ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

static членови

Објектно ориентирано програмирање

2015

Static членови на класа

Нешто слично како глобални променливи во рамки на една класа (надвор од класата, не можат да бидат пристапени)

- Променлива која е дел од класата, а не е дел од објект на таа класа, се нарекува **`static`** член.
- Постои една и единствена копија од static членот во рамки на една класа, наспроти постоењето на една копија за секој објект на обичните (**`non-static`**) членови.
  - Една единствена копија од променливата делена од страна на сите објекти во класата
    - “Class-wide” информација
- Функција која може да пристапи до членовите на класата, но не мора да има инстанцирано објект преку кој таа ќе биде повикана се нарекува **`static`** функциски член.

Static членови на класа

- Можат да бидат декларирани како **`public`**, **`private`** или **`protected`**
- Примитивни **`static`** податочни членови
  - Предефинирано се иницијализираат на 0
  - Ако сакаме друга иницијална вредност, static податочниот член може да биде иницијализиран единствено само еднаш (за сите објекти од класата)
- **`const static`** податочниот член од **`int`** или **`enum`** тип може да биде иницијализиран при неговата декларација во дефиницијата на класата
  - Алтернативно, може да биде иницијализиран надвор од класата
- Сите други **`static`** податочни членови мораат да бидат иницијализирани надвор од дефиницијата на класата
- **`static`** податочните членови од класен тип кои имаат предефиниран конструктор не мора да бидат иницијализирани бидејќи нивниот предефиниран конструктор ќе биде повикан

Static членови на класа

- За да се пристапи на **`public static`** податочен член кога не постои инстанциран објект од класата:
  - Се користи името на податочниот член и како префикс името на класата и scope операторот (::)

    `Employee::count`
- Исто така овие податочни членови можат да бидат пристапени од страна на секој објект од класата
  - Се користи името на објектот, операторот точка и името на податочниот член

    `Employee_object.count`
- **`static`** функциски член

- Пример: `SEmployee.h, SEmployee.cpp, static.cpp`

Static функциски член

- Функциите декларирани како **`static`** смеат да пристапуваат само до **`static`** податочните елементи на класата.
- За разлика од другите функции членки на класата, **`static`** функциите не мора (но може) да се повикаат преку постоечки објект од класата. Може да се повикуваат преку името на класата со **`scope`** операторот.
- **`static`** функциите немаат **`this`** покажувач.
- **`static`** функциите не можат да бидат виртуелни.

SEmployee.h

- Employee има **`static`** функција и static податочен член

```
class Employee
{
public:
    Employee(const char *const, const char *const);
    ~Employee();
    const char* getFirstName() const;
    const char* getLastName() const;
    static int getCount();
private:
    char* firstName;
    char* lastName;
    static int count;                    // број на инстанцирани објекти
};
```

`Static` податочниот член чува информација за бројот на Employee објекти кои постојат;
`Static` функцијата може да биде повикана, дури и да нема инстанцирано објект од класата.

SEmployee.cpp (1/3)

```
// дефинирање и иницијализирање на static податочниот член во датотека
int Employee::count = 0; // не се пишува клучниот збор static

int Employee::getCount()
{
   return count;
}
```

Иако **`static count`** е **`private`**!

SEmployee.cpp (2/3)

```
Employee::Employee(const char *const first, const char *const last)
{
   firstName = new char[ strlen( first ) + 1 ];
   strcpy(firstName, first);

   lastName = new char[ strlen( last ) + 1 ];
   strcpy( lastName, last );

   count++;

   cout << "Employee constructor for " << firstName
      << ' ' << lastName << " called." << endl;
}
```

- Non-static член функциите (на пр. конструкторот) можат да ги менуваат static податочните членови на класата.

SEmployee.cpp (3/3)

```
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
      << ' ' << lastName << endl;

   delete[] firstName;
   delete[] lastName;

   count--;
}
```

static.cpp (1/2)

```
cout << "Number of employees before instantiation of any
  objects is " << Employee::getCount()  << endl;

Employee* e1Ptr = new Employee( "Susan", "Baker" );
Employee* e2Ptr = new Employee( "Robert", "Jones" );

cout << "Number of employees after objects are instantiated is "
     << e1Ptr->getCount();
```

- **`static`** функцискиот член може да се пристапи преку името на класата и операторот ::
- Повикувањето на **`static`** функциски член преку покажувач на објект го има истото значење како и повикувањето преку името на класата

static.cpp (2/2)

```
cout << "\n\nEmployee 1: " << e1Ptr->getFirstName() << " "
     << e1Ptr->getLastName() << "\nEmployee 2: " <<
  e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n\n";

delete e1Ptr;
e1Ptr = 0; // e1Ptr = NULL;
delete e2Ptr;
e2Ptr = 0;

cout << "Number of employees after objects are deleted is "
     << Employee::getCount() << endl;
```

- Дури и ако објект од класата не е инстанциран, static функцијата `getCount()` може да биде повикана

static.cpp Sample Output

```
Number of employees before instantiation of any objects is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after objects are instantiated is 2 (same as
calling Employee::getCount() = 2)

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after objects are deleted is 0
```

Const Static членови

```
#include <iostream>

using namespace std;

class F {
public:
  static int getcount();
  // static member function cannot have `const' method qualifier
private:
  const static int count;
};

// initialization of constant static variable: must be here; not in main()
const int F::count = 2;

int F::getcount() {
  cout << count;
}

int main() {
  F::getcount();  // print out 2
  F::getcount();  // print out 2
  cout << F::count;  // wrong as 'const int F::count' is private
  return 0;
}
```

ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

ИСКЛУЧОЦИ

Објектно ориентирано програмирање

2015

Програмирање со исклучоци

- Тешко е да се предвиди што сè може да тргне наопаку при извршувањето на програмата и што да се преземе во секој од случаите
- Исклучок – неочекуван настан кој го нарушува нормалниот тек на програмата

Традиционален начин на справување со исклучоци

- Мешање на програмската логика и логиката за справување со грешки
  - Псевдокод

```
Изврши определена задача
Ако задачата не се извршила коректно
   Преземи акци
Изврши следна
Ако задачата н
   Преземи акци
...
```

Забелешка:– Во најголемиот број на големи системи, кодот за справување со грешки и исклучоци претставува >80% од вкупниот код во системот

- Прави програмата да биде тешка за читање, менување, одржување и дебагирање
- Влијае на перформансите

Exception handling

- Справувањето со исклучоците (exception handling) претставува механизам (можност) на програмскиот јазик за справување со ситуации кои би го пореметиле нормалното извршување на програмата
- Исклучоците се користат како механизам за сигнализирање на „исклучителна“ (ненормална) состојба

Настапување на исклучок

- При настанување на исклучок, тековнта состојба на програмата се зачувува по што контролата се предава на предефиниран „справувач“ (handler) со исклучокот
- Зависно од ситуацијата справувачот може
  - подоцна да го продолжи извршувањето на програмата од оригиналната локација, користејќи ја снимената состојба откако причините за појава на исклучокот ќе бидат отстранети (page fault)
  - или да пријави грешка (division by zero)

Генерирање на исклучоци

- Програмскиот јазик C++ поддржува механизам за обработка на исклучоци. Исклучоците претставуваат неочекувано однесување на програмата (аномалија – недозволена операција, немање доволно меморија, ...). Делот од програмата во која се јавува исклучокот **генерира** сигнал - „фрла“ исклучок, односно го прави достапен на остатокот од програмата. Во **друг дел** од програмата фрлениот исклучок „се фаќа“ и се обработува (се преземаат соодветни акции).

Исклучоци во C++

- Исклучок се генерира со **`throw object`**;
- Во **`try`** блокот се ставаат наредбите кои може да предизвикаат исклучок. Веднаш по try блокот се наоѓа секвенца од **`catch`** изрази секој задолжен за обработка на различен вид исклучоци.

```
try
{
    . . .
    throw …
    f(…);
    . . .
}
catch(…)
{
    . . .
}
. . .
catch(…)
{
    . . .
}
```

Throw

- **`Throw`** изразот прифаќа еден параметар како свој аргумент и овој се предава на управувачот за исклучоци. Може да имаме произволен број на **`throw`** наредби со различни вредности во рамки на **`try`** блокот, така што справувачот со исклучоци при добивањето на параметарот ќе знае точно која акција треба да ја преземе.

```
try
{           // code
            if ( x )
              throw 10;
            // code
            if (y)
              throw 20;
            //code
 }
```

Catch() {}

- Справувачот со исклучоци може да биде идентификуван со клучниот збор **`catch`**. **`catch`** секогаш прима еден аргумент. Типот на catch параметарот е значаен за да се одлучи кој справувач ќе биде активиран.

```
try { // code here }
catch (int param) {
   cout << "int exception";
}
catch (char param) {
   cout << "char exception";
}
catch (...) {
   cout << "default exception";
}
```

- **`catch`**(…) справувачот ги фаќа сите исклучоци, без разлика на типот. Може да се користи како предефиниран справувач ако биде деклариран последен.

Исклучоци во C++ (пример)

```
int main()
{
    try {
        int i;
        cin >> i;
        switch(i) {
            case 1: throw 1;
            case 2: throw 2;
            case 3: throw 2.1;
            case -1: throw "qwe";
        }
        cout << i << endl;
    }
    catch(int)    { cout << "Exception type 1" << endl; }
    catch(double) { cout << "Exception type 2" << endl; }
    catch(char)   { cout << "Exception type 3" << endl; }
    cout << "END" << endl;
}
```

```
1 ↵
Exception type 1
END

2 ↵
Exception type 1
END

3 ↵
Exception type 2
END
```

```
4 ↵
4
END

-1 ↵
Runtime error!
```

```
class Exception0 {};
class Exception3 {};
int main() {
  try {
      int i;
      char *p="abc";
      int *ip;
      cin >> i;
      switch(i) {
       case 1: throw p;
       case 2: throw ip;
       case 3: throw Exception3();
       case -1: throw Exception0();
       case -2: throw 2;
      }
      cout << i << endl;
  }
  catch(char *s) { cout << "Exception type 1, char *=" << s << endl;}
  catch(void *v) { cout << "Exception type 2, void *=" << v << endl;}
  catch(Exception3) { cout << "Exception type 3" << endl; }
  catch(...) { cout << "Exception of unknown type" << endl; }
  cout << "END" << endl;
}
```

```
1
Exception type 1, char *=abc
END
```

```
2
Exception type 2, void *=0x431000
END
```

```
3
Exception type 3
END
```

```
4
4
END
```

```
-1
Exception of unknown type
END
```

```
-2
Exception of unknown type
```

```
1
Exception type 2, void *0x4012d0
END
```

Исклучоци во функции

```
#include <iostream>
using namespace std;

class Exception1 {}; class Exception2 {};
class Exception3 {}; class Exception4 {};

void f(int i) {
   try {
      switch(i) {
          case 1: throw Exception1();
          case 2: throw Exception2();
          case 3: throw Exception3();
          case -1: throw Exception4();
       }
       cout << i << endl;
   }
   catch(Exception1) {
      cout << "Exception in f(" << i << ") of type 1" << endl;
    //    throw;
   }
   catch(Exception2) {
      cout << "Exception in f(" << i << ") of type 2" << endl;
   }
   catch(Exception3) {
      cout << "Exception in f(" << i << ") of type 3" << endl;
   }
}
```

```
void main()
{
   int i;
   cin >> i;
   try {
      f(i);
   }
   catch(Exception1) {
      cout << "Exception of type 1" << endl;
   }
   catch(...) {
      cout << "Exception of unknown type" << endl;
   }
   cout << "END" << endl;
}
```

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

class Error { /* ... */ };
class ExcepA : public Error {};
class ExcepB : public Error {};
class ExcepC : public Error {};
class ExcepD : public Error {};
class ExcepE : public Error {};
class ExcepF : public Error {};
class ExcepG : public Error {};
void e();
void f();
void g();
void h();

int main() {
  cout << "Exception throwing demo" << endl;
  e();
}
```

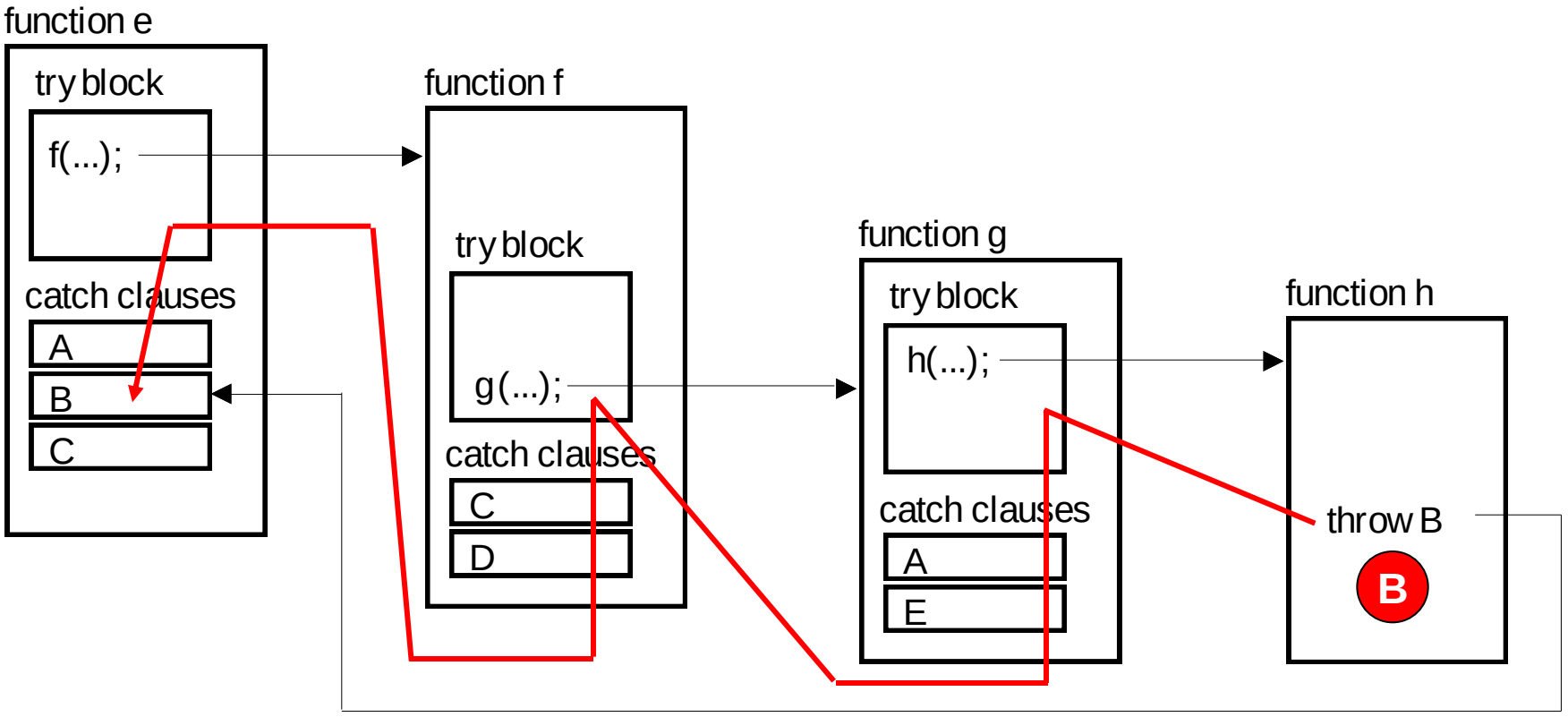
```
void e() {
  try { // do something
      f();
  }
  catch(ExcepA) { cerr << "Exception A caught in function e()" << endl; exit (-1); }
  catch(ExcepB) { cerr << "Exception B caught in function e()" << endl; exit (-1); }
  catch(ExcepC) { cerr << "Exception C caught in function e()" << endl; exit (-1); }
  catch(ExcepF) { cerr << "Exception F caught in function e()" << endl; exit (-1); }
}
void f() {
  try {  // do something
      g();
  }
  catch(ExcepC) { cerr << "Exception C caught in function f()" << endl; exit (-1); }
  catch(ExcepD) { cerr << "Exception D caught in function f()" << endl; exit (-1); }
  catch(ExcepF) { cerr << "Exception F SPOTED in function f()" << endl; throw; }
}
void g() {
  try {  // do something
      h();
  }
  catch(ExcepA) { cerr << "Exception A caught in function g()" << endl; exit (-1); }
  catch(ExcepE) { cerr << "Exception E caught in function g()" << endl; exit (-1); }
}
```

```
void h() {
  char e;
  do {
      cout << "Exception to throw [A,B,C,D,E,F,G]: ";
      cin >> e;
      cout << "Throwing Exception " << e << endl;
      switch(e) {
       case 'A': case 'a': throw ExcepA();
       case 'B': case 'b': throw ExcepB();
       case 'C': case 'c': throw ExcepC();
       case 'D': case 'd': throw ExcepD();
       case 'E': case 'e': throw ExcepE();
       case 'F': case 'f': throw ExcepF();
       case 'G': case 'g': throw ExcepG();
       default : cout << "No such Exception!" << endl;
      }
  }
  while(strchr("ABCDEabcde",e)==NULL); }
```

```
Exception throwing demo
Exception to throw [A,B,C,D,E,F,G]: A ↵
Throwing Exception A
Exception A caught in function g()

Exception throwing demo
Exception to throw [A,B,C,D,E,F,G]: B ↵
Throwing Exception B
Exception B caught in function e()

Exception throwing demo
Exception to throw [A,B,C,D,E,F,G]: C ↵
Throwing Exception C
Exception C caught in function f()

Exception throwing demo
Exception to throw [A,B,C,D,E,F,G]: D ↵
Throwing Exception D
Exception D caught in function f()
```

```
Exception throwing demo
Exception to throw [A,B,C,D,E,F,G]: E ◄┘
Throwing Exception E
Exception E caught in function g()

Exception throwing demo
Exception to throw [A,B,C,D,E,F,G]: F ◄┘
Throwing Exception F
Exception F SPOTED in function f()
Exception F caught in function e()

Exception throwing demo
Exception to throw [A,B,C,D,E,F,G]: G ◄┘
Throwing Exception G
Unhandled Exception…
```

Пропагација на исклучок



- Фаќање на исклучоци кај изведени класи
  - Потребно е да се биде внимателен на подредувањето на **catch** наредбите кога се обидуваме да фатиме некој исклучок со податочен тип од некоја изведена класа затоа што **catch** наредбата за основната класа ги фаќа исклучоците со податочни типови на сите изведени класи од неа

```
// Catching derived classes.
#include <iostream>
using namespace std;
class B {
};
class D: public B {
};
int main()
{
	D derived;
	try {
		throw derived;
	}
	catch(B b) {
		cout << "Caught a base class.\n";
	}
	catch(D d) {
		cout << "This won't execute.\n";
	}
return 0;
}
```

Restricting Exception

- Можност за ограничување на типовите на исклучоци кои што една функција може да ги генерира
- Овозможено со следнава синтакса

ret-type func-name(*arg-list*) throw(*type-list*)
{
// ...
}

```
// Restricting function throw types.
#include <iostream>
using namespace std;
// This function can only throw ints, chars, and doubles.
void Xhandler(int test) throw(int, char, double)
{
	if(test==0) throw test; // throw int
	if(test==1) throw 'a'; // throw char
	if(test==2) throw 123.23; // throw double
}
int main()
{
	cout << "start\n";
	try{
		Xhandler(0); // also, try passing 1 and 2 to Xhandler()
	}
	catch(int i) {
		cout << "Caught an integer\n";
	}
	catch(char c) {
		cout << "Caught char\n";
	}
	catch(double d) {
		cout << "Caught double\n";
	}
	cout << "end";
	return 0;
}
```

Rethrowing an Exception

```
// Example of "rethrowing" an exception.
#include <iostream>
using namespace std;
void Xhandler()
{
    try {
        throw "hello"; // throw a char *
    }
    catch(const char *) { // catch a char *
        cout << "Caught char * inside Xhandler\n";
        throw ; // rethrow char * out of function
    }
}
```

Rethrowing an Exception

```
int main()
{
    cout << "Start\n";
    try{
    Xhandler();
    }
    catch(const char *) {
            cout << "Caught char * inside main\n";
    }
    cout << "End";
    return 0;
}
```

Добри програмски практики со C++ исклучоци

- Не користете исклучоци за нормален програмски тек
  - Користете само кога нормалниот тек е невозможен
- Секогаш фрлајте некој тип со исклучокот
  - За истиот да може да биде фатен
- Користете ја спецификацијата за исклучоци
  - Тоа може да помогне при справувањето со исклучоците