Тема 7. Средства за организация на програмите

(продължение)

7. Именни пространства;
8. Конструиране на вградени обекти;
9. Вградени функции;
10. Подразбиращи се аргументи;
11. Декларация auto;

Въведение в STL – C++ основа

7. Именни пространства

Дефиниция за Именни пространства :

Програмно средство за създаване на множества от глобални класове, обекти и/или функции:

- Общото глобално пространство се разделя на подобласти, наречени именни пространства- namespaces.

Необходимост:

- Дублиране на имена на променливи и функции, които компилаторът не допуска;
- Възникване на колизии, вследствие използваните еднакви наименования, които трябва да се отстранят;
- Ефективност на програмирането.

Именни пространства

Пример: Файлове за декларация на функциите и класовете на двама програмисти:

```
//Program1.h
char f1(char);
int f1(int);
class CMyClass {/*…*/};
//Program2.h
char f1(char);
double f1(double);
class CMyClass {/*…*/};
```

Именни пространства

Стандартът C++ предлага механизъм за предпазване от колизии-имената пространства:

- Дефинират се с ключовата дума **namespace;**
- Всяко множество от дефиниции в библиотека от програми е “затворено” в именована област **namespace;**
- Позволява друга дефиниция да има същото наименование в друга (различна) именована област, колизии в имената не се получават.

Именни пространства

Ключовата дума, която се използва за дефиниране на именното пространство, което ще се използва е ***using.***

Всички стандартни C++ функции са описани в стандартната библиотека, наречена “Standard C++ libraries” и използват именована област, наречена **std** (получава се от **st**an**d**art).

Именни пространства

Пример: за всяка програма, използваща стандартни функции за вход/изход се използва директива:

using namespace std;

- Използване на стандартни **C** хедерни файлове:

#include <iostream.h>

Еквивалентен запис със STL хедери:

#include <iostream>

using namespace std;

Именни пространства

Общ синтаксис за декларация на именно пространство:

namespace [*идентификатор*] **{**
съдържание на именното пространство-декларации на променливи и функции
}

Именни пространства

Пример:

```
#include <iostream.h>
// декларация на именни пространства
namespace names1
{
  int var = 6;
}
namespace names2
{
  double var = 1.2345;
}
// използване
int main () {
  cout << names1::var << endl;
  cout << names2::var << endl;
  return 0;
}
```

Изход:

```
6
1.2345
```

8. Конструиране на вградени обекти

Дефиниция:

Вграждане на обект в един клас има тогава, когато се декларира обект като член променлива на друг клас - обект в друг обект.

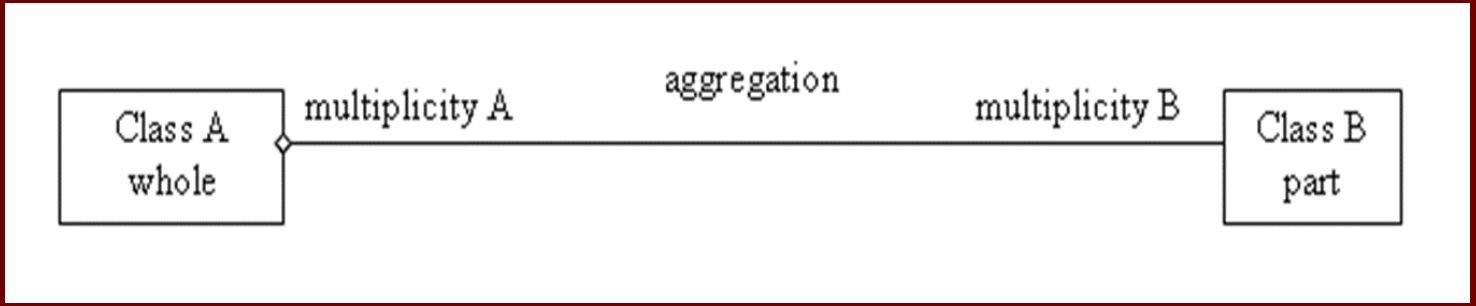
Конструиране на вградени обекти

Пример :

```
class CXY {
public:
  double x, y;
  CXY(void);
  ~CXY(void);
  CXY( double xarg, double yarg);
  CXY(const CXY& xy);
//
};
/** Клас Line: пример за вграждане на обекти в обект.
*/
class Line {
  CXY m_from;
  CXY m_to;
public:
  Line(){…}
};
```

Конструиране на вградени обекти

Обща схема:



За примера:

Конструиране на вградени обекти

Действие на компилатора при създаване на обектите:

- Определя се големината на необходимата памет;
- Стартират се конструкторите на вградените обекти m_from, m_to;
- Създават се обектите, членове на класа;
- Стартира се конструктора на Line.

Конструиране на вградени обекти

Действие на компилатора при унищожаване на обектите.

- Стартиране на деструктора на Line;
- Стартират се деструкторите на вградените обекти;
- Освобождава се паметта m_to, m_from;
- Освобождава се паметта на Line.

9. Вградени функции (Inline functions)

Дефиниция:

При решаване на проблеми чрез C++ концепциите на макросите на езика C се имплементира чрез вградени функции (*inline function*). Цялата функционалност, която се изисква от обикновената функция, може да се поеме от вградената функция:

Разлика:

Създава се като **препроцесорен макрос**, т.е. кода й се помества във всяко място на обръщение;

- Това изисква допълнителна памет за разполагане на кода на функцията;
- Икономисва се времето за изпълнение на функционалното обръщение и връщането от функция – печели се бързодействие.

Вградени функции

Правила:

- Всяка функция, която е дефинирана в тялото на класа е автоматично дефинирана като inline;
- Могат да се дефинират и други функции, извън класовете, но с изрични указания към компилатора- ключовата дума **inline** пред дефиницията на функцията;
- Необходимо е да се дефинира и тялото на функцията, в противен случай компилатора третира тази функция като обикновена декларация.

Вградени функции

Пример:

inline int plusOne(int x);

При дефиниране на тялото на функцията се изисква отново декларация **inline** :

```
inline int plusOne(int x) {
   return ++x;
}
```

Вградени функции

Ефективност на вградените функции:

inline int fact(int n) { return (n<2) ? 1 : fact(n-1); }

С извикване:

fact(5);

Резултат:

120

Вградени функции в класовете

Всяка функция с дефиниция в класовата декларация:

Пример: Включен фаил (Lectures1_16.h) :

```
#include <iostream>
#include <string>
using namespace std;
#define __forceinline // допълнителна декларация
class CXY {
 double m_x, m_y;
public:
  CXY(void) { m_x = 0; m_y = 0; }
  CXY(double xarg, double yarg) { m_x = xarg; m_y = yarg; }
  CXY(const CXY& xy) { m_x = xy.m_x; m_y = xy.m_y; }
  void print(const  string& msg) const {
    if (msg.length()>0)
      cout << msg << endl;
    cout << "x = " << m_x << ", " << "y = " << m_y << endl;
  }
  ~CXY(void){}
};
```

Вградени функции в класовете

Използване: Lectures1_16.cpp

```
#include "Lectures1_16.h"
int main(int argc, char* argv[])
{
  CXY xy3(1.0, 2.0), xy4(3.0, 4.0);
  xy3.print("value of xy3");
  xy4.print("value of xy4");
  return 0;
}
```

Вградени функции в класовете

Действие на компилатора:

- Стандартни проверки;
- Анализира тялото на функцията и генерирания за него код се поставя в таблицата със символи.
- Кодът на тялото на функцията се включва като се елиминира самото обръщение и връщането на резултата;
- Ако фунцията е член на клас, в мястото на обръщението се поставя адресът на обекта (**this**);

Вградени функции в класовете

Вградени функции и прегледност на кода,
Използване на *.h и *.cpp:

Пример:

```
/* Lectures1_17.h */
#include <iostream>
class CXY {
public:
  CXY(void);
  CXY(double xarg, double yarg);
  CXY(const CXY& xy);
  void print(const string& msg) const;
  ~CXY(void){}
  double m_x, m_y;
};
```

Вградени функции извън класовете

```
/* Lectures1_17.cpp */
#include <iostream>
#include <string>
using namespace std;
#include "Lectures1_17.h"
#define __forceinline
inline CXY::CXY(void) { m_x = 0; m_y = 0; }
inline CXY::CXY(double xarg, double yarg)  { m_x = xarg; m_y =
    yarg; }
inline CXY::CXY(const CXY& xy){ m_x = xy.m_x; m_y = xy.m_y; }
inline void CXY::print(const string& msg) const {
  if (msg.length()>0)
    cout << msg << endl;
  cout << "x = " << m_x << ", " << "y = " << m_y << endl;
}
```

Вградени функции извън класовете

(продължение)

```
int main(int argc, char* argv[])
{
  CXY xy3(1.0, 2.0), xy4(3.0, 4.0);
  xy3.print("value of xy3");
  xy4.print("value of xy4");
  return 0;
}
```

10. Подразбиращи се аргументи

Дефиниция:

C++ предоставя възможност за създаване на предефинирана функция с едно и също тяло, която може да се извика по различни начини чрез добавяне на фактически аргументи.

Подразбиращи се аргументи

Пример:

```
// Изход на цяло число с подразбираща се
//основа 10
void print( int iValue, int iBase=10 );
```

Използване:

```
print(31);
print(31,10);
print(31, 16);
print(31, 2);
```

Изход:

```
31 31 1f 11111
```

Подразбиращи се аргументи

Основни правила при използването на подразбиращите се аргументи:

- Само последните аргументи от списъка могат да бъдат подразбиращи се:
  - След подразбиращ се аргумент не може да има неподразбиращ се.
  - Да се осигурят подразбиращи се стойности на всички аргументи до края на списъка от аргументи на функцията (следва от 1);
- Подразбиращите се аргументи се описват само в декларацията на функцията (която обикновено се включва във включен файл със суфикс *.H), а не при дефиницията на функцията:
  - Компилаторът трябва да знае формата и подразбиращата се стойност преди да използва функционалното обръщение;
  - Обикновено при програмирането на дефиницията се запазва в коментар подразбиращата(ите) се стойност(и) за документиране на този факт.

Подразбиращи се аргументи

Декларативна част на функцията с подразбиращ се аргумент:

```
void fn(int x = 0 );
```

Дефиниция на функцията:

```
void fn(int x /* = 0 */) { }
```

Подразбиращи се аргументи

Правила:

- Използването на подразбиращи се аргументи вместо предефинирана функция е обосновано при функции, които имат еднакво действие при промяна в стойнос(тите) на предаваните аргументи;
- Ако функциите имат голямо различие в описанието (действието) си не е обосновано използването на подразбиращи се аргументи:
  - Използването на предефиницията на функцията също трябва да се обмисли в този случай – каква е причината, поради която те имат еднакво име.

11. Декларация auto cpp 11

- При деклариране на променливи в блок;
- В декларация на резултат от функция;
- В декларация на инициализации;
- В декларация на тип на променлива;
- Пред клас принадлежност;
- Аргумент на цикличен оператор;
- При lambda функции;

11. Декларация auto cpp 11

При деклариране на променливи в блок:

auto <*variable initializer*>

Заменя се с използване на типа на инициализацията. Пример:

```
{
auto a = 1 + 2;
std::cout << "type of a: " << typeid(a).name() << '\n';
}
int
```

11. Декларация auto cpp 11

В декларация на резултат от функция с шаблонни аргументи. Пример:

```
template<class T, class U>
auto add(T t, U u) -> decltype(t + u){
  return t + u;
}
auto b = add(1, 1.2);
std::cout << "type of b: " << typeid(b).name() << '\n';
//синтаксис за типа: на operator+(T, U) т.е. double
```

Въведение в STL

Създаване на библиотеката:

Alexander Stepanov & Meng Lee - 1992 г.

Основна идея:

Създаване на алгоритми, дефинирани като обобщени, без да се губи ефективността им.

STL и ANSI/ISO C++ стандарти

Специфични за STL програмни средства на C++

Основните понятия от C++ : Класове,обекти и щаблони.

2.1 Стандартизирани класове;
2.2 Обект функция;
2.3 Темплейти
2.3.1 Темплейтни функции;
2.3.2 Темплейтни класове за контейнери;
2.3.3 Специализирани темплейти

2.1 Стандартизирани класове

Дефиниция:

Един клас се нарича *стандартизиран* ако поддържа определено множество от стандартни член функции. Минималното множество от функции, които се изискват за да бъде един клас стандартен са следните:

- Копиращ конструктор (Copy constructor):

T (const T&)

- Оператор за присвояване (Assignment operator):

T& operator= (const T&)

- Оператор проверка за еквивалентност (Equality operator):

bool operator== (const T&) **const**

- Оператор проверка за нееквивалентност (Inequality operator):

bool operator!= (const T&) **const**

- Оператор проверка за по-малко (Less operator):

bool operator< (const T&) **const**

2.1 Стандартизирани класове

Свойства на основните функции:

- T a(b); осигуряващ (a == b); (копиращият конструктор да осигурява сравнението за равенство между обектите да връща истина).
- a = b; осигуряващ (a == b); (операторът за присвояване осигурява сравнението за равенство между обектите да е истина);
- a == a; (операторът за сравнение приложен за един и същ обект да осигурява истина);
- a == b ако b == a; (операторът за сравнение да е комутативен);
- (a == b) && (b == c) следва (a == c) (операторът за сравнение да е асоциативен);
- a != b ако ! (a == b) (операторите “различно” (!=) и “равно” (==) да са с инверсна логика).

2.1 Стандартизирани класове

Дефиниция:

Член функцията T::s(...) се нарича *еквивалентно представяне на оператора* == при следните условия:

За обекти от клас T – objA и objB

Ако е изпълнено objA == objB :

следва objA.s (...) == objB.s (...)

2.1 Стандартизирани класове

Пример:

```
class CXY {
  double m_x, m_y;
public:
  CXY(void) { m_x=0; m_y=0;}
  CXY( const double& xarg, const double& yarg)  { m_x=xarg; m_y=yarg;  }
  CXY(const CXY& xy){m_x=xy.m_x; m_y=xy.m_y; }
  void print() const {
    cout << "x = " << m_x << ", " << "y = " << m_y << endl;
  }
  bool operator == (const CXY& xy) const{
    return (m_x==xy.m_x && m_y==xy.m_y);
  }
  bool operator < (const CXY& xy) const {
    return (m_x<xy.m_x && m_y<xy.m_y);
  }
  bool operator != (const CXY& xy) const {
    return !(*this == xy);
  }
  double getX() const { return m_x; } // за четене
  double getY() const { return m_y; }
  void setX(const double& x ) { m_x=x; }
  void setY(const double& y ) { m_y=y; }
  ~CXY(void){}
};
```

2.1 Стандартизирани класове

Примерна тестова програма:

```
void main() {
  CXY oXY(1.0, 2.0);
  CXY oXY2(oXY);
  oXY.print();
  oXY2.print();
  cout<< endl;
  cout<< (oXY == oXY2 ?"true":"false") << endl;
  cout<< (oXY < oXY2 ?"true":"false") << endl;
  cout<< (oXY != oXY2 ?"true":"false") << endl;
  cout<< endl;
  oXY2 = oXY;
  cout<< (oXY == oXY2?"true":"false") << endl;
  cout<< (oXY < oXY2?"true":"false") << endl;
  cout<< (oXY != oXY2?"true":"false") << endl;
  cout<< endl;
  oXY=CXY(1.0, 2.0);
  oXY2=CXY(3.0, 4.0);
  cout<< (oXY == oXY2?"true":"false") << endl;
  cout<< (oXY < oXY2?"true":"false") << endl;
  cout<< (oXY != oXY2?"true":"false") << endl;
}
```

2.2 Обект функция

Дефиниция:

Обект функция се нарича обект от клас, който има дефиниран (или предекларирaн) *оператор за обръщение към функция* (operator()) .

Видове:

В зависимост от броя аргументи в библиотеката се разграничават три вида функции:

- Без аргументи, наричани още *генератори* (Generator);
- С един аргумент, наричани още *унарни функции* (Unary Function);
- С два аргумента, наричани още *бинарни функции* (Binary Function).
- Обозначенията на тези функции са респективно f(), f(x) и f(x,y). По принцип съществуват функции с повече аргументи, но те нямат практическо значение (не се използват) в алгоритмите на библиотеката STL.
- Обектите функции, които връщат логическа стойност имат специално предназначение, поради което се наричат *предикати*. Предикатите с два аргумента се наричат *бинарни предикати*.

2.2 Обект функция

Пример за инползване на глобална функция **rand** (като генератор)

- Пример за генериране на случайни числа в диапазон:

```
vector<int> V(100);
generate(V.begin(), V.end(), rand);
```

2.2 Обект функция

Пример: Предикатна функция за сравнение “по-малко”:

```
class less {
public:
   less (const double& val) : m_val (val) {}
   bool operator () (const double& val) {
      return val < m_val;
   }
private:
   double m_val;
};
```

2.2 Обект функция

Образуване на обект функция:

```
less less_than_five (5.);
// тестова програма
cout<< "2 е по-малко от 5:
    "<<(less_than_five(2.)?“да" : “не");
```

2.3 Темплейти (Шаблони)

Основа за STL са шаблоните, разгледани в лекция 5:

- 2.3.1 Темплейтни функции;
- 2.3.2 Темплейтни класове за контейнери;
- 2.3.3 Специализирани темплейти.

ВЪПРОСИ ?