# Programare orientată obiect

Cursul 11

rev 1.1/PP/20131214

# Sumar

- Clase generice
- Biblioteca C++ standard
  - Excepții
  - Clasa string
  - I/O C++

# Clase generice

- Șabloane pentru crearea de clase cu tipuri de date concrete
  - Instanțiere pe bază de șablon
- Șablonul poate include:
  - Tipuri generice
  - Tipuri concrete (pentru valori constante)
    - Toate pot fi inițializate cu valori implicite
- Declararea și implementarea clasei trebuie să fie accesibile compilatorului
  - În același fișier (antet, sursă)

# Clase generice

```
template <typename T, typename X, …, tip CT, …>
class NumeClasaGenerica
{
//…
        tip functie(param);
//…
};
```

```
template < typename T, typename X, …, tip CT, … >
tip NumeClasaGenerica<T, X, …, CT,…>::functie(param) { … }
```

# Clase generice

- Utilizare clasă cu un tip concret de dată:
  - NumeClasaGenerica<tip_concret> obiect;
- Utilizare clasă cu tipul implicit de dată (dacă este cazul):
  - NumeClasaGenerica<> obiect;

# Clase generice – Exemplul 1

```
template <typename T>
class Fisier
{
        T articol;
public:
        Fisier();
        //…
};

template <typename T>
Fisier<T>::Fisier () { … }
```

```
Fisier<int> fis1;
Fisier<double> fis2;
Fisier<Produs> fis3;
```

# Clase generice – Exemplul 2

```
template <typename T>
class Nod<T>
{
        T _info;
//…
}

template <typename T>
class Lista
{
        Nod<T> *cap;
public:
        Lista();
        //…
};
```

- **Implementare**

  ```
  template <typename T>
  Lista<T>::Lista() { … }
  ```

- **Utilizare**

  ```
  Lista<int> lista1;
  Lista<double> lista2;
  Lista<Produs> lista3;
  ```

# Clase generice – Exemplul 3

```
template <typename T, typename V>
class Pereche //pentru Dictionar
{
        T cheie;
        V valoare;
public:
         Pereche ();
        V valoare(T);
        //…
};

template <typename T, typename V>
Pereche <T, V>:: Pereche () { … }
```

```
Pereche<int, int> per1;

Pereche<int, Produs> per2;

Pereche<char *, char *> per3;
```

# Clase generice – Exemplul 4

```
template <typename T, int DMAX>
class Vector
{
        T data[DMAX];
public:
        Vector();
        //…
};

template <typename T, int DMAX>
Vector<T, DMAX>::Vector () { … }
```

- Vector<int, 10> v1;
- Vector<Produs, 1000> v2;
- Vector<double, 20> v3;

# Clase generice – Exemplul 5

```
template <typename T = int, int DMAX = 100>
class Vector
{
        T data[DMAX];
public:
        Vector();
        //…
};

template <typename T, int DMAX>
Vector<T, DMAX>::Vector () { … }
```

- Vector<> v1;
- Vector<double> v2;
- Vector<double, 20> v3;

# Specializarea claselor

- Comportament diferit pentru anumite tipuri de date
  - Apeluri de funcții specifice: comparații, sortări etc.
- Sînt create clase/metode noi cu tipuri de date concrete
  - în plus față de clasele generice pe care le specializează
- Prioritate față de clasa generică
- Specializare la nivel de
  - Clasă
  - Metode
- Specializare (pe tipuri, la nivel de clasă)
  - Integrală
  - Parțială

# Specializare claselor: integrală

```
template <>
class Pereche <char *, char *>
{
        char *cheie;
        char *valoare;
public:
         Pereche ();
        //…
};

Pereche <char *, char *>:: Pereche () {/* …*/ }
```

# Specializare claselor: parțială

```
template <class V>
class Pereche <char *, V>
{
        char *cheie;
        V valoare;
public:
         Pereche ();
        //…
};

template <class V>
Pereche <char *, V>:: Pereche () {/* …*/ }
```

# Specializare claselor: metode

```
template <class T, class V>
class Pereche <T, V> {
        T cheie;  V valoare;
public:
         Pereche();
         V valoare(T); //…
};
//…
template <typename T, typename V>
V Pereche<T, V>::valoare(T c) { /* */}

char * Pereche <char *,char *>:: valoare(char *c)  { /* specializare pe char * și char * */}
```

# Clase generice: Moștenire

| Clasa de bază | Clasa derivată | Observații |
|---|---|---|
| Generică | Generică | Derivare obișnuită. |
| Normală | Generică | Clasa derivată include facilități de parametrizare. |
| Generică | Normală | Clasa derivată va include o **specializare** a clasei de bază cu tipuri de date concrete |

# Clase generice: Moștenire – 1

```
template <typename T>
class Numar
{
protected:
        T _a;
public:
        Numar(T a) : _a(a) {}
};
```

```
template <typename T>
class Complex : public Numar<T>
{
protected:
        T _b;
public:
        Complex(T a, T b) :
                Numar(a), _b(b) {}
};
```

# Clase generice: Moștenire – 2

```
template <typename T>
class Numar
{
protected:
        T _a;
public:
        Numar(T a) : _a(a) {}
};
```

```
class Complex : public Numar<int>
{
protected:
        int _b;
public:
        Complex(int a, int b) :
                Numar(a), _b(b) {}
};
```

# Biblioteca C++ standard (cont.)

# Tratarea excepțiilor (2)

# Tratarea excepțiilor (2)

- Clasa de bază **exception**
- using namespace **std**;
- Include
  - constructor implicit,
  - constructor de copiere
  - destructor
  - operatorul de atribuire
  - metoda virtuală **what()** care returnează descrierea excepției (char *)
- Ierarhie de excepții

# Clase asociate excepțiilor

- **bad_alloc**: excepție la alocarea memoriei
- **bad_cast**: conversie dinamică
- **bad_exception**: excepție neașteptată
- **bad_typeid**: utilizare typeid cu pointer null
- **ios_base::failure**: clasă de bază pentru excepții la nivel de fluxuri de date

# Clase asociate excepțiilor

- **logic_error**: erori legate de logica internă a programului
  - **domain_error**: neîncadrarea în domeniu
  - **invalid_argument**: argument invalid
  - **length_error**: excepție de lungime
  - **out_of_range**: în afara limitelor

# Clase asociate excepțiilor

- **runtime_error**: excepție la execuție
  - **overflow_error**: excepție de depășire
  - **range_error:** excepții legate de limite
  - **underflow_error**: excepție de depășire negativă

# Excepții utilizator

- Posibilitatea derivării de noi clade de tip excepție
- Supraîncărcarea metodei what():

```
class init_invalida: public exception
{
public:
  virtual const char* what() const throw()
  {
    return "Initializare invalida!";
  }
}
```

# Excepții utilizator

```
void main()
{
	try
	{
		//cod care poate genera exceptie prin throw
		//direct sau din alta functie
	}
	catch(init_invalida ie) {/*tratare exceptie: init_invalida */  cout<<ie.what(); }
	catch(exception ex) {/*tratare exceptie: exception */ }
	catch(…) { /*tratare exceptie: orice tip */}
}
```

# Clasa string/wstring

# Clasa string

- Șiruri de caractere
- Supraîncarcă operatorul de atribuire
- Supraîncarcă operatorii >> și <<
- Supraîncarcă operatorii + și += pentru concatenare
- Supraîncarcă operatorii relaționali pentru comparații
- #include <string>

# Metode

- Numărul de caractere
  - **size**(), **length**()
- Înlocuirea de conținut
  - **replace**(poz, lung, sir)
- Test șir vid
  - **empty**()
- Referirea elementelor
  - Operatorul **[]**
  - Metoda **at**()

# Operații

- Inserare
  - **push_back**(char)
  - **insert**(poz, sir/subsir/caracter)
  - **append**(sir/subsir/caracter)
- Ștergere
  - **erase**(poz =0 , lung = npos)
  - **clear**()
- Obținerea unui șir de caractere
  - char * **c_str**()
  - char ***data**()

# Operații: Căutări

- Prima apariție
  - size_t **find**({string|char*|char}, poz = 0)
- Ultima apariție
  - size_t **rfind**({string|char*|char}, poz = 0)
- Prima apariție a oricărui caracter din/care nu este în listă
  - size_t **find_first_of**({string|char*|char}, poz = 0)
  - size_t **find_first_not_of**({string|char*|char}, poz = 0)
- Ultima apariție a oricărui caracter din/care nu este în listă
  - size_t **find_last_of**({string|char*|char}, poz = 0)
  - size_t **find_last_not_of**({string|char*|char}, poz = 0)
- Generează un subșir/șir nou
  - string **substr**(start = 0, stop = string::npos)