Chapter 4 - Built-in Type Constructors 2
Dynamically Allocating Data Members
Example: a Ztring Class
#ifndef ZSTRING_H
#define ZSTRING_H
#include <iostream>

class Ztring {
public:
    Ztring(const char* cstring = 0); // constructor
    Ztring(const Ztring&); // copy constructor
    ~Ztring(); // destructor
    Ztring& operator=(const Ztring&); // assignment operator
    const char* data() const; // why return const char*?
    unsigned int size() const;
    void print(std::ostream&) const;
    void concat(const Ztring&); // concatenates to *this

private:
    char* data_; // a C-style string
    char* init(const char* string); // return a copy of string
};

//...
// ...

// Auxiliary functions: overloaded operators
// E.g.
// Ztring x("abc");
// Ztring y("def");
// cout << x + y << endl;
Ztring
operator+(const Ztring& s1, const Ztring& s2);

std::ostream&
operator<<(std::ostream& os, const Ztring& s);

#endif
#include "zstring.h"
#include <string.h>  // for strlen

// constructor
Ztring::Ztring(const char* cstring) : data_(init(cstring)) {}  

// copy constructor
Ztring::Ztring(const Ztring& s) : data_(init(s.data())) {}  

// destructor
Ztring::~Ztring() {
    // very important: avoid memory leak (init uses new[]!)
    delete[] data_;  
}

// ...

zstring.cpp
Ztring: Assignment Operator

// ...

// assignment operator
Ztring&
Ztring::operator=(const Ztring& s) {
    if (this == &s) { // why test this?
        return *this;
        delete[] data_; // avoid memory leak
    data_ = init(s.data());
    return *this;
}

// ...

ztring.cpp
// ... 

// constant public member functions 
const char* Ztring::data() const {
    return data_; 
}

unsigned int Ztring::size() const {
    if (data_ == 0) // if no data_ was set (== NULL)
        return 0;
    return strlen(data_); // compute the # of “non-/0” chars
}

void Ztring::print(std::ostream& os) const {
    // can be made shorter, illustrates low-level implementation 
    for (unsigned int i = 0; (i < size()); ++i)
        os << data_[i] ;
}
Ztring: Concatenation

```
concat(s)
```

```
data_
*this
```

```
data_
s
```

```
old
```

```
0
```

```
0
```
void Ztring::concat(const Ztring& s) {
    // save old data of this ztring
    unsigned int old_size = size();
    char* old = data_;

    // allocate buffer large enough to hold both + trailing \0
    data_ = new char[old_size + s.size() + 1];
    unsigned int j = 0;

    for (unsigned int i = 0; (i < old_size); ++i)
        data_[j++] = old[i]; // copy original string

    for (unsigned int i = 0; (i < s.size()); ++i)
        data_[j++] = s.data_[i]; // after that the argument s

    data_[j] = '\0'; // don’t forget trailing ’\0’ character
    delete[] old; // avoid memory leak
}
char*
Ztring::init(const char* s) {  
    if (s == 0)  // remember: as opposed to references,  
        return 0;  // pointers can point to nil. Always check!
    else {  
        unsigned int len(strlen(s) + 1);  
        char* p(new char[len]);  
        for (unsigned int i = 0; (i < len); ++i)  
            p[i] = s[i];  // s is an address, copy its contents
        return p;
    }
}
// auxiliary functions (overloaded operators)

Ztring operator+(const Ztring& s1, const Ztring& s2) {
    Ztring s(s1);
    s.concat(s2);
    return s;
}

std::ostream&
operator<<(std::ostream& os, const Ztring& s) {
    s.print(os);
    return os;
}

Notice how the implementation of these functions is simple and elegant because of modularisation and abstraction. Often it is a good idea to refactor long function bodies (e.g., "extract method" refactoring). Check out www.refactoring.com [M. Fowler]
If a class `C` contains dynamically allocated data members then it should have:

- A **copy-constructor**
  
  ```cpp
  C::C(const C&)
  ```
  
  to avoid unwanted sharing of data

- A tailored **assignment operator**
  
  ```cpp
  C& C::operator=(const C&)
  ```
  
  to avoid unwanted sharing of data

- A **destructor**
  
  ```cpp
  C::~C()
  ```
  
  to avoid memory leaks
Encapsulating Free Objects and Overloading `new`, `delete`  
Example: Object Pool for Rationals
class Rational {
    // ...
public:
    void* operator new(size_t) {
        return pool_.alloc();
    }
    void operator delete(void* p, size_t size) {
        assert(size == sizeof(Rational)); // sanity check
        if (p) // do not attempt to delete a null pointer
            pool_.dealloc(p);
    }
    // ...
private:
    static Pool pool_; // a Pool of reusable Rational Objects
    // ...
};

// will allocate from Rational::pool_
Rational* p(new Rational(1, 3));

The void* type indicates a ‘universal’ pointer.
Must be static_cast to a particular type before dereferencing

size_t is the unsigned integral type returned by the operator sizeof()

alloc() and dealloc() are member functions of our user-defined type Pool

Can we achieve better performance than the default new and delete?
#ifndef POOL_H
#define POOL_H
#include "rational.h"

// a pool of reusable areas, each of size sizeof(Rational)
class Pool {
public:
    Pool(unsigned int number_of_areas); // constructor
    ~Pool(); // destructor
    bool is_full() const { return free_ < 0; }
    Rational* alloc(); // return pointer to free area
    void dealloc(void* p); // free an area

private:
    Pool(const Pool&); // copying Pools is forbidden
    Pool& operator=(const Pool&); // assigning Pools is forbidden

    Rational* slots_; // holds an array of reusable Rational areas.
    int* next_; // if i equals the index of a free slot, then
    // next_[i] is the index of another free
    // slot or -1 if full.
    int free_; // index of first free slot or <0 if pool is full.
};
#endif
Pool After Construction

slots_ holds an array of reusable Rational objects
next_ holds an array where each slot holds the index of the next free slot after you use the selected slot
free_ is the index of the next free slot in the pool (-1 if full)
// note: this code assumes that Rational::operator new[]
// and Rational::operator delete[]
// are NOT overloaded (see earlier example)!

#include "pool"
#include <stdlib.h>  // for abort()

// constructor
Pool::Pool(unsigned int size) :
   slots_(new Rational[size]), next_(new int[size]), free_(0) {
   // initially, the free list is 0, 1, 2, ..
   for (unsigned int i=0;(i<(size-1));++i)
      next_[i] = i+1;
   next_[size-1] = -1;  // end of free list
}

// destructor
Pool::~Pool() {
   delete[] next_;
   delete[] slots;
}
(De)Allocation from a Pool

Rational*
Pool::alloc() {
    if (is_full())
        abort();
    Rational* r(&slots_[free_]);
    free_ = next_[free_];
    return r;
}

void Pool::dealloc(void* p) {
    // compute index of pointer p in slots_ array
    int index(static_cast<Rational*>(p) - slots_);
    // add index of deallocated area to front of free list
    next_[index] = free_;
    free_ = index;
}
Smart Pointers
A **smart pointer** gives a user-defined type some of the functionality of a pointer (e.g. dereferencing with -> and *, pointer arithmetic)

The interpretation of \( x->name \) if \( x \) is of type \( T \) is 
\[
(x\cdot\operatorname{operator}->())->name
\]
where \( -> \) needs to be overloaded with \( T::\operatorname{operator}->() \)

```cpp
class Url; // defined elsewhere

class HtmlPage { // a page is uniquely identified by its URL
  friend class Proxy;
  public:
    std::string title() const;
  private:
    static HtmlPage* fetch(const Url&); // retrieve page at URL
    HtmlPage(const HtmlPage&); // forbidden to copy
    HtmlPage& operator=(const HtmlPage&); // forbidden to assign
};
```
// Proxy acts as a logical pointer to a HtmlPage
// which is fetched on demand. The address is in this case a URL.
class Proxy {
public:
    Proxy(const std::string& url) : key_(url) {}  

    // smart pointer
    HtmlPage* operator->() const {
        return HtmlPage::fetch(url());
    }

    const Url& url() const {
        return key_;  
    }

private:
    Url key_; }

Proxy proxy("http://www.vub.ac.be/index.html");
// call HtmlPage::fetch(http://www.vub.ac.be/index.html);
proxy->title();