran variedad de problemas y hacen más sencillo la escritura y lectura de código, en C++ la definición e inicialización son conceptos que están unificados.
- RAII: Resource Acquisition Is Initialization.

# Cleanup

- Existen problemas potenciales y reales de no hacer un cleanup adecuado de los objetos creados. C++ provee el destructor como una forma de realizar el cleanup de una instancia.

```
class Y {
        int i;
    public:
        ~Y();
};
```

- El destructor es llamado automáticamente por el compilador cuando el objeto sale de scope. Aún saltando con `goto` (saltos locales, no cuando se utiliza `setjmp/longjmp`).

# Inicialización y Cleanup

```
class Tree {
    int height;
public:
    Tree(int initialHeight);
    ~Tree();  // Destructor
    void grow(int x);
    void printsize();
};

Tree::Tree(int initialHeight) {
    height = initialHeight;
}
Tree::~Tree() {
    cout << "Tree destructor\n";
    printsize();
}
void Tree::grow(int x) {
    height += x;
}
void Tree::printsize() {
    cout << "h:" << height << endl;
}
```

```
int main() {
    cout << "before opening\n";
    {
        Tree t(12);
        cout << "after Tree creation\n";
        t.printsize();
        t.grow(4);
        cout << "before closing\n";
    }
    cout << "after closing\n";
    return 0;
}
```

```
before opening
after Tree creation
h: 12
before closing
Tree destructor
h: 16
after closing
```

# Stash con constructor y destructor

```
#ifndef STASH2_H
#define STASH2_H 1

class Stash {
    int size;       // Size of each space
    int quantity;   // Number of storage
                    // spaces
    int next;       // Next empty space
    // Dynamically allocated
    // array of bytes:
    unsigned char* storage;
    void inflate(int increase);
public:
    Stash(int size);
    ~Stash();
    int add(void* element);
    void* fetch(int index);
    int count();
};

#endif // STASH2_H
```

```
Stash::Stash(int sz) {
    size = sz;
    quantity = 0;
    storage = nullptr;
    next = 0;
}


Stash::~Stash() {
    if( storage != 0 ) {
        cout << "freeing storage\n";
        delete [] storage;
    }
}
```

# Stash con constructor y destructor

```
#include "Stash2.h"
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main() {

  Stash intStash(sizeof(int));

  for( int i = 0; i < 100; ++i )
      intStash.add(&i);

  for(int j = 0; j < intStash.count(); ++j)
    cout << "intStash.fetch(" << j << ")="
         << *((int*) intStash.fetch(j))
         << endl;
```

```
  const int bufsize = 80;

  Stash stringStash(sizeof(char)*bufsize);

  ifstream in("Stash2Test.cpp");

  string line;

  while( getline(in, line) )
      stringStash.add((char*)line.c_str());

  int k = 0;
  char* cp;

  while((cp=(char*)stringStash.fetch(k++)))
    cout << "stringStash.fetch(" << k
         << ")=" << cp << endl;

  return 0;
}
```

# Stack con constructor y destructor

```
#ifndef STACK3_H
#define STACK3_H

class Stack {
    struct Link {
        void* data;
        Link* next;
        Link(void* dat, Link* nxt);
        ~Link();
    }* head;
public:
    Stack();
    ~Stack();
    void push(void* dat);
    void* peek();
    void* pop();
};
#endif
```

```
Stack::Link::Link(void* dat, Link* nxt) {
    data = dat;
    next = nxt;
}
Stack::Link::~Link() { }
Stack::Stack() { head = nullptr; }
void Stack::push(void* dat) {
    head = new Link(dat, head);
}
void* Stack::peek() {
    assert(head != nullptr);
    return head->data;
}
void* Stack::pop() {
    if( head == nullptr )
        return nullptr;
    void* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
}
Stack::~Stack() {
    assert(head == nullptr);
}
```

# Constructor por defecto

- Un constructor por defecto es aquel que puede ser llamado sin argumentos.

```
class A {
        int i;
    public:
        A(int i_) { i = i_; };
};

//...

A a(1);                     // ok
A b;                        // error no default constructor
A va[2] { A(1), A(4) };     // ok
A va2[2];                   // error, no default constructor
A va3[2] = { A(3) };        // error, no default constructor for va3[1]
```

# Miembros `static`

- Existe la posibilidad de crear una variable que es miembro de la clase y no de cada objeto. Estos son los miembros **`static`**.
- Existe una única copia de los miembros **`static`** en vez de una por objeto, como de los miembros no **`static`**.
- De la misma manera, una función miembro que no necesita ser invocada sobre un objeto en particular es una función miembro **`static`**.
- **`static`** en el sentido de *static data storage*.
- Miembros **`static`** tienen que ser definidos fuera de la clase (previo a C++17).
- Funciones miembro **`static`** no tienen **`this`**.

# Miembros `static`

```
class Date {
          int d, m, y;
          static Date default_date;              // Declaración
     public:
          Date(int dd = 0, int mm = 0, int yy = 0);
          static void set_default(int, int, int);
};

Date Date::default_date(25, 5, 1810);            // Definición

void Data::set_default(int d, int m, int y) {
    Date::default_date = Date(d, m, y);
}

void f() {
    Date::set_default(1, 1, 2000);
}
```

# Miembros constantes estáticos

- Una única constante para todos los objetos.
- Constantes “de clase“.
- Inicializadas al momento de compilación.

```
class X {
    static const int size = 18;
    static const double pi;
    static constexpr double e = 2.71;
};

const double X::pi = 3.14;
```

# Objetos constantes

- Se puede definir un objeto como constante o tener un puntero a un objeto constante o una referencia a un objeto constante.

```
const Complex i(0,1);
const Complex *pi = &i;
const Complex &ri = i;
```

# Funciones miembro constantes

```
class X {
        int ival;
    public:
        X(int iv);
        int get_i() const;
        void set_i(int iv);
};

X::X(int iv)            { ival = iv;    }
int X::get_i() const    { return ival; }
void X::set_i(int iv)   { ival = iv;    }
```

# Funciones miembro constantes

```
X x1(18);
const X x2(25);
const X *px = &x1;
const X &rx = x2;

void f() {
    x1.set_i(5);
    cout << x1.get_i() << endl;
    x2.set_i(45);                                   // Error
    px->set_i(34);                                  // Error
    cout << px->get_i() << rx.get_i() << endl;
}
```

# Problemas con miembros constantes

- Clases lógicamente constantes pero con representación variable.

```
class Date {
        int d, m, y;
        bool cache_valid;
        string cache;
        void compute_string_cache();
    public:
        Date(int day, int month, int year);
        string string_rep() const;
};
```

# Problemas con miembros constantes

```
string Date::string_rep() const
{
    Date *me = const_cast<Date*>(this);
               // (Date *) this;
    if( ! me->cache_valid ) {
        me->compute_string_cache();
        me->cache_valid = true;
    }
    return cache;
}
```

# Problemas con miembros constantes

```
void f(const Date &cdr) {
    cout << cdr.string_rep() << endl;
}

const Date epoch(1,1,1970);

int main() {
    Date hoy(27,8,2020);
    f(hoy);
    f(epoch);
    return 0;
}
```

# Mutables

- Miembros declarados **mutables**.

```
class Date {
        int d, m, y;
        mutable bool cache_valid;
        mutable string cache;
        void compute_string_cache();
    public:
        string string_rep() const;
};
```

# Mutables

```
string Date::string_rep() const
{
     if( ! cache_valid ) {
          computee_string_cache();
          cache_valid = true;
     }
     return cache;
}
```

# Miembros constantes de clases

- Se pueden declarar miembros constantes.
- ¿Dónde inicializo el valor de la constante?

```
class X {
        const int size;
    public:
        X(int sz);
};

X::X(int sz) {
    size = sz;       // Error
}
```

# Constructor initializer list

- Lista de inicialización de miembros anterior al cuerpo del constructor.

```
class X {
        const int size;
    public:
        X(int sz);
};

X::X(int sz)
    : size(sz)
{ }
```