Тема 6. Средства за организация на програмите (продължение)

4. Статични членове;
5. Типови преобразувания;
6. Изключения

4. Статични членове

Дефиниция ***static*** :

В езика C++ ***static служи за*** **модификация на декларацията**, пред която се поставя

Статичните членове в C++ се дефинират чрез ключовата дума ***static,*** поставена **пред** декларация

Статични членове

Видове декларации:

- Преди декларация на **локална променлива**. В този случай се получават ***статични локални променливи***;
- Преди декларацията на **функции, които не са членове на класове**. В резултат се получават ***статични функции***;
- Преди декларацията на **член променливи на клас**. В този случай се получават ***статични член променливи***;
- Преди декларациите на **член функции** на клас. В този случай се получават ***статични член функции***;

Статични членове

- Синтаксис:

static *декларация*;

Когато се прилага към функция (независимо дали е член функция на клас или не, тя се поставя ***само пред прототипа(декларацията)*** Не е необходимо да се записва отново пред **дефиницията на функцията**.

Статични членове

- **Статични локални променливи:**

Използва се за да се обяви локална променлива, с време на живот, както при глобалните променливи, но с област на видимост в рамките на обхвата, в който е обявена.

Променливата запазва стойността си между различните обръщения към мястото, където е декларацията.

Статични членове

Пример: Функция, изчисляваща номера на извикването си:

```
int fCalcCallNumber() {
  static int iNumCalls = 0;
  iNumCalls++;
  return iNumCalls;
}
void main() {
  for(int i=0; i<10; i++)
       cout << fCalcCallNumber() << "  " ;
}
```

Статични членове

Пример:

Fib(n) със следната математическа дефиниция:

Fib(0) = 0;

Fib(1) = 1;

Fib(n) = Fib(n-1) + Fib(n-2) за всяко n>1;

Статични членове

Реализация:

```
int Fibonacci(void) {
   static int temp;
   static int fib1 = 0;
   static int fib2 = 1;
   temp = fib1 + fib2 ;
   fib1 = fib2 ;
   fib2 = temp ;
   return fib1 ;
}
void main() {
  cout << "Fibonacci: { " ;
  for(int it = 0; it < 9; it++)
     cout << Fibonacci() << " " ;
  cout << " }\n" << endl ;
}
```

Статични членове

- **Статични функции**:

Предназначение: За намаляване на областта на видимост на една глобална функция.

Целта е да се **ограничават колизиите**, които се получават между имена на функции в приложението. Когато се приложи този модификатор към декларацията на глобална функция, областта на видимост на тази функция се ограничава до модула (файл с изходен код на C++), в който е дефинирана.

Статични членове

Пример MyMathFunctions.cpp

```
#include <iostream>
using namespace std;
// декларация на функцията ( в началото на модула)
static int Fibonacci(void);
.....
// дефиниция на функцията (не е нужно да се повтаря
// статичната декларация)
int Fibonacci(void) {
   static int temp;
   static int fib1 = 0;
   static int fib2 = 1;
   temp = fib1 + fib2 ;
   fib1 = fib2 ;
   fib2 = temp ;
   return fib1 ;
}
```

Статични членове

- **Статични член променливи на класовете**:

За дефиниране на член променлива, която да се използва като глобална, но да бъде асоциирана към даден клас и невидима извън класа, за член променливата на класа се използва декларацията ***static***. Обявяването на дадена член променлива от класа заделя място в общия даннов сегмент на програмата за всички обекти, които се образуват от дадения клас, независимо от техния брой. Всичките създадени обекти ***използват едно и също*** място за съхраняване на данни за тази член променлива (interface).

Статични членове

Правила:

- При поставяне на декларацията пред член променливата, компилаторът ***не заделя място за променливата в обекта***.

  *При свързване на програмата свързващият редактор ще генерира грешка, че статичната член променлива **не е дефинирана**.*

- Дефиницията трябва да се направи извън класа, като е допустима само една такава. *Най-логично е да се направи във файла с класа и допълнително дефиниране при описанието на класа.*

Статични членове

```
// Пример:
#include <iostream>

using namespace std;

class CStudent  {
  public:
    static int iNumStudents; // декларация
    CStudent()  { // конструктор на класа
      ++iNumStudents;
    }
//.....
};
//дефиниция на статичната член променлива
int CStudent::iNumStudents=0;
```

Статични членове

```
// тестова програма

void main()
{
  CStudent oSt1;  // създаване на обект 1
  CStudent oSt2; // създаване на обект 2
  cout << CStudent::iNumStudents;
// извежда брой създадени обекти
// клас принадлежността
// (CStudent::) е  задължителна
}
```

Статични членове

- **Статични член функции в класовете**:

Тези функции имплементират операции, валидни за класа като цяло (глобални), а не за отделния обект от този клас.

Вместо да се създава глобална функция, която да е видима в цялото програмно пространство, обявява се функция, която се асоциира към съответния клас.

Извикването на статичната функция се извършва:

- Чрез точка (стеково базирани) ;
- Стрелка (чрез указател) от съответния обект, към който е асоциирана;
- Освен това много типичен начин за извикване на статичната член функция е с използването на разделителя за принадлежност (::) и име на класа.

Статични членове

- Статични член функции

Пример:

```
class CStaticSample {
public:
static void fStatic();
};
// извикване на статична член функция
// чрез клас принадлежност
CStaticSample::fStatic();
```

Статични членове

Когато се срещне статична член функция в клас, трябва да се разбира, че проектиращият класа е имал идеята да асоциира тази функция към класа като цяло. Правила за използването на статичните член функции:

- Статичните член функции **не могат да правят достъп до обикновените член променливи на класа**, а само до статичните.
- **Могат да извикват само други статични член функции**.

За всеки клас адресът на текущия обект (**this**) може да се използва за достъп до всеки елемент на класа, но статичните член функции нямат указател към обекта, поради това **нямат достъп до обикновените член променливи**.

5. Типови преобразувания

Дефиниция:

Управлението на типовите преобразувания, които програмистът може да използва при програмирането се осигурява от C++ ***оператори за преобразуване на типове***. **Видове**:

reinterpret_cast <new_type> (expression)

static_cast <new_type> (expression)

const_cast <new_type> (expression)

dynamic_cast <new_type> (expression)

Типови преобразувания

Където new_type е типът, към който се преобразува expression.

Основа:

Известните от езика C преобразувания на типовете, дефинирани с *cast operator* със съответните обозначения:

(new_type) expression

new_type (expression)

Типови преобразувания-видове

- *reinterpret_cast*

- Преобразува указател от какъв да е тип към друг тип указател. Може да преобразува указатели между обекти, които нямат релация помежду си.
- Резултатът от операцията е двоичното копие от един указател на друг.
- Съдържанието по указателя не се проверява, не се правят някакви трансформации между типовете. Близък е до използваните в езика C, но се използва за указатели.

Типови преобразувания-видове

Пример:

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

Типови преобразувания-видове

- ***static_cast***
- Подразбиращо се (от компилатора). Прилага се към указатели към класовете, като позволява да се преобразува указател към клас наследник към указател към базов клас
- В обратен ред (да се преобразува указател към базов клас към такъв сочещ наследник). В този случай не се осигурява проверка за определяне дали е правилно или не съответното преобразуване.

Типови преобразувания-видове

Пример:

```
class Base {};
class Derived: public Base {};
Base * a = new Base;
Derived * b = static_cast<Derived*>(a);
```

static_cast между прости типове:

```
double d=3.14159265;
int i = static_cast<int>(d);
```

Типови преобразувания-видове

- const_cast

Добавят константни атрибути на декларирания обект.

Пример:

```
class C {};
const C * a = new C;
C * b = const_cast<C*> (a);
```

Не се допуска с помощта на другите видове преобразувания да се промени константността на указателя към обекта

Типови преобразувания видове

- **dynamic_cast**

За преобразувания между указатели и референции към обекти.

- Може да преобразува указател към базов клас към такъв към наследник;
- Осигурява проверка за валидността на преобразуващата операция:
  - Може да се преобразува-резултат;
  - Нулев указател.

Типови преобразувания видове

Пример:

```
class Base { virtual dummy(){}; };
class Derived : public Base { };

Base* b1 = new Derived;
Base* b2 = new Base;
Derived* d1=dynamic_cast<Derived*>(b1);
// d1 е успешно преобразуван

Derived* d2=dynamic_cast<Derived*>(b2);
// неуспешно-NULL
```

Типови преобразувания видове

Ако преобразуването се приложи към тип референция, резултатът (нулев указател) не е валиден, поради което се генерира изключение от тип bad_cast , пример:

```
class Base { virtual dummy(){}; };
class Derived : public Base { };

Base* b1 = new Derived;
Base* b2 = new Base;
Derived d1 = dynamic_cast<Derived&*>(b1);
// успешно преобразуване
Derived d2=dynamic_cast<Derived&*>(b2);
// неуспешно-изключение bad_cast
```

Проверка на типовете

Дефинира се с ***typeid***:

Определяне типа на параметъра по време на изпълнение.

Проверка на типовете

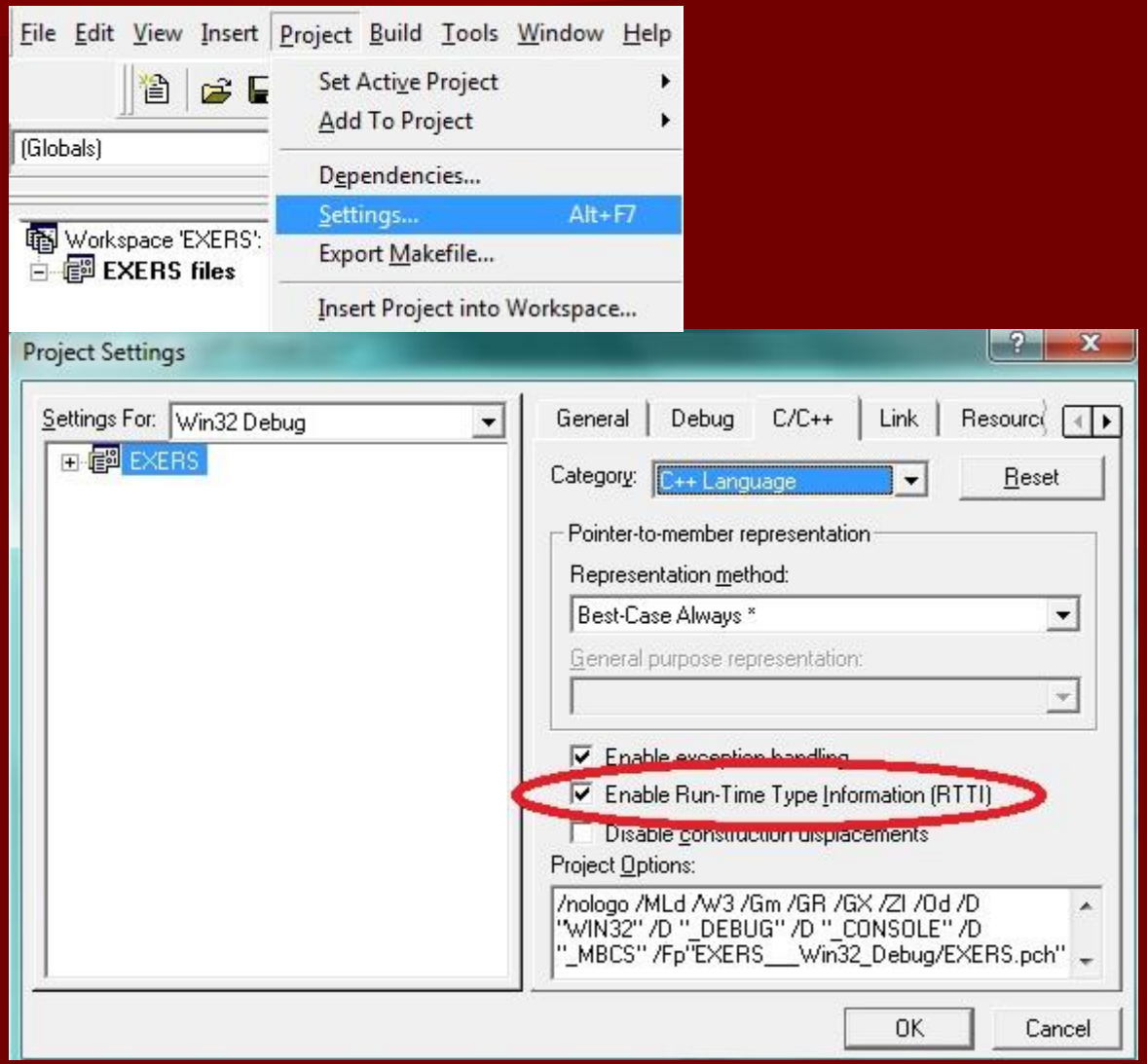
```
// Пример:
#include <iostream>
#include <typeinfo>
using namespace std;
int main () {
  int * a,b;
  a=0; b=0;
  if (typeid(a) != typeid(b))
  {
    cout << "a and b are of different types:\n";
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
  }
  return 0;
}
```

Проверка на типовете

```
// Пример за получаване на името на типа CBase* (CDerived*) и
// CBase (CDerived)
class CBase { virtual void f(){} };
class CDerived : public CBase {};
  try {
    CBase* a = new CBase;
    CBase* b = new CDerived;
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
    cout << "*a is: " << typeid(*a).name() << '\n';
    cout << "*b is: " << typeid(*b).name() << '\n';
  } catch (exception& e) { cout << "Exception: " << e.what() << endl; }
```

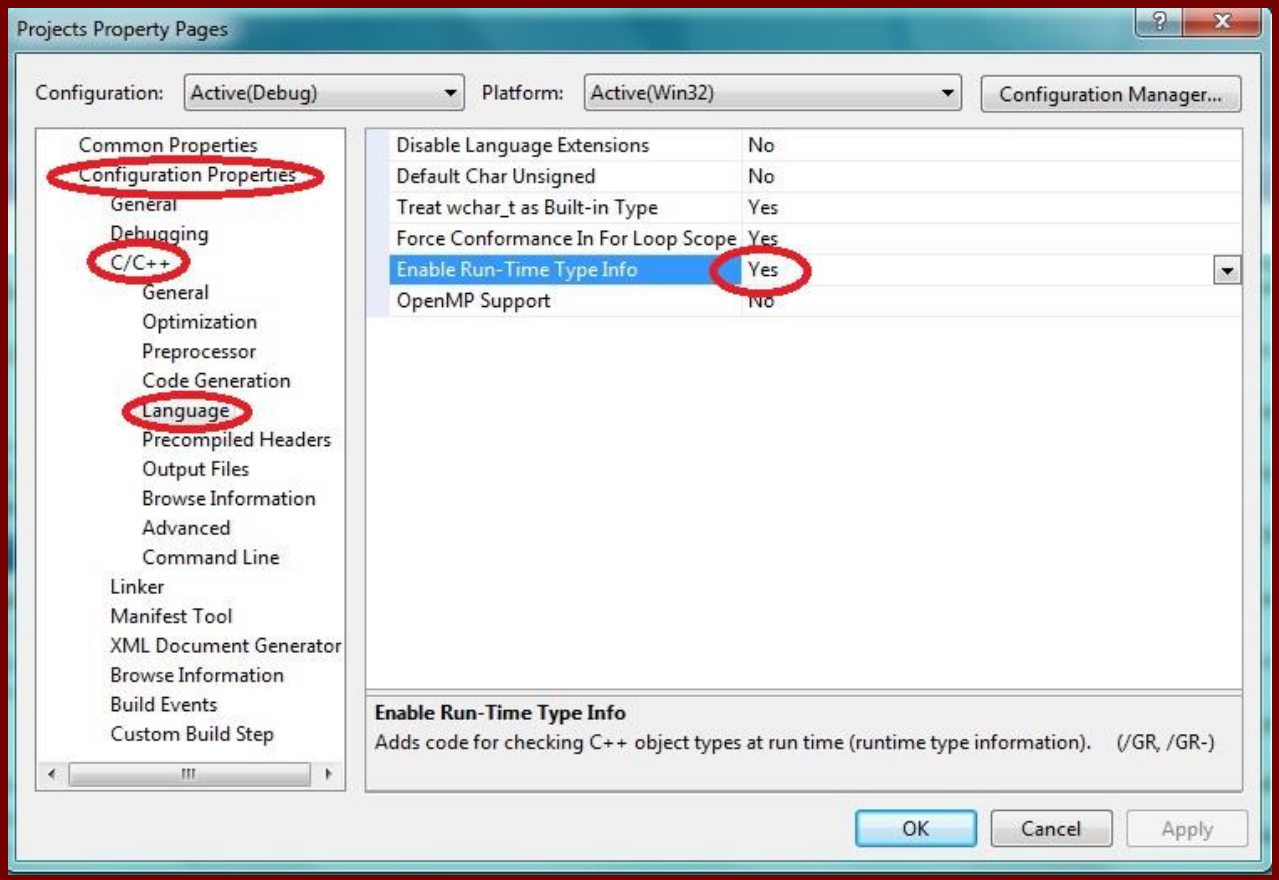
Типови преобразувания

Установяване на настройките на проекта (VS 6.0) с цел допускане на типови преобразувания в реално време- **typeid** и **dynamic_cast**



Типови преобразувания

Установяване на настройките на проекта (VS 2008+) с цел допускане на типови преобразувания в реално време- **typeid** и **dynamic_cast**



6. Изключения

За обработка на непредвидени ситуации при изпълнението на дадена програма.

Дефиниция:

Изключенията предлагат чист начин за откриване и справяне с неочаквани ситуации при изпълнението на приложенията. Всяка програма, когато открие грешка, може да генерира изключение. Когато се предизвика изключение, контролът на програмата се предава на съответният прихващач на изключението (*exception handler*). Чрез дефиниране на функция, която *прихваща* изключения, програмистът може да напише код, който да обработи грешката. Операционната система също продуцира изключения, които могат да се прихващат.

От гледна точка на програмирането на C++ изключението е обект.

Изключения

Обикновенно изключенията са производни (дирекни или индиректни) на базов клас, наречен **exception**, който е дефиниран в стандартната C++ библиотека, във файла <exception>.

Пример за функция, която изпраща изключение използвайки оператора **throw**:

```
class domain_error : public exception {};
void f ()
{
    // ...
    throw domain_error ();
}
```

Изключения

```
void g ()
{
    try {
        f ();
    }
    catch (domain_error) {
        // прихващач на изключението
    }
    catch (range_error) {
        // прихващач на изключението
    }
}
```

Изключения

Действие:

- Когато се генерира изключение, поредицата от извикани функции се претърсва в обратен ред (от викан към викащ), за да се намери най-близкият оператор catch.
- Докато се развива стека на виканите функции, деструкторите на локалните променливи, декларирани във функциите се стартират автоматично.
- Когато програма предизвика изключение, което не се прихваща, се прекъсва изпълнението и.
- Общо прихващане на всички изключения, без значение от генериралия изключението се дефинира чрез оператора catch с анонимен аргумент:

```
catch (…) {
  // прихващач на всички изключения
  }
```

Изключения

```
#include <iostream.h>
int main () {
  try  {
    char * mystring;
    mystring = new char [10];
    if (mystring == NULL) throw "Грешка при заемане на памет";
// несъобразено с размера на масива присвояване:
    for (int n=0; n<=10; n++)  {
      if (n>9) throw n;
      mystring[n]='z';
    }
  }  catch (int i)  { // прихващане на прости типове изключения
    cout << "Индекс " << i << " е извън границите" << endl;
  }  catch (char * str)  {// прихващане на прости типове изключения
// ще изведе “Грешка при заемане на памет”
    cout << str << endl;
  }
  return 0;
}
```

Стандартни изключения

Класова йерархия на стандартните изключения:

- exception
  - bad_alloc (генериран от оператор new)
  - bad_cast (генериран от оператор dynamic_cast)
  - bad_exception (когато не се прихване от друго изключение)
  - bad_typeid (генериран от typeid)
  - logic_error (логически грешки)
    - domain_error
    - invalid_argument
    - length_error
    - out_of_range
  - runtime_error (решки по време на изпълнение на програмата)
    - overflow_error
    - range_error
    - underflow_error
- ios_base::failure (генериран от ios::clear)

Стандартни изключения

Пример:

```
#include <iostream>
#include <exception>
#include <typeinfo>
class A {virtual f() {}; };
int main () {
  try {
    A * a = NULL;
    typeid (*a);
  } catch (std::exception& e)   {
    cout << "Exception: " << e.what();
  }
  return 0;
}
```

Резултат:

```
Exception: Attempted typeid of NULL pointer
```

Средства за организация на програмите

Въпроси?