Bitset, String, Excepții - plan

- bitset
- string
- Tratarea erorilor
- Excepții
- Componentele mecanismului
- Ierarhii
- Excepții în constructori
- Excepții în destructor
- Specificarea excepțiilor
- Excepții standard

bitset

- <bitset> definit în std
- un obiect de tipul bitset<n> este un tablou de n biți
- diferă de:
  - un vector<bool> prin faptul că are dimensiune fixă
  - un set prin faptul că biții sunt indexați de un întreg și nu de o valoare
- are operații specifice pentru biți

constructori

```
bitset<10> b1; // 10 biti 0
bitset<16> b2 = 0xaaaa; //1010101010101010
bitset<32> b3 = 0xaaaa;
      //10101010101010101010101010101010
bitset<10> b4(string("1010101010"));
bitset<10> b5(string("101101110111110", 4));
bitset<10> b6(string("101101110111110", 2, 8));
bitset<10> b7(string("badstring"));
//se genereaza exceptia invalid_argument
```

Operații

- Operatorii: [], &=, |=, ^=, <<=, >>=, ~, <<, >>, ==, !=
- Metode
  - set() : setează biții la 1
  - set(pos, val): setează bitul de pe pos la val(implicit 1)
  - reset(): setează biții la 0
  - reset(pos)
  - flip()
  - flip(pos)
  - count(): numără biții 1
  - size(): numărul de biți
  - test(pos): true dacă la pos este 1
  - any(): true dacă este măcar un bit 1
  - none(): true dacă nu este nici un bit 1
  - to_ulong(), to_string(): operații inverse constructorilor

Exemplu

```
bitset<4> first (string("1001"));
bitset<4> second (string("0011"));

cout << (first^=second) << endl;          // 1010 (XOR,assign)
cout << (first&=second) << endl;          // 0010 (AND,assign)
cout << (first|=second) << endl;          // 0011 (OR,assign)

cout << (first<<=2) << endl;              // 1100 (SHL,assign)
cout << (first>>=1) << endl;              // 0110 (SHR,assign)

cout << (~second) << endl;                // 1100 (NOT)
cout << (second<<1) << endl;              // 0110 (SHL)
cout << (second>>1) << endl;              // 0001 (SHR)

cout << (first==second) << endl;          // false (0110==0011)
cout << (first!=second) << endl;          // true  (0110!=0011)

cout << (first&second) << endl;           // 0010
cout << (first|second) << endl;           // 0111
cout << (first^second) << endl;           // 0101
```

Biblioteca **`string`**

- Un string este o secventa de caractere
- Biblioteca string contine :
  - Operatii pentru manipularea sirurilor
  - Asignare
  - Comparari
  - Adaugare caractere la sfarsit (append)
  - Concatenare
  - Cautare subsiruri
- Se poate folosi si stilul C: sirurile sunt tablouri de **`char`** iar bibliotecile corespunzatoare au prefixul c: **`cstring, cctype, cwtype, cstdlib`**

Structura char_traits

- **`char_traits`** este o specializare pentru E = **`char`** a template-ului:

```
template <class E>
struct char_traits{
   typedef E char_type;
   typedef T1 int_type;
   //…
   static int compare(const E *x,
                      const E *y, size_t n);
   static size_t length(const E *x);
   static E *copy(E *x, const E *y, size_t n);
   static int_type eof();
};
```

Clasa basic_string

```
template<class E,
  class T = char_traits<E>,
  class A = allocator<T> >
class basic_string { };

typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

Clasa basic_string - constructori

```
explicit basic_string();
basic_string(const basic_string& rhs);
basic_string(const basic_string& rhs,
      size_type pos, size_type n);
basic_string(const E *s, size_type n);
basic_string(const E *s);
basic_string(size_type n, E c);
basic_string(const_iterator first, const_iterator
   last);
```

Clasa basic_string - constructori

```
string s0
string s1 = "";
string s2("Facultatea de Informatica");
string s3(s2);
string s4(s2, 2, 3);
string s5(20, 'a');
string s6 = s2;
string s7(p+7, 3); // char* p;
string s8(v.begin(), v.end());
string s9 = 'a'; // eroare
string s10(5); // eroare
```

Clasa basic_string - iteratori

- Iterator: pointer “smart” la un element al unei secvențe, în stare să:
  - ofere elementul la care pointează (operatori * şi ->)
  - pointeze la următorul element (operatori ++)
  - verifce egalitatea a 2 iteratori (operatori ==)

```
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin()const;
reverse_iterator rend();
const_reverse_iterator rend() const;
```

Clasa basic_string - operatori

```
basic_string& operator=(const basic_string& rhs);
basic_string& operator=(const E *s);
basic_string& operator=(E c);
basic_string& operator+=(const basic_string& rhs);
basic_string& operator+=(const E *s);
basic_string& operator+=(E c);
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
```

- Acces cu verificarea domeniului de valori:

```
const_reference at(size_type pos) const;

reference at(size_type pos);
```

Clasa basic_string – Alte funcții

```
append() asign() insert()
compare() find() rfind()
copy()     swap()
find_first_of() find_first_not_of()
find_last_of() find_last_not_of()
replace() erase() substr()
size()     max_size() resize()
length() empty() capacity()
reserve()
```

Clasa basic_string – Alte funcții

```
s.copy(p, n, m); // p = s[m]…
s.insert(s.begin(), '');
s.insert(0, s1);
string s = "accdcde";
int i1 = s.find("cd");              //i1=2
int i2 = s.rfind("cd");             //i2=4
int i3 = s.find_first_of("cd");     //i3=1
int i4 = s.find_last_of("cd");      //i4=5
int i5 = s.find_first_not_of("cd"); //i5=0
```

Operatii I/O cu stringuri

```
template<class E, class T, class A>
basic_ostream<E, T>& operator>>( basic_ostream
  <E, T>& os, basic_string<E, T, A>& str);

template<class E, class T, class A>
basic_ostream<E, T>& operator<<( basic_ostream
  <E, T>& os,const basic_string<E, T, A>& str);
```

Operatii I/O cu stringuri

```
template<class E, class T, class A>
basic_istream<E, T>& getline( basic_istream <E,
  T>& is, basic_string<E, T, A>& str);

template<class E, class T, class A>
basic_istream<E, T>& getline( basic_istream <E,
  T>& is, basic_string<E, T, A>& str, E delim);
```

Tratarea erorilor

- Program = module separate, modulele pot proveni din diverse biblioteci
- Tratarea erorilor trebuie să fie separată în 2 părți distincte:
  - Raportarea erorilor ce nu pot fi tratate local
  - Tratarea erorilor detectate undeva, oriunde
- Mecanismul ”exception-handling” pune la dispoziție o alternativă la tehnicile tradiționale de tratare a erorilor (insuficiente, neelegante, predispuse la erori)

Politici de tratare a erorilor în C

- Terminarea programului
  - `abort()` (termină imediat programul)
  - `exit()` (golește memoriile buffer , închide fișierele deschise și termină programul)
- Returnarea unei valori ce reprezintă eroare – utilizată rar deoarece de fiecare dată trebuie un test pentru această valoare
- Returnarea unei valori legale și trecerea programului într-o stare ilegală
  - în header-ul `<errno.h>`, se definește o variabilă globală `errno` și funcția `perror()`
- Apelul unei funcții desemnată a fi invocată în cazul apariției unei erori

Excepții

- Mecanismul de tratare a excepțiilor în C++ este proiectat pentru:
  - a fi suport pentru tratarea erorilor
  - a fi suport pentru tratarea altor condiții excepționale ce pot apare
- Se pot trata doar excepțiile sincrone (erori I/O, domenii tablouri, etc.). Evenimentele asincrone(întreruperi, erori aritmetice etc.) necesită alte mecanisme

Excepții

- O excepție este un obiect al unei clase ce reprezintă evenimente excepționale
- Codul care detectează o eroare ”aruncă” un astfel de obiect (cu expresia **`throw`**)
- Codul care ”dorește” să trateze o excepție trebuie să conțină o clauză **catch**
- Efectul unei expresii throw este de a desfășura stiva (to unwind the stack) până se găsește o clauză catch corespunzătoare, într-o funcție care, direct sau indirect, a invocat funcția ce a aruncat excepția
  - căutarea produce distrugerea obiectelor locale

Componentele mecanismului

- Blocul **`try`** – conține o secvență de cod ce poate genera excepții:

```
try{
        // cod ce poate genera exceptii
}
```

- Blocul **`try`** este urmat de una sau mai multe *secvențe handler* care gestionează excepțiile:

```
try{
        //cod care poate genera excepţii
}
catch(tip1 id){
        //tratarea unei excepţii de tipul tip1
}
catch(tip2& id){
        // tratarea unei excepţii de tipul tip2&
}
catch(tip3* id){
        // tratarea unei excepţii de tipul tip3*
}
```

Componentele mecanismului

- Handler-ul **`catch`** se comportă ca o funcție ce “prinde” un obiect excepție fie prin valoare fie prin referință
- O listă `catch` seamănă într-un fel cu o instrucțiune `switch`; nu este nevoie de `break`
- Tipul excepției generată în **`try`** determină handler-ul ce o va trata; celelalte sunt inactive
- Există un handler ce prinde orice excepție (și care asigură desfășurarea stivei):

```
catch(…){/*…*/}
```

Structurarea excepțiilor în ierarhii

```
class Matherror{};
class Overflow:public Matherror{};
class Underflow:public Matherror{};
class Zerodivide:public Matherror{};

void f()
{
  try{
    //…
}
catch(Overflow){
  // tratare Overflow
}
catch(Matherror){
  //tratare orice exceptie care nu este Overflow
}
```

Structurarea excepțiilor în ierarhii

```
class Matherror{
public:
   virtual void debug_print()const{cerr<< "Math error" ;}
};
class Int_overflow:public Matherror{
public:
   Int_overflow(const char* p, int a, int b) {
        op = p; a1 = a; a2 = b;
   }
   virtual void debug_print()const{
        cerr<< "Depasire superioara: " << op << '(' << a1 <<
   ',' << a2 <<')' ;
   }
private:
   const char* op;
   int a1, a2;
};
```

Structurarea excepțiilor în ierarhii

```
int add(int x, int y)
{
   if( (x>0 && y>0 && x>INT_MAX-y)|| (x<0 && y<0 && x<INT_MIN-y))
        throw Int_overflow("+", x, y);
   return x+y;
}
void f()
{
   try{
        int i1 = add (100,200);
        int i2 = add(INT_MIN, -2);
        int i3 = add(INT_MAX, 2);
   }
   catch (Matherror& m){     //catch(Matherror m)
        m.debug_print();
   }
}
```

Tratarea excepțiilor

```
void f(){
   try{
        throw E();
   }
   catch(H){
        //
   }
}
```

- Handler-ul catch este invocat dacă:
  1. H are același tip cu E
  2. H este bază publică neambiguă a lui E
  3. H și E sunt tipuri pointeri și are loc 1 sau 2 pentru tipurile la care se referă
  4. H este referință și 1 sau 2 are loc pentru tipul la care se referă

Tratarea excepțiilor

- În general, o excepție este copiată când este aruncată încât handler-ul tratează o copie a excepției
  - O excepție este copiată de mai multe ori până este tratată
  - Nu se pot arunca excepții ce nu pot fi copiate
- Dacă o excepție nu poate fi tratată complet de către un handler, acesta poate decide aruncarea sa din nou (după ce a executat operațiile pe care le poate face)
  - `throw` fără operand

Tratarea excepțiilor

```
catch(Matherror){
    if(/*…*/){
          // tratare completa
          return;
    }
    else{
          //tratare partiala
          throw; // re-throw
    }
}
```

- Dacă este re-aruncată o excepție care nu există atunci se lansează funcția `std::terminate()`

Tratarea excepțiilor

- Ordinea în care este încercată potrivirea unui handler catch este cea în care acestea sunt scrise în cod
  - Este importantă această ordine în cazul unei ierarhii de excepții
- O funcție care alocă resurse (deschide fișiere, alocă memorie, etc.) este esențial să elibereze resursele în ordinea inversă alocării lor
  - Asta asigură comportarea acestora ca și modul de creare și distrugere a obiectelor locale ceea ce conferă robustețe aplicației

Excepții în constructori

- Gestionarea resurselor folosind obiecte locale este cunoscută sub numele de ”resource acquisition is initialization”
  - Se bazează pe proprietățile constructorilor și destructorilor și interacțiunea lor cu tratarea excepțiilor
- Dacă un constructor nu poate să completeze sarcina de a construi în întregime un obiect (datorită unor excepții), destructorul nu este apelat și astfel se ajunge la fenomenul ”memory leaks”
- Constructorul trebuie să fie proiectat astfel ca toate excepțiile să fie tratate corect, (execuția unei secvențe de cod pentru eliberarea resurselor şi repropagarea excepției)

Excepții în constructori

```
class X{
   int* p; void init();
public:
   X(int s) {p = new int[s]; init();}
   ~X() {delete []p;}
   //…
}
```

- Dacă `init` aruncă o excepție, constructorul nu-și termină treaba și destructorul nu se va invoca. O soluție:

```
class Y{
   vector<int> p; void init();
public:
   Y(int s) :p(s) {init();}
   //…
};
```

Excepții în constructori

- Biblioteca standard pune la dispoziție clasa template `auto_ptr` care oferă suport pentru tehnica ”resource acquisition is initialization”
  - În loc de pointer la un tip, `T* p,` se declară în clasă `auto_ptr<T> p`, el poate fi dereferențiat ca și pointerul la T iar obiectele pointate vor fi implicit șterse la ieșirea din bloc
  - `auto_ptr` este definită în fișierul `memory`

```
class X{
  auto_ptr<int> p; void init();
public:
  X(int s) {p = new int[s]; init();}
  ~X() {delete []p;}
  //…
}
```

Excepții în destructor

- Un destructor este invocat:
  - Apel normal: la ieșirea din blocul unde a fost definit obiectul, delete
  - Apel la tratarea unei excepții: la desfășurarea stivei
- Dacă în al doilea caz destructorul ar genera o excepție se consideră că mecanismul eșuează și se apelează terminate()
- Dacă destructorul apelează funcții ce generează excepții, se poate proteja prin includerea acestora într-un bloc try și adăugarea unui handler **`catch(…){}`**
- Se poate folosi funcția `uncauth_exception()` din `<stdexcept>` pentru a verifica dacă o anume excepție a fost tratată sau nu

Excepții care nu sunt erori

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

void main(){
     int nr;
     cout << "Numar> ";
     cin >> nr;
     cout << endl;
     try{
        if (nr == 0) throw "zero";
        if (nr == 1) throw "unu";
        if (nr % 2 == 0) throw "par";
        for (int i = 3; i < sqrt(nr); i++)
                if (nr % i == 0) throw "neprim";
        throw "prim";
     }
     catch (char *concluzie) {
        cout << " Numarul introdus este " << concluzie;
        cout << endl;
     }
}
```

Specificarea excepțiilor

- O funcție poate specifica în declarația sa mulțimea de excepții pe care o generează:
  - *tip nume_functie(param) throw(lista tipuri);*

```
void f(int a) throw(t1, t2);
```

  - Funcția f poate genera doar excepții de tip t1, t2 și excepții derivate din aceste tipuri
  - Prin specificarea excepțiilor funcția oferă garanții apelantului său
  - În cazul în care la execuție se ajunge la o violare a acestei specificații, se apelează std::unexpected() care de fapt pointează către std::terminate()

Specificarea excepțiilor

```
void f() throw (t1, t2)
{
  //…
}
```

este echivalent cu:

```
void f()
try
{
  //…
}
catch(t1) {throw;} // rethrow
catch(t2) {throw;} // rethrow
catch (…) { std::unexpected(); }
```

Specificarea excepțiilor

- O funcție virtuală poate fi extinsă doar de o funcție care are specificația de excepții cel puțin cu restricția celei din bază

```
class B{
public:
  virtual void f();
  virtual void g() throw(X, Y);
  virtual void h throw(X);
}
class D: public B{
public:
  void f();
  void g() throw(X); // OK
  void h() throw(X, Y); // error
}
```

Excepții standard

- **exception**
  - **logic_error**
    - **length_error**
    - **domain_error**
    - **aut_of_range**
    - **invalid_argument**
  - **runtime_error**
    - **range_error**
    - **overflow_error**
    - **underflow_error**
  - **bad_alloc**
  - **bad_exception**
  - **bad_cast**
  - **bad_typeid**
  - **ios_base::failure**

Clasa de bază – interfața

```
class exception {
      public:
        exception() throw();
        exception(const exception&) throw();
        exception& operator=(const exception&) throw();
        virtual ~exception() throw();
        virtual const char* what() const throw();
      };
```

- **`what()`** returnează o descriere a excepției sub forma unui șir de caractere

Excepții generate de limbaj

- bad_alloc
  - generată de new (eşec de alocare de memorie)
- bad_cast
  - generată de dynamic_cast (expresie invalidă)
- bad_typeid
  - generată de typeid (returnare pointer nul)
- bad_exception
  - generată de ”exception specification”

```
class X { };
class Y { };
void f() throw (X, std::bad_exception)
{
  //…
  throw Y(); // throw “bad” exception
}
```

Excepții generate de std

- aut_of_range
  - generată de bitset, deque, string, vector.
- invalid_argument
  - generată de bitset, fstream
- length_error
  - generată de string, vector (număr de elemente din container mai mare decât dimensiunea sa)
- overflow_error
  - generată de bitset<>::to_ulong()
- ios_base::failure
  - generată de ios_base::clear()

Excepții netratate

- funcția std::terminate() se va apela dacă:
  - o excepție este generată cu throw dar nu este tratată;
  - mecanismul de tratare a excepțiilor constată că stiva este coruptă
  - un destructor apelat în procesul desfășurării stivei cauzată de o excepție încearcă să înceteze execuția folosindu-se de o excepție
- Comportamentul lui terminate() se poate modifica prin std::set_terminate() care are ca argument o funcție de tip void fără parametri
- Prin apelul lui set_terminate() pot fi eliberate resursele (apelul explicit al destructorilor) înainte de apela abort()

Sfaturi (Stroustrup)

- Folosiți excepții pentru tratarea erorilor
- Acolo unde sunt suficiente informații pentru a gestiona erorile, renunțați la excepții
- Minimizați utilizarea blocurilor try
- Generați excepții pentru a pune în evidență eșuările în constructori
- Evitați generarea de excepții în destructori
- Proiectați main() încât să prindă și să raporteze toate excepțiile
- separați codul ordinar de cel ce tratează erorile
- Folosiți specificații de excepție în interfață
- Dezvoltați o strategie de tratarea a erorilor la proiectarea aplicației