Лекція 14

Тема лекції: Динамічна інформація про тип. Покажчики на функції-члени класу. Статичні члени класу. Розміщення об'єктів за заданими адресами. Ініціалізація членів об’єкту класу. Вкладені оголошення в класі.

Динамічна інформація про тип

Динамічна інформація про тип призначена для надання інформації про тип об’єкту під час виконання програми. Можна використовувати її для порівняння типів об'єктів і для доступу до рядків, у яких вони описуються.

Оператор *typeid* повертає об'єкт класу *Type_info*, що описує тип об’єкту. Можна використовувати *typeid* як для простих об'єктів вмонтованих типів, так і для класів, структур, масивів, власних типів даних для порівняння на еквівалентність (дорівнює/не дорівнює) або за абеткою. Для використання *typeid* потрібно включити до програми заголовочний файл *TYPEINFO.H.*

У класі *Typeinfo* є функції-члени, до яких можна звернутися. Наприклад, *name()* - функція, призначена для доступу до імені об’єкту:

```
int count;
cout << typeid(count).name();
```

Оператор виведення відобразить *int* - ім'я типу даних змінної *count*.

Порівняння типів даних об'єктів:

```
int i;
int j;
if(typeid(i)==typeid(j))
cout << "однакові типи";
```

Функція *before()* викликається для порівняння типів за абеткою

```
if(typeid(i).before(typeid(d)));
```

Якщо *typeid* не може створити об'єкт класу *Type_info*, оператор збуджує виключну ситуацію типу *Bad_typeid*. Отже, при використанні *typeid* необхідно запрограмувати оператор *catch*, що обробляє цю помилку.

Покажчики на функції-члени класу

Можна посилатися на функції-члени класу за допомогою покажчиків, але не так, як на звичайні функції. Функції-члени викликаються для об'єктів класу, і вони одержують прихований покажчик *this*. Тому адресація функцій-членів потребує нового підходу.

Покажчик на функцію-член пов'язаний з ім'ям її класу. Наприклад, для класу *A* можна оголосити покажчик на функцію-член у таким чином:

```
double(A::*myp)(void);
```

Це означає, що покажчик *myp* посилається на функцію-член класу *A*, що не має параметрів і повертає дійсне значення подвійної точності. Для оголошення *myp* покажчиком, що посилається на функцію, яка нічого не повертає і має два цілих параметри, можна записати:

```
void(A::*myp)(int,int);
```

У цих оголошеннях не вказується конкретно, на які функції-члени посилається покажчик *myp*, задається тільки вигляд функції, адреса якої може бути привласнена покажчику. Залишається створити об'єкт класу і привласнити адресу його члена покажчику, що має відповідний вигляд. Наприклад:

```
class TFirstClass
 {
  private:
   int count;
  public:
   TFirstClass()
    {
     count=0;
    }
   int Acs(void)
   {
    return count++;
   }
 };
int (TFirstClass::*myfp)(void);
main()
```

```
{
 int i;
 TFirstClass fc;
 cout fc.Acs(); //Виклик функції-члена звичайним засобом
 myfp = &TFirstClass::Acs; //Адреса функції
 cout << (fc.*myfp)();
 TFirstClass *fp = new TFirstClass;
 cout << (fp->*myfp)();
 return 0;
}
```

Тут *myfp* оголошений покажчиком на функцію-член класу *TFirstClass*, що не має аргументів і повертає ціле значення. Покажчик може посилатись на будь-які функції-члени класу, що мають подібний вигляд.

У функції *main()* оголошується об'єкт *fc* класу *TFirstClass*. Далі викликається функція-член *Acs()* класу *TFirstClass* звичайним чином.

Потім викликається ця ж сама функція за допомогою покажчика *myfp*. Для ініціалізації покажчика йому привласнюється адреса функції-члену *Acs()*. Можна замінити оголошення покажчика і привласнення одним оператором функції *main()*:

```
int (TFirstClass::*myfp)(void) = &TfirstClass::Acs();
```

При виклику за покажчиком функції-члена необхідно притримуватися таких правил:

- Звертатися до об’єкту класу.
- Взяти в круглі дужки виклик функції.

Наприклад, якщо *n* - ціла змінна, і *fc* - об'єкт класу *TFirstClass*, то запис

```
n=(fc.*myp)();
```

привласнить *n* результат функції *Acs()*. Двохсимвольний вираз “*.**” називається оператором посилання на член і служить для розміщення покажчика на функцію-член об’єкту класу. Круглі дужки необхідні тому, що оператор виклику функції *()* має більш високий пріоритет, ніж “*.**”. Раніше записаний оператор еквівалентний оператору

```
n=fc.Acs();
```

Можна також викликати функції-члени за допомогою покажчиків, коли

об'єкти адресуются іншими змінними-покажчиками. У записаній програмі спочатку покажчику *fp* привласнюється адреса нового об’єкту класу *TFirstClass*. Потім у виразі

```
(fp->*myp)();
```

в операторі виведення в потік викликається функція-член *Acs()* для об’єкта, адресованого покажчиком *fp*. Трьохсимвольний вираз ->* - інший вид оператора посилання на член. Круглі дужки знову необхідні. Оператор еквівалентний такому:

```
fp -> Acs();
```

У доповнення до адресації функцій-членів покажчиками, можна також посилатися на інші відкриті члени класу. Наприклад, якби клас *TFirstClass* мав відкритий член типу *double* на ім'я *balans*, можна було б оголосити покажчик на цей член так:

```
double TFirstClass::*dataPtr;
```

Покажчик *dataPtr* оголошується для адресації будь-яких відкритих даних-членів у класі *TFirstClass*, що мають тип дійсних значень подвійної точності. Для ініціалізації покажчика треба використовувати оператор:

```
dataPtr=&TFirstClass::balance;
```

Або можна оголосити покажчик і привласнити йому адресу члена *balance*:

```
double TFirstClass::*dataPtr=&TFirstClass:balance;
```

У будь-якому випадку, покажчик *dataPtr* тепер посилається на член *balance* класу *TFirstClass*. Насправді *dataPtr* не містить адреси розміщення в пам'яті, замість цього в ньому зберігається зсув, за яким член *balance* розміщується в об'єкті класу *TFirstClass*.

Залишається звернутися до адресованого члена через об'єкт класу:

```
fc.balance = 1234.56;
cout << fc.*dataPtr;
```

Запис *fc.*dataPtr* аналогічний запису, використаному для виклику функції-члена. У ньому використовується оператор посилання на член “.*”. Тут не потрібні додаткові круглі дужки, тому що вираз не містить конфліктуючих операторів.

Статичні члени класу

Статичні функції-члени зазвичай виконують глобальні дії або ініціалізують глобальні дані для всіх об'єктів класу. Для оголошення статичної функції-члену необхідно зазначити ключове слово *static* перед оголошенням функції-члена:

```
class TAnyClass
{
 public:
  static void Global(void);
};
```

Статичні функції-члени не можуть бути віруальними. Реалізуються вони так само, як і інші функції-члени:

```
void TAnyClass::Global(void)
 {
 ...
 }
```

Статичні функції-члени можуть виконувати будь-які оператори, але вони не одержують покажчика *this* і не мають доступу ні до яких даних-членів або функцій-членів класів. Для виклику статичної функції-члена в програмі використовується не об'єкт, а ім'я класу:

```
TAnyClass::Global();
```

Передбачається, що статична функція-член виконує дії, що стосуються всіх об'єктів типу *TAnyClass*. У класах можуть оголошуватись і статичні дані-члени:

```
class TAnyClass
 {
  private:
   static char c;
  public:
   char Get(void)
    {
     return c;
```

```
    }
};
```

Тут член *c* оголошений закритим статичним членом класу. Статичні дані-члени також можуть бути відкритими і захищеними. Існує лише одна копія *TAnyClass::c* незалежно від того, скільки визначається об'єктів класу *TAnyClass* у програмі. Статичні дані-члени повинні визначатися й ініціалізуватися в програмі за допомогою глобальних оголошень:

```
char TAnyClass::c='q';
```

Насправді ця глобальна змінна доступна тільки функціям-членам *TAnyClass*. Далі можна визначити оголошення класу *TAnyClass*, що використовує *c*:

```
TAnyClass x;
cout << x.Get();
```

Розміщення об'єктів за заданими адресами

Іноді може знадобитися розмістити об'єкт у задану ділянку пам'яті, наприклад у глобальний буфер у сегменті даних. Подібні об'єкти продовжують існувати і поза областю їхнього оголошення й іноді їх називають постійними об’єктами.

Приклад: розміщення об’єкту за заданою адресою.

```
class TPersist
 {
  private:
   int x,y;
  public:
   TPersist(int a, int b)
    {
     x=a;
     y=b;
    }
   ~TPersist()
    {
     x=0;
```

```
     y=0;
    }
   void *operator new(size_t,void *p)
    {
     return p;
    }
   friend ostream& operator<<(ostream &os, TPersist &p);
 };
char object[sizeof(TPersist)];
main()
 {
  TPersist *p=new(object) TPersist(10,20);
  cout << *p;  //Буфер object:
  cout << "адреса:" << &object;
  cout << "адреса *p:" << &(*p);
  p->TPersist::~TPersist();
  return 0;
 }
 ostream& operator<<(ostream& os,TPersist &p)
  {
   os << "x==" << p. x << ",y==" << p. y;
   return os;
  }
```

У класі *TPersist* оператор *new* перевантажується незвичним засобом. Замість того, щоб виділити пам'ять для покажчика, функція перевантаження оператора просто повертає покажчик *p* типу *void*. У програмі оператор *new* використовується для зберігання об’єкту класу *TPersist* у глобальному буфері. Вираз *new(object)* викликає перевантажений оператор *new*, передаючи йому адресу символьного буферу *object*. Оскільки перевантажений оператор *new* просто повертає переданий йому покажчик *p*, цей оператор у дійсності привласнює покажчику *p* адресу об’єкту, а також викликає конструктор класу *TPersist* для ініціалізації нового, виділеного в буфері, об’єкту.

У програмі відображаються адреси символьного буферу *object* і об’єкта в буфері. Ці адреси ідентичні, що свідчить про розміщення об’єкта в зазначеному місці. Якщо запустити програму з налагоджувача і простежити за вмістом

буфера *object* під час покрокового виконання коду, то можна виявити, що в глобальному буфері з'явилися значення 10 і 20. Це доводить, що перевантажений оператор *new* не користувався стандартними засобами виділення простору в купі.

У програмі наводиться приклад безпосереднього виклику деструктора. Цей прийом використовується для очищення об’єкту, створеного перевантаженим оператором *new*. C++ не розпізнає область дії об’єкту, збереженого в буфері і не може викликати деструктор класу автоматично.

Для віртуальних деструкторів, навіть у випадках з об’єктами похідних класів, можна використовувати таку форму виклику деструктора:

```
p->~TPersist();
```

Ініціалізація членів об’єкту класу

Зазвичай дані-члени об’єкту класу ініціалізуються у конструкторі. У якості альтернативи можна ініціалізувати члени так, як якби вони мали конструктори:

```
class A
 {
  private:
   int x,y;
  public:
   A():x(0),y(0){}
 };
```

У цій версії конструктора *A* членам *x* і *y* привласнюються нульові значення. Ініціалізатори даних-членів задаються до виконання тіла конструктора. Цей прийом корисний для ініціалізації об'єктів інших класів. Наприклад, припустимо, що клас *A* має член-об'єкт класу *B* і конструктору класу *B* необхідний один цілий параметр:

```
class B
 {
  int x;
  public:
```

```
    B(int n)
     {
      x=n;
     }
};
```

Якщо в класі *A* оголошений член типу *B*, конструктор *A* зобов'язаний ініціалізувати цей об'єкт. Оскільки конструктор не може бути викликаний безпосередньо, об'єкт повинен бути ініціалізований за допомогою альтернативного засобу.

```
class A
 {
  private:
   int x;
   B z;
  public:
   A(int n):z(n)
    {
     x=n;
    }
 };
```

Член *z* - об'єкт класу *B*, Конструктор класу *A* оголошений з цілим параметром *n*, що використовується для ініціалізації об’єкту *z* перед виконанням тіла конструктора. Якщо в класі є декілька членів-об'єктів, можна ініціалізувати їх, перерахувавши і розділивши комами.

Вкладені оголошення в класі

Всередині класу можуть оголошуватись не тільки дані і функції, але й інші елементи: *typedef, struct* і інші класи. Такі вкладені оголошення в класі бувають корисні, якщо вони тісно пов'язані з класом. Вони також дозволяють двом або більше класам оголошувати елементи з однаковою назвою, що відрізняються тільки іменем відповідних їм класів.

Вкладений *typedef* треба використовувати для експорту типів даних із класу. Наприклад у класі *A typedef* використовується для оголошення C_C для

int:

```
class A
{
public:
 typedef int C_C;
 static C_C ClassCount;
 A()
  {
   ClassCount++;
  }
 ~A()
  {
   ClassCount--;
  }
};
```

У класі оголошується статична змінна типу *C_C* з ім'ям *ClassCount*. Щоб виділити пам'ять для цієї змінної у програмі необхідно зробити глобальне оголошення:

```
A::C_C
A::ClassCount;
```

Ім'я класу передує типу даних і імені змінної. Оскільки в класі *A* існує *typedef C_C*, у програмі можна оголосити змінну *ClassCount*, не знаючи дійсного типу змінної. Без допомоги з боку програми клас може самостійно підраховувати число створених об'єктів його типу:

```
A *cp1,*cp2,*cp3;
cp1 = new A;
cp2 = new A;
cp3 = new A;
cout << A::ClassCount;
```

Останній оператор відобразить *3*.

Можливі складні вкладені оголошення. Всередині класу може оголошуватись структура:

```
class A
 {
  public:
```

```
   struct CS
    {
     int x;
     int y;
    };
     };
```

Потім у програмі може бути оголошений об'єкт типу *A::*CS:

```
A::CS k;
```

k можна використовувати, як звичайну структуру:

```
k.x=1;
k.y=2;
```

Також один клас може оголошуватись всередині іншого:

```
class A
 {
  public:
   class C
    {
     int x;
     int y;
     public:
      C()
       {
        x=1;
        y=2;
       }
      int GetX(void)
       {
        return x;
       }
      int GetY(void)
       {
        return y;
       }
    };
 };
```

Тут у класі *A* оголошується внутрішній клас *C*. У програмі можна

використовувати цей вкладений клас так:

```
A::C k;
cout << k.GetX();
cout << k.GetY();
```

Можливі й інші види вкладених оголошень у класі, такі як перерахування констант. До використання вкладених оголошень у класі застосовується правило: посиланню на вкладений елемент повинні передувати ім'я класу й оператор розширення області бачення.