Presentation of LC-4

Assistants 2009

May 6, 2014

Presentation of LC-4

1. Overview of the tarball
2. Implementation details

Overview of the tarball

1. Overview of the tarball
2. Implementation details

The tree structure of TC-4

- Only 'src/type' directory has been added.
- That is where the TypeChecker lays.

The tree structure of TC-4

- Only 'src/type' directory has been added.
- That is where the TypeChecker lays.

Code to write

- Types classes.
- `type::TypeChecker`
- `ast::Typeable` and `ast::TypeConstructor` (Update the AST accordingly).

Implementation details

1. Overview of the tarball

2. Implementation details
    - Template Method
    - Design Pattern Proxy
    - Stack unwinding
    - Vtables
    - Singleton
    - Architecture of TC-4

Template Method

1 Overview of the tarball

2 Implementation details
- Template Method
- Design Pattern Proxy
- Stack unwinding
- Vtables
- Singleton
- Architecture of TC-4

Template Method: the return

- Do you recall the template method vs method template mess?
- ... well you really should :)
- ... because it is useful for TC-4.
- Otherwise, check TC-3 presentation.

Template Method: the return

- Do you recall the template method vs method template mess?
- . . . well you really should :)
- . . . because it is useful for TC-4.
- Otherwise, check TC-3 presentation.

Template Method: the return

- Do you recall the template method vs method template mess?
- ... well you really should :)
- ... because it is useful for TC-4.
- Otherwise, check TC-3 presentation.

Template Method: the return

- Do you recall the template method vs method template mess?
- . . . well you really should :)
- . . . because it is useful for TC-4.
- Otherwise, check TC-3 presentation.

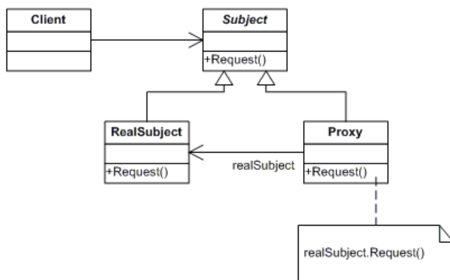
Design Pattern Proxy

1. Overview of the tarball
2. Implementation details
   - Template Method
   - Design Pattern Proxy
   - Stack unwinding
   - Vtables
   - Singleton
   - Architecture of TC-4

The design pattern *Proxy*

The design pattern *Proxy*

Provide a surrogate or placeholder for another object to control access to it.



Usage

- Access control.
- Hiding informations about the real object (Example: image).

Stack unwinding

1. Overview of the tarball
2. Implementation details
   - Template Method
   - Design Pattern Proxy
   - Stack unwinding
   - Vtables
   - Singleton
   - Architecture of TC-4

Exception handling and stack unwinding

- What does happen when throwing an exception?
  - Construction of the thrown object.
  - “Stack unwinding”.
  - Execution of the first catch code found.

Exception handling and stack unwinding

- What does happen when throwing an exception?
  - Construction of the thrown object.
  - “Stack unwinding”.
  - Execution of the first catch code found.

Exception handling and stack unwinding

- What does happen when throwing an exception?
  - Construction of the thrown object.
  - “Stack unwinding”.
  - Execution of the first catch code found.

Exception handling and stack unwinding

- What does happen when throwing an exception?
  - Construction of the thrown object.
  - “Stack unwinding”.
  - Execution of the first catch code found.

Stack unwinding

Until a catch block is found, exit the current function, which means that:

- All automatic variables are deallocated.
- If an automatic variable is an object, its destructor is automatically called.
- Stack is popped.
- warning: If a pointer on an allocated memory block is present in the function, this memory block will *leak*.

Stack unwinding

Until a catch block is found, exit the current function, which means that:

- All automatic variables are deallocated.
- If an automatic variable is an object, its destructor is automatically called.
- Stack is popped.
- warning: If a pointer on an allocated memory block is present in the function, this memory block will *leak*.

Stack unwinding

Until a catch block is found, exit the current function, which means that:

- All automatic variables are deallocated.
- If an automatic variable is an object, its destructor is automatically called.
- Stack is popped.
- warning: If a pointer on an allocated memory block is present in the function, this memory block will *leak*.

Stack unwinding

Until a catch block is found, exit the current function, which means that:

- All automatic variables are deallocated.
- If an automatic variable is an object, its destructor is automatically called.
- Stack is popped.
- warning: If a pointer on an allocated memory block is present in the function, this memory block will *leak*.

Stack unwinding: Example

```
void throwing_function () {
  // huge_get delegates the deallocating
  // of the object to the caller.
  HugeObject* huge = huge_get ();
  throw("Your program leaks a lot :(.");
  delete huge;
}

void function_catcher () {
  try
  { throwing_function(); }
  catch (std::string str)
  { std::cerr << str << std::endl; }
}

int main() {
  function_catcher();
}
```

A solution to the memory leak: auto_ptr

- auto_ptr implements the design pattern Proxy.
- Included in STL.
- Behave like a pointer but automatically free object at the end of the scope.
- Alert: Copies of auto_ptr are not equivalent. Only one of them has the ownership of the object.

A solution to the memory leak: auto_ptr

- auto_ptr implements the design pattern Proxy.
- Included in STL.
- Behave like a pointer but automatically free object at the end of the scope.
- Alert: Copies of auto_ptr are not equivalent. Only one of them has the ownership of the object.

A solution to the memory leak: `auto_ptr`

- `auto_ptr` implements the design pattern Proxy.
- Included in STL.
- Behave like a pointer but automatically free object at the end of the scope.
- Alert: Copies of auto_ptr are not equivalent. Only one of them has the ownership of the object.

A solution to the memory leak: `auto_ptr`

- `auto_ptr` implements the design pattern Proxy.
- Included in STL.
- Behave like a pointer but automatically free object at the end of the scope.
- Alert: Copies of auto_ptr are not equivalent. Only one of them has the ownership of the object.

Stack unwinding: a leak-free example

```
void throwing_function() {
  // huge_get delegates the deallocating
  // of the object to the caller.
  std::auto_ptr<HugeObject> huge = huge_get ();
  throw("Your program does not leak :).");
  // huge is automatically freed.
}

void function_catcher() {
  try
  { throwing_function(); }
  catch (std::string str)
  { std::cerr << str << std::endl; }
}

int main() {
  function_catcher();
}
```

1 Overview of the tarball

2 Implementation details
- Template Method
- Design Pattern Proxy
- Stack unwinding
- Vtables
- Singleton
- Architecture of TC-4

Why vtables?

```
struct A {
  virtual void test() {std::cout << "Finished :)" << std::endl};
};

struct B : public A {
  virtual void test() {
    while (1) std::cout << "You loose :(" << std::endl
  };
};

int main() {
  A* a = (rand() % 2) ? new A() : new B();
  a->test();
}
```

- How to resolve the good call to test?
- Vtables are a solution.

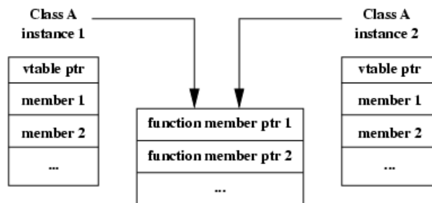
Why vtables?

```
struct A {
  virtual void test() {std::cout << "Finished :)" << std::endl};
};

struct B : public A {
  virtual void test() {
    while (1) std::cout << "You loose :(" << std::endl
  };
};

int main() {
  A* a = (rand() % 2) ? new A() : new B();
  a->test();
}
```

- How to resolve the good call to test?
- Vtables are a solution.

vtables

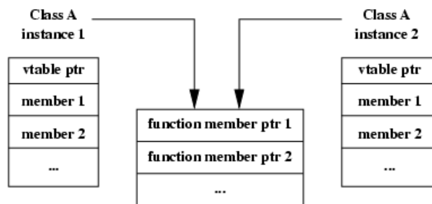
- Vtables are used to resolve virtual function member calls.
- A vtable is an array of function member pointers. Each function member has an unique entry in the vtable.



- Each class that owns a virtual member function has a vtable.
- Each instance of a class has a pointer on that vtable.

vtables

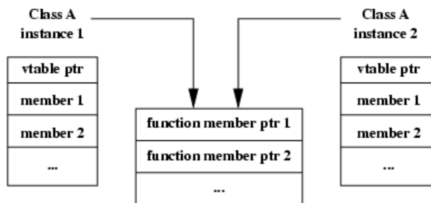
- Vtables are used to resolve virtual function member calls.
- A vtable is an array of function member pointers. Each function member has an unique entry in the vtable.



- Each class that owns a virtual member function has a vtable.
- Each instance of a class has a pointer on that vtable.

vtables

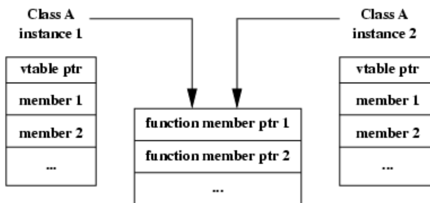
- Vtables are used to resolve virtual function member calls.
- A vtable is an array of function member pointers. Each function member has an unique entry in the vtable.



- Each class that owns a virtual member function has a vtable.
- Each instance of a class has a pointer on that vtable.

vtables

- Vtables are used to resolve virtual function member calls.
- A vtable is an array of function member pointers. Each function member has an unique entry in the vtable.



- Each class that owns a virtual member function has a vtable.
- Each instance of a class has a pointer on that vtable.

A small reminder on templates

- Code is not generated upon definition but only when the template is used.

```
// No code is generated.
template <class T>
T max (T a, T b);

// max<int> is generated.
max (1, 2);

// max<float> is generated.
max (1f, 2f);
```

Virtual template method

- 'problem.hh'

```
struct A {
  template<class T>
  virtual void test (T& t) {std::cout << "A" << std::endl};
};
```

- 'lib.cc'

```
#include "problem.hh"
int fun () {
  A* a = new A ();
  a->test(3f);
}
```

- 'main.cc'

```
#include "problem.hh"
int main () {
  A* a = new A ();
  a->test(3);
}
```

- This is not valid in C++.

Virtual template method

- 'problem.hh'

```
struct A {
 template<class T>
 virtual void test (T& t) {std::cout << "A" << std::endl};
};
```

- 'lib.cc'

```
#include "problem.hh"
int fun () {
 A* a = new A ();
 a->test(3f);
}
```

- 'main.cc'

```
#include "problem.hh"
int main () {
 A* a = new A ();
 a->test(3);
}
```

- This is not valid in C++.

Virtual template method

- 'problem.hh'

```
struct A {
  template<class T>
  virtual void test (T& t) {std::cout << "A" << std::endl};
};
```

- 'lib.cc'

```
#include "problem.hh"
int fun () {
  A* a = new A ();
  a->test(3f);
}
```

- 'main.cc'

```
#include "problem.hh"
int main () {
  A* a = new A ();
  a->test(3);
}
```

- This is not valid in C++.

Virtual template method

- 'problem.hh'

```
struct A {
  template<class T>
  virtual void test (T& t) {std::cout << "A" << std::endl};
};
```

- 'lib.cc'

```
#include "problem.hh"
int fun () {
  A* a = new A ();
  a->test(3f);
}
```

- 'main.cc'

```
#include "problem.hh"
int main () {
  A* a = new A ();
  a->test(3);
}
```

- This is not valid in C++.

Why virtual template methods are invalid?

- When compiling main.cc, g++ does not know all the uses of the templates (here: int, float), hence it does not know how many virtual function members exist in a class.
- Vtable sizes cannot be known.
- Since it cannot work, it is not valid.

Why virtual template methods are invalid?

- When compiling main.cc, g++ does not know all the uses of the templates (here: int, float), hence it does not know how many virtual function members exist in a class.
- Vtable sizes cannot be known.
- Since it cannot work, it is not valid.

Why virtual template methods are invalid?

- When compiling main.cc, g++ does not know all the uses of the templates (here: int, float), hence it does not know how many virtual function members exist in a class.
- Vtable sizes cannot be known.
- Since it cannot work, it is not valid.

Singleton

1 Overview of the tarball

2 Implementation details
- Template Method
- Design Pattern Proxy
- Stack unwinding
- Vtables
- Singleton
- Architecture of TC-4

The design pattern: singleton

Singleton
-instance : Singleton
-Singleton() +Instance() : Singleton

- Ensure a class has only one instance and provide a global point of access to it.
- Save space.
- In Tiger, it is used for built-in types (nil, void, int and string).

Example:

```
class Nil : public Type
{
public:
  /// Return the unique instance of Nil.
  static const Nil& instance ();
private:
  Nil ();
};
```

The design pattern: singleton

Singleton
-instance : Singleton
-Singleton() +Instance() : Singleton

- Ensure a class has only one instance and provide a global point of access to it.
- Save space.
- In Tiger, it is used for built-in types (nil, void, int and string).

Example:

```
class Nil : public Type
{
public:
  /// Return the unique instance of Nil.
  static const Nil& instance ();
private:
  Nil ();
};
```

The design pattern: singleton

Singleton
-instance : Singleton
-Singleton() +Instance() : Singleton

- Ensure a class has only one instance and provide a global point of access to it.
- Save space.
- In Tiger, it is used for built-in types (nil, void, int and string).

Example:

```
class Nil : public Type
{
public:
  /// Return the unique instance of Nil.
  static const Nil& instance ();
private:
  Nil ();
};
```

The design pattern: singleton

Singleton
-instance : Singleton
-Singleton() +Instance() : Singleton

- Ensure a class has only one instance and provide a global point of access to it.
- Save space.
- In Tiger, it is used for built-in types (nil, void, int and string).

Example:

```
class Nil : public Type
{
public:
  /// Return the unique instance of Nil.
  static const Nil& instance ();
private:
  Nil ();
};
```

Architecture of TC-4

1 Overview of the tarball

2 Implementation details
- Template Method
- Design Pattern Proxy
- Stack unwinding
- Vtables
- Singleton
- Architecture of TC-4

Typeable

- Interface for all nodes related to a type.
- It is used in two cases:
  - Nodes that have a type.
  - Nodes that define a type.

Typeable

- Interface for all nodes related to a type.
- It is used in two cases:
  - Nodes that have a type.
  - Nodes that define a type

Typeable

- Interface for all nodes related to a type.
- It is used in two cases:
  - Nodes that have a type.
  - Nodes that define a type

Typeable

- Interface for all nodes related to a type.
- It is used in two cases:
  - Nodes that have a type.
  - Nodes that define a type

TypeConstructor

- In the AST some nodes construct types.
- All other nodes just reference them, hence *must* not free them.
- TypeConstructor is a class that factors the deallocation of type objects.
- Only nodes that create types derive from TypeConstructor.

TypeConstructor

- In the AST some nodes construct types.
- All other nodes just reference them, hence *must* not free them.
- TypeConstructor is a class that factors the deallocation of type objects.
- Only nodes that create types derive from TypeConstructor.

TypeConstructor

- In the AST some nodes construct types.
- All other nodes just reference them, hence *must* not free them.
- TypeConstructor is a class that factors the deallocation of type objects.
- Only nodes that create types derive from TypeConstructor.

TypeConstructor

- In the AST some nodes construct types.
- All other nodes just reference them, hence *must* not free them.
- TypeConstructor is a class that factors the deallocation of type objects.
- Only nodes that create types derive from TypeConstructor.

The Type hierarchy

- Grammar of type:

```
<Type > ::=
        Int | String | Void | Nil
        | Array <Type> | Record ( Id<Type> )+ | Name <Type>
        | Function ( Id<Type> )+ : <Type>
;
```

- Functions create a type, then it must be part of the hierarchy.
- `nil` and `void` are anonymous types that cannot be used directly.

The Type hierarchy

- Grammar of type:

```
<Type > ::=
        Int | String | Void | Nil
        | Array <Type> | Record ( Id<Type> )+ | Name <Type>
        | Function ( Id<Type> )+ : <Type>
;
```

- Functions create a type, then it must be part of the hierarchy.
- nil and void are anonymous types that cannot be used directly.

The Type hierarchy

- Grammar of type:

```
<Type > ::=
        Int | String | Void | Nil
        | Array <Type> | Record ( Id<Type> )+ | Name <Type>
        | Function ( Id<Type> )+ : <Type>
;
```

- Functions create a type, then it must be part of the hierarchy.
- `nil` and `void` are anonymous types that cannot be used directly.

Strange constructs with defined types

- Those types use only defined type, but are either invalid or unusable.
  - Recursive aliased types. Example:

    ```
    let
      type a = b
      type b = a
    in
    end
    ```

    is not valid.
  - Recursive arrays. Example:

    ```
    let
      type my_array = array of my_array
    in
    end
    ```

    cannot be used (but is valid).

Strange constructs with defined types

- Those types use only defined type, but are either invalid or unusable.
  - Recursive aliased types. Example:
    ```
    let
      type a = b
      type b = a
    in
    end
    ```
    is not valid.
  - Recursive arrays. Example:
    ```
    let
      type my_array = array of my_array
    in
    end
    ```
    cannot be used (but is valid).

Strange constructs with defined types

- Those types use only defined type, but are either invalid or unusable.
  - Recursive aliased types. Example:

    ```
    let
      type a = b
      type b = a
    in
    end
    ```

    is not valid.
  - Recursive arrays. Example:

    ```
    let
      type my_array = array of my_array
    in
    end
    ```

    cannot be used (but is valid).

The sound method

- Implement it in ‘`type/named.cc`’
- Check that a type is not a recursive aliased type.

The sound method

- Implement it in ‘`type/named.cc`’
- Check that a type is not a recursive aliased type.