75.04/95.12 Algoritmos y Programación II

Guía 1, Ejercicios 8 y 9 modificados: Constructores

Universidad de Buenos Aires - FIUBA

Primer Cuatrimestre de 2018

Dadas una estructura S y una clase foo definidas a continuación, explicar qué imprime el programa cuando, al momento de preprocesar el código fuente, las macro "`MISSING_COPY_CONSTRUCTOR`" y "`MISSING_ASSIGNMENT_OPERATOR`"

(a) están definidas.

(b) no están definidas.

```
#include <iostream>

struct S
{
    S() {std::cout << "S::S()" << std::endl;}
    S(const S *s) {std::cout << "S::S(const S *)" << std::endl;}
    S(const S &s) {std::cout << "S::S(const S &)" << std::endl;}
    ~S() {std::cout << "S::~S()" << std::endl;}
    S const &operator=(const S &s)
    {
        std::cout << "S const &S::operator=(const S &)" << std::endl;
        return *this;
    }
};

class foo
{
    S s;
    public:
        foo() {std::cout << "foo::foo()" << std::endl;}

        #ifndef MISSING_COPY_CONSTRUCTOR
        foo(const foo &f) {std::cout << "foo::foo(const foo &)" << std::endl;}
        #endif

        foo(const foo *f) {std::cout << "foo::foo(const foo *)" << std::endl;}

        #ifndef MISSING_ASSIGNMENT_OPERATOR
        foo const &operator=(foo const &f)
        {std::cout << "const &foo::operator=(foo const &)" << std::endl;}
        #endif

        ~foo() {std::cout << "foo::~foo()" << std::endl;}

        static foo bar(foo A)
        {
            std::cout << "static foo bar(foo)" << std::endl;
            return A;
        }

        static foo &bar(foo *A)
        {
```

```
            std::cout << "const static foo bar(foo *)" << std::endl;
            return *A;
        }
};

int main(int argc, char const *argv[])
{
    foo A;
    foo B(A);
    foo C = A;
    foo *ptrA, *ptrB;
    foo V[2];
    foo E(&B);
    ptrA = &A;
    ptrB = new foo(B);
    foo &D = A;
    delete ptrB;

    V[0] = foo::bar(B);

    V[1] = foo::bar(ptrA);

    foo *F(&A);
    foo &G(E);
    foo *W = new foo[3];
    foo H = *W;
    delete []W;

    return 0;
}
```

En primer lugar es importante aclarar que en `C++` `struct` es idéntico a `class` con la única diferencia siendo que en el primer caso los miembros son públicos por defecto, mientras que en el segundo son privados. Teniendo esto en cuenta, como `foo` tiene un atributo de tipo `S`, con cada nuevo objeto de tipo `foo` se debe crear un dato de tipo `S`, lo que implica llamar a su constructor. Análogamente, cuando se destruye un objeto de tipo *foo*, a continuación se destruye `s`, por lo que se llama al destructor de `S`.

Con estas consideraciones, las invocaciones de los constructores y destructores (que imprimen sus respectivos mensajes en pantalla según sus definiciones), línea por línea, son las siguientes:

- Se crea el objeto `A`, que invoca en primer lugar al constructor `S()` y luego al constructor `foo()`, constructores sin argumentos definidos.
- Se crea el objeto `B` por copia de `A`. Si las macros están definidas, el constructor por copia de `foo` que se invoca es el que existe por defecto por ser `foo` una clase. Éste copia todos los atributos de `A` a `B`, lo que implica copiar un dato de tipo `S`. Para esto se invoca al constructor por copia de `S`, que está definido y que imprime un mensaje en pantalla. En caso de no estar definidas las macros, el constructor copia de `foo` que se usa está definido explícitamente, y por lo tanto se invoca éste, que imprime un mensaje en pantalla. Sin embargo, este constructor no especifica qué hacer con el atributo `s` (no aclara que tiene que hacer una copia de él) y entonces el constructor de `S` que se invoca es el constructor sin argumentos. Todo esto sucede cada vez que se crea un objeto por copia.
- Se crea un objeto `C` por copia de `A`, a pesar de que se use el `operator=`.
- Se definen punteros `ptrA` y `ptrB`. A pesar de que sean punteros a objetos, siguen siendo punteros, por lo que no se invoca a ningún constructor.
- Se define el array estático `V` de tamaño 2, lo que implica crear 2 objetos, cada uno como si se lo creara con el constructor sin argumentos.

- Se crea un objeto `E` por copia de puntero con la dirección de memoria de `B` mediante el constructor definido. Nuevamente, el constructor que se invoca no especifica nada sobre el atributo `s`, por lo que el constructor de `S` que se invoca es el constructor sin argumentos.
- Se le asigna un valor a un puntero, no se trabaja sobre ningún objeto.
- El operador `new` le asigna el espacio necesario al puntero `ptrB` como para que quepa un objeto de tipo `foo` y crea un objeto por copia de `B` (con lo que esto implica, como en la segunda línea) que pasa a ser apuntado por `ptrB`.
- Se define la referencia `D` que, al igual que con los punteros, no implica crear un objeto.
- El operador `delete` destruye el objeto apuntado por `ptrB`, y luego se libera la memoria. Para esto se invoca primero al destructor de `foo` y luego al de `S`, en orden inverso al de construcción.
- La función `bar(foo A)` recibe un objeto de tipo `foo` por copia, lo que implica crear una copia del mismo modo que en la definición de `B`. Luego de imprimir el mensaje, la función devuelve un objeto por copia, nuevamente invocando al constructor por copia correspondiente. Luego se invoca al `operator=`. El caso es análogo al del constructor por copia de `foo`: si las macros están definidas, el operador que se invoca es el que existe por defecto en cualquier clase, y lo que hace es asignar cada atributo del objeto copiado al objeto nuevo. Esto implica que se invoque al `operator=` definido en `S`. En cambio, si las macros no están definidas, se invoca al `operator=` de `foo` definido, que imprime un mensaje pero no especifica qué hacer con `s`, que ya existe, por lo que no invoca a ningún método de `S`. Cuando termina la sentencia, se destruyen los objetos que existen en el scope de la función `bar`.
- La función `bar(foo &A)` recibe un objeto por puntero, imprime su mensaje, y devuelve un objeto por referencia, por lo que no se crea ningún objeto. Finalmente se invoca al operador `=` del mismo modo que en el caso anterior.
- Se crea un puntero `F` que apunta a `A`. Para esto no se invoca a ningún constructor.
- Se crea una referencia `G` de `E`. No se invoca a ningún constructor.
- Se crea un array dinámico `W` de tamaño 3 y de tipo `foo`, lo que implica crear 3 objetos de ese tipo con constructor sin argumentos.
- Se crea un objeto `J` por copia del primer elemento del array `W`.
- Se destruye el array dinámico, invocando 3 veces a los destructores de `foo` y `S` definidos.
- Finalmente se destruyen todos los objetos que todavía existen al terminar el programa: `A`, `B`, `C`, `E`, `*F`, `H`, y los dos objetos de `V`.

Salida para macros definidas:

```
S::S()
foo::foo()
S::S(const S &)
S::S(const S &)
S::S()
foo::foo()
S::S()
foo::foo()
S::S()
foo::foo(const foo *)
S::S(const S &)
foo::~foo()
S::~S()
S::S(const S &)
static foo bar(foo)
S::S(const S &)
S const &S::operator=(const S &)
foo::~foo()
S::~S()
foo::~foo()
S::~S()
const static foo bar(foo *)
S const &S::operator=(const S &)
S::S()
foo::foo()
S::S()
foo::foo()
S::S()
foo::foo()
S::S(const S &)
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
```

Salida para macros no definidas:

```
S::S()
foo::foo()
S::S()
foo::foo(const foo &)
S::S()
foo::foo(const foo &)
S::S()
foo::foo()
S::S()
foo::foo()
S::S()
foo::foo(const foo *)
S::S()
foo::foo(const foo &)
foo::~foo()
S::~S()
S::S()
foo::foo(const foo &)
static foo bar(foo)
S::S()
foo::foo(const foo &)
const &foo::operator=(foo const &)
foo::~foo()
S::~S()
foo::~foo()
S::~S()
const static foo bar(foo *)
const &foo::operator=(foo const &)
S::S()
foo::foo()
S::S()
foo::foo()
S::S()
foo::foo()
S::S()
foo::foo(const foo &)
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
foo::~foo()
S::~S()
```