Jaap van der Leest
Timon van der Berg
October 22, 2017

C++ Course
Assignment 6

Exercise 42

Problem statement. *What are the variants of new/delete? For each of the variants provide a (short!) example in which the used new/delete is appropriate and provide a short explanation why it is appropriately used.*
Solution.

new/delete

`new` is used to allocate memory for a primitive type or object. When allocating for an object it will call the object constructor. Example of `new` use:

```
int *ptr = new int;
```

This allocation is appropriate since we want to allocate memory for a single int.

`delete` is used to deallocate memory that was allocated using `new`. If called on an object (not a primitive type) it will also call that objects' destructor. Example of `delete` use:

```
std::string *ptr = new std::string;
delete ptr;
```

This is appropriate because `delete` is used on memory allocated using `new`.

new[], delete[]

`new[]` is used to allocate memory for arrays. Like `new` it is type-safe: the type of the element has to be declared. Like `new`, it calls constructors. An example of using `new`:

```
int *aoi = new int[20];
```

`delete[]` is used to delete memory allocated using `new[]`. Unlike `new`, new[] saves the size of the array it allocates. `delete[]` uses this to delete the array. Destructors are called[1]. An example of `delete[]` usage:

```
string *strp = new string[550];
delete[] strp;
```

This is appropriate because `delete[]` is used on an array allocated using `new[]`.

operator new, operator delete

`operator new` is used to allocate raw bytes of memory. To actually use this memory, a static cast is required. An example of using `operator new`:

```
size_t *sp = static_cast<size_t *>(operator new(5 * sizeof(size_t)));
```

[1] If the array contains a primitive type no destructors are called. Therefore an array of pointers require manual destruction of whatever is pointed to.

This is appropriate because `operator new` is used to allocate raw memory. Here we first calculate the number of bytes needed for 5 `size_t` variables. We then allocate the memory. `operator new` does not care for types.

`operator delete` is used to deallocate memory that was allocated using `operator new`. Like `operator new`, `operator delete` does not care for types. Because `operator new` saves the number of bytes allocated, `operator delete` knows how much memory to deallocate. `operator delete` does not call any destructors. An example of using `operator delete`:

```
string *sp = static_cast<string *>(operator new(5 * sizeof(string)));
operator delete(sp);
```

This is appropriate because we are using operator delete to deallocate memory that was allocated using operator new.

placement new

placement `new` is found in `<memory>` and overloads `new`. Placement `new` is used to place objects in previously allocated memory. An example of using placement `new`:

```
string *sp = static_cast<string *>( operator new(15 * sizeof(string)));
new sp string("Donald Knuth");
```

This is appropriate because we are using operator new on memory of the correct type that was previously allocated. We have placed a single string in this memory, leaving room for 14 more.

Exercise 43

Problem statement. *Fix the memory leak in the 'Strings' class.*
Solution. Because our own implementation of 'Strings' was not perfect, we instead modified the official solution provided in the answers of set 5.

strings.h

```
#ifndef INCLUDED_STRINGS_
#define INCLUDED_STRINGS_

#include <iosfwd>

class Strings
{
    size_t d_size;
    std::string *d_str;

    public:
        struct POD
        {
            size_t      size;
            std::string *str;
        };

        Strings();
        ~Strings();
        Strings(int argc, char *argv[]);
        Strings(char *environLike[]);
        Strings(std::istream &in);
```

```
23
24          void swap(Strings &other);
25
26          size_t size() const;
27          std::string const *data() const;
28          POD release();
29
30          std::string const &at(size_t idx) const;    // for const-objects
31          std::string &at(size_t idx);                // for non-const objects
32
33          void add(std::string const &next);          // add another element
34
35      private:
36          void fill(char *ntbs[]);                    // fill prepared d_str
37
38          std::string &safeAt(size_t idx) const;      // private backdoor
39          std::string *enlarge();
40          void destroy();
41
42          static size_t count(char *environLike[]);   // # elements in env.like
43
44  };
45
46  inline size_t Strings::size() const         // potentially dangerous practice:
47  {                                           // inline accessors
48      return d_size;
49  }
50
51  inline std::string const *Strings::data() const
52  {
53      return d_str;
54  }
55
56  inline std::string const &Strings::at(size_t idx) const
57  {
58      return safeAt(idx);
59  }
60
61  inline std::string &Strings::at(size_t idx)
62  {
63      return safeAt(idx);
64  }
65
66
67  #endif
```

strings5.cc

```
1   #include "strings.ih"           // using namespace std;
2
3   Strings::~Strings()
```

```
{
    delete[] d_str;             // 0 pointer allowed
}
```

Exercise 44

Problem statement. *gi* **Solution.** go

Exercise 45

Problem statement. *gi* **Solution.** go

Exercise 46

Problem statement. *gi* **Solution.** go

Exercise 47

Problem statement. *Replace the switches in the 'CPU' class using function pointers.*
Solution. Because our own implementation of CPU was imperfect, we used the official solutions for Exercise 31. Our modified header is found below, followed byany new or modified helper functions. Everything not shown is assumed to be the unchanged.

cpu.h

```
#ifndef INCLUDED_CPU_
#define INCLUDED_CPU_

#include "../tokenizer/tokenizer.h"

class Memory;

class CPU
{
    enum
    {
        NREGISTERS = 5,                                          // a..e at indices 0..4, respecti
        LAST_REGISTER = NREGISTERS - 1
    };

    struct Operand
    {
        OperandType type;
        int value;
    };

    Memory &d_memory;
    Tokenizer d_tokenizer;

    int d_register[NREGISTERS];

```

```
27      public:
28          CPU(Memory &memory);
29          void start();
30
31      private:
32          bool error();                                       // show 'syntax error', and prepa
33                                                              // next input line
34                                                              // return a value or a register's
35                                                              // memory location's value
36          int dereference(Operand const &value);
37
38          bool rvalue(Operand &lhs);                          // retrieve an rvalue operand
39          bool lvalue(Operand &lhs);                          // retrieve an lvalue operand
40
41                                                              // determine 2 operands, lhs must
42          bool operands(Operand &lhs, Operand &rhs);
43
44          bool twoOperands(Operand &lhs, int &lhsValue, int &rhsValue);
45
46                                                              // store a value in register or m
47          void store(Operand const &lhs, int value);
48          void mov();                                         // assign a value
49          void add();                                         // add values
50          void sub();                                         // subtract values
51          void mul();                                         // multiply values
52          void div();                                         // divide values (remainder: last
53                                                              // div a b computes a /= b, last
54          void neg();                                         // negate a value
55          void dsp();                                         // display a value
56  };
57
58  #endif
```

dereference.cc

```
1   #include "cpu.ih"
2
3   int CPU::dereference(Operand const &value)
4   {
5       switch (value.type)
6       {
7           default:
8           // FALLING THROUGH (not used, but satisfies the compiler)
9           case OperandType::VALUE:
10          return value.value;
11
12          case OperandType::REGISTER:
13          return d_register[value.value];
14
15          case OperandType::MEMORY:
16          return d_memory.load(value.value);
17      }
18  }
```

```
19
20
21  // above is official solution
22  // below is rewritten using function pointers.
23  // todo:
24  //  move to files
25  //  add to header
26
27  int CPU::dereference(Operand const &value)
28  {
29      return readOperand[value.type](Operand const &value);
30  }
31
32  int (*CPU::readOperand[])(Operand const &value) // order as in enums.h
33  {
34      nullptr,                                    // padding for syntax, will never be called ; should it
35      &valueReturn,                               // could make it like store.cc
36      &registerReturn,
37      &memoryReturn
38  }
39
40
41  int CPU::valueReturn(Operand const &value)
42  {
43      return value.value;
44  }
45
46  int CPU::registerReturn(Operand const &value)
47  {
48      return d_register[value.value];
49  }
50
51  int CPU::memoryReturn(Operand const &value)
52  {
53      return d_memory.load(value.value);
54  }
```

start.cc

```
1   #include "cpu.ih"
2
3   void CPU::start()
4   {
5       while (true)
6       {
7           switch (d_tokenizer.opcode())
8           {
9               case Opcode::ERR:
10                  error();
11              break;
12
13              case Opcode::MOV:
14                  mov();
```

```
15              break;
16
17              case Opcode::ADD:
18                  add();
19              break;
20
21              case Opcode::SUB:
22                  sub();
23              break;
24
25              case Opcode::MUL:
26                  mul();
27              break;
28
29              case Opcode::DIV:
30                  div();
31              break;
32
33              case Opcode::NEG:
34                  neg();
35              break;
36
37              case Opcode::DSP:
38                  dsp();
39              break;
40
41              case Opcode::STOP:
42              return;
43          } // switch
44
45          d_tokenizer.reset();        // prepare for the next line
46
47      } // while
48  }
49
50
51  // code above is official solution
52  // d_tokenizer.opcode() is opcode from enums.h
53  //  enum class Opcode
54  //     {
55  //         ERR,
56  //         MOV,
57  //         ADD,
58  //         SUB,
59  //         MUL,
60  //         DIV,
61  //         NEG,
62  //         DSP,
63  //         STOP,
64  //     };
65  // below is rewrite using function pointers
66  void CPU::Start()
67  {
68      while (true)
```

```
69          execute[d_tokenizer.opcode()];
70          d_tokenizer.reset();
71  }
72
73  void (*CPU::execute[])() // order as in enums.h
74  {                        // seperate file, add to header
75      &error,
76      &mov,
77      &add,
78      &sub,
79      &mul,
80      &div,
81      &neg,
82      &dsp,
83      &stp
84  }
85
86  void CPU::stp()          // seperate file, add to header
87  {
88      break;
89  }
```

store.cc

```
1   #include "cpu.ih"
2
3   void CPU::store(Operand const &lhs, int value)
4   {
5       switch (lhs.type)
6       {
7           default: // not used, but satisfies the compiler
8           break;
9
10          case OperandType::REGISTER:
11              d_register[lhs.value] = value;
12          break;
13
14          case OperandType::MEMORY:
15              d_memory.store(lhs.value, value);
16          break;
17      }
18  }
19
20  // code above is original from official solutions
21  // lhs is struct 'Operand'
22  // lhs.type is 'Operandtype' from enums.h
23  // code below is rewrite using function pointers
24  // enum class OperandType
25  //     {
26  //         SYNTAX,              // syntax error while specifying an operand
27  //         VALUE,               // direct value
28  //         REGISTER,            // register index
29  //         MEMORY               // memory location (= index)
```

```
//    };
//
void CPU::store(Operand const &lhs, int value)                                  // shou
{
    storeValue[lhs.type](lhs.value, value);                                     // stor
}

void (*storeValue[])(int place, int value)
{
    nullptr,
    nullptr,
    &storeRegister,
    &storeMemory
}

void CPU::storeRegister(int place, int value)                                   // sep
{
    d_register[place] = value;
}

void CPU::storeMemory(int place, int value)                                     // sep
{
    d_memory.store(place, value);
}
```

Exercise 48

Problem statement. *gi* **Solution.** go