Clases – Técnicas avanzadas

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20

Clases

Filosofía de trabajo con **clases**:

- La implementación de la clase (tipo de datos) debe facilitar todo lo posibe el uso de esa clase en el desarrollo de software.
- Eso implica automatizar y acomodar todo lo posible el uso de esa clase al resto del lenguaje.

```
1 Thing a,b;
2
3 a.init("Hola");
4 b = a.clone();
```

vs

```
1 Thing a("Hola"),b;
2
3 b = a;
```

Buffer: clase con datos dinámicos

buffer.h:

```
1  class Buffer
2  {
3  private:
4    int   sz;
5    int*  data;
6  public:
7    Buffer(int _sz = 0);
8  };
```

buffer.cc:

```
1  Buffer::Buffer(int _sz)
2          : sz(_sz)
3  {
4    if (sz>0)
5      data = new int[sz];
6    else
7      data = nullptr;
8  }
```

Vamos a ver una serie de situaciones problemáticas para el *usuario* (el programador que utiliza la clase `Buffer`), y cómo podemos resolverlas.

La solución siempre será añadir o modificar la funcionalidad de la clase `Buffer`, de forma transparente para el usuario.

Destructor

Destructor

Tarea: Definición de objetos como variable local.

```
1  void f()
2  {
3    Buffer a(1024);
4
5    ...
6  }
```

Pregunta:
¿ Problema ?

Destructor

Tarea: Definición de objetos como variable local.

```
1  void f()
2  {
3    Buffer a(1024);
4
5    ...
6  }
```

Pregunta:
¿ Problema ?

Memory leak:
La memoria reservada por el objeto no se libera.

Destructor

Tarea: Definición de objetos como variable local.

```
1  void f()
2  {
3    Buffer a(1024);
4
5    ...
6  }
```

Respuesta:
Solución:

Destructor

```
~Buffer()
```

Destructor

Tarea: Definición de objetos como variable local.

```
1 Buffer::~Buffer()
2 {
3   delete[] data;
4 }
```

Destructor

Tarea: Definición de objetos como variable local.

```
1  void f()
2  {
3    Buffer a;
4
5    ...
6  }
```

Pregunta:
¿ Problema ?

Destructor

Tarea: Definición de objetos como variable local.

```
1  void f()
2  {
3    Buffer a;
4
5    ...
6  }
```

Pregunta:
¿ Problema ?

```
delete[] data
```

Tarea: Definición de objetos como variable local.

```
1  void f()
2  {
3    Buffer a;
4
5    ...
6  }
```

Pregunta:
¿ Problema ?

`delete[] data`

NO:

```
::(optional) delete expression
::(optional) delete[] expression
```

If expression evaluates to a null pointer value, no destructors are called, and the deallocation function may or may not be called (it's unspecified), but the default deallocation functions are guaranteed to do nothing when passed a null pointer.

Inicialización

Inicialización

Tarea: El programador desea crear un duplicado de un objeto que ya existe.

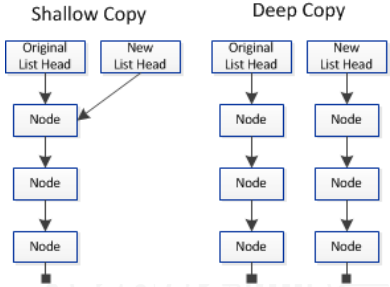
```
1 Buffer a;
2
3 // Posibilidades
4 Buffer b{a};
5 Buffer b(a);
6 Buffer b = a;
```

Pregunta:
¿ Problema ?

Tarea: El programador desea crear un duplicado de un objeto que ya existe.

```
1  Buffer a;
2
3  // Posibilidades
4  Buffer b{a};
5  Buffer b(a);
6  Buffer b = a;
```

Pregunta:
¿ Problema ?



Tarea: El programador desea crear un duplicado de un objeto que ya existe.

```
1 Buffer a;
2
3 // Posibilidades
4 Buffer b{a};
5 Buffer b(a);
6 Buffer b = a;
```

Respuesta:
Solución:

Constructor de copia
(*copy constructor*)

```
Buffer(const Buffer&)
```

Semántica estricta de *deep copy*.

Inicialización

Tarea: El programador desea construir un duplicado de un objeto que ya existe.

```
1  Buffer::Buffer(const Buffer& that)
2            : sz(that.sz)
3  {
4    data = new int[sz];
5    std::copy(that.data,that.data+sz,data);
6  }
```

Inicialización

Tarea: El programador desea inicializar un objeto con una lista de valores.

```
1 Buffer a {1,2,3};
```

Pregunta:
¿ Problema ?

Inicialización

Tarea: El programador desea inicializar un objeto con una lista de valores.

```
1 Buffer a {1,2,3};
```

Pregunta:
¿ Problema ?

Esa sintaxis sólo funciona con los tipos predefindos.

Tarea: El programador desea inicializar un objeto con una lista de valores.

```
1 Buffer a {1,2,3};
```

Respuesta:
Solución:

Constructor de lista de inicialización
(*initializer list constructor*)

```
Buffer(initializer_list<...>)
```

Inicialización

Tarea: El programador desea inicializar un objeto con una lista de valores.

```
1  #include <initializer_list>
2
3  Buffer::Buffer(const initializer_list<int>& il)
4            : sz(il.size())
5  {
6    data = new int[sz];
7    std::copy(std::begin(il),std::end(il),data);
8  }
```

Asignación

Asignación

Tarea: El programador desea copiar un objeto que ya existe.

```
1 Buffer a,b,c;
2
3 c = b = a;
```

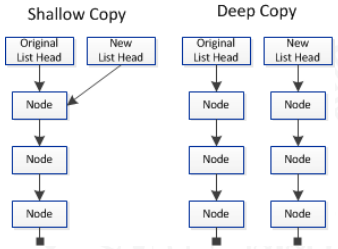
Pregunta:
¿ Problema ?

Asignación

Tarea: El programador desea copiar un objeto que ya existe.

```
1  Buffer a,b,c;
2
3  c = b = a;
```

Pregunta:
¿ Problema ?



Asignación

Tarea: El programador desea copiar un objeto que ya existe.

```
1 Buffer a,b,c;
2
3 c = b = a;
```

Respuesta:
Solución:

Operador de asignación
(*copy assignment operator*)

```
operator=(const Buffer&)
```

Semántica estricta de *deep copy*.

Asignación

Tarea: El programador desea copiar un objeto que ya existe.

```
1  const Buffer& Buffer::operator=(const Buffer& that)
2  {
3    delete[] data;
4    sz   = that.sz;
5    data = new int[sz];
6    std::copy(that.data,that.data+sz,data);
7
8    return *this;
9  }
```

Asignación

Tarea: El programador desea asignar a un objeto una lista de valores.

```
1 Buffer a,b;
2
3 b = a = {1,2,3};
```

Pregunta:
¿ Problema ?

Asignación

Tarea: El programador desea asignar a un objeto una lista de valores.

```
1 Buffer a,b;
2
3 b = a = {1,2,3};
```

Pregunta:
¿ Problema ?

Esa sintaxis sólo funciona con los tipos predefindos.

Asignación

Tarea: El programador desea asignar a un objeto una lista de valores.

```
1 Buffer a,b;
2
3 b = a = {1,2,3};
```

Respuesta:
Solución:

Asignación de lista de inicialización
(*initializer list assignment*)

`operator=(initializer_list<...>)`

Asignación

Tarea: El programador desea asignar a un objeto una lista de valores.

```
1  const Buffer& Buffer::operator=(const initializer_list<int>& il)
2  {
3    delete[] data;
4    sz   = il.size();
5    data = new int[sz];
6    std::copy(std::begin(il),std::end(il),data);
7
8    return *this;
9  }
```

Funciones

Funciones

En el uso de funciones se utilizan los mismos operadores.

```
1  void work(Buffer par)
2  {
3    ...
4  }
5
6  Buffer arg;
7  work(arg);
```

Operadores implicados:

- construcción por copia del parámetro

  `Buffer(const Buffer&)`

- destrucción del parámetro

  `~Buffer()`

Funciones

En el uso de funciones se utilizan los mismos operadores.

```
1  void work(Buffer& par)
2  {
3    ...
4  }
5
6  Buffer arg;
7  work(arg);
```

Operadores implicados:

- ninguno

En el uso de funciones se utilizan los mismos operadores.

```
1  Buffer build()
2  {
3    Buffer res;
4    ...
5    // fill buffer w things
6    ...
7    return res;
8  }
9
10 Buffer b;
11 b = build();
```

Operadores implicados (versión estricta):

- construcción de `res`
  `Buffer()`
- construcción de copia de `tmp(res)` en la pila
  `Buffer(const Buffer&)`
- destruccion de `res`
  `~Buffer()`
- asignacion de copia `tmp -> b`
  `operator=(const Buffer&)`
- destrucción de `tmp`
  `~Buffer()`

Funciones

En el uso de funciones se utilizan los mismos operadores.

```
1  Buffer build()
2  {
3    Buffer res;
4    ...
5    // fill buffer w things
6    ...
7    return res;
8  }
9
10 Buffer b;
11 b = build();
```

Operadores implicados (versión optimizada):

- construcción de `res`.
  `Buffer()`
- asignacion de copia `res -> b`
  `operator=(const Buffer&)`
- destrucción de `res`
  `~Buffer()`

En el uso de funciones se utilizan los mismos operadores.

```
1  Buffer build()
2  {
3    Buffer res;
4    ...
5    // fill buffer w things
6    ...
7    return res;
8  }
9
10 Buffer b;
11 b = build();
```

Ineficiencia:

- asignacion de copia `res -> b`
- destrucción de `res`

Se duplica un objeto para elimiar el original inmediatamente.

En el uso de funciones se utilizan los mismos operadores.

```
1  Buffer build()
2  {
3    Buffer res;
4    ...
5    // fill buffer w things
6    ...
7    return res;
8  }
9
10 Buffer b;
11 b = build();
```

Posible optimización:

- **transferencia** de res -> b
- destrucción de res (sin borrado)

En el uso de funciones se utilizan los mismos operadores.

```
1  Buffer build()
2  {
3    Buffer res;
4    ...
5    // fill buffer w things
6    ...
7    return res;
8  }
9
10 Buffer b;
11 b = build();
```

Respuesta:
Solución:

Constructor de transferencia
(*move constructor*)
`Buffer(Buffer&&)`

Asignación de transferencia
(*move assignment*)
`operator=(Buffer&&)`

Move constructor

```
1  Buffer::Buffer(Buffer&& that)
2  {
3    sz   = that.sz;
4    data = that.data;
5    that.sz   = 0;
6    that.data = nullptr;
7  }
```

Move assignment

```
1  const Buffer& Buffer::operator=(Buffer&& that)
2  {
3    delete[] data;
4
5    sz   = that.sz;
6    data = that.data;
7    that.sz   = 0;
8    that.data = nullptr;
9
10   return *this;
11 }
```

Move operators

El compilador los usa a discreción si están disponibles, por ejemplo en sentencias `return`.
Se puede forzar la semántica de las operaciones:

```
1    Buffer a,b;
2
3    b = std::move(a);        // move assign
4
5    Buffer c(std::move(a));  // move ctor
```

Clases – Técnicas avanzadas

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20