4. Supraîncărcarea operatorilor

Obiective

- Înțelegerea modului în care se pot redefini (supraîncărca) operatorii pentru a lucra cu noi tipuri de date
- Înțelegerea modului în care se pot converti obiectele de la o clasă la alta
- Înțelegerea situațiilor în care trebuie sau nu trebuie supraîncărcați operatorii
- Studierea unor clase care folosesc operatori supraîncărcați

- *Introducere*

În capitolele anterioare am introdus noțiunile necesare definirii claselor C++. Am văzut apoi că manipularea obiectelor se face prin trimiterea de mesaje obiectelor sub forma apelurilor de funcții membre. Pentru unele clase, în special cele care implementează operații matematice, înlocuirea operațiilor prin astfel de funcții membre este greoaie. Ar fi de dorit ca setul de operații C++ să poată fi folosit şi pentru tipurile abstracte de date. În acest capitol vom vedea cum putem permite operatorilor să lucreze cu obiecte ale claselor introduse de noi. Acest proces se numeşte *supraîncărcarea operatorilor*.

În limbajul C++, operatorii `+` şi `-` au diverse funcții dependente de context: operații aritmetice pentru valori întregi, reale, operații cu pointeri. Acesta este un exemplu de supraîncărcare a operatorilor.

Limbajul C++ permite programatorilor să supraîncarce majoritatea operatorilor pentru ca aceştia să poată fi folosiți în contextul unor noi clase. Unii operatori sunt supraîncărcați mai frecvent decât alții, cum ar fi de exemplu operatorul de asignare sau cei aritmetici de adunare sau scădere. Acțiunile implementate de operatorii supraîncărcați pot fi realizate la fel de bine şi prin apeluri explicite de funcții, însă folosirea notației cu operatori este mai clară şi mai intuitivă.

Vom vedea în acest capitol când trebuie să supraîncărcăm un operatori şi când nu trebuie să facem acest lucru. Vom arăta cum se supraîncarcă operatorii şi vom da câteva exemple complete.

- *Supraîncărcarea operatorilor – noțiuni fundamentale*

Aşa cum operatorii pot fi folosiți în limbajul C++ pentru tipurile de date predefinite, ei pot fi folosiți şi pentru tipurile de date introduse de programatori. Cu toate că nu se pot crea noi operatori, majoritatea celor existenți pot fi supraîncărcați ca să poată fi folosiți şi pentru noile clase. Aceasta este unul dintre cele mai puternice atribute ale limbajului C++. Chiar dacă supraîncărcarea operatorilor face programele mai clare decât folosirea unor funcții pentru aceleaşi operații, folosirea excesivă a acestei tehnici poate face uneori ca aplicațiile să fie criptice şi dificil de înțeles.

Operația de adunare `+` funcționează pentru variabile de tip `int`, `float`, `double` şi un număr de alte tipuri de dată predefinite deoarece operatorul `+` a fost supraîncărcat chiar în limbajul de programare C++.

Operatorii se supraîncarcă scriind o definiție de funcție obişnuită, singura excepție fiind aceea că numele său este unul special, format din cuvântul cheie operator urmat de simbolul operatorului care urmează să fie supraîncărcat. De

exemplu, numele de funcție `operator+` se foloseşte atunci când se supraîncarcă operatorul `+`.

Pentru ca un operator să poată lucra asupra obiectelor unei clase el trebuie să fie supraîncărcat pentru acea clasă. Această regulă are două excepții. Operatorul `=` poate fi folosit pentru orice clasă iar funcția sa implicită este de copiere membru cu membru. Această metodă de copiere a obiectelor nu este potrivită pentru clasele care au pointeri ca date membre şi, în acest caz, se preferă supraîncărcarea explicită a acestui operator. A doua excepție este operatorul adresă `&` poate fi de asemenea folosit pentru obiecte din orice clasă şi returnează adresa de memorie a obiectului. Dacă programatorul doreşte, poate supraîncărca şi acest operator.

Scopul supraîncărcării operatorilor este acela de a putea scrie expresii concise pentru obiecte care fac parte din clase definite de programatori în acelaşi fel în care se scriu pentru tipurile predefinite. Programatorul trebuie să decidă când este oportun sau nu să supraîncarce un operator pentru o operație. Trebuie evitate, însă, situațiile extreme, în care operația implementată de operatorul supraîncărcat nu corespunde din punct de vedere semantic operatorului. Exemple de astfel de greşeli de concepție ar fi supraîncărcarea operatorului `+` printr-o operație similară scăderii sau supraîncărcarea operatorului `/` ca să implementeze o înmulțire. Asemenea variante de supraîncărcare a operatorilor fac un program extrem de dificil de înțeles. Este recomandabilă, aşadar, folosirea intuitivă a operatorilor.

- *Restricții la supraîncărcarea operatorilor*

Cei mai mulți operatori din C++ pot supraîncărcați. Lista lor ese următoarea:

Operatorii care pot fi supraîncărcați							
`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`
`~`	`!`	`=`	`<`	`>`	`+=`	`-=`	`*=`
`/=`	`%=`	`^=`	`&=`	`|=`	`<<`	`>>`	`>>=`
`<<=`	`==`	`!=`	`<=`	`>=`	`&&`	`||`	`++`
`--`	`->*`	`,`	`->`	`[]`	`()`	`new`	`delete`
`new[]`	`delete[]`						

Operatorii care nu pot fi supraîncărcați sunt următorii:

Operatorii care nu pot fi supraîncărcați				
`.`	`.*`	`::`	`?:`	`sizeof`

Trebuie să precizăm faptul că precedența, aritatea (numărul de operanizi) şi asociativitatea operatorilor nu poate fi schimbată prin supraîncărcare.

Nu este posibil să creăm noi operatori, doar cei existenți putând fi supraîncărcați. De exemplu, nu putem introduce notația ** pentru exponențiere şi să folosim acest operator care există în limbaje de programare precum FORTRAN sau BASIC.

Operațiile realizate de operatorii tipurilor de date predefinite nu pot fi modificate. Programatorul nu poate, de exemplu, să schimbe modalitatea în care se adună doi întregi. Supraîncărcarea operatorilor este valabilă doar pentru tipuri de date definite de programator sau pentru operații care combină tipuri de date definite de programator cu tipuri de date predefinite.

- *Operatori ca funcții membre ale claselor şi operatori ca funcții `friend`*

Funcțiile operatori pot fi funcții membre sau funcții nemembre. De regulă, funcțiile nemembre care implementează operatori sunt declarate `friend`. Operatorii `()`, `[]`, `->` şi operatorii de asignare trebuie să fie funcții membre. Pentru ceilalți operatori, se poate opta şi pentru varianta în care operatorul nu este funcție membră.

Chiar dacă un operator este implementat ca funcție membră sau nu, el este folosit în acelaşi fel în expresii. Aşadar, care variantă este mai potrivită?

Atunci când un operator este implementat ca funcție membră, operandul din stânga operației, sau unicul operand atunci când operația este unară, trebuie să fie obiectul clasei sau o referință la un obiect al clasei din care face parte operatorul. Dacă operandul din partea stângă a operației trebuie să fie un obiect al altei clase sau dintr-un tip predefinit, atunci operatorul trebuie implementat ca funcție nemembră a clasei. Un operator declarat ca funcție nemembră trebuie declarat `friend` dacă trebuie să acceseze în mod direct membri private sau protected ai clasei. Dacă, însă, clasa are funcții *get* şi *set* pentru aceşti membri, funcția poate să nu fie declarată `friend`.

Un exemplu este cel al operatorilor << şi >> care au în stânga operației stream-uri de ieşire sau de intrare. Operatorul << trebuie să aibă un operand stâng de tip `ostream&`, de exemplu `cout` din expresia `cout << classObject`, de aceea trebuie declarat ca funcție nemembră. În mod similar, operatorul >> are un operand stând de tip `istream&`, de exemplu `cin` din expresia `cin >> classObject`, astfel că şi el trebuie supraîncărcat ca funcție nemembră.

Un alt motiv pentru care putem decide ca un operator să nu fie membru al clasei este dorința ca acel operator să fie comutativ. O operație de adunare dintre un obiect `number` de tip `long int` şi un obiect `bigInteger1` de tip `HugeInteger`, o clasă pentru manipularea întregilor mai mari decât valoare limitată de calculator, trebuie să permită comutativitatea. Pentru expresia `number + bigInteger1` operatorul trebuie supraîncărcat ca funcție nemembră. Pentru operația `bigInteger1 + number`, funcția `operator+` poate să fie implementată ca membră a clasei `HugeInteger`.

- *Supraîncărcarea operatorilor de inserare în stream şi de extragere din stream*

Limbajul C++ dă posiblitatea supraîncărcării celor doi operatori, de inserare in stream << şi extragere din stream >>, pentru o clasă definită de utilizator.

Clasa `PhoneNumber` din exemplul următor este un tip de dată care defineşte un număr de telefon prin prefix de 4 cifre şi număr de 6 cifre. Vom supraîncărca operatorii << şi >> ca să îi putem folosi pentru obiecte ale noii clase. Presupunem că numerele de telefon sunt introduse întotdeauna corect. Ca exercițiu, puteți completa programul ca să verifice corectitudinea datelor.

Exemplu

```
#include <iostream>

using std::cout;
using std::cin;
using std::endl;
using std::ostream;
```

```
using std::istream;

#include <iomanip>

using std::setw;

class PhoneNumber
{
  friend ostream& operator<<(ostream&, const PhoneNumber&);
  friend istream& operator>>(istream&, PhoneNumber&);
  private:
    char areaCode[5];//4 cifre si null
    char number[7];  //6 cifre si null
};

ostream& operator<<(ostream& out, const PhoneNumber& num)
{
  out << num.areaCode << "/" << num.number;
  return out;//pentru cout << a << b << c;
}

istream& operator>>(istream& in, PhoneNumber& num)
{
  in >> setw(5) >> num.areaCode;
  in.ignore(1);//ignora /
  in >> setw(7) >> num.number;
  return in;//pentru cin >> a >> b >> c;
}

int main()
{
  PhoneNumber phone;//creeaza obiectul phone
  cout << "Introduceti numarul de telefon"
       << " in format 0711/123456:\n";

  //cin >> phone invoca functie operator>>
  //prin apelul operator>>(cin, phone)
  cin >> phone;

  //cout << phone invoca functie operator<<
  //prin apelul operator<<(cout, phone)
  cout << "Numarul de telefon introdus este: "
       << phone << endl;

  return 0;
}
```

Rulând programul obținem următorul rezultat:

```
Introduceti numarul de telefon in format 0711/123456:
0268/478705
Numarul de telefon introdus este: 0268/478705
```

Funcția care implementează operatorul de extracție din stream `operator>>` primeşte ca argument o referință la un `istream` numită `in` şi o referință la `PhoneNumber` care se numeşte `num` şi returnează o referință la un `istream`. Funcția `operator>>` este folosită în program pentru a permite introducerea numerelor de telefon în format `0711/123456` în obiectele clasei `PhoneNumber`. Atunci când compilatorul întâlneşte expresia

```
cin >> phone
```

generează apelul de funcție

```
operator>>(cin, phone);
```

La acest apel, parametrul referință `in` devine un alias pentru `cin` şi parametrul referință `num` devine alias pentru `phone`. Funcția care implementează operatorul încarcă cele două părți ale numărului de telefon în datele membre `areaCode` şi `number` ale obiectului de tip `PhoneNumber`: referința `num` din funcția operator şi obiectul phone din `main`.

Funcția `operator>>` returnează referința `in` la `istream`, în exemplu aceasta fiind `cin`. Acest lucru permite cascadarea operațiilor de intrare a obiectelor `PhoneNumber` sau a obiectelor altor tipuri de date. De exemplu, două obiecte `PhoneNumber` ar putea fi introduse astfel:

```
cin >> phone1 >> phone2;
```

Prima dată se execută operația `cin >> phone1` prin apelul

```
operator>>(cin, phone1);
```

Acest apel returnează o referință la cin ca valoare a lui `cin >> phone1`, astfel că a doua porțiune a expresiei inițiale ar putea fi interpretată ca `cin >> phone2`. Acesta se realizează prin apelul

```
operator>>(cin, phone2);
```

Operatorul de inserare în stream primeşte ca argumente o referință la un obiect de tip `ostream`, în exemplul nostru referință la `out`, şi o referință numită `num` la un obiect de tip `PhoneNumber`. Funcția `operator<<` afişează obiectele de tip `PhoneNumber`. Când compilatorul întâlneşte expresia

```
cout << phone;
```

în `main`, compilatorul generează apelul

```
operator<<(cout, phone);
```

Funcțiile operator>> şi operator<< sunt declarate în clasa PhoneNumber ca funcții nemembre, friend. Aceşti operatori trebuie declarați nemembri pentru că obiectul din clasa PhoneNumber apare de fiecare dată ca operand în dreapta operatorului. Operandul care este obiect al clasei trebuie să fie în stânga operației pentru ca operatorul să poată fi implementat ca funcție membră a clasei.

Am optat pentru varianta în care referința la obiectul `PhoneNumber` din lista de parametri ai funcției `operator<<` este `const` pentru că nu trebuie modificat. Referința la obiectul `PhoneNumber` din lista de parametri ai funcției `operator>>` este non-const pentru că obiectul trebuie modificat de funcția de citire a datelor.

- *Supraîncărcarea operatorilor unari*

Un operator unar pentru o clasă poate fi implementat ca funcție membră fără argumente sau ca funcție nemembră cu un argument. Acest argument trebuie să fie un obiect al clasei sau o referință la un obiect al clasei.

Vom defini în acest capitol clasa `String` şi vom implementa pentru aceasta operatorul unar `!` prin care vom testa dacă un obiect este vid sau nu. Presupunând

că `s` este un obiect al clasei `String`, la întâlnirea expresiei !s compilatorul va genera apelul `s.operator!()`. Operandul `s` este obiectul clasei pentru care a fost invocat apelul funcției membre `operator!` a clasei `String`. Funcția este declarată în definiția clasei astfel:

```
class String
{
  public:
    bool operator!() const;
    ...
};
```

Acest operator unar poate fi implementat şi ca funcție nemembră cu un argument, în două feluri: fie cu un argument care este obiect al clasei, fie cu argument ca referință la obiect. Pentru un obiect `s` din clasa `String`, apelul `!s` este tratat prin apelul `operator!(s)` care invocă funcția nemembră friend a clasei `String` declarată astfel:

```
class String
{
    friend bool operator!(const String&);
    ...
};
```

Ca regulă practică în alegerea uneia dintre cele două variante, trebuie evitată folosirea funcțiilor nemembre `friend` dacă folosirea lor nu este absolut necesară.

- ***Supraîncărcarea operatorilor binari***

Un operator binar pentru o clasă poate fi implementat ca funcție membră cu un argument sau ca funcție nemembră cu două argumente dintre care unul este un obiect al clasei sau o referință la un obiect al clasei.

Vom implementa operatorul `+=` pentru concatenarea a două obiecte de tip String. Când optăm pentru varianta implementării acestui operator ca funcție membră cu un singur argument, presupunând că `y` şi `z` sunt obiecte ale clasei `String`, `y += z` este tratată ca şi cum am fi scris `y.operator+=(z)`. Este invocată funcția membră `operator+=` declarată astfel:

```
class String
{
  public:
    const String& operator+=(const String&);
    ...
};
```

Dacă `+=` este implementat ca operator binar, operația `y += z` este tratată ca şi cum în program am fi avut instrucțiunea `operator+=(y, z)`, invocându-se funcția `friend` nemembră `operator+=` declarată astfel:

```
class String
{
    friend const String& operator+=(String&,
                                     const String&);
    ...
};
```

- *Conversii între tipuri de date*

Majoritatea programelor procesează informații care sunt stocate sub forma datelor de diverse tipuri. Adeseori este necesară conversia datelor de la un tip la altul. Acest lucru se întâmplă în asignări, calcule, transmiterea parametrilor către funcții, returnarea valorilor de către funcții. Compilatorul cunoaşte modul în care se fac aceste conversii între date care au tipuri predefinite.

Pentru tipurile de dată definite de programator, compilatorul nu cunoaşte regulile după care trebuie să facă aceste conversii. Programatorul clasei trebuie să specifice aceste reguli. Conversiile pot fi realizate prin *constructorii de conversie*, constructori cu un singur argument care convertesc obiecte de alte tipuri în obiecte ale clasei. Clasa `String` va include un constructor care converteşte valori `char*` la obiecte de tip `String`.

Un *operator de conversie*, numit şi *operator cast*, poate fi folosit pentru conversia unui obiect al unei clase în obiect al altei clase sau într-un tip predefinit. Un asemenea operator trebuie să fie declarat ca funcție membră în clasă, el neputând să fie declarat `friend`.

Prototipul de funcție

```
A::operator char*() const;
```

declară o funcție cast supraîncărcată prin care se pot crea obiecte temporare de tip `char*` dintr-un obiect de tip `A`. O funcție operator cast supraîncărcată nu specifică niciun tip al valorii returnate, tipul returnat fiind cel la care este convertit obiectul.

Pentru `s` un obiect al clasei `A`, atunci când compilatorul întâlneşte expresia `(char*) s` generează apelul `s.operator char*()`. Operandul `s` este obiectul `s` pentru care este invocată funcția membră `operator char*`.

Una dintre facilitățile operatorilor de cast şi a operatorilor de conversie este aceea că, de fiecare dată când este necesar, compilatorul apelează aceste funcții pentru a crea obiecte temporare. De exemplu, dacă un obiect `s` din clasa `String` apare într-un program într-un loc în care trebuie să apară de fapt o dată de tip `char*`, ca în instrucțiunea

```
cout << s;
```

compilatorul apelează funcția operator cast supraîncărcată operator `char*` pentru a converti obiectul la `char*`. Având acest operator de conversie în clasa `String`, operatorul de inserare în stream nu mai trebuie supraîncărcat pentru afişarea obiectelor de tip `String`.

- *Studiu de caz: clasa `String`*

Clasa `string` este parte a bibliotecii standard C++. În exemplul următor vom defini o nouă clasă `String` pentru a prezenta supraîncărcarea operatorilor. Fişierul `string1.h` conține definiția clasei `String`, fişierul `string1.cpp` conține definițiile funcțiilor membre, iar `test_string1.cpp` este un program care foloseşte obiecte din clasa `String`.

Exemplu

string1.h

```
#ifndef STRING1_H
#define STRING1_H
#include <iostream>
using std::ostream;
using std::istream;
```

```
class String
{
  friend ostream& operator<<(ostream&, const String&);
  friend istream& operator>>(istream&, String&);

  public:
    //constructor implicit/de conversie
    String(const char* = "");
    String(const String&);//constructor de copiere
    ~String();//destructor
    const String& operator=(const String&);//asignare
    const String& operator+=(const String&);//concatenare
    bool operator!() const;//String este gol?
    bool operator==(const String&) const;//test s1==s2
    bool operator<(const String&) const;//test s1<s2

    //test s1!=s2
    bool operator!=(const String& right) const
      { return !( *this == right ); }

    //test s1>s2
    bool operator>(const String& right) const
      { return right < *this; }

    //test s1<=s2
    bool operator<=(const String& right) const
      { return !( right < *this ); }

    //test s1>=s2
    bool operator>=(const String& right) const
      { return !( *this < right ); }

    char& operator[](int);//indexare
    const char& operator[](int) const;//indexare
    String operator()(int, int);//returneaza un substring
    int getLength() const;//lungimea stringului
  private:
    int length;//lungimea stringului
    char* sPtr;//pointer la inceputul stringului
    void setString(const char*);//functie utilitara
};
#endif
```

string1.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;
```

```
#include <cstring>
#include <cassert>
#include"string1.h"
//Conversie char*->String
String::String(const char* s) : length(strlen(s))
{
  cout << "Constructor de conversie: " << s << '\n';
  setString(s);//apelul functiei utilitare
}
//Constructor de copiere
String::String(const String& copy) : length(copy.length)
{
  cout << "Constructor de copiere: " << copy.sPtr << '\n';
  setString(copy.sPtr);//apelul functiei utilitare
}
//Destructor
String::~String()
{
  cout << "Destructor: " << sPtr << '\n';
  delete[] sPtr;//eliberarea zonei de memorie
}
//operatorul =; se evita autoasignarea
const String& String::operator=(const String& right)
{
  cout << "operator= apelat\n";
  if(&right != this)//evitarea autoasignarii
  {
    delete[] sPtr;//eliberarea memoriei
    length = right.length;//lungimea noului String
    setString(right.sPtr);//functia utilitara
  }
  else
    cout << "Nu se poate asigna un string lui insusi\n";
  return *this;//permite cascadarea asignarilor
}
//concateneaza operandul drept cu obiectul this
//rezultatul este stocat in this
const String& String::operator+=(const String& right)
{
  char* tempPtr = sPtr;//salveaza adresa din sPtr
  length += right.length;//noua lungime
  sPtr = new char[length+1];//aloca spatiu
  assert(sPtr!=0);//termina daca nu a fost alocata memoria
  strcpy(sPtr, tempPtr);//partea stanga a noului String
  strcat(sPtr, right.sPtr);//partea dreapta
  delete[] tempPtr;//elibereaza vechiul spatiu de memorie
  return *this;//permite cascadarea apelurilor
}
//String-ul este gol?
bool String::operator!() const
  {return length==0;}
```

```
//Acest string este egal cu cel din partea dreapta?
bool String::operator==(const String& right) const
  {return strcmp(sPtr, right.sPtr)==0;}
//Compararea stringurilor
bool String::operator<(const String& right) const
  {return strcmp(sPtr, right.sPtr)<0;}
//Intoarce o referinta la un caracter dintr-un String
//ca un lvalue
char& String::operator[](int subscript)
{
  //Testeaza daca indicele se incadreaza in domeniu
  assert(subscript>=0 && subscript<length);

  return sPtr[subscript];//creeaza lvalue
}
//Intoarce o referinta la un caracter dintr-un String
//ca un rvalue
const char& String::operator[](int subscript) const
{
  //Testeaza daca indicele se incadreaza in domeniu
  assert(subscript>=0 && subscript<length);

  return sPtr[subscript];//creeaza rvalue
}
//Intoarce un substring incepand cu index
//si de lungime subLength
String String::operator()(int index, int subLength)
{
  //index este in domeniu si subLength>=0 ?
  assert(index>=0 && index<length && subLength>=0);

  //determina lungimea substring-ului
  int len;
  if((subLength==0) >> (index+subLength>length))
    len = length-index;
  else
    len = subLength;

  //aloca un tablou temporar pentru substring si null
  char* tempPtr = new char[len+1];
  assert(tempPtr!=0);//verifica daca spatiul a fost alocat

  //copiaza substring-ul in array si adauga null
  strncpy(tempPtr, &sPtr[index], len);
  tempPtr[len]='\0';

  //Creeaza temporar un obiect String continand
  //substring-ul
  String tempString(tempPtr);
  delete[] tempPtr;//sterge tabloul temporar
```

```
   return tempString;//returneaza copia temporara a String-
ului
}
//Intoarce lungimea unui String
int String::getLength() const { return length; }
//Functie utilitara apelata de constructor si
//de operatorul de asignare
void String::setString(const char* string2)
{
  sPtr = new char[length+1];//aloca memoria
   assert(sPtr!=0);//termina programul daca memoria nu a
fost alocata
  strcpy(sPtr, string2);
}
//Operatorul de afisare
ostream& operator<<(ostream& output, const String& s)
{
  output << s.sPtr;
  return output;
}
//Operatorul de citire
istream& operator>>(istream& input, String& s)
{
  char temp[100];//pastreaza temporar intrarea
  input >> setw(100)>>temp;
  s = temp;//foloseste operatorul de asignare din clasa
  return input;//permite cascadarea citirilor
}
```

test_string1.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include"string1.h"

int main()
{
  String s1("happy"), s2(" birthday"), s3;
  //Testeaza operatorii relationali
  cout << "s1 este \"" << s1 << "\"; s2 is \"" << s2
       << "\"; s3 este \"" << s3 << '\"'
       << "\nRezultatele operatorilor relationali:"
       << "\ns2 == s1 este "
       << (s2==s1 ? "true" : "false")
       << "\ns2 != s1 este "
       << (s2!=s1 ? "true" : "false")
       << "\ns2 > s1 este "
        << (s2>s1 ? "true" : "false")
       << "\ns2 < s1 este "
       << (s2<s1 ? "true" : "false")
       << "\ns2 >= s1 este "
```

```
        << (s2>=s1 ? "true" : "false")
        << "\ns2 <= s1 este "
        << (s2<=s1 ? "true" : "false");
    //Testeaza operatorul !
    cout << "\n\nTestarea !s3:\n";
    if(!s3)
    {
      cout << "s3 este vid; asigneaza s1 lui s3;\n";
      s3 = s1;//testeaza operatorul de asignare
      cout << "s3 este \"" << s3 << "\"";
    }
    //Testeaza operatorul de concatenare
    cout << "\n\ns1 += s2 este s1 = ";
    s1 += s2;//testeaza operatorul de concatenare
    cout << s1;
    //Testeaza constructorul de conversie
    cout << "\n\ns1 += \" to you\" este\n";
    s1 += " to you";//testeaza constructorul de conversie
    cout << "s1 = " << s1 << "\n\n";
    //Testeaza operatorul ()
    cout << "Substring-ul din s1 care incepe la \n"
        << "locatia 0 si are 14 caractere, s1(0,14), "
        << "este:\n" << s1(0,14) << "\n\n";
    //Testeaza operatorul ()
    //cu optiunea "pana la sfarsitul sirului";
    //0 inseamna "pana la sfarsitul sirului"
    cout << "Substring-ul din s1 care incepe la \n"
        << "locatia 15, s1(15,0), este: "
        << s1(15,0) << "\n\n";
    //Testeaza constructorul de copiere
    String* s4Ptr = new String(s1);
    cout << "*s4Ptr = " << *s4Ptr << "\n\n";
    //Testeaza operatorul = cu optiunea de autoasignare
    cout << "asignarea *s4Ptr catre *s4Ptr\n";
    *s4Ptr = *s4Ptr;//testeaza asignarea
    cout << "s4Ptr = " << *s4Ptr << '\n';
    //Testeaza destructorul
    delete s4Ptr;
    //Testeaza operatorul de indexare
    s1[0] = 'H';
    s1[6] = 'B';
    cout << "\ns1 dupa s1[0] = 'H' si s1[6] = 'B' este: "
        << s1 << "\n\n";
    //Testeaza indicele in afara limitelor
    cout << "Intentie de a asigna 'd' lui s1[30]: " << endl;
    s1[30] = 'd';//Eroare: indice in afara limitelor

    system("pause");
    return 0;
}
```

Acest program testează operatorii supraîncărcați pentru clasa `String`. Rulând programul de mai sus, obținem următoarele rezultate:

```
Constructor de conversie: happy
Constructor de conversie:  birthday
Constructor de conversie:
s1 este "happy"; s2 is " birthday"; s3 este ""
Rezultatele operatorilor relationali:
s2 == s1 este false
s2 != s1 este true
s2 > s1 este false
s2 < s1 este true
s2 >= s1 este false
s2 <= s1 este true

Testarea !s3:
s3 este vid; asigneaza s1 lui s3;
operator= apelat
s3 este "happy"

s1 += s2 este s1 = happy birthday

s1 += " to you" este
Constructor de conversie:  to you
Destructor:  to you
s1 = happy birthday to you

Constructor de conversie: happy birthday
Substring-ul din s1 care incepe la
locatia 0 si are 14 caractere, s1(0,14), este:
happy birthday

Destructor: happy birthday
Constructor de conversie: to you
Substring-ul din s1 care incepe la
locatia 15, s1(15,0), este: to you

Destructor: to you
Constructor de copiere: happy birthday to you
*s4Ptr = happy birthday to you

asignarea *s4Ptr catre *s4Ptr
operator= apelat
Nu se poate asigna un string lui insusi
s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 dupa s1[0] = 'H' si s1[6] = 'B' este: Happy Birthday to
you

Intentie de a asigna 'd' lui s1[30]:
Assertion  failed:  subscript>=0  &&  subscript<length,  file
```

```
string1.cpp, line 70

Abnormal program termination
```

Clasa `String` declară doi constructori. Primul este un *constructor de conversie*:

```
String(const char* = "");
```

Orice *constructor cu un singur argument* poate fi interpretat ca un constructor de conversie. Acest constructor converteşte un şir de caractere de tip `char*` la un obiect de tip `String`. Apeluri ale acestui constructor sunt:

```
s1("happy");
s1 += " to you";
```

În primul caz se creează obiectul `s1` prin apelul constructorului cu un şir de caractere. În al doilea caz conversia este cerută de funcția membră `operator+=` care are ca parametru un obiect de tip `String`. Şirul `" to you"` este convertit la String înainte ca funcția membră operator să fie apelată.

Cel de-al doilea este un *constructor de copiere*:

```
String(const String&);
```

Acesta inițializează un obiect de tip `String` cu o copie a altui obiect de tip `String` care a fost deja definit. Constructorul de copiere este invocat oricând programul cere crearea unei copii a unui obiect, cum ar fi apelul prin valoare, când o funcție returnează un obiect prin valoare sau când un obiect este inițializat ca o copie a altui obiect din aceeaşi clasă. Un apel al acestui constructor este:

```
String* s4Ptr = new String(s1);
```

Acest apel este echivalent cu

```
String* s4Ptr(new String(s1));
```

În general, o declarație

```
String s4(s3);
```

este echivalentă cu declarația

```
String S4 = s3;
```

Exercițiu: Rulați acest program, parcurgeți fiecare linie de cod din funcția `main`, examinați fereastra de ieşire şi înțelegeți cum se foloseşte fiecare operator supraîncărcat.