Assignment 3:
Use `make` instead of `gmake` on kits.few.vu.nl

Outline
- About object oriented programming (OOP)
- Structures in C
- Classes and objects in C++
- Separate compilation

Reading
- Prata book, Chapter 10: Objects and classes

Procedural programming
- The problem is decomposed into smaller units named procedures
- Data is assembled in packages, called structures.
- Data and procedures are separated.
- C, Pascal, Fortran, Modula2 are procedural languages
- Complex programs created in procedural languages are difficult to manage, hard to maintain, and expensive to extend.

Object oriented programming
- A new way to program: object oriented programming (OOP)
- OOP combines data and procedures in one single unit = object
- Simula (introduced classes), Smalltalk, C++ are class-based OOP languages
Object oriented programming is a tool for new challenges in software development
- offers a closer fit to the way humans think
- improves communication
- improves the quality of software

Goal: to design high quality software at low cost

Important OOP concepts:
- data hiding - compiler-enforced public or private data access
- encapsulation - public interface, private data implementation
- inheritance - classes derive functionality from parents
- polymorphism - access derived classes through parent classes

Essence of OOP:
Don’t think about data and functions separately, think about objects. Objects are small bundles of data that know how to do things with themselves.

Example:
You can think: A car is a collection of wheels, doors, seats, windows. But think what a car can do: move, speed up, slow down, stop, park, etc. Put everything you know about a car in one object.

You don’t say: the computer moves the car. You say: the car moves itself.

Examples of objects
- a window: close, open, resize, move
- a data structure (list, tree): find, remove, sort, insert
- a file: open, close, read, write, rename
- 3D object: color, rotate, resize

C has structures
```c
struct Car
{
    string name;
    int top_speed;
    int capacity;
};
```

C++ has classes
- A class is a user-defined type. The variables of this type are objects (or instances).
- A class can be obtained from a structure if some member functions are added.

Outline
- About object oriented programming (OOP)
- Structures in C
- Classes and objects in C++
- Separate compilation
Classes

- All cars are objects of the same type
- The description of this type is a class: class Car
- Advantage: clients of your class can use it without worrying about how it works (Encapsulation)

First example with classes

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

class Car
{
private:
    int weight, capacity;
    float top_speed;
public:
    void printData();
}

int main()
{
    Car my_car;
    my_car.printData();
    return 0;
}

void Car::printData()
{
    cout << "weight = " << weight
         << " capacity = " << capacity
         << " top speed = " << top_speed
         << endl;
}
```

Encapsulation

- Combining a number of items (variables) into a single package such as a class
  - Data is accessed only through member functions
  - Implementation is independent of interface!
  - Data variables are not accessible to users of class

Abstract data types

- An abstract data type (ADT) does not allow access to the details of how the values and functions are implemented
  - An ADT is an example of encapsulation
  - Example: C++ predefined types (int, double, etc) and C++ operators

OOP tip: Make sure all the classes you define are ADTs

How to make an ADT?

- Separate the interface from the implementation:
  - Interface: how the type is used by a programmer
  - Implementation: the details of how the type is implemented
- Make all member variables private to the class (inaccessible outside class)

Private and public

- A member (variable or function) can be private or public
  - Access is enforced by the compiler only
- A "good" class keeps its member variables private = data hiding
  - Use public member functions to access or change each private variables.
  - These public member functions are called accessors and mutators.
Member functions

- Accessor function is usually named get***...
  - Returns data without changing object
    - `getAge()`, `getSize()`, `getDate()`, ...
- Mutator function is usually named set***...
  - Changes object!
    - `setAge(x)`, `setSize(x)`, `setDate(x, y, z)`

```cpp
class Vehicle
{
  private:
  int weight, capacity;
  float topspeed;
  
  public:
  void printData();
  
  // here are 3 accessor functions
  const int getWeight() { return weight; }
  const int getCapacity() { return capacity; }
  const float getTopSpeed() { return topspeed; }
  // and now the 3 mutator functions
  int setWeight(int new_weight) { weight = new_weight; }
  int setCapacity(int new_capacity);
  int setTopSpeed(float new_topspeed);
};
```

Example: data hiding

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

class Vehicle
{
  public:
  int weight, capacity;
  float top_speed;
  void printData();
};

int main()
{
  Vehicle my_car;
  my_car.printData();
  return 0;
}

void Vehicle::printData()
{
  cout << "weight = " << weight <<", capacity = " << capacity <<", top speed = " << top_speed << endl;
}
```

Test

What is the output of this program?

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

class Vehicle
{
  public:
  int weight, capacity;
  float top_speed;
  void printData();
};

int main()
{
  Vehicle my_car;
  my_car.printData();
  return 0;
}

void Vehicle::printData()
{
  cout << "weight = " << weight <<", capacity = " << capacity <<", top speed = " << top_speed << endl;
}
```

Constructors

- A constructor is a special member function automatically called when an object is created.
- Constructors are used for object initialization.
- Rules:
  - Constructors should be public
  - Constructors must have the same name as the class
  - Constructors have no type for the return value (not even void)
  - Classes can have more than 1 constructor (overloading)
  - If no constructor is declared, C++ generates a default constructor that does nothing (no initialization at all)
# Example: constructors & destructors

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

class Vehicle
{
private:
    int weight, capacity;
    float topspeed;
public:
    void printData();
    Vehicle(); // default constructor
    // another constructor with 3 parameters
    Vehicle(int new_weight, int new_cap, float new_topspeed);
    ~Vehicle(); // destructor
};

void Vehicle::printData()
{
    cout << "weight = " << weight <<", capacity = " << capacity <<", top speed = " << topspeed << endl;
}

Vehicle::Vehicle()
{
    topspeed = weight = capacity = 0;
    cout << "Object created" << endl;
}

Vehicle::Vehicle(int new_weight, int new_cap, float new_topspeed)
{
    weight = new_weight;
    capacity = new_cap;
    topspeed = new_topspeed;
    cout << "object created & initialized" << endl;
}

Vehicle::~Vehicle()
{
    cout << "Object destroyed" << endl;
}
```

```cpp
Example: constructors & destructors

int main()
{
    Vehicle mycar1;
    Vehicle mycar2(10000, 4, 125);
    mycar1.printData();
    mycar2.printData();
    return 0;
}
```

Gives the output:

Object created
Object created & initialized
weight = 0, capacity = 0, top speed = 0
weight = 10000, capacity = 4, top speed = 125
Object destroyed
Object destroyed

---

# Example: friends

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

const int MAX_SIZE = 100;

class TemperatureArray
{
private:
    double array[MAX_SIZE];
    int size;
public:
    TemperatureArray(); // constructor
    void add_temperature(double temperature);
    bool full();
    friend void print (const TemperatureArray& the_object);
};

void TemperatureArray::add_temperature(double temperature)
{
    if (size < MAX_SIZE)
    {
        array[size] = temperature;
        size++;
    }
}

bool TemperatureArray::full()
{
    return size == MAX_SIZE;
}

friend void print (const TemperatureArray& the_object)
{
    cout << "TemperatureArray: " << endl;
    for (int i = 0; i < the_object.size; i++)
    {
        cout << the_object.array[i] << " ";
    }
    cout << endl;
}
```

---

# Friend functions

- **A friend function** of a class is an ordinary function which has access to the private members of this class.
  - Exception to compiler-enforced access control
  - Warning: This breaks encapsulation!
- **How to make a function friend of a class?**
  - List its name in the class definition by using the keyword `friend`
  - Can also be applied to classes:
    - Friend class `Helper`;
- **OOP Tip:** Use friends sparingly!
TemperatureArray::TemperatureArray() {
    size = 0;
}

void TemperatureArray::add_temperature(double temperature) {
    if (full()) {
        cout << "Array is full" << endl;
        exit(1);
    } else {
        array[size] = temperature;
        size = size + 1;
    }
}

bool TemperatureArray::full() {
    return (size == MAX_SIZE);
}

Example: friends

void print(const TemperatureArray& the_object) {
    for (int i = 0; i < the_object.size; i++)
        cout << the_object.array[i] << endl;
}

int main() {
    TemperatureArray my_array;
    for (int i = 1; i < 10; i++)
        my_array.add_temperature(i);
    print(my_array);
    return 0;
}

Example: friends

Inheritance

- C++ classes support inheritance
  - Like Java extends, but with access
  - A: public B
    - A is the base class (or parent class)
    - B is the derived class (or subclass)
- Used for:
  - Specialization: a dog is-a animal, etc.
  - Overriding: subclass has a different implementation

Polymorphism

- C++ inheritance supports polymorphism
  - Like Java Interface
    - A: public B
      - Use type B as reference to all subclasses
      - Allows subclasses to specialize implementation while code using B doesn’t need to change

Using inheritance

Class Animal {
    public: run();
    speak() { cout << "blurb "; } }
Class Duck: public Animal {
    public: speak() { cout << "quack "; } }
Class Dog: public Animal {
    public: speak() { cout << "woof "; } }
Class Cat: public Animal {
    public: speak() { cout << "meow "; } }

Polymorphism example

Animal ani; Animal *aniP; Duck duck; Dog dog; Cat cat;
duck.speak(); dog.speak(); cat.speak();
cout <<endl;
ani.speak();
aniP = &duck; ani->speak();
aniP = &dog; ani->speak();
aniP = &cat; ani->speak();
cout <<endl;
Output:
quack, woof, meow
blurb, quack, woof, meow
Outline

- About object oriented programming (OOP)
- Structures in C
- Classes and objects in C++
- Separate compilation

Separate compilation

A small object-oriented program should have:
- an interface file: usually with the extension .h or .hpp - a header file that describes the services provided by the class.
- an implementation file: contains the bodies for the member functions, usually .cpp
- an application file: creates and uses objects of this class, usually .cpp

Rule: Every file using the class must include the appropriate interface file
#include <libClass.h> ; for system interface file, part of standard C++ library
#include "myClass.h" ; " " for immediate interface files, in the same directory as the source code

What happens to a C++ program?

Implementation: source code
Interface: header file

Option 1: Compiling by hand:
g++ car.cpp wheels.cpp doors.cpp windows.cpp body.cpp –o car.exe

How to compile multiple files?

We usually separate different classes into different files to simplify locating the code and to encourage reuse (including in multiple programs).

Option 1: Compiling by hand:
g++ car.cpp wheels.cpp doors.cpp windows.cpp body.cpp –o car.exe

Multiple files: make

Option 2: Using make utility
make reads info from a makefile:
# Build the car program
car: car.o wheels.o doors.o windows.o body.o
g++ -c car.cpp
../wheels.cpp
g++ -c doors.cpp
g++ -c windows.cpp
g++ -c body.cpp
g++ -o car car.o wheels.o doors.o windows.o body.o
clean:
rm -f *.o

using #ifndef

To prevent headers from being included more than once, C++ needs some construction to tell the compiler: if you have included this stuff before, do not include it again.

Use this sequence in the header file:
#ifndef STACK_H_
#define STACK_H_
....class definition
#endif // ifndef STACK_H_
Problem: write a class to implement a stack. Use this class in a program which reads a word as a sequence of letters and types this word in reversed order. Use multiple files and separate compilation.

Example multiple files: stack

```c++
# ifndef STACK_H
# define STACK_H
struct StackNode {
    char data;
    StackNode *next;
};
typedef StackNode* StackNodePtr;
class Stack {
private:
    StackNodePtr top;
public:
    Stack();
    ~Stack();
    void push(char symbol); // push element onto stack
    char pop(); // remove and return top of stack
    bool empty(); // returns whether stack contains data
};
#endif
```

The interface file: stack.h

```c++
#include <iostream>
#include <cstddef>
#include "stack.h"
using namespace std;
```

```c++
void Stack::push(
    char symbol) {
    StackNodePtr temp_ptr;
    temp_ptr = new StackNode;
    temp_ptr->data = symbol;
    temp_ptr->next = top;
    top = temp_ptr;
    return;
}
```

Implementation file: stack.cpp (1/2)

```c++
void Stack::pop(){
    if(empty()){
        cout <<"Error popping" <<" empty stack" <<endl;
        exit(1);
    }
    char result = top->data;
    StackNodePtr temp_ptr;
    temp_ptr = top;
    top = top->next;
    delete temp_ptr;
    return result;
}
```

Implementation file: stack.cpp (2/2)

```c++
int main() {
    Stack s;
    char next_char, ans;
    do {
        cout <<"Enter a word: ";
        cin.get(next_char);
        while (next_char != 'n') {
            s.push(next_char);
            cin.get(next_char);
        }
        cout <<"Written backward is: ";
        while (!s.empty()) {
            cout << s.pop();
        }
        cout << endl;
        cout <<"Again? (y/n): ";
        cin >> ans;
    } while (ans != 'n' && ans != N');
    return 0;
}
```

Application program: test.cpp

```c++
```

Questions?

Programming in C/C++ / Lecture 3 / 16-02-2009