Presentation of TC-5

Assistants 2009

May 6, 2014

Presentation of TC-5

Overview of the tarball

The tree structure of TC-5

- New directories:
  - 'src/frame': Definition of classes representing frames.
  - 'src/temp': Classes representing labels, temporaries, ...
  - 'src/tree': Intermediate representation (the second AST).
  - 'src/translate': Translation to intermediate code.

The tree structure of TC-5

- New directories:
  - ‘`src/frame`’: Definition of classes representing frames.
  - ‘`src/temp`’: Classes representing labels, temporaries, ...
  - ‘`src/tree`’: Intermediate representation (the second `AST`).
  - ‘`src/translate`’: Translation to intermediate code.

The tree structure of TC-5

- New directories:
  - ‘`src/frame`’: Definition of classes representing frames.
  - ‘`src/temp`’: Classes representing labels, temporaries, . . .
  - ‘`src/tree`’: Intermediate representation (the second `AST`).
  - ‘`src/translate`’: Translation to intermediate code.

The tree structure of TC-5

- New directories:
  - ‘`src/frame`’: Definition of classes representing frames.
  - ‘`src/temp`’: Classes representing labels, temporaries, ...
  - ‘`src/tree`’: Intermediate representation (the second `AST`).
  - ‘`src/translate`’: Translation to intermediate code.

The tree structure of TC-5

- New directories:
  - '`src/frame`': Definition of classes representing frames.
  - '`src/temp`': Classes representing labels, temporaries, ...
  - '`src/tree`': Intermediate representation (the second `AST`).
  - '`src/translate`': Translation to intermediate code.

Code to write

- ‘`src/temp/*`’: Complete identifier and factory classes.
- ‘`src/frame/*`’: Some code to do.
- ‘`src/translate/fragment.hh`’: Finish the class.
- `Translator`: The core of TC-5.

C++ notions

Memory management

Problematics

- How to handle an object deallocation?
  - In the class destructor: well suited for simple cases, but is a nightmare when several pointers reference the same object.
  - Reference counting.
  - Garbage collection.

Problematics

- How to handle an object deallocation?
  - In the class destructor: well suited for simple cases, but is a nightmare when several pointers reference the same object.
  - Reference counting.
  - Garbage collection.

Problematics

- How to handle an object deallocation?
  - In the class destructor: well suited for simple cases, but is a nightmare when several pointers reference the same object.
  - Reference counting.
  - Garbage collection.

Problematics

- How to handle an object deallocation?
  - In the class destructor: well suited for simple cases, but is a nightmare when several pointers reference the same object.
  - Reference counting.
  - Garbage collection.

Reference counting

- Each object knows how many pointers reference it.
- A pointer informs the object when it reference it, so that the counter can be incremented, and when it stops reference it, so that the counter can be decremented.
- When the counter reaches 0, the object can be deallocated.

Reference counting

- Each object knows how many pointers reference it.
- A pointer informs the object when it reference it, so that the counter can be incremented, and when it stops reference it, so that the counter can be decremented.
- When the counter reaches 0, the object can be deallocated.

Reference counting

- Each object knows how many pointers reference it.
- A pointer informs the object when it reference it, so that the counter can be incremented, and when it stops reference it, so that the counter can be decremented.
- When the counter reaches 0, the object can be deallocated.

Advantages and drawbacks

- Easy to implement.
- Low memory footprint.
- Do not use much CPU.
- But cannot handle all cases: circular references.

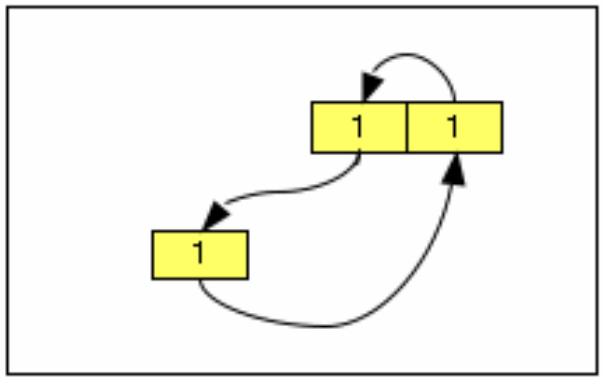

Figure: Numbers are the reference counter of each object

Advantages and drawbacks

- Easy to implement.
- Low memory footprint.
- Do not use much CPU.
- But cannot handle all cases: circular references.

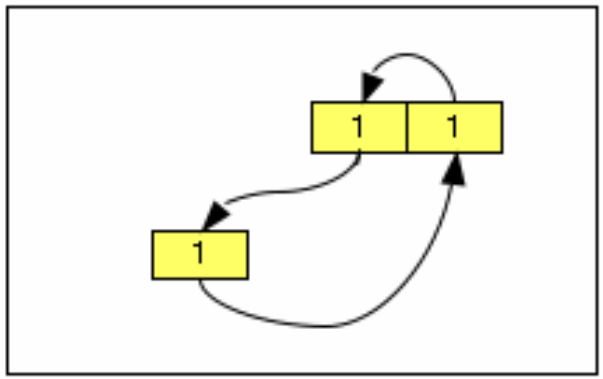

Figure: Numbers are the reference counter of each object

Advantages and drawbacks

- Easy to implement.
- Low memory footprint.
- Do not use much CPU.
- But cannot handle all cases: circular references.

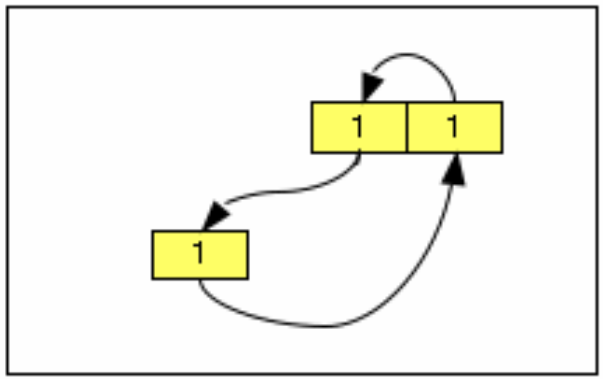

Figure: Numbers are the reference counter of each object

Advantages and drawbacks

- Easy to implement.
- Low memory footprint.
- Do not use much CPU.
- But cannot handle all cases: circular references.

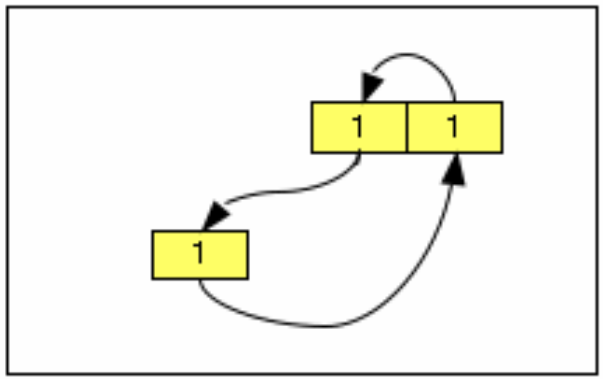

Figure: Numbers are the reference counter of each object

Common implementation in C++

- Reference counting is implemented using a Proxy design pattern.
- The proxy behaves like a pointer, by overloading `operator*` and `operator->`.

Common implementation in C++

- Reference counting is implemented using a Proxy design pattern.
- The proxy behaves like a pointer, by overloading `operator*` and `operator->`.

Garbage collecting

- Aims at determining which objects can be used by the program at any moment.
- Builds a graph of objects, where nodes represent objects and edges represent references.
- Accessible objects are those whose nodes can be reached from root nodes:
  1. Global variables
  2. Objects stored in the stack

Garbage collecting

- Aims at determining which objects can be used by the program at any moment.
- Builds a graph of objects, where nodes represent objects and edges represent references.
- Accessible objects are those whose nodes can be reached from root nodes:
  - Global variables.
  - Objects stored in the stack.

Garbage collecting

- Aims at determining which objects can be used by the program at any moment.
- Builds a graph of objects, where nodes represent objects and edges represent references.
- Accessible objects are those whose nodes can be reached from root nodes:
  - Global variables.
  - Objects stored in the stack.

Garbage collecting

- Aims at determining which objects can be used by the program at any moment.
- Builds a graph of objects, where nodes represent objects and edges represent references.
- Accessible objects are those whose nodes can be reached from root nodes:
  - Global variables.
  - Objects stored in the stack.

Garbage collecting

- Aims at determining which objects can be used by the program at any moment.
- Builds a graph of objects, where nodes represent objects and edges represent references.
- Accessible objects are those whose nodes can be reached from root nodes:
  - Global variables.
  - Objects stored in the stack.

Garbage collecting

Numerous languages integrate a garbage collecting:

- Java
- C#
- Caml
- Ruby
- Python
- Eiffel
- D
- Lisp
- ...

Advantages and drawbacks

- Perfect handling of deallocation.
- *Slow.*
- Complex implementation.

Advantages and drawbacks

- Perfect handling of deallocation.
- *Slow*.
- Complex implementation.

Advantages and drawbacks

- Perfect handling of deallocation.
- *Slow*.
- Complex implementation.

What about TC?

- Nodes of the Tree structure will be created then destroyed.
- We want to handle it with little effort thanks to `boost::shared_ptr` and `misc::ref`.
- Quite like the `Smptr` you did during the C++ Workshop.
- Pay special attention to implicit calls to the constructor.

What about TC?

- Nodes of the Tree structure will be created then destroyed.
- We want to handle it with little effort thanks to `boost::shared_ptr` and `misc::ref`.
- Quite like the `Smptr` you did during the C++ Workshop.
- Pay special attention to implicit calls to the constructor.

What about TC?

- Nodes of the Tree structure will be created then destroyed.
- We want to handle it with little effort thanks to `boost::shared_ptr` and `misc::ref`.
- Quite like the `Smptr` you did during the C++ Workshop.
- Pay special attention to implicit calls to the constructor.

What about TC?

- Nodes of the Tree structure will be created then destroyed.
- We want to handle it with little effort thanks to `boost::shared_ptr` and `misc::ref`.
- Quite like the `Smptr` you did during the C++ Workshop.
- Pay special attention to implicit calls to the constructor.

Variant types

Variants

- Remember `boost::variant`.
- And the famous boost::bad_get.
- We need a compile-time type verification.

Visiting Variant: Visitor

- Design pattern Visitor aims at executing an action on an object.
- Prevent dispatching of the action code in many classes definitions.
- Well suited for working on Variant.

```
#include "boost/variant.hpp"
#include <iostream>

typedef boost::variant<ast::IntExp, ast::StringExp> scalar_type;

struct my_visitor : public boost::static_visitor<void> {
  void operator() (const ast::IntExp&) const {
    std::cout << "IntExp";
  }
  void operator() (const ast::StringExp&) const {
    std::cout << "StringExp";
  }

};
int main() {
  scalar_type value = "a string";
  boost::apply_visitor (my_visitor (), value);
}
```

Visiting Variant: Visitor

- Design pattern Visitor aims at executing an action on an object.
- Prevent dispatching of the action code in many classes definitions.
- Well suited for working on Variant.

```
#include "boost/variant.hpp"
#include <iostream>

typedef boost::variant<ast::IntExp, ast::StringExp> scalar_type;

struct my_visitor : public boost::static_visitor<void> {
  void operator() (const ast::IntExp&) const {
    std::cout << "IntExp";
  }
  void operator() (const ast::StringExp&) const {
    std::cout << "StringExp";
  }

};
int main() {
  scalar_type value = "a string";
  boost::apply_visitor (my_visitor (), value);
}
```

Visiting Variant: Visitor

- Design pattern Visitor aims at executing an action on an object.
- Prevent dispatching of the action code in many classes definitions.
- Well suited for working on Variant.

```
#include "boost/variant.hpp"
#include <iostream>

typedef boost::variant<ast::IntExp, ast::StringExp> scalar_type;

struct my_visitor : public boost::static_visitor<void> {
  void operator() (const ast::IntExp&) const {
    std::cout << "IntExp";
  }
  void operator() (const ast::StringExp&) const {
    std::cout << "StringExp";
  }

};
int main() {
  scalar_type value = "a string";
  boost::apply_visitor (my_visitor (), value);
}
```

Tiger implementation

The second AST

- Intermediate representation is a different language than Tiger.
- Hence, has its own AST.

The second AST

- Intermediate representation is a different language than Tiger.
- Hence, has its own AST.

The second `AST`

- Different implementation than the first `AST`.
- Base class `Tree` has an `enum` indicating the kind of the object.
- Prevent numerous `dynamic_cast` in TC-6 and TC-7.

The second `AST`

- Different implementation than the first `AST`.
- Base class `Tree` has an `enum` indicating the kind of the object.
- Prevent numerous `dynamic_cast` in TC-6 and TC-7.

The second `AST`

- Different implementation than the first `AST`.
- Base class `Tree` has an `enum` indicating the kind of the object.
- Prevent numerous `dynamic_cast` in TC-6 and TC-7.

The second AST

```
/Tree/
    /Exp/
        Const (int value)
        Name  (const temp::Label &label)
        Temp  (const temp::Temp &temp)
        Binop (Oper oper, Exp &left, Exp &right)
        Mem   (Exp &exp)
        Call  (Exp &func, std::list<Exp *> &args)
        Eseq  (Stm &stm, Exp &exp)

    /Stm/
        Move  (Exp &dst, Exp &src)
        Sxp   (Exp &exp)
        Jump  (Exp &exp, std::list<temp::Label> &targets)
        CJump (Relop relop, Exp &left, Exp &right,
               Label &iftrue, Label &iffalse)
        Seq   (std::list<Stm *>)
        Label (temp::Label &label)
```