Programare Orientata pe Obiect
Lucrarea de laborator Nr. 5

Clase derivate. Mosteniri.

Într-o lucrare anterioara era ilustrat unul din principiile modelului obiect, si anume *ierarhizarea* ca modalitate de a ordona abstractizarile (tipurile abstracte de date - tipurile definite de utilizator - clasele). Ierarhiile pot sa denote relatii de tip sau relatii de agregare. Relatiile de agregare specifica compunerea unui obiect din mai multe obiecte mai simple. Altfel spus, ierarhia de agregare este o ierarhie în care se poate afirma despre un obiect ca este o parte a altui obiect, mai complex. Exemplul dat al unei astfel de descompuneri era cel al unui calculator. Acesta poate fi studiat prin descompunerea lui în subansamble componente: placa de baza, placa video, monitor, etc; la rândul ei, placa de baza este compusa din placheta de circuit imprimat, procesor, memorie, etc. Pe de alta parte, fiecare obiect poate fi încadrat într-o categorie (clasa, tip) mai larga, care contine mai multe obiecte care au proprietati comune. O discutie despre disk-drive-uri ar putea începe prin examinarea caracteristicile disk-drive-urilor în general. Ar putea fi studiate detaliile unui hard-drive si diferentele pe care le are un floppy-disk-drive. Aceasta ar implica mostenirea pentru ca multe dintre caracteristicile drive-urilor pot caracteriza drive-uri în general, dupa care apar diferente dependente de cazul particular (floppy, hard, etc.). Astfel, relatiile de tip sunt definite prin mostenirile între clase, prin care o clasa (clasa derivata) mosteneste structura sau comportarea definita în alta clasa (clasa de baza).

1. Clase derivate

Derivarea permite definirea într-un mod simplu, eficient si flexibil a unor clase noi prin adaugarea unor functionalitati claselor deja existente, fara sa fie necesara reprogramarea sau recompilarea acestora. *Clasele derivate* exprima relatii ierarhice între conceptele pe care acestea le reprezinta si asigura o interfata comuna pentru mai multe clase diferite. O clasa care asigura proprietati comune mai multor clase se defineste ca o *clasa de baza*. O *clasa derivata* mosteneste de la una sau mai multe clase de baza toate caracteristicile acestora, carora le adauga alte caracteristici noi, specifice ei. Astfel, posibilitatea de a include într-o clasa date descrise într-o alta clasa are în limbajele obiect orientate un suport mai eficient si mai simplu de utilizat decât includerea unui obiect din tipul dorit: derivarea claselor, care mostenesc date si functii membre de la clasa de baza.

În general, derivarea unei clase se specifica în felul urmator:

```
class  nume_clasa_derivata : specificator_acces  nume_clasa_baza {
      /*  corpul clasei  */
};
```

Specificatorul de acces poate fi unul din cuvintele-cheie: `public`, `private`, `protected`. Când o clasa este derivata dintr-o clasa de baza, clasa derivata mosteneste toti membrii clasei de baza (cu exceptia unora: constructorii, destructorii si functia operator de asignare). Tipul de acces din clasa derivata la membrii clasei de baza este dat de specificatorul de acces. Daca nu este indicat, specificatorul de acces implicit este `private`.

Când specificatorul de acces este `public`, toti membrii de tip `public` ai clasei de baza devin membrii de tip `public` ai clasei derivate; toti membrii `protected` ai clasei de baza devin membrii `protected` ai clasei derivate. Membrii `private` ai clasei de baza ramân `private` în clasa de baza si nu sunt accesibili membrilor clasei derivate. Aceasta restrictie de acces ar putea pare surprinzatoare, dar, daca s-ar permite accesul dintr-o clasa derivata la membrii `private` ai clasei de baza, notiunile de încapsulare si ascundere a datelor nu ar mai avea nici o semnificatie. Într-adevar, prin simpla derivare a unei clase, ar putea fi accesati toti membrii clasei respective. Exemplificare:

```
class Baza {
      int a;
protected:
      int b;
public:
      int c;
      void seta(int x){a = x; cout << "seta din baza\n";}
      void setb(int y){b = y; cout << "setb din baza\n";}
      void setc(int z){c = z; cout << "setc din baza\n";}
};
class Derivata : public Baza {
      int d;
public:
      void seta(int x) { a = x;} // eroare, a este private in clasa de baza
      void setb(int y) { b = y; cout << "setb din derivata\n";}
      void setc(int z) { c = z; cout << "setc din derivata\n";}
};
void main(){
      Derived obd;
      obd.a = 1;        // eroare, a este private in baza
      obd.seta(2);      // corect se apeleaza Baza::seta()
      obd.b = 3;        // eroare, b este protected
      obd.Baza::setb(5);//corect, Base::setb este public
      obd.setb(4);      // corect, Derivata::setb() este public
      obd.c = 6;        // corect, c este public
      obd.Baza::setc(7);//corect, Baza::setc() este public
      obd.setc(8);      // corect, Derivata::setc() este public
}
```

Daca se comenteaza liniile de program care provoaca erori si se executa functia `main()`, se obtin urmatoarele mesaje la consola:

```
seta din baza
setb din baza
setb din derivata
setc din baza
setc din derivata
```

Daca specificatorul de acces din declaratia clasei derivate este `protected`, atunci toti membrii de tip `public` si `protected` din clasa de baza devin membri `protected` în clasa derivata. Membrii de tip `private` în clasa de baza nu pot fi accesati din clasa derivata. Mai mult, în mostenirea `protected` a unei clase de baza nu mai pot fi accesati din afara clasei derivate nici unul dintre membrii clasei de baza. Exemplificare:

```
class Baza {
      int a;
protected:
      int b;
public:
      int c;
      void seta(int x){a = x; cout << "seta din baza\n";}
      void setb(int y){b = y; cout << "setb din baza\n";}
      void setc(int z){c = z; cout << "setc din baza\n";}
};
class Derivata : protected Baza {
      int d;
```

```
public:
      void seta(int x) { a = x;} // eroare, a este private in clasa de baza
      void setb(int y) { b = y; cout << "setb din derivata\n";}
      void setc(int z) { c = z; cout << "setc din derivata\n";}
};
void main(){
      Derived obd;
      obd.a = 1;              // eroare, a este private in baza
      obd.seta(2);            // eroare, Baza::seta()este protected
      obd.b = 3;              // eroare, b este protected
      obd.Baza::setb(5);  // eroare, Baza::setb() este protected
      obd.setb(4);         // corect, Derivata::setb() este public
      obd.c = 6;              // eroare, c este protected
      obd.Baza::setc(7);  // eroare, Baza::setc() este protected
      obd.setc(8);        // corect, Derivata::setc() este public
}
```

Astfel ca, daca se comenteaza toate liniile care produc erori, la executie se afiseaza urmatoarele rezultate:

```
setb din derivata
setc din derivata
```

În cazul specificatorului `private`, toti membrii de tip `public` si `protected` din clasa de baza devin membrii de tip `private` în clasa derivata si pot fi accesati numai din functiile membre si `friend` ale clasei de baza. Din nou trebuie reamintit faptul ca membrii de tip `private` în clasa de baza nu pot fi accesati din clasa derivata. Din punct de vedere al clasei derivate, mostenirea de tip `private` este echivalenta cu mostenirea de tip `protected`.

Exercitiul 1:

Reluati exemplul de mai sus modificând specificatorul de access din `protected` în `private` si urmariti mesajele de erori de compilare si de executie ale functiei `main()`.

Ceea ce diferentiaza mostenirea `private` de mostenirea `protected` este modul cum se transmit mai departe, într-o noua derivare, membrii clasei de baza. Toti membrii clasei de baza fiind mosteniti de tip `private`, o noua clasa derivata (care mosteneste indirect clasa de baza) nu va mai avea acces la nici unul din membrii clasei de baza.

Exercitiul 2:

Reluati exercitiul de mai sus, adaugând o noua clasa derivata din clasa Derivata. Accesati datele membre ale clasei de baza si ale clasei derivate în noua clasa si în functia `main()` si urmariti mesajele de erori de compilare si de executie ale functiei `main()`.

Specificatorii de acces anteriori se refera la toti membrii clasei de baza. Se pot modifica drepturile de acces la membrii mosteniti din clasa de baza prin declararea individuala a tipului de acces. Aceste declaratii nu pot modifica, însa, accesul la membrii de tip `private` ai clasei de baza. Exemplificare:

```
class Baza {
            int a;
      protected :
            int b;
      public :
```

```
            int c;
}
      class Derivata : private Baza {
            int d;
      public :
            B :: a;           // eroare, a nu poate fi declarat public
            B :: b;           // corect
            B :: c;           // corect

}
void main(){
      Derived obd;
      obd.a = 1;   // eroare, a este private
      obd.b = 5;   // corect, b este public
      obd.c = 6;   // corect, c este public
      obd.setb(5); // corect, Derived::setb este public
      obd.setc(8); // corect, Derived::setc este public
}
```

Variabilele `b` si `c` din clasa de baza au fost transformate în membrii publici ai clasei derivate prin declaratiile `B :: a; B :: b;` în zona de declaratii de tip `public` a clasei derivate.

2. Constructori si destructori în clasele derivate

Constructorii si destructorii sunt functii membre care nu se mostenesc. La crearea unei instante a unei clase derivate (obiect) se apeleaza implicit mai întâi constructorii clasei de baza si apoi constructorul clasei derivate. La distrugerea unui obiect al unei clase derivate se apeleaza implicit destructorii în ordine inversa: mai întâi destructorul clasei derivate, apoi destructorul clasei de baza. La instantierea unui obiect al unei clase derivate, dintre argumentele care se transmit constructorului acesteia, o parte sunt utilizate pentru initializarea datelor membre ale clasei derivate, iar alta parte sunt transmise constructorilor clasei de baza. Argumentele necesare pentru initializarea clasei de baza sunt plasate în definitia (nu în declaratia) constructorului clasei respective.

```
class  Baza {
            public: Baza(tip1 arg1, tip2 arg2,...,tipk argk);
                    Baza(Baza& r);
            };

class  Derivata:public Baza {
                  public: Derivata(tip1 arg1,...,tipk argk,...,tipn argn);
                          Derivata(Derivata& r);
                  };

Derivata::Derivata(tip1 arg1,..,tipk argk,..,tipn argn):Baza(arg1,.,argk){…}

Derivata::Derivata(Derivata& r):Baza(r) {…}
```

Daca nici în clasa de baza nici în clasa derivata nu au fost definiti constructori, compilatorul genereaza câte un constructor implicit pentru fiecare clasa si îi apeleaza în ordinea baza, apoi derivata. O clasa derivata trebuie sa aiba prevazut cel putin un constructor în cazul în care clasa de baza are un constructor care nu este implicit. Se observa ca argumentul contructorului de copiere al bazei este o referinta la un obiect derivate, lucru posibil deoarece o referinta la clasa derivata poate fi convertita în referinta la clasa de baza.

Exercitiul 3:

Definiti o clasa `Persoana` care sa contina vârsta unei persoane, ca data membra privata. Definiti constructorii necesari, destructorul si functia operator de scriere în stream. Din clasa `Persoana` derivati o noua clasa, `Student`, care sa contina datele membre private: numele studentului si facultatea (ca siruri de caractere terminate cu nul). Pentru clasa derivata definiti constructorii, destructorul si functia operator de scriere în stream. În toti constructorii si destructorii adaugati o instructiune care sa afiseze la consola un mesaj care sa precizeze ce operatie se executa. Creati în mod dinamic în memoria libera un obiect din clasa `Persoana` cu vârsta de 28 de ani si un obiect din clasa `Student` cu numele Popescu, vârsta 23 de ani, facultatea Electronica. Afisati la consola continutul acestor obiecte folosind functia operator de scriere în stream, apoi stergeti obiectele create.

3. Mostenirea multipla

Este posibila mostenirea din mai multe clase de baza:

```
class  nume_derivata : specificator_acces_1  nume_baza_1,
specificator_acces_2 nume_baza_2,..,specificator_acces_n  nume_baza_n
       {/*  corpul clasei  */};
```

Specificatorii de acces pot sa difere (pot fi oricare din `public`, `private`, `protected`).
O clasa de baza se numeste *baza directa* daca ea este mentionata în lista de clase de baza. Ordinea în care sunt apelati constructorii claselor de baza este cea din lista claselor de baza din declaratia clasei derivate. La distrugerea unui obiect al unei clase derivate se apeleaza implicit destructorii în ordine inversa: mai întâi destructorul clasei derivate, apoi constructorii claselor de baza, în ordinea inversa celei din lista din declaratie. La instantierea unui obiect al clase derivate, dintre argumentele care se transmit constructorului acesteia, o parte sunt utilizate pentru initializarea datelor membre ale clasei derivate, iar alta parte sunt transmise constructorilor claselor de baza.

```
class  Baza1 {
            public: Baza1(tip1 arg1, tip2 arg2,...,tipk argk);
            };
class  Baza2 {
            public: Baza2(tipk+1 argk+1, tipk+2 argk+2,...,tipk+p argk+p);
            };
class  Derivata:public Baza1,public Baza2 {
                  public: Derivata(tip1 arg1,...,tipk argk,...,tipn argn);
                        };
Derivata::Derivata(tip1 arg1,..,tipk argk,..,tipn argn):Baza1(arg1,.,argk),
Baza1(argk+1,.,argk+n),{. . . }
```

O clasa de baza se numeste *baza indirecta* daca nu este baza directa, dar este clasa de baza pentru una din clasele mentionate în lista claselor de baza. Într-o clasa derivata se mostenesc atât membrii bazelor directe cât si membrii claselor indirecte si, de asemenea, se pastreaza mecanismul de redefinire si ascundere a membrilor redefiniti.

```
class Baza_directa{/* . . . */};
class Baza_indirecta {/* . . . */};
class Derivata1 : public Baza_indirecta {/* . . . */};
class Derivata2 : public Baza_directa,public Derivata1 {/* . . . */};
```

Într-o mostenire este posibil ca o clasa sa fie mostenita indirect de mai multe ori, prin intermediul unor clase care mostenesc, fiecare în parte, clasa de baza. De exemplu:

```
class B { public : int x;}
class D1 : public B {/* . . . */};
class D2 : public B {/* . . . */};
class D3 : public D1 , public D2 {/* . . . */};
```

Un obiect din clasa `D3` va contine membrii clasei `B` de doua ori: o data prin clasa `D1` si o data prin clasa `D2`. În aceasta situatie, accesul la un membru al clase `B` (de exemplu variabila `x`), al unui obiect de tip `D3` este ambiguu si deci interzis:

```
D3 ob;
ob.x=2;//Eroare: D3::x poate fi în baza B clasei D1 sau in baza B a clase D2
```

Se pot elimina ambiguitatile si deci erorile prin specificarea clasei careia îi apartine variabila :

```
D3 ob;
ob.D1::x=2; // corect x din D1
ob.D2::x=3; // corect x din D2
```

O alta solutie pentru eliminarea ambiguitatilor în mostenirile multiple este de a impune crearea unei singure copii a clasei de baza în clasa derivata. Pentru aceasta este necesar ca acea clasa care ar putea produce copii multiple prin mostenire indirecta (clasa B de exemplu) sa fie declarata clasa de baza de tip *virtual* în clasele care o introduc în clasa cu mostenire multipla. De exemplu:

```
class B { public : int x;}
class D1 : virtual public B {/* . . . */};
class D2 : virtual public B {/* . . . */};
class D3 : public D1 , public D2 {/* . . . */};
```

O clasa de baza virtuala este mostenita o singura data si creaza o singura copie în clasa derivata. O clasa poate avea atât clase de baza virtuale, cât si nevirtuale, chiar de acelasi tip:

```
class L { public: int x; };
class A : virtual public L { /*    */ };
class B : virtual public L { /*    */ };
class C : public L { /*    */ };
class D : public A, public B, public C { /*    */ };
```

Clasa `D` mosteneste indirect clasa `L`: de doua ori ca o clasa de baza virtuala prin mostenirea din clasele `A` si `B` si înca o data direct, prin mostenirea clasei `C`. Un obiect din clasa `D` va contine doua copii a clasei `L`: o singura copie prin mostenirea de tip *virtual* prin intermediul claselor `A` si `B` si o alta copie prin mostenirea clasei `C`. Ambiguitatile care pot apare în astfel de situatii se rezolva prin calificarea membrilor cu numele clasei (domeniului) din care fac parte.

Exercitiul 4:

Definiti o clasa `Componente` care contine ca data membra un pointer la un sir de caractere, `nume`. Din aceasta clasa derivati doua clase, `Procesor`, cu data membra de tip întreg, `frecventa`, si `Monitor`, cu data membra de tip întreg, `diagonala`. Fiecare din cele doua clase derivate contine o functie de afisare a numelui si frecventei procesorului, respectiv a numelui si diagonalei monitorului. Definiti apoi o clasa `Calculator` ce contine ca date membre un obiect `Procesor` si un obiect `Monitor` si o functie de afisare a obiectelor continute. Definitii constructorii necesari si functiile de afisare si construiti în functia

`main()` un obiect de tip `Calculator` cu procesor Pentium la 800 MHz si monitor Samsung de 17 inch. Afisati datele acestui obiect folosind functia de afisare definita în clasa `Calculator`.

4. Un caz practic de mostenire multipla

Creati o aplicatie Win32ConsoleApplication, Empty Project. Includeti în proiect fisierele urmatoare:

//DATE.H

```
#ifndef DATE_H
#define DATE_H

class date {

protected:
   int month;
   int day;
   int year;
   static char out_string[25];
   static char format;
   int days_this_month(void);

public:
   date(void);
   int set_date(int in_month, int in_day, int in_year);
   int get_month(void) { return month; };
   int get_day(void)   { return day;   };
   int get_year(void)  { return year;  };
   void set_date_format(int format_in) { format = format_in; };
   char *get_date_string(void);
   char *get_month_string(void);
};

#endif
```

//DATE.CPP

```
#include <stdio.h>
#include <time.h>
#include "date.h"

char date::format;
char date::out_string[25];

date::date(void)
{
time_t time_date;
struct tm *current_date;
   time_date = time(NULL);
   current_date = localtime(&time_date);
   month = current_date->tm_mon + 1;
   day = current_date->tm_mday;
   year = current_date->tm_year + 1900;
   format = 1;
}
```

```
int date::set_date(int in_month, int in_day, int in_year)
{
int temp = 0;
int max_days;
      if (in_year < 1500) {
      year = 1500;
      temp = 1;
   } else {
      if (in_year > 2200) {
         year = 2200;
         temp = 1;
      } else
         year = in_year;
   }
   if(in_month < 1) {
      month = temp = 1;
   } else {
      if (in_month > 12) {
         month = 12;
         temp = 1;
      } else
         month = in_month;
   }
   max_days = days_this_month();
   if (in_day < 1) {
      day = temp = 1;
   } else {
      if (in_day > max_days) {
         day = max_days;
         temp = 1;
      } else
         day = in_day;
   }
   return temp;
}

static char *month_string[13] = {" ", "Jan", "Feb", "Mar", "Apr","May", "Jun",
                                 "Jul", "Aug","Sep", "Oct", "Nov", "Dec"};
char *date::get_month_string(void)
{
   return month_string[month];
}
char *date::get_date_string(void)
{
   switch (format) {
      case 2  : sprintf(out_string, "%02d/%02d/%02d",month, day, year - 1900);
                break;
      case 3  : sprintf(out_string, "%02d/%02d/%04d",month, day, year);
                break;
      case 4  : sprintf(out_string,"%d%s %04d",day,month_string[month], year);
                break;
      case 1  :
      default : sprintf(out_string,"%s%d,%04d",month_string[month],day, year);
                break;
   }
   return out_string;
}
```

```
int days[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int date::days_this_month(void)
{
   if (month != 2)
      return days[month];
   if (year % 4)
      return 28;
   if (year % 100)
      return 29;
   if (year % 400)
      return 28;
   return 29;
}
```

//TIME.H

```
#ifndef TIME_H
#define TIME_H

class time_of_day {

protected:
   int hour;                   // 0 through 23
   int minute;                 // 0 through 59
   int second;                 // 0 through 59
   static char format;         // Format to use for output
   static char out_string[25]; // Format output area

public:
         // Constructor - Set time to current time and format to 1
   time_of_day(void);
   time_of_day(int H) {hour = H; minute = 0; second = 0; };
   time_of_day(int H, int M) {hour = H; minute = M; second = 0; };
   time_of_day(int H, int M, int S) {hour = H; minute = M; second = S; };
         // Set the time to these input values
         //  return = 0 ---> data is valid
         //  return = 1 ---> something is out of range
   int set_time(void);
   int set_time(int hour_in);
   int set_time(int hour_in, int minute_in);
   int set_time(int hour_in, int minute_in, int second_in);
         // Select string output format
   void set_time_format(int format_in) { format = format_in; };
         // Return an ASCII-Z string depending on the stored format
         //   format = 1    13:23:12
         //   format = 2    13:23
         //   format = 3     1:23 PM
   char *get_time_string(void);
};

#endif
```

//TIME.CPP

```
#include <stdio.h>         // For the sprintf function
#include <time.h>          // For the time & localtime functions
#include "time.h"          // For the class header

char time_of_day::format;  // This defines the static data member
char time_of_day::out_string[25];  // This defines the string
         // Constructor - Set time to current time
         //               and format to 1

time_of_day::time_of_day(void)
{
time_t time_date;
struct tm *current_time;
   time_date = time(NULL);
   current_time = localtime(&time_date);
   hour     = current_time->tm_hour;
   minute   = current_time->tm_min;
   second   = current_time->tm_sec;
   format   = 1;
}
         // Set the time to these input values
         //  return = 0 ---> data is valid
         //  return = 1 ---> something out of range
int time_of_day::set_time(void) {return set_time(0, 0, 0); };
int time_of_day::set_time(int H)        {return set_time(H, 0, 0); };
int time_of_day::set_time(int H, int M) {return set_time(H, M, 0); };
int time_of_day::set_time(int hour_in, int minute_in, int second_in)
{
int error = 0;
   if (hour_in < 0) {
      hour_in = 0;
      error = 1;
   } else if (hour_in > 59) {
      hour_in = 59;
      error = 1;
   }
   hour = hour_in;
   if (minute_in < 0) {
      minute_in = 0;
      error = 1;
   } else if (minute_in > 59) {
      minute_in = 59;
      error = 1;
   }
   minute = minute_in;
   if (second_in < 0) {
      second_in = 0;
      error = 1;
   } else if (second_in > 59) {
      second_in = 59;
      error = 1;
   }
   second = second_in;
   return error;
}
```

```
         // Return an ASCII-Z string depending on the stored format
         //   format = 1    13:23:12
         //   format = 2    13:23
         //   format = 3     1:23 PM
char *time_of_day::get_time_string(void)
{
   switch (format) {
      case 2  : sprintf(out_string, "%2d:%02d", hour, minute);
                break;
      case 3  : if (hour == 0)
                   sprintf(out_string, "12:%02d AM", minute);
                else if (hour < 12)
                   sprintf(out_string, "%2d:%02d AM", hour, minute);
                else if (hour == 12)
                   sprintf(out_string, "12:%02d PM", minute);
                else
                   sprintf(out_string, "%2d:%02d PM",
                                              hour - 12, minute);
                break;
      case 1  : // Fall through to default so the default is also 1
      default : sprintf(out_string, "%2d:%02d:%02d",hour, minute, second);
                break;
   }
   return out_string;
}
```

//NEWDATE.H

```
//This class inherits the date class and adds one variableand one method to it
#ifndef NEWDATE_H
#define NEWDATE_H
#include "date.h"

class new_date : public date {

protected:
   int day_of_year;             // New member variable
public:
   int get_day_of_year(void);   // New method
};

#endif
```

//NEWDATE.CPP

```
#include "newdate.h"
extern int days[];
         // This routine ignores leap year for simplicity, and adds the days
         //in each month for all months less than the current month, then adds
         //the days in the current month up to today.
int new_date::get_day_of_year(void)
{
int index = 0;
   day_of_year = 0;
   while (index < month) day_of_year += days[index++];
   return (day_of_year += day);
}
```

//DATETIME.H

```
#ifndef DATETIME_H
#define DATETIME_H
#include "newdate.h"
#include "time.h"

class datetime : public new_date, public time_of_day {

public:
   datetime(void) { ; };        // Default to todays date and time
   datetime(int M, int D, int Y, int H, int Mn, int S) :
                    new_date(),              // Member initializer
                    time_of_day(H, Mn, S)    // Member initializer
                    { set_date(M, D, Y); };  // Constructor body
};

#endif
```

//USEDTTM.CPP

```
#include <iostream.h>
#include "datetime.h"

datetime now, birthday(10, 18, 1938,  1, 30, 17);
datetime       special( 2, 19, 1950, 13, 30,  0);

void main(void)
{
      cout << "Now = " << now.get_date_string() << " "
            << now.get_time_string() << " and is day "
            << now.get_day_of_year() << "\n";


      cout << "Birthday = " << birthday.get_date_string() << " "
            << birthday.get_time_string() << " and is day "
            << birthday.get_day_of_year() << "\n";

      cout << "Special = " << special.get_date_string() << " "
            << special.get_time_string() << " and is day "
            << special.get_day_of_year() << "\n";
}
```

Compilati. Rulati programul.