ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Εισαγωγή στον Προγραμματισμό
Introduction to Programming

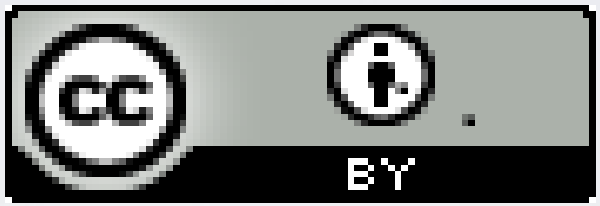
Διάλεξη 8: Κλάσεις κ.λ.π.

Γ. Παπαγιαννάκης

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται στην άδεια χρήσης **Creative Commons** και ειδικότερα

Αναφορά Δημιουργού 3.0 - Μη εισαγόμενο Ελλάδα (Attribution 3.0– Unported GR)



- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

ΗΥ-150 Προγραμματισμός
CS-150 Programming

Lecture 8:
Technicalities: Classes, etc.

G. Papagiannakis

Abstract

- This lecture presents language technicalities, mostly related to user defined types; that is, classes and enumerations.

Overview

- Classes
  - Implementation and interface
  - Constructors
  - Member functions
- Enumerations
- Operator overloading
- Useful headers in C++
- 2nd Assignment

Classes

- The idea:
  - A class directly represents a concept in a program
    - If you can think of “it” as a separate entity, it is plausible that it could be a class or an object of a class
    - Examples: vector, matrix, input stream, string, FFT, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
  - A class is a (user-defined) type that specifies how objects of its type can be created and used
  - In C++ (as in most modern languages), a class is the key building block for large programs
    - And very useful for small ones also
  - The concept was originally introduced in Simula67

Members and member access

- One way of looking at a class;

```
class X {     // this class' name is X
   // data members (they store information)
   // function members (they do things, using the information)
};
```

- Example

```
class X {
public:
   int m;       // data member
   int mf(int v) { int old = m; m=v; return old; }   // function member
};

X var;                    // var is a variable of type X
var.m = 7;                // access var's data member m
int x = var.mf(9);        // call var's member function mf()
```

Classes

- A class is a user-defined type

```
class X {       // this class' name is X
public:         // public members -- that's the interface to users
                //      (accessible by all)
    // functions
    // types
    // data (often best kept private)
private:        // private members -- that's the implementation details
                //      (accessible by members of this class only)
    // functions
    // types
    // data
};
```

Struct and class

- Class members are private by default:

```
class X {
        int mf();
        // ...
};
```

- Means

```
class X {
private:
        int mf();
        // ...
};
```

- So

```
X x;                // variable x of type X
int y = x.mf();     // error: mf is private (i.e., inaccessible)
```

Struct and class

- A struct is a class where members are public by default:

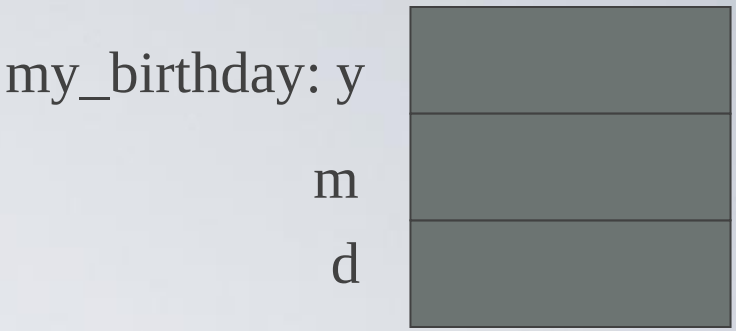
```
struct X {
        int m;
        // ...
};
```

- Means

```
class X {
public:
        int m;
        // ...
};
```

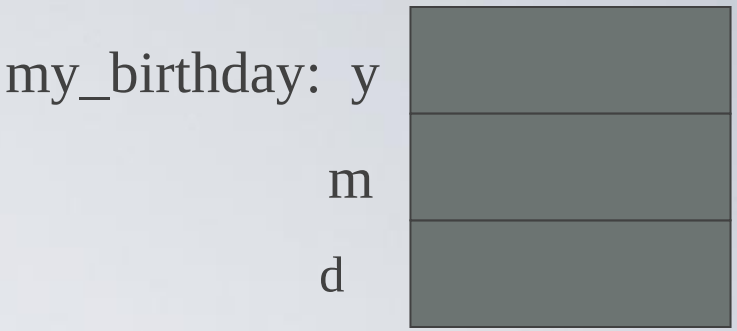
- **struct**s are primarily used for data structures where the members can take any value

Structs



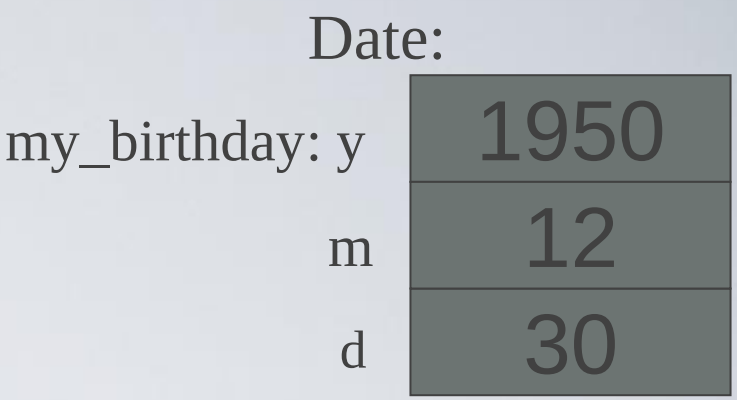

```
// simplest Date (just data)
struct Date {
   int y,m,d;      // year, month, day
};

Date my_birthday;      // a Date variable (object)

my_birthday.y = 12;
my_birthday.m = 30;
my_birthday.d = 1950;// oops! (no day 1950 in month 30)
                     // later in the program, we'll have a problem
```

Structs



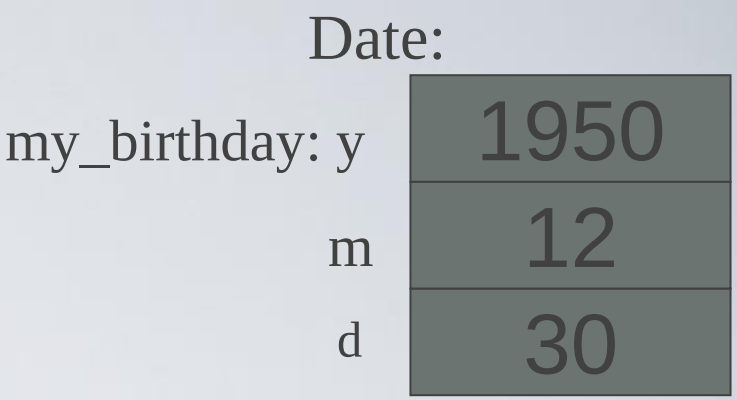

```
// simple Date (with a few helper functions for convenience)
struct Date {
    int y,m,d;          // year, month, day
};

Date my_birthday;           // a Date variable (object)


// helper functions:

void init_day(Date& dd, int y, int m, int d); // check for valid date and initialize

void add_day(Date&, int n);         // increase the Date by n days
// …

init_day(my_birthday, 12, 30, 1950); // run time error: no day 1950 in month 30
```

Structs




```
// simple Date

//          guarantee initialization with constructor
//          provide some notational convenience
struct Date {
   int y,m,d;                    // year, month, day
   Date(int y, int m, int d);  // constructor: check for valid date and initialize
   void add_day(int n);          // increase the Date by n days
};

// …
Date my_birthday;                  // error: my_birthday not initialized
Date my_birthday(12, 30, 1950);  // oops! Runtime error
Date my_day(1950, 12, 30);         // ok
my_day.add_day(2);                 // January 1, 1951
my_day.m = 14;                     // ouch! (now my_day is a bad date)
```

Classes

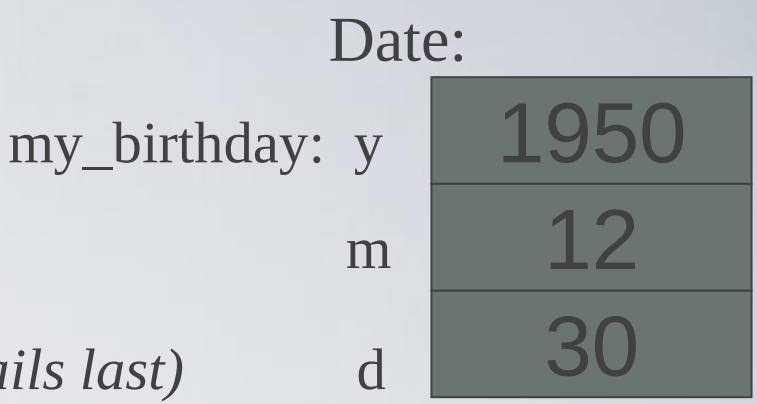



```
// simple Date (control access)

class Date {
   int y,m,d;        // year, month, day
public:
   Date(int y, int m, int d);  // constructor: check for valid date and initialize

   // access functions:
   void add_day(int n);        // increase the Date by n days
   int month() { return m; }
   int day() { return d; }
   int year() { return y; }
};

// …
Date my_birthday(1950, 12, 30);          // ok
cout << my_birthday.month() << endl;     // we can read
my_birthday.m = 14;                      // error: Date::m is private
```

Classes

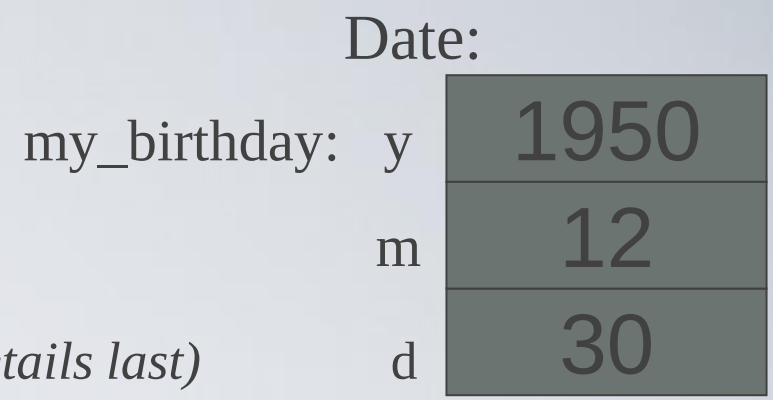
- The notion of a “valid Date” is an important special case of the idea of a valid value
- We try to design our types so that values are guaranteed to be valid
  - Or we have to check for validity all the time
- A rule for what constitutes a valid value is called an “invariant”
  - The invariant for Date (“Date must represent a date in the past, present, or future”) is unusually hard to state precisely
    - Remember February 28, leap years, etc.
- If we can’t think of a good invariant, we are probably dealing with plain data
  - If so, use a struct
  - Try hard to think of good invariants for your classes
    - that saves you from poor buggy code

Classes




```
// simple Date (some people prefer implementation details last)
class Date {
public:
   Date(int y, int m, int d);  // constructor: check for valid date and initialize
   void add_day(int n);        // increase the Date by n days
   int month();
   // ...
private:
   int y,m,d;      // year, month, day
};

Date::Date(int yy, int mm, int dd)                  // definition; note :: "member of"
   :y(yy), m(mm), d(dd) { /* ... */ };       // note: member initializers
void Date::add_day(int n) { /* ... */ };  // definition
```

Classes




```
// simple Date (some people prefer implementation details last)
class Date {
public:
   Date(int y, int m, int d);  // constructor: check for valid date and initialize
   void add_day(int n);        // increase the Date by n days
   int month();
   // …
private:
   int y,m,d;        // year, month, day
};


int month() {  return m; }    // error: forgot Date::
                              // this month() will be seen as a global function
                              //  not the member function, can't access members

int Date::season() { /* ... */ }        // error: no member called season
```

Classes

```
// simple Date (what can we do in case of an invalid date?)
class Date {
public:
   class Invalid { };                  // to be used as exception
   Date(int y, int m, int d);          // check for valid date and initialize
   // …
private:
   int y,m,d;                          // year, month, day
   bool check(int y, int m, int d);    // is (y,m,d) a valid date?
};

Date:: Date(int yy, int mm, int dd)
   : y(yy), m(mm), d(dd)                        // initialize data members
{
   if (!check(y,m,d)) throw Invalid();          // check for validity
}
```

Classes

- Why bother with the public/private distinction?
- Why not make everything public?
  - To provide a clean interface
    - Data and messy functions can be made private
  - To maintain an invariant
    - Only a fixed set of functions can access the data
  - To ease debugging
    - Only a fixed set of functions can access the data
    - (known as the “round up the usual suspects” technique)
  - To allow a change of representation
    - You need only to change a fixed set of functions
    - You don’t really know who is using a public member

Enumerations

- An **enum** (enumeration) is a very simple user-defined type, specifying its set of values (its enumerators)
- For example:

```
enum Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month m = feb;
m = 7;                      // error: can't assign int to Month
int n = m;                  // ok: we can get the numeric value of a Month
Month mm = Month(7);        // convert int to Month (unchecked)
```

Enumerations

- Simple list of constants:

```
enum { red, green };          // the enum { } doesn't define a scope
int a = red;                  // red is available here
enum { red, blue, purple };   // error: red defined twice
```

- Type with list of constants

```
enum Color { red, green, blue, /* ... */  };
enum Month { jan, feb, mar, /* ... */ };

Month m1 = jan;
Month m2 = red;      // error red isn't a Month
Month m3 = 7;        // error 7 isn't a Month
int i = m1;          // ok: an enumerator is converted to its value, i==0
```

Enumerations – Values

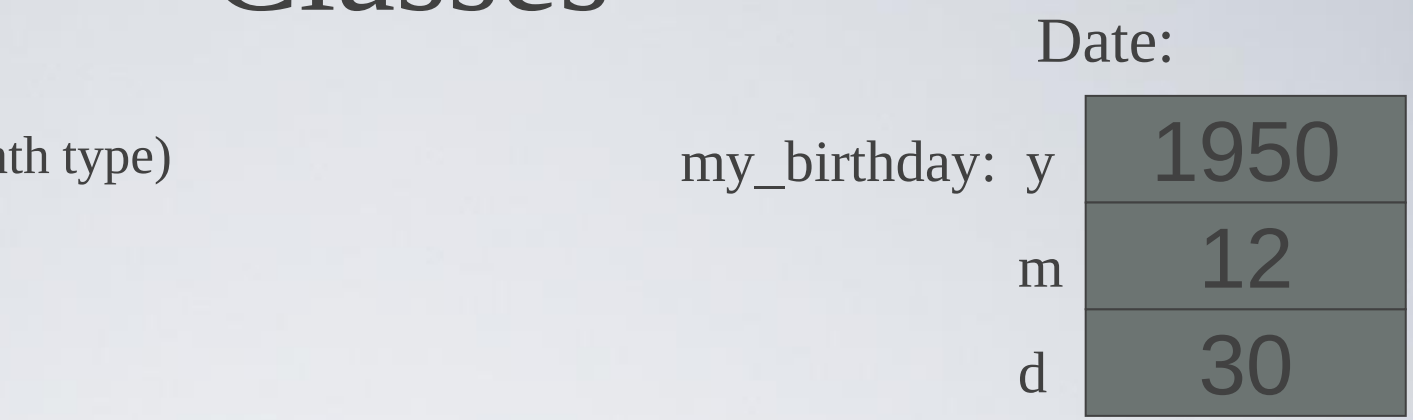
- By default

```
// the first enumerator  has the value  0,
// the next enumerator has the value "one plus the value of the
// enumerator before it"
enum { horse, pig, chicken };          // horse==0, pig==1, chicken==2
```

- You can control numbering

```
enum { jan=1, feb, march /* ... */ };          // feb==2, march==3
enum stream_state { good=1, fail=2, bad=4, eof=8 };
int flags = fail+eof;                  // flags==10
stream_state s = flags; // error: can't assign an int to a stream_state
stream_state s2 = stream_state(flags);   // explicit conversion (be careful!)
```

Classes




```
// simple Date (use Month type)
class Date {
public:
   enum Month {
         jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
   };
   Date(int y, Month m, int d);  // check for valid date and initialize
   // …
private:
   int y;              // year
   Month m;
   int d;  // day
};

Date my_birthday(1950, 30, Date::dec);    // error: 2nd argument not a Month
Date my_birthday(1950, Date::dec, 30);    // ok
```

Const

```
class Date {
public:
   // ...
   int day() const { return d; }       // const member: can't modify
   void add_day(int n);                // non-const member: can modify
   // ...
};

Date d(2000, Date::jan, 20);
const Date cd(2001, Date::feb, 21);

cout << d.day() << " – " << cd.day() << endl;        // ok
d.add_day(1);     // ok
cd.add_day(1);    // error: cd is a const
```

Const

```
//
Date d(2004, Date::jan, 7);              // a variable
const Date d2(2004, Date::feb, 28);      // a constant
d2 = d;          // error: d2 is const
d2.add(1);       // error d2 is const
d = d2;          // fine
d.add(1);        // fine

d2.f();  // should work if and only if f() doesn't modify d2
         // how do we achieve that? (say that's what we want, of course)
```

Const member functions

```
// Distinguish between functions that can modify (mutate) objects
// and those that cannot ("const member functions")
class Date {
public:
    // …
    int day() const; // get (a copy of) the day
    // …
    void add_day(int n);      // move the date n days forward
    // …
};

const Date dx(2008, Month::nov, 4);
int d = dx.day();   // fine
dx.add_day(4);     // error: can't modify constant (immutable) date
```

Classes

- What makes a good interface?
  - Minimal
    - As small as possible
  - Complete
    - And no smaller
  - Type safe
    - Beware of confusing argument orders
  - Const correct

Classes

- Essential operations
  - Default constructor (defaults to: nothing)
    - No default if any other constructor is declared
  - Copy constructor (defaults to: copy the member)
  - Copy assignment (defaults to: copy the members)
  - Destructor (defaults to: nothing)
- For example

```
Date d;          // error: no default constructor
Date d2 = d;     // ok: copy initialized (copy the elements)
d = d2;          // ok copy assignment (copy the elements)
```

Interfaces and “helper functions”

- Keep a class interface (the set of public functions) minimal
  - Simplifies understanding
  - Simplifies debugging
  - Simplifies maintenance
- When we keep the class interface simple and minimal, we need extra “helper functions” outside the class (non-member functions)
  - E.g. == (equality) , != (inequality)
  - **next_weekday(), next_Sunday()**

Helper functions

```
Date next_Sunday(const Date& d)
{
   // access d using d.day(), d.month(), and d.year()
   // make new Date to return
}

Date next_weekday(const Date& d) { /* … */ }

bool operator==(const Date& a, const Date& b)
{
   return a.year()==b.year()
          && a.month()==b.month()
          && a.day()==b.day();
}

bool operator!=(const Date& a, const Date& b) { return !(a==b); }
```

Operator overloading

- You can define almost all C++ operators for a class or enumeration operands
  - that's often called "operator overloading"

```
enum Month {
   jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month operator++(Month& m)        // prefix increment operator
{
   m = (m==dec) ? jan : Month(m+1);          // "wrap around"
   return m;
}

Month m = nov;
++m;          // m becomes dec
++m;          // m becomes jan
```

Operator overloading

- You can define only existing operators
  - *E.g.*, + - * / % [] () ^ ! & < <= > >=
- You can define operators only with their conventional number of operands
  - *E.g.*, no unary <= (less than or equal) and no binary **!** (not)
- An overloaded operator must have at least one user-defined type as operand
  - **int operator+(int,int);** // error: you can't overload built-in +
  - **Vector operator+(const Vector&, const Vector &);** // ok
- Advice (not language rule):
  - Overload operators only with their conventional meaning
  - + should be addition, * be multiplication, [] be access, () be call, etc.
- Advice (not language rule):
  - Don't overload unless you really have to

Date class

```
//
// This is example code from Chapter 9.8 "The Date class" of
// "Programming -- Principles and Practice Using C++" by Bjarne Stroustrup
//

#include <iostream>

using namespace std;

//------------------------------------------------------------------------------

namespace Chrono {

//------------------------------------------------------------------------------

class Date {
public:
    enum Month {
        jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
    };

    class Invalid { };                  // to throw as exception

    Date(int y, Month m, int d);        // check for valid date and initialize
    Date();                             // default constructor
    // the default copy operations are fine

    // non-modifying operations:
    int   day()   const { return d; }
    Month month() const { return m; }
    int   year()  const { return y; }

    // modifying operations:
    void add_day(int n);
    void add_month(int n);
    void add_year(int n);
private:
    int   y;
    Month m;
    int   d;
};

//------------------------------------------------------------------------------

bool is_date(int y, Date::Month m, int d); // true for valid date

//------------------------------------------------------------------------------

bool leapyear(int y);                   // true if y is a leap year

//------------------------------------------------------------------------------

bool operator==(const Date& a, const Date& b);
bool operator!=(const Date& a, const Date& b);

//------------------------------------------------------------------------------

ostream& operator<<(ostream& os, const Date& d);
istream& operator>>(istream& is, Date& dd);

//------------------------------------------------------------------------------

} // Chrono
```

```
int main()
try
{
    Chrono::Date holiday(1978, Chrono::Date::jul, 4); // initialization
    Chrono::Date d2 = Chrono::next_Sunday(holiday);
    Chrono::Day  d  = day_of_week(d2);
    cout << "holiday is " << holiday << " d2 is " << d2 << endl;
    return holiday != d2;
}
catch (Chrono::Date::Invalid&) {
    cerr << "error: Invalid date\n";
    return 1;
}
catch (...) {
    cerr << "Oops: unknown exception!\n";
    return 2;
}
```

```
#include "Chrono.h"

namespace Chrono {

// member function definitions:

//------------------------------------------------------------------------------

Date::Date(int yy, Month mm, int dd)
    : y(yy), m(mm), d(dd)
{
    if (!is_date(yy,mm,dd)) throw Invalid();
}

//------------------------------------------------------------------------------

const Date& default_date()
{
    static const Date dd(2001,Date::jan,1); // start of 21st century
    return dd;
}

//------------------------------------------------------------------------------

Date::Date()
    :y(default_date().year()),
     m(default_date().month()),
     d(default_date().day())
{
}

//------------------------------------------------------------------------------

void Date:: add_day(int n)
{
    // ...
}

//------------------------------------------------------------------------------

void Date::add_month(int n)
{
    // ...
}

//------------------------------------------------------------------------------

void Date::add_year(int n)
{
    if (m==feb && d==29 && !leapyear(y+n)) { // beware of leap years!
        m = mar;        // use March 1 instead of February 29
        d = 1;
    }
    y+=n;
}

//------------------------------------------------------------------------------
```

```
bool operator==(const Date& a, const Date& b)
{
    return a.year()==b.year()
        && a.month()==b.month()
        && a.day()==b.day();
}

//------------------------------------------------------------------------------

bool operator!=(const Date& a, const Date& b)
{
    return !(a==b);
}

//------------------------------------------------------------------------------

ostream& operator<<(ostream& os, const Date& d)
{
    return os << '(' << d.year()
              << ',' << d.month()
              << ',' << d.day()
              << ')';
}

//------------------------------------------------------------------------------

istream& operator>>(istream& is, Date& dd)
{
    int y, m, d;
    char ch1, ch2, ch3, ch4;
    is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;
    if (!is) return is;
    if (ch1!='(' || ch2!=',' || ch3!=',' || ch4!=')') { // oops: format error
        is.clear(ios_base::failbit);                    // set the fail bit
        return is;
    }
    dd = Date(y,Date::Month(m),d);     // update dd
    return is;
}

//------------------------------------------------------------------------------

enum Day {
    sunday, monday, tuesday, wednesday, thursday, friday, saturday
};

//------------------------------------------------------------------------------

Day day_of_week(const Date& d)
{
    // ...
    return sunday;
}

//------------------------------------------------------------------------------

Date next_Sunday(const Date& d)
{
    // ...
    return d;
}

//------------------------------------------------------------------------------

Date next_weekday(const Date& d)
{
    // ...
    return d;
}
```

```
// helper functions:

bool is_date(int y, Date::Month  m, int d)
{
    // assume that y is valid

    if (d<=0) return false;             // d must be positive

    int days_in_month = 31;             // most months have 31 days

    switch (m) {
case Date::feb:                         // the length of February varies
    days_in_month = (leapyear(y))?29:28;
    break;
case Date::apr: case Date::jun: case Date::sep: case Date::nov:
    days_in_month = 30;                 // the rest have 30 days
    break;
    }

    if (days_in_month<d) return false;

    return true;
}

//------------------------------------------------------------------------

bool leapyear(int y)
{
    // See exercise ???
    return false;
}
```

Useful C++ headers: ctype

cctype (ctype.h)

Character handling functions

This header declares a set of functions to classify and transform individual characters.

All these functions take as parameter the `int` equivalent of one character and return an `int`, that can either be another character or a value representing a boolean value: an `int` value of `0` means false, and an `int` value different from `0` represents true.

There are two sets of functions:

First a set of classifying functions that check whether the character passed as parameter belongs to a certain category. These are:

isalnum	Check if character is alphanumeric (function)
isalpha	Check if character is alphabetic (function)
iscntrl	Check if character is a control character (function)
isdigit	Check if character is decimal digit (function)
isgraph	Check if character has graphical representation using locale (function template)
islower	Check if character is lowercase letter (function)
isprint	Check if character is printable (function)
ispunct	Check if character is a punctuation character (function)
isspace	Check if character is a white-space (function)
isupper	Check if character is uppercase letter (function)
isxdigit	Check if character is hexadecimal digit (function)

And secondly, two functions to convert between letter cases:

tolower	Convert uppercase letter to lowercase (function)
toupper	Convert lowercase letter to uppercase (function)

Useful C++ headers: iomanip

manipulator function

setw

<iomanip>

```
smanip setw ( int n );
```

Set field width

Sets the number of characters to be used as the *field width* for the next insertion operation.

Behaves as if a call to the stream's member ios_base::width with *n* as its argument was made.

The *field width* determines the minimum number of characters to be written in some output representations. If the standard width of the representation is shorter than the field width, the representation is padded with *fill characters* (see setfill) at a point determined by the format flag `adjustfield` (`left`, `right` or `internal`).

This manipulator is declared in header `<iomanip>`, along with the other parameterized manipulators: resetiosflags, setiosflags, setbase, setfill and setprecision. This header file declares the implementation-specific `smanip` type, plus any additional operator overload function needed to allow these manipulators to be inserted and extracted to/from streams with their parameters.

Parameters

n

Number of characters to be used as field width.

Return Value

Unspecified. This function should only be used as a stream manipulator.

Example

```
1  // setw example
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  int main () {
7    cout << setw (10);
8    cout << 77 << endl;
9    return 0;
10 }
```

This code uses `setw` to set the *field width* to 10 characters.

Useful C++ headers: stdexcept

header

stdexcept

Exception classes

This header defines a set of standard exceptions that both the library and programs can use to report common errors.

They are divided in two sets:

Logic errors:

logic_error	Logic error exception (class)
domain_error	Domain error exception (class)
invalid_argument	Invalid argument exception (class)
length_error	Length error exception (class)
out_of_range	Out-of-range exception (class)

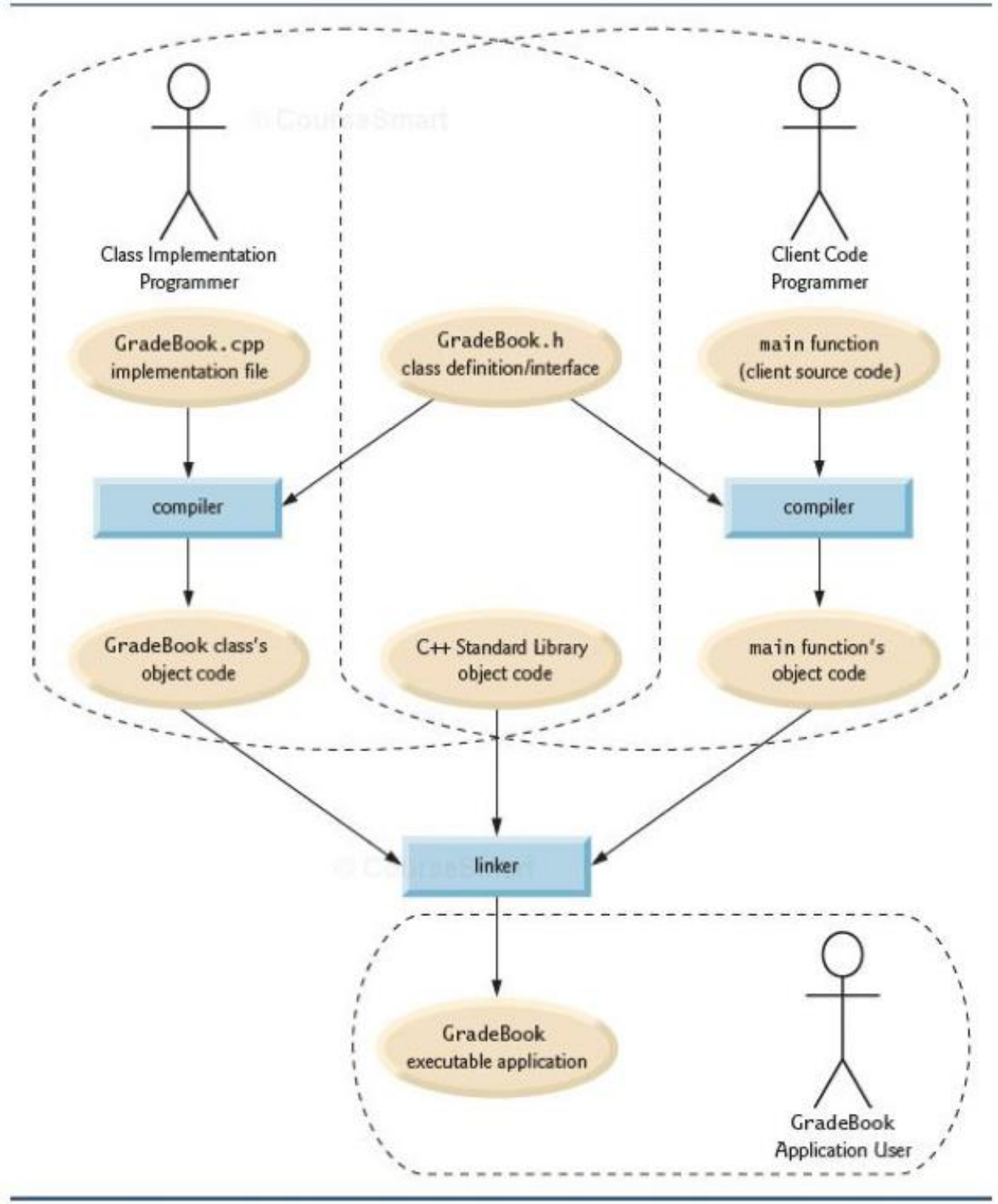
Runtime errors:

runtime_error	Runtime error exception (class)
range_error	Range error exception (class)
overflow_error	Overflow error exception (class)
underflow_error	Underflow error exception (class)

C++ compiler and linker with more than one files

//build executable out of two files PartA.cpp and GradeBook.cpp
// (single stage)
g++ -Wall PartA.cpp GradeBook.cpp -o PartAB.exe

//build an executable out of two input //files (two stages)
g++ -Wall -time -O3 -DNDEBUG -o PartA.o -c PartA.cpp
g++ -Wall -time -O3 -DNDEBUG -o GradeBook.o -c GradeBook.cpp
g++ -o PartAB.exe PartA.o GradeBook.o -lstdc++



Acknowledgements

Bjarne Stroustrup

Programming -- Principles and Practice Using C++

http://www.stroustrup.com/Programming/

Thank you!


BY


Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ
2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης