Unified Modeling Language
a case study

1. an online phone book
   - use case diagram
   - encapsulating a file
2. Command Line Arguments
   - arguments of main
   - arrays of strings
3. Class Definition
   - the files `phonebook.h` and `phonebook.cpp`
   - the main program

MCS 360 Lecture 5
Introduction to Data Structures
Jan Verschelde, 24 January 2020

Unified Modeling Language

1. an online phone book
   - use case diagram
   - encapsulating a file

2. Command Line Arguments
   - arguments of main
   - arrays of strings

3. Class Definition
   - the files `phonebook.h` and `phonebook.cpp`
   - the main program

An Online Phone Book

As case study we consider the management and the consultation of an online phone book.

Two types of use:

1. manager: add and delete entries;
2. reader: lookup phone numbers.

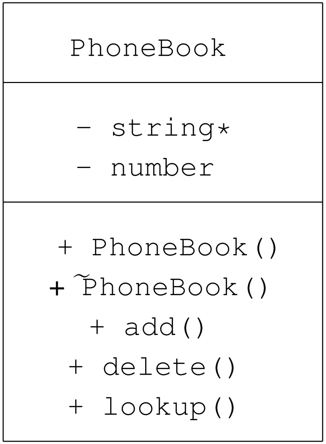
Two types of diagrams in UML:

1. class diagram: defines data and methods;
2. use case diagram: who uses what methods.

the class `PhoneBook`

We use an array of strings to represent the entries in a phone book.

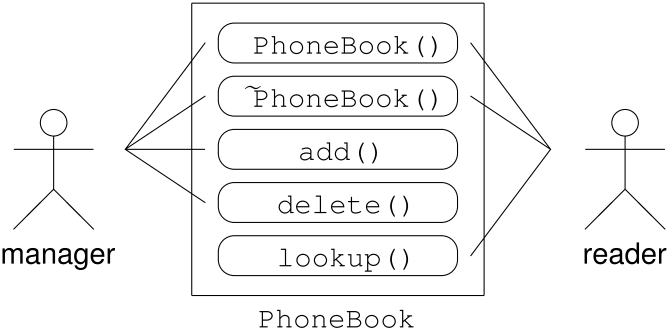
A class diagram:
−: private
+: public



Use Case Diagram for PhoneBook

a behavior modeling diagram

Managers and readers differ in their use of the PhoneBook:



Unified Modeling Language

1. an online phone book
   - use case diagram
   - encapsulating a file
2. Command Line Arguments
   - arguments of main
   - arrays of strings
3. Class Definition
   - the files `phonebook.h` and `phonebook.cpp`
   - the main program

Encapsulating a File

What does the creator `PhoneBook()` do?

1. read number of entries from file;
2. allocate memory;
3. read data from file into array.

The destructor `~PhoneBook()` deallocates the memory.

Principle of information hiding:

1. actual file and its format hidden from the user;
2. programmer considers array of strings.

Command Line Arguments

Main program is used in two different modes:

1. administrative mode by manager;
2. reader only consults the phone book.

One possible way of implementation: `-a` is command line argument of `phonebook` program.

Run in administrative mode:

```
$ phonebook -a
```

Without option, program runs in reader mode.

Unified Modeling Language

1. an online phone book
   - use case diagram
   - encapsulating a file
2. Command Line Arguments
   - arguments of main
   - arrays of strings
3. Class Definition
   - the files `phonebook.h` and `phonebook.cpp`
   - the main program

Arguments of `main`

```
int main ( int argc, char *argv[] )
```

Two optional arguments:

1. `argc`: the number of arguments, `argc` ≥ 1
   If there are no command line arguments, then argc $= 1$.
2. `argv`: the arguments of the command line
   `argv` is an array of strings,
   `argv[0]` is the name of the program
   `argv[i]` is the (i-1)-th argument of the program.

Example at the command prompt `$`:

```
$ command_line_args -a somefile.txt
```

Unified Modeling Language

1. an online phone book
   - use case diagram
   - encapsulating a file
2. Command Line Arguments
   - arguments of main
   - arrays of strings
3. Class Definition
   - the files `phonebook.h` and `phonebook.cpp`
   - the main program

arrays of strings

```
#include <iostream>
#include <string>
using namespace std;

int main ( int argc, char *argv[] )
{
   cout << "name of the program : \""
        << argv[0] << "\"" << endl;

   cout << "number of command line arguments : "
        << argc-1 << endl;

   if(argc > 1)
      for(int i=1; i<argc; i++)
         cout << "argument " << i << " : \""
              << argv[i] << "\"" << endl;

   return 0;
}
```

two exercises

Exercise 1:

1. Modify `copy_files.cpp` of lecture 3 so that the names for the two files are given on the command line.

Exercise 2:

2. Add the command line option `-v` to the `hello_world.cpp` program of the first lecture.
    - Without `-v`, the program executes as before.
    - With `-v`, the program prints the version number, e.g.: `Release 1.0`.

Unified Modeling Language

1. an online phone book
   - use case diagram
   - encapsulating a file
2. Command Line Arguments
   - arguments of main
   - arrays of strings
3. Class Definition
   - the files `phonebook.h` and `phonebook.cpp`
   - the main program

Files

The data is stored on file, in `phonebook_data.txt`.
Example:

```
2
111-222-4444 Elliot Koffman
333-666-9999 Paul Wolfgang
```

Number of entries comes first, one entry per line.
On each line, a phone number is in the first 11 characters followed by a name.

Other files:

- `phonebook.h`: public and private attributes;
- `phonebook.cpp`: defines the class methods;
- `use_phonebook.cpp`: the main program.

the file `phonebook.h`

```
#ifndef PHONEBOOK_H
#define PHONEBOOK_H
#include <string>
class PhoneBook
{
   public:
      PhoneBook();
      ~PhoneBook();
      int length() const;
      std::string operator[](size_t k) const;
      void add(const std::string s);
   private:
      int number;            // number of entries
      std::string *data;     // array of strings
};
#endif
```

specifications of methods

Add as documentation in `phonebook.h`:

```
PhoneBook();
/*
   Reads phone book entries from file.

   Precondition:
      file phonebook_data has valid entries.
   Postcondition:
      for PhoneBook b, there are b.length()
      entries b[k], with 0 <= k < b.length(). */

~PhoneBook();
/*
   Deallocates memory occupied by entries.

   Postcondition:
      b.number == 0 after b.~PhoneBook(). */
```

specifications continued

```
int length() const;
/*
   Returns the number of entries in phone book.

   Precondition:
      constructor PhoneBook() executed correctly.
   Postcondition: length() >= 0. */

std::string operator[](size_t k) const;
/*
   Returns element at index k in phone book.

   Precondition: k < b.length() for PhoneBook b.
   Postcondition:
      b[k] is k-th entry in phone book,
      matching appropriate line on file. */
```

specifications of `add`

```
void add(const std::string s);
/*
   Adds a new entry defined by the data in s.

   Precondition:
      s matches the data format for file,
      contains phone number and name.
   Postcondition:
      after PhoneBook b; b[b.length()-1] == s. */
```

In a more elaborate design, a separate class would define the layout of the strings on file.

constructor and destructor

```
#include <limits>
#include <fstream>
#include "phonebook.h"
PhoneBook::PhoneBook()
{
   std::ifstream ins("phonebook_data.txt");
   ins >> number;
   data = new std::string[number];
   ins.ignore(std::numeric_limits<int>::max(),'\n');
   for(int k=0; k<number; k++)
      getline(ins,data[k],'\n');
   ins.close();
}
PhoneBook::~PhoneBook()
{
   delete[] data;
   number = 0;
}
```

selectors and modifier

```
int PhoneBook::length() const
{
   return number;
}
std::string PhoneBook::operator[](size_t k) const
{
   return data[k];
}
void PhoneBook::add(const std::string s)
{
   std::ofstream outs("phonebook_data.txt");
   number = number + 1;
   outs << number << std::endl;
   for(int k=0; k<number-1; k++)
      outs << data[k] << std::endl;
   outs << s << std::endl;
   outs.close();
}
```

Unified Modeling Language

1. an online phone book
   - use case diagram
   - encapsulating a file
2. Command Line Arguments
   - arguments of main
   - arrays of strings
3. Class Definition
   - the files `phonebook.h` and `phonebook.cpp`
   - the main program

the main program

```
#include <iostream>
#include "phonebook.h"
using namespace std;

int main ( int argc, char* argv[] )
{
   PhoneBook b;
   int n = b.length();

   cout << "number of entries : " << n << endl;
   if(argc == 1)
      for(int k=0; k<n; k++)
         cout << "  entry " << k << " : "
              << b[k] << endl;
```

running as manager

```
   else
   {
      string new_entry;

      cout << "give new entry : ";
      getline(cin,new_entry);

      b.add(new_entry);
   }

   return 0;
}
```

Dynamic Allocation and Deallocation

```
int main ( int argc, char* argv[] )
{
   PhoneBook *b;
   // statements welcoming user
   b = new PhoneBook;     // allocation
   // statements using b
   b->~PhoneBook();       // deallocation
```

For `PhoneBook *b`, need to replace

- `b.length()` by `b->length()`
- `b[k]` by `(*b)[k]`
- `b.add(new_entry)` by `b->add(new_entry)`

the makefile defines the `make use_phonebook`

1. which c++ compiler to use,
2. the compilation of the definition of the class,
3. the compilation of the main program,
4. the linking of the files with the object code.

```
gpp=g++

use_phonebook:
        $(gpp) -c phonebook.cpp
        $(gpp) -c use_phonebook.cpp
        $(gpp) -o /tmp/use_phonebook \
                 phonebook.o use_phonebook.o

clean:
        del *.o
```

Summary + Additional Exercises

We ended Chapter 1: *Introduction to Software Design*.

Additional Exercises:

3. Provide another constructor to the class `PhoneBook` with allows the name of the file as input parameter.
4. Add a method `delete(size_t k)` to remove an entry from file, given the index `k` in the array.
5. Develop a `search(const string name)` method to search a phone number given a name.