C++ visto como un C mejorado

Instituto Balseiro

Compatibilidad C / C++

- Con pocas excepciones, C++ es un superset de C.
- La mayoría de las diferencias provienen del mayor énfasis de C++ en el chequeo de tipos.
- Programas bien escritos en C son en general programas válidos en C++
- La mayoría de las diferencias entre C y C++ pueden ser encontradas y diagnosticadas por el compilador.

Funciones

- Uso de prototipos como forma de darle información al compilador.

```
int translate(float x, float y, float z);
```

ó

```
int translate(float, float, float);
```

- En la definición C++ soporta “unnamed” arguments.
- Funciones sin argumentos:

```
int f1(void);  // o simplemente int f1()  (C++)
```

- Funciones con lista variable de argumentos:

```
int f1(...);  // hay mejores alternativas
```

Ejemplo de argumentos variables

```
void ShowVar(const char* szTypes, ...)
{
    va_list vl;
    int i;

    //  szTypes is the last argument
    //   others must be accessed using va_list
    va_start(vl, szTypes);

    // Step through the list.
    for (i = 0; szTypes[i] != '\0'; ++i) {
        union Printable_t {
            int     i;
            float   f;
            char    c;
            char* s;
        } Printable;

        switch (szTypes[i]) {   // Type.
            case 'i':
                Printable.i = va arg(vl, int);
                printf("%i\n", Printable.i);
                break;
            case 'f':
                Printable.f = va arg(vl, double);
                printf("%f\n", Printable.f);
                break;
            case 'c':
                Printable.c = va arg(vl, char);
                printf("%c\n", Printable.c);
                break;
            case 's':
                Printable.s = va arg(vl, char*);
                printf("%s\n", Printable.s);
                break;
            default:
                break;
        }
    }
    va_end(vl);
}


////////////////////////////////
int main()
{
    ShowVar("fcsi", 32.4f, 'a', "Test", 4);
    return 0;
}
```

Funciones, retorno

- Las funciones en C++ deben especificar el tipo de retorno.

```
int f1(void);                  // Returns an int, no arguments

int f2();                      // Like f1() in C++ but not in C

float f3(float, int, char); // Returns a float

void f4(void);                 // no arguments, returns nothing
```

Control de flujo

- Expresiones lógicas: similar al C. Puede contener sentencias algebraicas, relacionales, lógicas o de bits.
- Bloques condicionales: **`if`**, **`if`**...**`else`**, **`switch`**, if aritmético (`?:`).
- Bloques repetitivos: **`while`**, **`do`**...**`while`**, **`for`**.
- **`break`**, **`continue`** y **`goto`**.

Tipos Fundamentales

- Tipo booleano (**`bool`**)
- Tipos caracter (**`char`**)
- Tipos enteros (**`int`**)
- Tipos reales (**`float/double`**)
- Tipos Enumerados para representar conjuntos de valores (**`enums`**)
- Tipo que indique ausencia de información de tipo (**`void`**)

tipos integrales

tipos aritméticos

Tipos Fundamentales

- Tipos puntero (`int *`)
- Tipos arreglo (`char []`)
- Tipos referencia lvalue (`float &`)
- Tipos referencia rvalue (`float &&`)
- Estructuras de datos, clases y uniones

Tipo bool

- Puede tener uno de dos valores posibles: **`true`**, **`false`**

```
void f(int a, int b)
{
    bool b1 = (a == b);
    // ...
}

bool greater(int a, int b)
{
    return a > b;
}

// true convertido a entero      => 1
// false convertido a entero     => 0
// int != 0 convertido a bool    => true
// int == 0 convertido a bool    => false


bool b = 7;
int i = true;
```

Tipo char y literales

- 3 variantes: **`char`**, **`signed char`** y **`unsigned char`**
- Casi universalmente, un char son 8 bits

```
char k = 'a';  // inicialización con una constante literal


void printHex(int val) {
    static const char alpha[] = "0123456789ABCDEF";
    if( val ) {
        printHex(val >> 4);
        printf("%c", alpha[val & 0xF]);
    }
}
```

- Raw string literals, backslash **`'\'`** no funciona como escape char:

```
string regexp = R"(\w\\w)";
const char *p = R"("quoted string")";      // "quoted string"
```

- Soporte para Unicode con **`wchar_t`**, UTF-8, UTF-16 y UTF-32 literals.

Tipo int y literales

- 3 variantes: `int`, `signed int` y `unsigned int`
- 4 tamaños: `short int`, "plain" `int`, `long int` y `long long int`

- Literales:

decimal :	`0`	`2`	`63`	`83`
octal:	`0`	`02`	`077`	`0123`
hexadecimal:	`0x0`	`0x2`	`0x3f`	`0x53`

Tipo reales

- Por defecto un tipo real es **`double`**
- 3 tamaños: **`float`**, **`double`** y **`long double`**

- Literales:

```
1.23  .23f  0.23  1.f  1.0  1.2e10  1.23e-15L
```

Enumeraciones

- Plain enums: importan los nombres al namespace que contiene su definición

```
// Sin nombre
enum { ASM , AUTO , BREAK };
     // ASM == 0, AUTO == 1, BREAK == 2

// Con nombre (define tipo)
enum EnumName { ASM , AUTO , BREAK };
```

- Class enums: con nombre, scope y tamaño (también define tipo). C++ 11.

```
enum class keyword : int { ASM , AUTO , BREAK };

void f(keyword key)  {
    switch( key ) {
        case keyword::ASM:
            // do something
            break;
        case keyword::BREAK:
            // do something
            break;
        default:
            // do something
            break;
    }
}
```

Class Enums

```
enum class Traffic_light { red, yellow, green };     // default int

enum class Warning { green, yellow, orange, red };   // default int

Warning a1 = 7;               // error: no int->Warning conversion
int a2 = green;               // error: green not in scope
int a3 = Warning::green;      // error: no Warning->int conversion
Warning a4 = Warning::green; // OK

void f(Traffic_light x)
{
    if( x == 9 ) {                    // error: 9 not a Traffic_light
        /* ... */
    }
    if( x == red ) {                  // error: no red in scope
        /* ... */
    }
    if( x == Warning::red ) {         // error: x is not a Warning
        /* ... */
    }
    if( x == Traffic_light::red ) {   // OK
        /* ... */
    }
}
```

Plain Enums

- Plain enums: rango de una enumeración [0:2^k-1] ó [-2^k:2^k-1] si hay negativos. No está definido el tamaño.

```
enum e1 { dark, light };            // range 0:1
enum e2 { a=3, b=9 };               // range 0:15
enum e3 { min=-10, max=1000000 };   // range -1048576:1048575


// Conversiones
enum flag { x=1, y=2, z=4, e=8 };   // range 0:15
flag f1 = 5 ;                       // type error: 5 is not type flag
flag f2 = flag(5);                  // ok: flag(5) is of type flag and
                                    // within the range of flag
flag f3 = flag(z|e );               // ok: flag(12) is of type flag and
                                    // within the range of flag
flag f4 = flag(99);                 // undefined: 99 is not within the
                                    // range of flag
```

Plain Enums: Problemas

```
enum Traffic_light { red, yellow, green };
enum Warning { green, yellow, orange, red };
// problem: two definitions of yellow (to the same value)
// problem: two definitions of red (to different values)

Warning a1 = 7;                 // error: no int->Warning conversion
int a2 = green;                 // OK: green converts to int
int a3 = Warning::green;        // OK: Warning->int conversion
Warning a4 = Warning::green;    // OK

void f(Traffic_light x)
{
    if (x == 9) {                       // OK (but Traffic_light has no 9)
        /* ... */
    }
    if (x == red) {                     // error: two reds in scope
        /* ... */
    }
    if (x == Warning::red) {            // OK (Ouch!)
        /* ... */
    }
    if (x == Traffic_light::red) {      // OK
        /* ... */
    }
}
```

const

- C y C++ soportan constantes simbólicas (sin tipo, sin address).

```
#define PI 3.141592
```

- C++ introduce el concepto de “named constant”, cualquier intento de modificar el valor será interpretado como error por el compilador.

```
const int x = 10;
```

- En C una constante **const** global tiene linkage **extern**.
  En C++ no, el linkage por default es **static** y debe ser inicializada salvo que esté explícitamente declarada **extern**.

Constantes

Dos tipos de “constantes”:

- **`constexpr`**: asegura su evaluación en tiempo de compilación. C++ 11.
  - Permite nombrar valores literales o guardados en variables.
  - Requeridos por C++ para tamaño de arrays, **`case`** y argumentos de templates.
  - Mejora de performance o evita problemas de inicialización cuando hay multithreading.
  - Se puede utilizar para funciones evaluadas en tiempo de compilación.
- **`const`**: Inmutabilidad en el scope y en definición de interfaces.

constexpr

```
constexpr int isqrt_helper(int sq, int d, int a)
{
    return sq <= a ? isqrt_helper(sq+d,d+2,a) : d;
}

constexpr int isqrt(int x)
{
    return isqrt_helper(1,3,x)/2 - 1;
}

constexpr int s1 = isqrt(9);       // 3
constexpr int s2 = isqrt(1234);

const int x = 7;
const string s = "asdf";
const int y = sqrt(x);

constexpr int xx = x;     // OK
constexpr string ss = s; // error: s not a constant expression
constexpr int yy = y;     // error: sqrt(x) not a constant expression
```

`constexpr` literal types

```
struct Point {
      int x,y,z;
      constexpr Point up(int d) { return {x,y,z+d}; }
      constexpr Point move(int dx, int dy) { return {x+dx,y+dy}; }
      // ...
  };

  constexpr Point origo {0,0};
  constexpr int z = origo.x;

  constexpr Point a[] = {
      origo , Point{1,1} , Point{1,1}, origo.move(3,3)
  };

  constexpr int x = a[1].x;
  constexpr Point xy{0,sqrt(2)};  // error: sqrt(2) not a constant expression
```

- Provee programación con tipos al momento de compilación.
  Antes de C++ solo se podía con **`int`**s y sin funciones.

Objetos y punteros const

```
const int model = 90;           // model is a const
const int v[] = { 1, 2, 3 };    // v[i] is a const
const int x;                    // error: no initializer

// const: No modifica el tipo, restringe como se utiliza el objeto.
void g(const X *p)
{
    // no se puede modificar *p
}

void h()
{
    X val;      // val puede ser modificado acá
    g(&val);    // la firma de g() asegura que no modificará val
}

//Cuando se utiliza un puntero, hay 2 objetos: el puntero mismo y el objeto apuntado.
// El prefijo const se refiere al objeto apuntado, no al puntero.
// Para declarar el puntero constante, hay que utilizar * const.
char *const cp;           // const pointer to char
char const *pc;           // pointer to const char
const char *pc2;          // pointer to const char
const char *const cpc;    // const pointer to const char
```

Referencias

- Pasaje de argumentos por valor

```
void swap(int x, int y)
{
    int t = x;
    x = y;
    y = t;
    cout << "In swap x: " << x << " y: " << y << endl;
}

void func()
{
    int a = 3,
        b = 5;
    swap(a, b);
    cout << "In func a: " << a << " b: " << b << endl;
}
```

Referencias

- Pasaje de argumentos por puntero

```
void swap(int *px, int *py)
{
    int t = *px;
    *px = *py;
    *py = t;
    cout << "In swap x: " << *px << " y: " << *py << endl;
}

void func()
{
    int a = 3,
        b = 5;
    swap(&a, &b);
    cout << "In func a: " << a << " b: " << b << endl;
}
```

Referencias

- Los punteros permiten pasar gran cantidad de datos a bajo costo. En vez de pasar los datos, pasamos su dirección de memoria.
- Diferencias en el uso de punteros con respecto a objetos:
  - Utilizar `*p` en vez de `obj` y `p->m` en vez de `obj.m`.
  - Más cuidado cuando se utiliza un puntero, puede ser `nullptr`.
  - El puntero puede ser apuntado a distintos objetos.
  - Utilización natural de operadores `x+y`, no `&x+&y`.
- Referencias: nombre alternativo a un objeto, un alias.
  - Misma sintaxis que objetos.
  - Siempre refiere al objeto con el que fue inicializada.
  - No existe un “null reference”.

Referencias

- Para distinguir entre lvalue/rvalue y **`const`**/no-**`const`**, hay 3 tipos de referencias:
- *lvalue references*: para referir a objetos cuyo valor se puede o se quiere modificar. **`X&`** es una referencia a **`X`**.
- **`const`** *references*: para referir a objetos cuyo valor no se quiere modificar. **`const X&`** es una referencia constante a **`X`**.
- *rvalue references*: para referir a objetos cuyo valor no necesita ser preservado después de usarlo (un valor temporario). Se asume que el objeto no va a ser utilizado nuevamente. **`X&&`** es una *rvalue reference* a **`X`**.

Referencias

```
string var{"Cambridge"};
string f();

string &r1;                // error, r1 no está inicializada
extern string &r2;         // OK, definida en otro lado
string &r3{var};           // OK, lvalue (r3 es una referencia a var)

string &r4{f()};           // error, el retorno de f() es un string temporario
string &r5{"Princeton"};   // error, temporario

string &&rr1{f()};         // OK, temporario

string &&rr2{var};         // error, lvalue

string &&rr3{"Oxford"};    // OK, temporario

const string &cr1{"Harvard"}; // OK
// string temp{"Harvard"}; const string &cr1{temp};
```

Referencias

- Pasaje de argumentos por referencia

```
void swap(int &x, int &y)
{
    int t = x;
    x = y;
    y = t;
    cout << "In swap x: " << x << " y: " << y << endl;
}

void func()
{
    int a = 3,
        b = 5;
    swap(a, b);
    cout << "In func a: " << a << " b: " << b << endl;
}
```

Referencias

```
template<class T>

class vector
{
        T* elem;
        // ...
    public:
        T& operator[](int i) { return elem[i]; }              // return reference to element
        const T& operator[](int i) const { return elem[i]; } // return reference
                                                              // to const element
        void push_back(const T& a);                           // pass element to be added by reference
        // ...
};

void f(const vector<double>& v)
{
    double d1 = v[1];       // copy the value of the double referred to by v.operator[](1) into d1

    v[2] = 7;               // place 7 in the double referred to by the result of v.operator[](2)

    v.push_back(d1);        // give push_back() a reference to d1 to work with
}
```

Scope

```
// How variables are scoped
int main() {
    int scp1;
    // scp1 visible here
    {
        // scp1 still visible here
        //.....
        int scp2;
        // scp2 visible here
        //.....
        {
            // scp1 & scp2 still visible here
            //..
            int scp3;
            // scp1, scp2 & scp3 visible here
            // ...
        } // <-- scp3 destroyed here
        // scp3 not available here
        // scp1 & scp2 still visible here
        // ...
    } // <-- scp2 destroyed here
    // scp3 & scp2 not available here
    // scp1 still visible here
    //..
} // <-- scp1 destroyed here
```

Definición de variables al vuelo

```
int main()
{
    {   // Begin a new scope
        int q = 0; // C requires definitions here
        //..

        // Define at point of use:
        for( int i = 0; i < 100; i++ ) {

            q++; // q comes from a larger scope

            // Definition at the end of the scope:
            int p = 12;
        }

        int p = 1;  // A different p
    } // End scope containing q & outer p

    // ...
}
```

Storage allocation

- Variables globales: declarada fuera de toda función y disponible para todo el programa (incluso para código en otros archivos).
- Variables de namespace: declaradas fuera de toda función, dentro de un **`namespace`**.
- Variables locales: declaradas dentro de una función
- Va

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

void func() {
    static int i = 0;
    cout << "i = " << ++i << endl;
}
int main() {
    for( int x = 0; x < 10; x++ )
        func();
}
```

Linkage

- Cuando **`static`** es aplicado a una función o una variable que está fuera de cualquier función define a ese nombre solo visible dentro de esa unidad de compilación (file scope).

```
// FileStatic.cpp
// File scope demonstration. Compiling and linking this file with FileStatic2.cpp
// will cause a linker error

// File scope means only available in this file:
static int fs;
int main() {
    fs = 1;
}

----
// FileStatic2.cpp : Trying to reference fs
extern int fs;
void func(){
    fs = 100;
}
```

Linkage

- **`extern`**: le indica al compilador que la variable o función existe.

```
// Forward.cpp
// Forward function & data declarations
#include <iostream>
using namespace std;

// This is not actually external, but the
// compiler must be told it exists somewhere:
extern int i;
extern void func();

int main() {
    i = 0;
    func();
}

int i; // The data definition
void func() {
    i++;
    cout << i;
}
```

`volatile`

- El calificador **`volatile`** aplicada a una variable evita que el compilador realice optimizaciones basadas en la estabilidad de una variable.

- Debe usarse este calificador cuando se debe leer un valor que está fuera del control del código (registro en una placa).

- Otro ejemplo es en programas con múltiples threads.

Operadores

- Los operadores de C son soportados por C++:
  - Algebraicos: `=  ++  --  +  -  *  /  %.` (y auto-operadores)
  - Relacionales: `==  >=  <=  >  <  !=`
  - Lógicos: `&&  ||  !`
  - De bits: `&  |  ~  ^`
  - De shifteo: `>>  <<`
  - `sizeof()`

```
#include <iostream>
void printBinary(const unsigned char val)
{
  for( int i = 7; i >= 0; i-- )
    if( val & (1 << i) )
      std::cout << '1';
    else
      std::cout << '0';
}
```

Operadores de cast

- Originalmente, el comité de estandarización de C++ quiso "deprecar" el casting de C, forzando al uso de los nuevos operadores de cast, pero debido a que esos son ampliamente usados en código legacy y a que muchos compiladores C++ sirven también como compiladores C es que se continúa soportándolos, pero se recomienda el uso de los nuevos operadores de cast.

```
void *p = &x;
int n = (int) p; // C-style cast
```

Operadores de cast

- Hay casteos como por ejemplo la promoción de **`int`** a **`double`** que son seguros, pero también se pueden hacer cosas poco felices:

```
const char *msg = "don't touch!";
unsigned char *p = (unsigned char *) msg; // intentional?
```

Nuevos operadores de cast

- `static_cast:` realiza casteo seguros y relativamente portables. Realiza el casteo/conversión en tiempo de compilación.

```
// Se pueden usar generalmente para documentar explícitamente
// un casteo que de todas maneras ocurriría automáticamente:

bool b = true;
int n = static_cast<int>(b);

// En otros contextos, es mandatorio:
int n = 4;
void *pv = &n;
int *pi2 = static cast<int *>(pv); // mandatorio, problema
// El cast puede hacer cambiar la representación binaria.
float p = 4.1;
int pi2 = static cast<int>(p);     // mandatorio
int k = p;      // warning: conversion to int from float

// O agarrar errores:
const char *msg = "don't touch!";
u8 *p = static_cast<u8 *>(msg);    // error!
```

Nuevos operadores de cast

- **`const_cast:`** operador que remueve o agrega el calificador **`const o volatile.`**

```
struct A {
    void func() {}                      // non-const member function
};
void f(const A& a) {
    a.func();                           // error, calling a non-const function
}

void f2(const A& a) {
    A &ref = const_cast<A&>(a);         // remove const
    ref.func();                         // now fine
}
```

- Ojo! No quiere decir que la variable ahora puede modificarse.

Nuevos operadores de cast

- **`reinterpret_cast`**: realiza casteos peligrosos y no portables. No cambia la representación binaria del objeto fuente.

```
float f = 10;
unsigned char *p = reinterpret_cast <unsigned char*>(&f);
for( int j=0; j<4; ++j )
    cout << p[j] << endl;
```

Nuevos operadores de cast

- **`dynamic_cast:`** es utilizado cuando la conversión debe acceder a la información de tipo en runtime (RTTI) de un objeto más que a su tipo estático. Es un cast seguro, si no puede realizarse, retorna NULL, o dispara una excepción en el caso de referencias.

  Dos usos típicos son:

  - "downcasting": es decir castear un puntero a clase base o referencia a un puntero o referencia de un tipo derivado
  - "crosscasting": convertir un objeto a una de sus clases bases secundarias.

Creación de tipos compuestos

- Aliases (no definen un nuevo tipo en realidad)

```
typedef unsigned long ulong;
using ulong = unsigned long;  // C++11
```

- Combinando tipos nativos con **`struct`**

```
struct St1 {
    char   c;
    int    i;
    float  f;
    double d;
};

void f() {
    St1 s1, s2;                    // en C: struct St1 s1, s2;
    s1.c = 'a';  s1.i = 123; s1.f = 0.2; s1.d = 3.14;

    St1 s3{ 'b', 234, 0.1, 2.7 };  // en C, haría falta un =
}
```

Creación de tipos compuestos

```
void g()
{
    St1 s1;
    s1.c = 'a';
    s1.i = 123;
    s1.f = 0.2;
    s1.d = 3.14;

    St1 *ps1 = &s1;
    ps1->i *= ps1->i;

    St1 &rs1 = s1;
    rs1.d *= 2;

    const St1 &crs1 = s1;
    crs1.f = 1.0;              // Error
}
```

Creación de tipos compuestos

- Ahorrando memoria con **`union`**

```
#include <iostream>
using namespace std;
struct St1 {
    char   c;
    int    i;
    float  f;
    double d;
};
union Un1 {
    char   c;
    int    i;
    float  f;
    double d;
};
int main() {
    St1 s1;
    Un1 u1;
    cout << (void *) &s1.c << " " << &s1.i << " " << &s1.f << " " << &s1.d << endl;
    cout << (void *) &u1.c << " " << &u1.i << " " << &u1.f << " " << &u1.d << endl;
    return 0;
}
```

```
0x28cd00  0x28cd04  0x28cd08  0x28cd10
0x28ccf8  0x28ccf8  0x28ccf8  0x28ccf8
```

Creación de tipos compuestos

- Ahorrando memoria con **union**

```
#include <iostream>
using namespace std;
struct __atribute__(__packed__)  St1 {
    char   c;
    int    i;
    float  f;
    double d;
};
union Un1 {
    char   c;
    int    i;
    float  f;
    double d;
};
int main() {
    St1 s1;
    Un1 u1;
    cout << (void *) &s1.c << " " << &s1.i << " " << &s1.f << " " << &s1.d << endl;
    cout << (void *) &u1.c << " " << &u1.i << " " << &u1.f << " " << &u1.d << endl;
    return 0;
}
```

```
0x28cd00  0x28cd01  0x28cd05  0x28cd09
0x28ccf8  0x28ccf8  0x28ccf8  0x28ccf8
```

Arreglos

- Conjunto de variables del mismo tipo referenciadas a través de un nombre único.

```
int a[10];

// Acceso indexado (zero-based y no checkeado)
a[5] = 47;  // a[13] = 3;   a[-2] = 5;

// arreglo de estructuras
struct P3D {
    int x, y, z;
};
int main()
{
    P3D p[10];
    for( int i = 0; i < sizeof(p)/sizeof(p[0]); ++i ) {
        p[i].x = i + 1;
        p[i].y = i + 2;
        p[i].z = i + 3;
    }
}
```

Punteros y Arreglos

- Arreglo ≡ puntero constante al primer elemento (salvo que no ocupa memoria)

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

int main()
{
    int a[10];

    cout << "a:      " << a << endl;
    cout << "&a[0]: " << &a[0] << endl;

    return 0;
}
```

```
a:      0x28ccf0
&a[0]: 0x28ccf0
```

Aritmética de punteros

- Comparación de punteros.
- Suma de un entero a un puntero. Resultado: puntero.
- Resta de 2 punteros. Resultado: entero con signo (**`intptr_t`**).
- Incremento/decremento de un puntero (++ --).

```
int main()
{
    int a[10] = { 1, 2, 3, 4, 5 };
    int *pa = &a[0],                    // pa = a;
        *pb;
    pb = pa + 3;                        // pb = &pa[3];
    assert((pb - pa) == 3);

    cout << "pa: " << pa << " *pa: " << (*pa)
         << "  (pa+1): " << (pa+1) << "  *(pa+1): " << *(pa+1) << endl;

    return 0;
}
```

```
pa: 0x28ccf0 *pa: 1   (pa+1): 0x28ccf4   *(pa+1): 2
```

Punteros a función

- Una función compilada y cargada para ser ejecutada ocupa un bloque de memoria y esa memoria tiene una dirección que puede ser almacenada en un puntero.

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

void func() {
    cout << "func() called..." << endl;
}

int main() {
    void (*fp)();           // Define un puntero a función como variable

    fp = func;              // Inicialización
    fp();                   // Llamado a la función, operador ()

    void (*fp2)() = func;   // Define e inicializa otra variable
    fp2();
}
```

Arreglo de punteros a función

```
#include <iostream>
using namespace std;
// A macro to define dummy functions:
#define DF(N) void N() { \
   cout << "function " #N " called..." << endl; }

DF(a); DF(b); DF(c); DF(d); DF(e); DF(f); DF(g);

void (*func_table[])() = { a, b, c, d, e, f, g };

int main() {
    while( 1 ) {
        cout << "press a key from 'a' to 'g' "
                "or q to quit" << endl;
        char c, cr;
        cin.get(c); cin.get(cr);  // second one for CR
        if( c == 'q' )
            break;                // ... out of while(1)
        if( c < 'a' || c > 'g' )
            continue;
        func_table[c - 'a']();
    }
}
```

Arreglo de punteros a función

```
#include <iostream>
using namespace std;
// A macro to define dummy functions:
#define DF(N) void N() { \
   cout << "function " #N " called..." << endl; }

DF(a); DF(b); DF(c); DF(d); DF(e); DF(f); DF(g);

// void (*func_table[])() = { a, b, c, d, e, f, g };

// typedef void (*FunPtr)();
using FunPtr = void(*)();

FunPtr func_table[] { a, b, c, d, e, f, g };

// ...
```

C style

- Biblioteca en C:

```
// CLib.h
typedef struct CStashTag {
    int size;                   /* Size of each space        */
    int quantity;               /* Number of storage spaces */
    int next;                   /* Next empty space          */
    unsigned char *storage;
} CStash;

void initialize(CStash *s, int size);
void cleanup(CStash *s);
int add(CStash *s, const void *element);
void *fetch(CStash *s, int index);
int count(CStash *s);
void inflate(CStash *s, int increase);
```

C style

```
// Clib.c
#include "CLib.h"

const int increment = 100;

void initialize(CStash *s, int sz)
{
    s->size = sz;
    s->quantity = 0;
    s->storage = 0;
    s->next = 0;
}

int add(CStash *s, const void *element)
{
    if( s->next >= s->quantity )
        inflate(s, increment);
    int startBytes = s->next * s->size;
    unsigned char *e = (unsigned char *) element;
    for( int i = 0; i < s->size; i++ )
        s->storage[startBytes + i] = e[i];
    s->next++;
    return s->next - 1;
}
```

```
// CLibTest.c
#include "CLib.h"

int main()
{
    int i;
    CStash intStash;

    initialize(&intStash, sizeof(int));

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

    for( i = 0; i < count(&intStash); i++ )
        cout << "fetch ( " << i << " ) = "
             << *(int *) fetch(&intStash, i)
             << endl;

    cleanup(&intStash);
}
```

Problemas

- Pasaje como argumento de la dirección de memoria de la estructura. El mecanismo de la biblioteca se mezcla con la semántica de las funciones.
- Colisión de nombres. Solucionable con decoración de los nombres: `CStash_initialize()` `CStash_clenaup()`.
- No hay inicialización ni cleanup automáticos. El programador tiene la responsabilidad.

Objeto

```
// CppLib.h

struct Stash {
    int size;
    int quantity;
    int next;
    unsigned char *storage;

    // Funciones o métodos
    void initialize(int size);
    void cleanup();
    int add(const void *element);
    void *fetch(int index);
    int count();
    void inflate(int increase);
};
```

Objeto

```
// CppLib.cpp
#include "CppLib.h"

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

int Stash::count()
{
    return next;
}

void Stash::cleanup()
{
    delete [] storage;
}
```

```
// CppLibTest.cpp
#include "CppLib.h"

int main()
{
    Stash intStash;

    intStash.initialize(sizeof(int));

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

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

    intStash.cleanup();
}
```

Diferencias / cosas nuevas

- No hace falta **`typedef`**. C++ utiliza el nombre de la estructura como un nuevo tipo automáticamente.
- Los datos miembros son exactamente igual que en C.
- Aparecen funciones en el cuerpo de la **`struct`**.
- Estas funciones (o métodos) no tienen el primer argumento con la dirección de la estructura. C++ se encarga, el mecanismo no se nota.
- No hay colisión de nombres, **`Stash::initialize()`** no colisiona con otras **`initialize()`**.

Diferencias / cosas nuevas

- Nuevo operador `::` de resolución de scope.
- Acceso a los miembros directo dentro de las funciones miembro, no hace falta **`s->size = sz`**, alcanza con poner **`size = sz`**.
- Si se quiere utilizar la dirección de memoria de la estructura sobre la que se está operando, existe el puntero **`this`**. Explícitamente: **`this->size = sz`**.