Composición y Herencia

Instituto Balseiro

Funciones `inline`

- Eficiencia de macros de preprocesador en C.
- Parece una llamada a función, pero sin su costo asociado.
- Problemas para encontrar bugs con macros.
- Usos no esperados de macros, malos argumentos.
- Macros no tienen scope, son globales.
- C++ provee funciones **`inline`**:
  - Eficiencia.
  - Mismos mecanismos de control que cualquier función.
  - Sugerencia al compilador.

Inlines adentro de clases

```
class Point {
        int i, j, k;

    public:
        Point() : i(0), j(0), k(0) { }

        Point(int ii, int jj, int kk)
            : i(ii), j(jj), k(kk) {}

        void print(const string& msg = "") const {
            if(msg.size() != 0)
                cout << msg << endl;
            cout << "i = " << i << ", " << "j = " << j << ", "
                 << "k = " << k << endl;
        }
};
```

Inlines afuera de clases

```
class Point {
        int i, j, k;
    public:
        void print(const string& msg) const;
};

inline void Point::print(const string& msg) const {
    if(msg.size() != 0)
        cout << msg << endl;
    cout << "i = " << i << ", " << "j = " << j << ", "
         << "k = " << k << endl;
}
```

- Tienen linkage **`static`**. Hay que definirlas en headers.

Funciones Inline

- Tienen linkage **`static`**. Hay que definirlas en headers.

```
inline double max(double a, double b) {
    if( a > b )
        return a;
    return b;
}

#define MAX(a,b) ((a)>(b)?(a):(b))

void f(double a, double b) {
    double c = max(++a,++b);
    double d = MAX(++a,++b);
}
```

Name control

- En C el único control sobre los nombres era declarar variables globales o funciones como **`static`**.
- Dos significados para el keyword **`static`**:
  - Para variables dentro de funciones. Estas variables no son de stack, están creadas en una *static data area*.
    - Concepto de static storage.
  - Para variables globales y funciones. Son sólo visibles dentro de esa unidad de compilación. No pueden ser accedidas desde otro archivo fuente.
    - Concepto de static linkage.

Namespaces

- En C existe un único espacio por ejecutable.
- C++ ofrece un mecanismo para evitar colisiones de nombres, provisto por el keyword **`namespace`**.
- Calificación del nombre de identificadores.

```
namespace A { int x = 5; }

namespace B { int x;     }

void f() {
    B::x = A::x;
}
```

Namespaces

- Keyword **`using`** para facilitar el acceso a los miembros de un namespace:

```
namespace A {
    struct X { int x; };
    int f(struct X *pX);
    int a;
}

void g() {
    using namespace A;
    X x;
    int a = f(&x);
    A::a = 3;
}
```

```
void f() {
    using namespace A;
    X x;
    a = f(&x);
}

void h() {
    using A::X;
    X x;
    int b = f(&x);
    A::a = 3;
}
```

```
std::cout << "Test\n";
```

Namespaces

```
namespace sp {
    int f(int a) { return a; }
    int x;
}

int f(int);

int g();

namespace sp {
    int z;
}

int g() {
    using namespace sp;
    return sp::f(z);
}
```

- Los namespaces pueden ser reabiertos y continuar con sus definiciones y/o declaraciones.
- Inclusive en distintas unidades de compilación.

Namespace `std`

```
#include <iostream>
#include <cstdlib>
#include <cassert>

using namespace std;

int main()
{
      int *p = (int*) malloc(8);
      assert(p);

      cout << " p: "
           << p << endl;

      free(p);
      return 0;
}
```

- Las funciones de las biblioteca standard de C++ están definidas en el namespace **`std`**.
- Los headers de la biblioteca standard de C++ no tienen la terminación **`.h`**.
- Los headers compatibles con los standard de C, sin la extensión **`.h`**, comienzan con '**`c`**' y en el namespace **`std`**.

Namespaces anidados y alias

```
namespace MyNamespace {
    int n;
    namespace Inner {
        int z;
        int g(int);
    }
}

int MyNamespace::Inner::g(int a) {
    return a << 1 + z;
}

void f() {
    namespace MNI = MyNamespace::Inner;
    MNI::z = 5;
}
```

Unnamed namespaces

```
namespace {
    int y;
    void f() {
        y *= 5;
    }
} // namespace

void g() {
    y = 1;
    f();
}
```

- En C++ es preferible a utilizar el calificador **`static`** para obtener visibilidad local a la unidad de compilación.

Composición y desambiguación

```
namespace Her_lib {
    class String { /* ... */ };
    template <class T> class Vector { /* ... */ };
}
namespace His_lib {
    class String { /* ... */ };
    template <class T> class Vector { /* ... */ };
}
namespace My_lib {
    using namespace His_lib;
    using namespace Her_lib;
    using Her_lib::String;          // disambiguation
    using His_lib::Vector;          // disambiguation
    template<class T> class List { /* ... */ };
}
```

Reutilización de código

- Mejorar el copy/paste/change.
- Creación de nuevas clases en base a las ya existentes, sin necesidad de tocar el código preexistente.
  - *Composición*: crear nuevas clases compuestas por objetos de clases ya existente.
  - *Herencia*: crear nuevas clases del tipo de una clase ya existente, sin modificarla.
- La herencia es uno de los aspectos más importantes de la programación orientada a objetos.

Composición

```
class X {
        int i;
    public:
        X() { i = 0; }
        void set(int ii) { i = ii;   }
        int read() const { return i; }
};
class Y {
        int i;
    public:
        X x;
        Y() { i = 0; }
        void f( int ii) { i = ii;   }
        int g() const   { return i; }
};
```

```
int main()
{
    Y y;
    y.f(47);
    y.x.set(37);
}
```

Composición

```
class X {
        int i;
    public:
        X() { i = 0; }
        void set(int ii) { i = ii;   }
        int read() const { return i; }
};
class Y {
        X x;
    public:
        Y() { }
        void f( int i) { x.set(i);         }
        int g() const  { return x.read(); }
};
```

```
int main()
{
    Y y;
    y.f(47);
    y.g();
}
```

Herencia

```
class X {
        int i;
    public:
        X() { i = 0; }
        void set(int ii) { i = ii; }
        int read() const { return i; }
};
class Y : public X {
        int i;
    public:
        Y() { i = 0; }
        void set(int ii) {
           i = ii; X::set(ii);
        }
        int g() const {
           return X::read();
        }
};
```

```
int main(){
    Y y;
    y.set(47);
    y.read();
}
```

Herencia

- **Y** hereda de **X** - **X** es una clase base de **Y**.
  - **Y** contiene todos los atributos de **X**
  - **Y** contiene todos los métodos de **X**.
  - Todos los elementos privados de **X** son privados de **Y**.
  - La herencia se especificó como **public**, por lo que todos los miembros públicos de **X** son accesibles como públicos a través de **Y**.
  - El método **set** está redefinido en **Y**.
  - El método **read** no está redefinido en **Y**, se activa el de **X** cuando se llama a través de un objeto de clase la clase **Y**.

Lista de inicialización en el constructor

- El compilador garantiza la ejecución de los constructores.
- El constructor de la clase base se ejecuta antes que el de la clase derivada.
- Para mandarle argumentos al constructor de una clase base se hace en la lista de inicialización.

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

class Y : public X {
    public:
        Y(int ii) : X(i) { }
};

class Z {
        X myx;
    public:
        Z(int ii) : myx(ii) { }
};
```

Herencia y composición

```
class A {
        int i;
    public:
        A(int ii)
            : i(ii) { }
        void f() const { }
};

class B {
        int i;
    public:
        B(int ii)
            : i(ii) { }
        void f() const { }
};
```

```
class C : public B {
        A a;
   public:
        C(int ii)
            : B(ii), a(ii) { }

        void f() {
            a.f();
            B::f();
        }
};
```

Orden de construcción y destrucción

- Primero se ejecutan los constructores de las clases base.
- Luego se ejecutan los constructores de objetos miembros.
- Finalmente se ejecuta el constructor de la clase derivada.
- Los destructores se ejecutan automáticamente en el orden inverso a los constructores.
- Este comportamiento de llamado automático de constructores y destructores no es tal para los métodos comunes. Deben activarse explícitamente.

Nombres de métodos (Name hiding)

```
class Base {
    public:
        int f() const {
            cout << "Base::f\n";
            return 1;
        }
        int f(string) const {
            return 1;
        }
        void g() {}
};

class D1 : public Base {
    public:
        // redefinicion
        void g() const {}
};
```

```
class D2 : public Base {
    public:
        int f() const { // redefinición
            cout << "D2::f\n";
            return 2;
        }
};
class D3 : public Base {
    public:
        void f() { cout << "D3::f\n"; }
};
class D4 : public Base {
    public:
        int f(int) {
            cout << "D4::f\n";
            return 4;
        }
};
```

Nombres de métodos (Name hiding)

```
int main() {
    string s("hello");
    D1 d1;
    int x = d1.f();
    d1.f(s);
    D2 d2;
    x = d2.f();
    // d2.f(s);      // error, hidden
    D3 d3;
    // x = d3.f();   // error, hidden
    d3.f();
    D4 d4;
    // x = d4.f();   // error, hidden
    d4.f(1);
}
```

```
Base::f
D2::f
D3::f
D4::f
```

Métodos que no se heredan automáticamente

- Constructores.
- Destructores.
- `operator=()`.

- Las funciones **`static`** o de clase cumplen las mismas reglas que los métodos no **`static`**.

Composición vs. herencia

- Composición se utiliza cuando un objeto está compuesto o contiene otros, pero no se quiere sus interfaces expuestas. Típicamente los objetos embebidos son privados.

  Ejemplo: un auto tiene motor, ruedas, puertas, etc. en este caso se puede pensar que no sean objetos privados.

- Se utiliza herencia cuando el objeto derivado cumple la relación de "*es un*" con el objeto padre. El objeto derivado hereda la interface del padre.

  Ejemplo: un auto es un vehículo.

Herencia privada

- Una clase derivada puede heredar private de su base.
- La interface de su base no puede ser accedida.
- Es probable que sea mejor utilizar composición.

```
class Pet {
    public:
        void eat();
        void speak();
        int sleep();
        int sleep(int);
};
```

```
class Goldfish : private Pet {
    public:
        using Pet::eat;
        using Pet::sleep;
};
```

```
int main() {
    Goldfish bob;
//    bob.speak() // error
    bob.eat(); bob.sleep();
    bob.sleep(1);
}
```

`protected`

- Los miembros **`protected`** se comportan como **`public`** para clases derivadas y como **`private`** para cualquier otro uso.
- Es preferible definir los atributos **`privados`** y los métodos **`protected`**. Para no poner dependencias en las clases derivadas de la implementación de la clase padre.
- Existe la herencia **`protected`**. Se utiliza raramente. Para las clases subderivadas se comporta como herencia **`public`**, para otros usos como **`private`**.

Herencia múltiple

- Se puede heredar de más de una clase.
- Parece simple, se agregan más clases en la lista de clases base separadas por comas. Sin embargo, esto introduce toda una nueva clases de problemas de ambigüedades, duplicaciones, etc.
- Es cuestión de debate si tiene sentido como parte del diseño.

```
class A { /* */ };
class B { /* */ };
class C : public A, public B {
    /* */
};
```

Upcasting

- Cuando una clase hereda de otra, cumple una relación de "*es un*".
- La interface de la clase base está presente en la clase derivada.
- El compilador acepta un puntero o una referencia a una clase derivada cuando necesita un puntero o una referencia a la clase base.
- Toda la funcionalidad que alguien puede esperar a través de un puntero o referencia de un objeto de una clase base está presente en un puntero o referencia a un objeto de la clase derivada.

Upcasting

```
enum note { do, re, mi };

class Instrument {
    public:
        void play(note) const {}
};

class Wind : public Instrument {
    /* ... */
};


void tune(Instrument &i) {
    // ...
    i.play(do);
}
```

```
int main() {
    Wind flute;
    tune(flute);
    return 0;
}
```

Upcasting

```
void f() {
    Wind w;
    Instrument *ip = &w;   // Upcast
    Instrument &ir = w;    // Upcast
    ip->play(mi);
    ir.play(re);
}
```

- El upcasting pierde información sobre un objeto.
- Hecho el upcast, sólo se puede utilizar la interface de la clase base a través del puntero o referencia.
- Si algún método fue redefinido, se pierde esa información y se llama al método de la clase base.

Composición vs. herencia

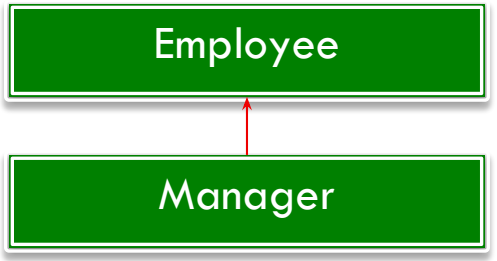
- Una manera de determinar si conviene utilizar herencia o composición es preguntarse si va a ser necesario hacer un upcast desde la nueva clase a la vieja.
  - Si es necesario utilizar upcast, la relación apropiada es la de herencia.
  - Si no es necesario o conveniente utilizar upcast, es preferible utilizar composición.

Clases Derivadas

```
struct Employee {
    string first_name, family_name;
    char   middle_initial;
    Date   hiring_date;
    short  department;
    // ...
};

struct Manager {
    Employee emp;           // manager's employee record
    set<Employee *> group; // people managed
    short level;
// ...
};

//    (Manager *) != (Employee *)
```

Clases Derivadas

```
struct Manager : public Employee {
    set<Employee *> group; // people managed
    short level;
/ / ...
};

void f(Manager &m1, Employee &e1)
{
    list<Employee *> elist { &m1, &e1 };
                //    elist.push_front(&m1);
                //    elist.push_front(&e1);
}
```



Clases Derivadas

```
void g(Manager mm, Employee ee)
{
    Employee *pe = &mm;                 // ok: every Manager is an Employee

    Manager *pm = &ee;                  // error: not every Employee is a Manager

    pm->level = 2;                      // disaster: ee doesn't have a 'level'

    pm = static_cast<Manager *>(pe);    // brute force: works because pe points
                                        // to the Manager mm

    pm->level = 2;                      // fine: pm points to the Manager mm that
                                        // has a 'level'

};
```

Clases Derivadas: funciones miembro

```
class Employee {
        string first_name, family_name;
        char middle_initial;
    // ...
    public:
        void print() const;
        string full_name() const {
            return first_name+ ' '  + middle_initial + ' ' +
                family_name;
        }
// ...
};
////////////////////////////////////////////////////
class Manager : public Employee {
    // ...
    public:
        void print() const;
};
```

Clases Derivadas: funciones miembro

```
void Manager::print() const  // OK
{
    cout << "name is " << full_name() << '\n';
// ...
};

void Manager::print() const   // Not OK
{
    cout << "name is " << family_name << '\n'; // error
// ...
};

void Manager::print() const  // OK
{
    Employee::print();  // print employ information
    cout << "level: " << level << "\n";   // manager info
};
```

Clases Derivadas: Copia

```
class Employee {
// ...
    Employee &operator=(const Employee &);
    Employee(const Employee &);
};

void f(const Manager &m)
{
    Employee e = m; // construct e from Employee part of m
    e = m;          // assign Employee part of m to e
};
```

Clases Derivadas: Jerarquías

```
class Employee {
/* ... */
};
class Manager : public Employee {
/* ... */
};
class Director : public Manager {
/* ... */
};
```

Clases Derivadas: Jerarquías

```
class Temporary { /* ... */ };
class Secretary : public Employee { /* ... */ };
class Tsec : public Temporary, public Secretary{ /* ... */ };
class Consultant : public Temporary,public Manager{/*... */ };
```