## Лекція 9

**Тема лекції:** Керування пам’яттю. Почленне привласнення. Шаблонні функції та шаблонні класи

### *Перевантаження та керування пам'яттю*

Оператори *new* і *delete* можуть переванажуватися з метою завдання нових можливостей керування пам'яттю для об'єктів класу. Перевантаження *new* в оголошенні класу вказує компілятору, що відтепер відповідальність за виділення пам'яті для об'єктів класу лежить на програмісті. Для перевантаження *new* треба використовувати прототип функції вигляду

```
void *operator new(size_t size);
```

Надалі звернення до оператора *new* для виділення пам'яті об’єктам класу будуть перенаправлені до перевантаженої функції. Функція повинна повертати адресу області пам'яті, яка виділена об'єкту. Якщо необхідного вільного простору немає, функція повинна повертати *null*.

Приклад: Замість купи, програма виділяє пам'ять об’єктам у глобальному буфері.

```
class BrandNew
{
 private:
  int x;
 public:
  BrandNew();
  void *operator new(size_t size);
};
char buf[512];
int index=1;
main()
{
 cout << "\nСтворений локальний об'єкт";
 BrandNew b1;
 cout << "\n Розташування простору, що виділений new";
```

```
 BrandNew * b2 = new BrandNew;
 BrandNew * b3 = new BrandNew;
 BrandNew * b4 = new BrandNew;
 BrandNew * b5 = new BrandNew;
 return 0;
}
BrandNew::BrandNew()
{
 x=index;
}
void *BrandNew::operator new(size_t size)
{
 if(index>=512-sizeof(BrandNew)) return 0;
  else
   {
    int k=index;
    index+=sizeof(BrandNew);
    return &buf[k];
   }
}
```

Тут функція перевантаження оператора *new* перевіряє наявність простору в глобальному буфері. Якщо його немає, функція повертає 0, що є підставою для повернення оператором *new* значення *NULL*. Якщо вільний простір є, то глобальний індекс збільшується на розмір необхідної пам'яті, що передається функції в параметрі *size*. Потім функція повертає адресу виділеної ділянки пам'яті. Можна перевантажити *delete* для відслідковування видалення об'єктів, які адресовані покажчиками. Прототип функції перевантаження оператора *delete* повинен мати вигляд:

```
void operator delete(void *p);
```

де *p* посилається на об'єкт, що видаляється. Можна оголосити функцію ще й так:

```
void operator delete(void *p,size_t size);
```

У цьому випадку C++ буде додатково передавати функції число байтів у видаленому об'єкті. У розглянутій програмі для додавання функції перевантаження оператора *delete* до класу *BrandNew* потрібно оголосити

функцію-член у відкритій частині класу:

```
void operator delete(void *p)
 {
  cout << "\n Об'єкт, що видаляється - " <<p;
 }
```

Оператор виведення буде відображати адресу кожного об'єкту, що видаляється. Перевантажений оператор *delete* насправді не звільняє ніякої пам'яті, тому що об'єкти не запам'ятовуються в купі. Для видалення декількох об'єктів буде потрібно декілька операторів *delete:*

```
delete b2;
delete b3;
delete b4;
delete b5; //(перед оператором return у main())
```

Щоб використовувати засоби керування пам'яттю C++ для виділення простору в купі для об'єктів, у яких перевантажений оператор *new*, потрібно поставити перед оператором подвійну двокрапку. Наприклад, у рядку

```
BrandNew *x =::new BrandNew;
```

ігнорується перевантажений в об'єктах класу *BrandNew* оператор *new*. Аналогічно, *::delete* служить для звернення до оператора *delete*, що використовується за замовчуванням.

Зазвичай, якщо *new* не може виконати запит на виділення пам'яті, оператор повертає *NULL*. Для того, щоб змінити дії, що виконуються за замовчуванням, треба привласнити адресу функції-обробника помилок покажчику *_new_handler*, визначеному таким чином:

```
typedef void(*vfp)(void);
vfp _new_handler;
```

Використання *typedef* не обов'язкове, але воно полегшує читання оголошення. Функція-обробник помилок нічого не повертає і не має аргументів. Вона встановлюється за допомогою виклику функції *set_new_handler()*. Її прототип утримується у файлі *new.h*:

```
vfp set_new_handler(vfp);
```

Обробник помилок реалізується так само, як і будь-яка інша функція C++. Наприклад, можна аварійно завершити програму з повідомленням про помилку

нестачі пам'яті:

```
void memerr(void)
{
 fputs("\n\nOut of memory\n",stderr);
 exit(1);
}
```

Щоб ця функція викликалася у випадку виникнення помилок, пов'язаних із нестачею пам'яті, потрібно передати її адресу функції *set_new_handler()*:

```
set_new_handler(memerr);
```

***Почленне привласнення***

Привласнення об’єкту класу іншому об'єкту може призвести до того, що члени-покажчики на дані будуть посилатися на ту саму ділянку пам'яті. Розглянемо фрагмент програми:

```
TAnyClass V1;
TAnyClass V2(1, 2, "Рядок");
V1 = V2;
```

Після ініціалізації об’єкту *V1* значеннями за замовчуванням і визначення *V2* з явними аргументами, оператор привласнення копіює *V2* у *V1*. Але привласнення не викликає жодного конструктора класу, навіть конструктора копії. Якщо в класі оголошуються або успадковуються члени-покажчики на дані, скопійовані покажчики будуть посилатися на ту саму ділянку пам'яті.

Конструктор копії не вирішить цієї проблеми, оскільки задіяні об'єкти вже створені. Розв’язання проблеми - перевантаження оператора привласнення "=". Треба використовувати оголошення вигляду *void operator =(const class&)*. Наприклад:

```
void operator =(const TAnyClass &copy);
```

Тут параметр посилається на об'єкт, що копіюється. Реалізація цієї функції:

```
void TAnyClass::operator =(const TAnyClass &copy)
 {
  cout << "... .
```

```
 if(this= =&copy) return;
  delete s;
  i = copy.i;
  r = copy.r;
  s = strdup(copy. s);
}
```

В операторі *if* порівнюється прихований покажчик *this* і адреса параметру *copy,* на який він посилається. Це необхідно для запобігання спроби привласнення об’єкту самому собі. В другій частині оператора *if* звільняється ділянка пам'яті, що адресується покажчиком s. Подальший код перевантаженої функції аналогічний коду конструктора копії. Конструктор копії можна змусити працювати спільно з перевантаженим оператором "=". Для цього всередині конструктора копії треба викликати функцію перевантаження *operator=* привласненням об’єкту, що копіюється, об'єктові *this*.

```
TAnyClass::TAnyClass(TAnyClass &copy)
 {
  s=null;
  *this=copy;
 }
```

Дуже важливо ініціалізувати всі члени, які перевіряє перевантажений оператор привласнення. Наприклад, якщо покажчик містить сміття замість *NULL*, функція перевантаження оператора намагається звільнити пам'ять, що не була виділена.

### *Шаблони*

Подібно до того, як клас є схематичним описом побудови об’єкта, так і шаблон є схематичним описом побудови функцій і класів. Шаблони вказують лише специфікації функцій і класів, але не деталі дійсної реалізації.

### *Шаблонні функції*

Шаблони функцій зазвичай оголошують у заголовочному файлі і вони

мають такий загальний вигляд:

```
template <class T> void f(T param)
 {
  //тіло функції
 }
```

Шаблонна функція починається рядком *template<class T>*, який вказує компілятору, що *T* - обумовлений користувачем тип функції. Необхідний принаймі один параметр типу *T* для передачі функції даних для обробки. Можна задати покажчик (*T* param*) або посилку (*T& param*). Функція може оголошувати декілька параметрів і повертати значення типу *T*:

```
template <class T> T f(int a, T b)
{
...
}
```

У цій версії шаблонна функція *f()* повертає значення типу *T* і має два параметри - ціле *a* і невизначений об'єкт *b*.

Користувачі шаблону зазначать дійсний тип даних для *T*. Наприклад, у програмі можна задати такий прототип:

```
double f(int a, double b);
```

Необхідно забезпечити реалізацію цієї функції, якщо вона - звичайна. Але, оскільки *f()* - шаблонна функція, компілятор реалізує код функції, замінивши *T* на *double*.

Приклад:

```
template <class T> T max(T a, T b)
 {
  if(a>b) return a;
   else
    return b;
 }
```

Тут *T* має невизначений тип, об'єкт якого повертає шаблонна функція *max()*. Функції *max()* необхідні два аргументи типу *T*. Оператори функції - це схема для реальних операторів, що будуть згенеровані пізніше, коли буде заданий дійсний тип *T*. Можна використовувати більше ніж один невизначений тип:

```
template <class T1, class T2> T1 max (T1 a, T2 b)
```

У цій версії функція *max()* буде повертати значення типу *T1* і їй необхідні два аргументи: один типу *T1*, другий - типу *T2*. У програмі, що використовує шаблонні функції, необхідно зазначити їхні прототипи, які компілятор використовує для створення дійсних тіл функцій.

***Шаблонні класи***

Шаблон класу забезпечує скелет узагальненого класу для його наступної реалізації. Найчастіше шаблони класів оголошують у заголовочному файлі. Розглянемо приклад створення шаблонного класу. Створимо заголовочний файл db.h:

```
template <class T> class TDataBase
 {
  private:
   T *rp;
   int num;
  public:
   TDataBase(int n)
   {
    rp=new T[num=n];
   }
   ~TDataBase()
    {
     delete[] rp;
    }
   T &GetRecord(int recnum);
};
template <class T>
T &TDataBase<T>::GetRecord(int recnum)
 {
  T *crp=rp;
  if(0<=recnum&&recnum<num)
   while(recnum-- > 0)
    crp++;
```

```
 return *crp;
}
```

Тут оголошення шаблону класу виглядає так:

```
template<class T> class TDataBase
 {
  ...
 };
```

де *T* - невизначений тип, що задається користувачем шаблону. Замість *T* можна використовувати будь-який інший ідентифікатор. *T* можна згодом замінити будь-яким вбудованим типом, іншим класом, покажчиком і т.д. *TDataBase* - ім'я шаблонного класу. Для ясності заголовок шаблону краще оголошувати в окремих рядках:

```
template <class T>
class AnyClass
 {
  ...
 };
```

Можна також зазначити декілька типів:

```
template <class T1, class T2, class T3>
class TAnotherClass
 {
  ...
 };
```

У шаблоні класу <*class T*> тип *T* можна використовувати для оголошення даних-членів, типів значень, що повертаються функціями-членами, параметрів і інших елементів невизначених типів. Наприклад, в оголошенні класу *TDataBase* оголошується покажчик типу *T* з ім'ям *rp*:

```
T *rp;
```

На цій стадії дійсна природа *T* ще невідома, тому у програмі його можна використовувати тільки в загальних випадках. Проте, конструктор *TDataBase()* виділяє пам'ять для масиву об'єктів *T*, привласнюючи адресу масиву покажчику *rp* і заодно встановлюючи член *num* рівним необхідному числу записів. У деструкторі цей масив видалиться.

Функція-член *GetRecord()* повертає посилку на об'єкт типу *T*, що

ідентифікується номером запису *recnum*. Це ще одна узагальнена операція, що не потребує завдання визначеного типу *T*.

Функції-члени шаблонного класу можуть бути вбудованими або реалізуватися окремо. І оскільки ці функції-члени шаблонного класу - тільки оголошення, то їх можна помістити до заголовочного файлу. Заголовок функції-члена випереджається фразою *template <class T>*, потім йде тип значення, яке повертається, ім'я класу й оператор області бачення. Останніми оголошується сама функція та її тіло.

Розглянемо функцію *GetRecord()*. Вона повертає посилку на об'єкт типу *T*. Всередині функції покажчик *crp* типу *T* привласнюється покажчику того ж типу з ім'ям *rp*. Тобто, покажчик *crp* посилається на перший запис, збережений в об'єкті класу *TDataBase*. Далі перевіряється, чи відповідає параметр *recnum* допустимому діапазону значень. Якщо так, то цикл *while* декрементує параметр *recnum* до 0 і одночасно пересуває покажчик *crp* на один запис у базі даних.

У класі *TDataBase* допустимий вираз

```
crp++;
```

навіть незважаючи на те, що тип об’єкту, на який посилається *crp*, невідомий. Пізніше, коли буде заданий дійсний тип для шаблону класу, компілятор зможе згенерувати відповідні інструкції для збільшення покажчика *crp* на розмір *sizeof(T)*.

Далі в програмі повертається розіменоване значення покажчика *crp*, тобто посилка на будь-який об'єкт, що адресується crp. Це завершує оголошення шаблону класу, в якому не робиться ніяких припущень про те, якого типу дані в ньому запам'ятовуються.

Тепер використаємо цей шаблон для створення об’єкту класу бази даних, спроможного запам'ятати деяке число записів.

```
class TRecord
{
 private:
  char name[41];
 public:
  TRecord(const char *s)
```

```
  {
   Assign(s);
  }
  void Assign(const char *s)
   {
    strncpy(name,s,40);
   }
  char *getName(void)
   {
    return name;
   }
};
main()
{
 int rn;
 TDataBase <TRecord> db(3);
 TDataBase <Trecord*> dbp(3);
 TDataBase <TRecord> *pdb;
 TDataBase <Trecord*> *ppdb;
 cout<<"\n\nDatabase of 3 TRecords\n";
 db.GetRecord(0).Assign(“One”);
 db.GetRecord(1).Assign(“Two”);
 db.GetRecord(2).Assign(“Three”);
 for(rn=0;rn<=2;rn++)
 cout << db.GetRecord(rn).GetName() << '\n';
 cout<<"\n\nDatabase of 3 TRecord pointers\n";
 dbp.GetRecord(0)=new TRecord(“One string”);
 dbp.GetRecord(1)=new TRecord("Two string ");
 dbp.GetRecord(2)=new TRecord("Three string ");
     for(rn = 0; rn <= 2; rn++)
     cout<<dbp.GetRecord(rn)->GetName()<<'\n';
 cout<<"\n\nPointer to database of 3 TRecords\n";
 pdb=new TDataBase <TRecord>(3);
 pdb->GetRecord(0).Assign(“string one”);
 pdb->GetRecord(1).Assign("string two ");
 pdb->GetRecord(2).Assign("string three ");
     for(rn = 0; rn<=2; rn++)
```

```
        cout << pdb->GetRecord(rn).GetName()<<'\n';
    ppdb=new TDataBase <TRecord*>(3);
    ppdb->GetRecord(0) = new TRecord(“string1”);
    ppdb->GetRecord(1) = new TRecord("string2");
    ppdb->GetRecord(2) = new TRecord("string3");
        for(rn = 0; rn <=2; rn++)
        cout<<ppdb->GetRecord(rn)->GetName()<<'/n';
        getch();
         return 0;
}
```

У програмі демонструється 4 засоби створення об’єкту класу за допомогою шаблону класу. У рядку

```
TDataBase <TRecord> db(3);
```

визначається об'єкт з ім'ям *db* шаблонного класу типу *TDataBase* і задається *TRecord* у якості класу, що заміщує *T* у шаблоні. Вираз у дужках (3) - ініціалізатор *db*, переданий конструктору класу *TDataBase*. Для створення бази даних необов'язково використовувати класи. Наприклад, створимо базу даних із 100 дійсних значень подвійної точності:

```
TDataBase <double> dbd(100);
```

Оскільки клас *TDataBase* написаний для зберігання об'єктів довільних типів, він називається контейнерним класом. Найкращі контейнерні класи є цілком узагальненими.

Далі в програмі створюються інші екземпляри шаблонного класу:

- об'єкт *dbp* класу *TDataBase*, що складається з 3х покажчиків на *Trecord*;
- покажчик *pdb* на об'єкт класу *TDataBase* із незаданим числом об'єктів типу *TRecord;*
- покажчик *ppdb* на базу даних покажчиків на *TRecord*.

У програмі об'єкт класу *TDataBase* використовується так само, як і будь-який інший не шаблонний об'єкт.

В усіх наведених прикладах не треба інформувати компілятор про типи даних, використаних класом *TDataBase*, за допомогою приведення типів.

Рекомендується не писати шаблони з чистого аркуша. Замість цього пишеться клас, що використовує задані об'єкти. Потім, після його

налагодження, можна перетворити його на універсальний шаблон. Це допомагає визначити істинно узагальнені властивості класів.