6 Architecture of C++ programs

Preview

- managing memory and other resources
  - "resource acquisition is initialization" (RAII)
  - using std::auto_ptr and other smart pointers
  - safe construction of an object revisited
  - on reference counting techniques
- physical structure of C++ programs
  - header files and physical dependencies
  - how to organize a C++ program into files
- on principles of object-oriented programming
  - programming to an interface
  - using Design Patterns
  - sample patterns: *Template Method*, *Singleton*

Unmanaged pointers

(1) *raw pointers* cannot have "destructors" that clean up

```
void f () {     // case 1: uses a raw pointer
   X * ptr = new X;
   . . . push_back (Student ("Joe"));  . . . // may throw
     //  calling other functions that may throw or not . .
     //  all code maintained and changing over time . .
   delete ptr;  // possible never executed: potential leak
}
```

(2) ctor exception safety works only for *class-type* members

```
X::X ()       // case 2:  uses pointer members p1 and p2
   : p1 (new A),        // may succeed or throw (but OK)
     p2 (new B) {  // if this throws, p1 is never released!
}               // note that p1 has no destructor to execute
```


Unmanaged pointers (cont.)

(3) trying to recover from an exception

```
try {         // case 3: use manual recovery
   typedef std::vector <int> IntStack;
   IntStack * stack = new IntStack;      // local pointer
   stack->push_back (20);  . . .         // may throw
}                                        // . .  potential leak
catch (std::exception const& e) {
   std::cout << e.what () << std::endl;
   //   . . . ?
}
```

- if *new* fails, no problem: space is released by the system
- but if push_back throws, reserved memory is here lost
  - move the pointer stack out of the *try* block (if can)
  - manual release is still a low-level and ad-hoc solution

Resource management in C++

- to manage resources in C++, use the RAII principle

  "*Resource Acquisition Is Initialization*"

- in general, your code may use several different resources, such as memory, files, locks, ports, etc.
- resources can have managers around them, implemented as classes ("resource wrappers")
  - resources are acquired by invoking a constructor that associates the resource with some data member
  - resources are released by destructors
- the manager provides operations to access and update the resource
- to prevent unwanted resource copies, the copy ctor and the assignment of the wrapper can be made *private*

Resource management (cont.)

The use of resources, r_1, r_2, ..., r_k, divided into three steps

1. acquire all resources r_1, r_2, ...,r_k
2. actual use of resources
3. release resources r_k, ..., r_2, r_1

Implemented in C++ as follows:

```
 . . . Resource1 r1 ( . . . );          // acquire a resource
       Resource2 r2 ( . . . );  . . .   // acquire another resource
       r1.useRes ();  . . .             // use these resources
 }  // automatic release of r2 and r1 (in reverse order)
```

- if, e.g. a ctor throws, all reserved resources are released
- works similarly for (1) declared local objects, (2) nested objects (data members), and (3) any dynamic objects *owned* by (1), (2), and (3) objects

Smart pointers

Problem: built-in pointers may be uninitialized, no automatic release for raw C pointers, create memory leaks, point to destroyed objects, etc.

Solution: smart pointers provide a much better behavior
- can be used as wrappers for primitive pointers
- objects look and behave as if they were pointers
- guarantee initialization (zero or otherwise) & clean up
- an application of the RAII principle
  - always properly set up resources by constructors
  - destructors are automatically called when they go out of scope (or are deleted - or during exceptions)

You can overload two pointer operators
- operator ->, for member access (call a method, etc.)
- operator *, for accessing the whole object (as an *lvalue*)

Use of *auto_ptr*

- include the header file <memory> to get std::auto_ptr
- *new* allocates memory and creates an object
- initialize auto_ptr with the object created

```
Student * p = new Student (321);
std::auto_ptr <Student> stud (p);  // not:  <Student*>
```

- better: always immediately pass the object to its owner

```
std::auto_ptr <Student> stud (new Student (321));
```

- stud becomes the *sole* owner of the object pointed to

auto_ptr features

- requires that there should be *at most one* owner
- can be used and acts like a regular pointer

```
std::cout << stud->getNumber ();        // output "321"
```

- when it goes out of scope (or is otherwise destructed), the object it owns is destroyed

```
void fun () {
    Student * ptr = new Student (783);          // bad idea
    auto_ptr <Student> stud (new Student (321));  // ok
     . . .
    std::cout << stud->getNumber () << std::endl;
}   // at end, deallocation for stud, but not for ptr
```


auto_ptr features (cont.)

- the reset function resets the state of the wrapped pointer, by deleting the old object and assigning a new one

```
auto_ptr <Student> ps;                 // initialized to zero
ps.reset (new Student (783));          // ps owns an object
ps.reset ();        // delete the object and set to zero (0)
```

- how to transform auto_ptr into a regular C pointer

```
T * get () const   // C-pointer to the owned object, or 0
if (ps.get () == 0)  . . .     // ps does not own any object
```

- can also release the ownership of the object

```
auto_ptr <Student> ps (new Student (321));   . . .
Student * ps1 = ps.release ();        // make unmanaged
// now used via regular pointer: no automatic deletion
```


Memory management using *auto_ptr*

```
. . .
{
    typedef   std::vector <int> IntStack;
    // use dynamic object as it were a local object
    std::auto_ptr <IntStack> s (new IntStack);  . . .
    s->push_back (20);                 // may throw
     . . .
    //  no problem: automatic release at exception
}
```


Safe construction using *auto_ptr*

```
class X {
public:
    X () : p1 (new A), p2 (new B) { }  . . .        // safe ctor
    X (X const& rhs)                            // safe copy ctor
      : p1 (new A (*rhs.p1)), p2 (new B (*rhs.p2)) { }  . . .
    X& operator = (X const& rhs) {      // safe assignment
      X tmp (rhs);                      // may throw but OK
      p1 = tmp.p1;  p2 = tmp.p2;        // nofail guarantee
      return  *this; }
       . . .
private:
    std::auto_ptr <A> p1;              // OK: auto_ptrs are
    std::auto_ptr <B> p2;       // class-type data members
};
```

auto_ptr features (cont.)

- the copy operators *transfers the ownership* of an object

```
auto_ptr<Student> stud1 (new Student (321));
auto_ptr<Student> stud2 (stud1);    // owned by stud2
auto_ptr<Student> stud3 = stud2;    // owned by stud3
```

- never can pass an auto_ptr object as a *value parameter*

```
void ShowStudent (auto_ptr <Student> s) { . . // error
  . . . // some nice display of s
auto_ptr <Student> stud (new Student (321));
ShowStudent (stud);          // after this call, stud is 0
std::cout << stud->getNumber ();              // error
```

- unfortunately, the compiler may not complain..
- also, cannot use an auto_ptr with STL containers (they require conventional copy semantics for their elements)

Reference counting

- uses a *handle-body* solution to share objects and deallocate them when they are no longer used by anyone
  - the handle keeps track of the number of references
  - have reference counters in the body - or as *separate shared* objects (also called "nonintrusive")
- define copying operations in the handle with the correct semantics (~ Java reference variables)

```
a = b;              // use reference semantics
```

  - increment the count of the right hand side object
  - decrement and check the count of the left hand side
  - assign pointer to the left hand side object
- implemented using C++ templates, of course


```
template <typename Body>          // simplified version
class CountedPtr {
public:
   explicit CountedPtr (Body *);        // bind owned object
   Body * operator -> ();               // to access members
   Body& operator * ();          // to access the whole object
   CountedPtr& swap (CountedPtr&);       // swap values
     // other operations . . .
   CountedPtr ();
   CountedPtr (CountedPtr const&);
   CountedPtr& operator = (CountedPtr const&);
   ~CountedPtr ();
private:
   Body * rep_;                          // the shared object
   std::size_t * count_;              // separate for generality
};
```


Reference counting (cont.)

Apply reference counting to Student objects:

```
CountedPtr <Student> p1 (new Student (123));
std::cout << p1->getNumber () << std::endl;
CountedPtr <Student> p2 (p1);           // shares the same
CountedPtr <Student> p3 (new Student (321)); // new one
  . . .
p3 = p2;                           // now shares the same
std::cout << *p2 << std::endl;           // print out object
std::cout << p3->getNumber () << std::endl;   // print no
 . . .
 // the Student destroyed when the count becomes zero
```


Reference counting (cont.)

```
template <typename Body>     // a sample implementation
CountedPtr <Body>::CountedPtr (Body * b)
 :  rep_(0), count_(0) {                   // or INVPTR_VAL
    try {          // prevent leak due to a throw from new
       rep_= b;                              // cannot fail
       count_= new std::size_t (1);            // may fail
    } catch (...) {               // or:  std:bad_alloc const&
        delete rep_;  rep_= 0;  // ownership has transferred
        throw;                           // rethrow exception
    }
 }
```


Notes on the implementation

Why used zero-initialization of pointers above?

- it seems that: when a failure occurs, no object and no pointers are returned or left behind to misuse
- still, initializing primitive values reduces unpredictable values and behavior (less random pointers around)
- behavior depends on implementation
  - the address of an object can be assigned *before* its constructor is executed

```
ptr = new CountedPtr . . //  ptr is assigned before ctor
               // and may be later "accidentally" used
```

  - of course, the behavior is undefined and no absolute guarantees are achieved
  - some C++ programming environments do such initializations as a part of their debugging support

Reference counting (cont.)

```
template <typename Body>     // a sample implementation
CountedPtr <Body>&           // uses reference semantics
CountedPtr <Body>::operator = (CountedPtr const& rhs) {
    if (rhs.count_)
        ++*rhs.count_;           // one more reference to rhs
    if (count_ && --*count_ == 0) { // check if lhs is garbage
        delete rep_;  delete count_;
    }
    rep_= rhs.rep_;                   // share the object
    count_= rhs.count_;               // share its counter
    return *this;
}
```


Notes on the implementation

- can here omit the self-assign test (even if "this == &rhs"):

```
//   if (this == &rhs || rep_ == rhs.rep_) return *this;
++*rhs.count_;           // increments rhs (/lhs) count_
if (--*count_ == 0) {    // decrements lhs (/rhs) count_
```

  - actually optimized by leaving out (mostly useless)
- some overhead for separately allocated counters

```
size_t * count_;         // shared reference count
```

- could be optimized by a special allocation strategy (overload *new* for a counter class [Stroustrup, p. 421])
- note that auto_ptr avoids any such overheads
- sometimes reference counts can be placed into the Body objects (say, for some *string* class implementation)

Problem: unnecessary header dependencies

```
// File: data.h
class Data {   . . .
};
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// File: client.h
#include "data.h"       // gets the Data class definition
class Client {           // compiles OK, but bad style
    Data query () const;   . . .
private:
    Data * ptrData_;
};
```

- changes to Data propagate to Client-related source code
- the physical dependence may create maintenance problems - or, at least, force recompilations

Required class information

What information is required from the class X in order to compile client code? For example:

```
X obj;      // compiler needs to know instance size
            // to allocate space for the object
```

However, this information *is not required* for:

(1) members with pointers or references, e.g., X * or X&

(2) function *declarations* with value parameters or results

X getX () or void print (X par) need only declaration

- only the *caller* of the operations needs the definitions to determine the required sizes (to actually pass values)
- thus, many times header files don't need to include full definitions to define their services and interfaces

Breaking unwanted dependencies

```
// File client.h
class Data;                   // forward declaration only
class Client {
public:
    Data query () const;  . . . // OK: no impl. needed here
private:
    Data * pData_;  . . .       // OK: no impl. needed here
};
```

- only source code that actually creates objects needs to include appropriate header files
  - e.g., "client.cpp" may need to include "data.h" (but no problem since it is an isolated translation unit)

Idiom: Pimpl (Pointer to impl.)

- also known as the *Handle-Body* idiom
- the class definition, with its private and protected parts, is often unnecessarily used to compile the client's code
- leads to a physical coupling between use of abstraction and its implementation, making maintenance difficult
- to avoid recompilation of the client's code, separate interface and implementation, and include only those header files that really are necessary for the client

```
class Abstraction {              // the handle part
    . . .
private:
    struct Impl * body_;         // hides the body
    . . .           // to be used in implementation unit
};
```

Problems with templates and headers

- not all "classes" can be forwarded with names only
  - std::string is really a *typedef* of a template instance and thus cannot be introduced by its name only
- the standard library provides the header <iosfwd> with minimal declarations for stream templates and their standard *typedef*s, such as std::ostream
- similar practice is recommended for user-defined headers
- no such forward header file exists for std::string
- C++ implementations sometimes #include extra header files along system headers, making code nonportable
  - in another platform missing headers break compilation
  - minimize dependencies on nested #includes by always including system headers as the last ones

Headers: summary

- #include only header files that are minimally needed to make your declarations and code to compile
  - let *users* include those header files *they need*
- prefer introducing classes by forward declarations ("class X;")
  - sufficient for declaring any functions, and for using pointers or references as data members
- always first *#include* user-defined (custom) header files, and after them system header files
  - the strategy ensures that custom headers are properly defined and stay independent from any hidden extra header file inclusions

OOP: programming to an interface

The basic principle of object-oriented programming:

Program to an interface, not an implementation.

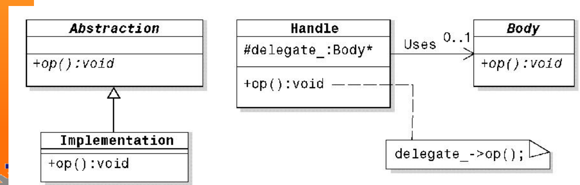
- access objects in terms of their abstract interfaces
- the code looks something like:

```
Interface * p = getObject ();
p->op ();  . . .     // avoid binding to specific impl.
```

- the actual type of an object is not known (possibly not even yet defined)
- the code is made independent from implementations
  - but remember that raw pointers are problematic: use smart pointers, such as boost::shared_ptr or the similar addition to the 200x version of C++ (C++0x)

Inheritance vs. delegation

- reuse through inheritance (~ white-box reuse)
  - uses a compile-time relationship between two classes (base and derived); define extensions and override
  - cannot easily make modifications at run-time
- reuse through delegation (~ black-box reuse)
  - uses a run-time relationship between two objects



Inheritance vs. delegation (cont.)

Object composition combined with inheritance is often a better alternative than just inheritance:

- the client uses an interface provided by the handle that delegates requests to the body
- the handle need not be changed for new implementations
- can define new functionality without affecting the client
  - when changes occur, the client's code need not be recompiled but only has to be relinked with the modified code

Inheritance vs. delegation (cont.)

- delegation decomposes an abstraction into separately manageable parts
- the body may be polymorphic (implemented by derived classes) and replaced at run time
- several design patterns are based on delegation techniques: *Bridge*, *Proxy*, *State*, etc.
- in C++, handle objects are also used for modularization and for management of memory, consider
  - the *Pimpl* idiom
  - smart pointers that are defined for resource management (see the part on RAII)

Source of Design Patterns

In software projects, we have concrete problems, e.g.:

- how to sequentially *access the elements* of an aggregate *without* the knowledge of its implementation
- how to represent a *shared* global *resource* as an object
- a way to provide for the *undoing* of actions
- how to provide a unified *interface to a set of services*

Design patterns are solutions to such common problems

(1) the *name* of the pattern,

(2) the *problem* and its *context* (circumstances, forces)

(3) the *solution*, usually as a set of collaborating objects

(4) the *consequences* (pros and cons) of using the pattern

Plus: sample code, implementation tips, related patterns, actual systems ("at least *two*"), etc.

Source of Design Patterns (cont.)

- solutions to design problems that you see over and over
- design patterns are not invented but found (in systems)
- patterns give solutions to (nontrivial) problems; these solutions have been tried and approved by experts, who have also named, analyzed, and catalogued these patterns
- often provide a level of indirection that keeps classes from having to know about each other's internals
- give ways to make some structures or behavior modifiable or replaceable
  - usually, by objectifying some aspect of the system
- generally, help you write more reusable programs

Philosophy of Design Patterns (cont.)

Different kinds of practices and reusable designs

(1) idioms - describe techniques for expressing low-level, mostly language-dependent ideas (ref. counts in C++)

(2) design patterns - medium-scale, mostly language-independent abstractions (solution "ideas")

- usually depend on object-oriented mechanisms
- essential features can often be described with a simple UML class diagram

(3) software frameworks - consist of source code with variant parts, into which the user can plug-in specific code to provide the required structure and behavior

- for example, GUI libraries in Java (with their Hollywood principle: use callbacks)

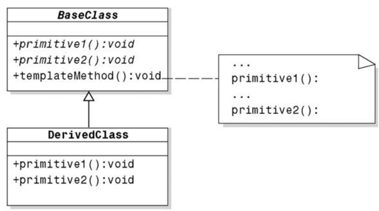
Basic Design Patterns

- the *Template Method* makes part(s) of an algorithm changeable: define the skeleton of an algorithm, deferring some ot its steps to subclasses (the related pattern *Strategy* uses delegation at run time)
- the Iterator pattern gives access to elements but hides internal organization (originally, a feature in *CLU*)
- the Singleton ensures a class only has one instance, and provides a global point of access to it (manages globals)
- the Bridge pattern separates interface and implementation hierarchies, and allows them vary independently (related to the *Handle-Body* idiom)
- the *Abstract Factory* provides an interface for creating families of related or dependent objects without specifying their concrete classes; uses *Factory Methods* or *Prototypes* to create the actual instances; the factory object is often a *Singleton*

Template Method pattern

- the *Template Method* design pattern defines a general algorithm, in terms of some calculation steps that are expressed as abstract operations, to be defined in derived classes
  - do not confuse with generics and C++ templates
- the most basic pattern: almost trivial use of OOP
  - but still a *pattern* that identifies a *problem* to solve
- *Template Method* is a behavioral class pattern
  - two kinds of behavioral patterns
    - class patterns that use inheritance, and
    - object patterns that use object composition and delegation (e.g., *Bridge*, *Strategy*)

Template Method pattern (cont.)



The *Template Method* design pattern

- abstract operations represent the variable parts
- template methods use these abstract operations
- e.g., an abstract operation could be a *Factory*
- the derived classes are responsible for implementing the abstract operations

Template Method pattern (cont.)

- the template method itself usually should not be overridden, and therefore it is not defined as *virtual*
- all deferred operations that are used by the template method are defined as *virtual*, as well as *private* (or *protected*), since the client is not supposed to directly call them

Note *white-box reuse* strategy

- must know what needs to be redefined and what not

UNIVERSITY OF HELSINKI

The *Singleton* pattern

- used to constrain object instantiation to ensure that the client cannot create more than one object of a particular class
- in Java, the class java.lang.Runtime is a singleton class with a static method getRuntime (), responsible for returning the single instance of this class
- an example of a *creational object pattern*
- in practice, is used to ensure that there is exactly
  - one window manager or print spooler,
  - a single-point entry to a database,
  - a single telephone line for an active modem, etc.

UNIVERSITY OF HELSINKI


```
class Singleton {
public:
    static Singleton& instance ();
    void op ();                // anything useful for the singleton
 private:
    Singleton ();               // implement but make private
    Singleton (Singleton&); // don't implement but declare
    void operator = (Singleton&);     // to avoid defaults
 };
 . . . // implementation file
 Singleton * onlyInstance_;               // = 0 by default
 Singleton& Singleton::instance () {
    if (onlyInstance_== 0)
       onlyInstance_= new Singleton;
    return *onlyInstance_;
 }
```

UNIVERSITY OF HELSINKI

Singleton (cont.)

- returns a reference: harder to destroy accidentally
- as an alternative solution, can use *static local* variables

```
class Singleton {
public:
   static Singleton& instance ();              // reference
    . . .
}; . . .
// implementation file:
Singleton& Singleton::instance () {
   static Singleton instance;      // created at first call
   return instance;
}                           // and destructed after main
```

UNIVERSITY OF HELSINKI

Notes on *Singleton* implementation

1. The client accesses features only through instance ()
   `Singleton::instance ().op ();`
2. The static method instance () uses *lazy evaluation*.
3. Public operations are application-specific services.
4. The constructors and the copy operations are defined as *private* or *protected*
   `Singleton * s = new Singleton ();          // error`
5. Often no use to delete and *recycle* a singleton (only one); and the following would create a dangling pointer
   `delete &Singleton::instance ();          // error`

   Solution: declare the destructor as *private*, too.
6. A *static* local object (within the function) is destructed after main; for the pointer solution, an automatic clean-up must be arranged separately (see next slide).

UNIVERSITY OF HELSINKI


```
static Singleton * onlyInstance_;       // = 0, by default
Singleton& Singleton::instance () {
   if (onlyInstance_== 0)
      onlyInstance_= new Singleton;
   return *onlyInstance_;
}
// deletion of a singleton for the pointer solution
struct SingletonCleanUp {                 // helper class
   ~SingletonCleanUp () {
      delete onlyInstance_;
      onlyInstance_= INVPTR_VAL;              // or:  = 0
   }
};
static SingletonCleanUp Dtor_;   // cleanup after main
```

- the singleton is destructed after the main terminates

UNIVERSITY OF HELSINKI

Application architecture: summary

- **RAII: use objects to manage resources**
- **keep your headers clean and minimal**
  (1) don't reveal unnecessary implementation details
  (2) use *class forward* declarations when sufficient
  (3) but use always full namespace paths
- **include only headers that really are needed to compile**
- **first include user-defined (custom) header files**
- **use namespaces, and *using* declarations (not *directives*)**
- **use *unnamed namespace* within implementation files**
- **use design patterns to make some aspects (structure or behavior) of software separate and manageable**
  - ***Singleton*, *Iterator, Template method, Factory***