Лекція 7

Класи

У цій лекції...

7.1. Члени класу і керування доступом
7.2. Конструктори і деструктор
7.3. Дружні функції і класи

Як ми уже відзначали, мова C++ є універсальною. У першій частині курсу розглянуті його імперативні (чи процедурно-орієнтовані) властивості, що, в основному, забезпечуються засобами мови C і деякими додатковими механізмами (посиланнями, inline-функціями, операторами приведення типів і ін.). Основу парадигми імперативного програмування складає *алгоритм*, що *обробляє дані*. У рамках цього підходу алгоритм і дані відділені друг від друга. Головна роль в імперативному програмуванні приділяється алгоритму, а дані знаходяться в тіні.

Загалом імперативний підхід можна описати в такий спосіб. Програма розбита на модулі, що реалізують різні алгоритми, як правило вузькоспеціалізовані. Модулі одержують дані, обробляють їх і передають іншим модулям. Координація роботи здійснюється головним модулем: він викликає підпрограми, передає їм дані і повертає результат програми. Модулі є відносно самостійними. Вони можуть викликати один одного, самих себе і навіть головний модуль, вводити з зовнішнього світу додаткові дані і виводити результати.

Потреби програмування, поставленого на промислову основу, неможливо повною мірою задовольнити за допомогою імперативного програмування. Навіть усередині однієї команди необхідно погоджувати деталі представлення даних і реалізації алгоритмів, спрямованих на рішення загальної задачі. Програмісти, що не погодили свої дії, ризикують створити несумісні модулі. Незначне відхилення від загального плану може привести до серйозних наслідків. У силу цих причин виникла необхідність створити таку концепцію програмування, що підвищила б продуктивність і надійність роботи програмістів, дозволивши застосувати конвеєрні методи роботи. Так виникла парадигма *об'єктно-орієнтованого програмування*.

В основу *об'єктно-орієнтованого* програмування покладене поняття *об'єкта* і принципи *інкапсуляції*, *приховання інформації*, *абстракції даних*, *успадкування* і *поліморфізму*.

Основна ідея об'єктно-орієнтованого програмування полягає в тім, що всі обчислення відбуваються усередині об'єктів, що обмінюються між собою повідомленнями. У деякому сенсі об'єктно-орієнтоване програмування можна вважати вищим ступенем розвитку модульного програмування; тільки роль модулів тут грають об'єкти. В імперативному програмуванні вся інформація оброблялася усередині модулів, причому дані не залежали від них. Тепер об'єкти наділені можливістю не тільки зберігати, але й обробляти дані. Наприклад, якщо в імперативних програмах структура являє собою просту сукупність записів, то в об'єктно-орієнтованих — це активний об'єкт, що не просто зберігає інформацію, але й обробляє її. У цьому і полягає *принцип інкапсуляції*.

Отже, *об'єкт — це об'єднання даних і алгоритмів, що обробляють ці дані*. Тепер ми можемо занурити масив і метод його сортування в деякий об'єкт, а потім звернутися до цього об'єкта з проханням упорядкувати елементи масиву і видати результат. Основна перевага такого підходу — гнучкість. Якщо даний модуль є частиною більш складної конструкції, необхідно лише погодити способи його оголошення з зовнішнім світом — те, що називають інтерфейсом. Його внутрішній світ інших програмістів не стосується. Як зберігаються дані усередині модуля і як вони обробляються, повинний знати тільки творець даного об'єкта. А якщо інші програмісти виявляться занадто безцеремонними і захочуть змінити внутрішню структуру об'єкта? Як захистити його від стороннього втручання? Для цього використовується принцип *приховання інформації*. В об'єктно-орієнтованому програмуванні внутрішню структуру об'єкта можна розділити на три розділи: відкритий, закритий і захищений. Не вдаючись у передчасні подробиці, помітимо, що відкритий розділ доступний усім сутностям програми (інакше кажучи, видний з будь-якої її точки), закритий розділ — лише внутрішнім сутностям об'єкта, а захищений — тільки об'єктам, створеним з даного об'єкта (спадкоємцям).

Для того щоб програма була ефективною, необхідно, щоб об'єкти добре відповідали розв'язуваній задачі. Однак згодом об'єкт може морально застаріти. Наприклад, може з'явитися новий надшвидкісний алгоритм сортування. Що робити? Створити новий об'єкт? Змінити старий? Для того щоб полегшити задачу, вирішили розділити схему об'єкта і його реалізацію. Це дозволяє програмісту описати, *що* може робити об'єкт, не конкретизуючи, *як* це робити. У цьому полягає принцип *абстракції даних*. Принципова схема об'єкта не повинна залежати від її конкретного наповнення. Необхідно лише вказати, що об'єкт повинний містити певну операцію. Якщо алгоритм знадобиться змінити, ми модифікуємо його реалізацію, але загальна структура об'єкта від цього не постраждає.

Уявимо собі, що нам знадобилося створити новий об'єкт, що володів би можливостями старого, але вмів би робити і щось нове. Що робили в минулому? Копіювали зміст старого модуля і додавали в нього нові функції.

Тепер усе це можна робити набагато простіше! Ми просто створюємо об'єкт, що *успадковує* усі властивості свого попередника, і додаємо в нього нові можливості. Цей механізм називається *успадкуванням*.

Об'єкт може одержувати різноманітну інформацію. Зрозуміло, для її обробки можна заздалегідь передбачити його реакцію, написавши відповідні функції. Однак в об'єктно-орієнтованих мовах існує можливість *перевантажувати* операції і функції, що дозволяє об'єкту самому конкретизувати їхній зміст у ході виконання програми. Що дозволяє спростити програмування. Наприклад, якщо перевантажити оператор +, то операція `a + b` може мати різний сенс. Якщо ці об'єкти — числа, застосовується звичайна операція додавання, якщо матриці — матрична (конечно, для цього необхідно визначити цю операцію). У цьому (і не тільки) полягає сутність *поліморфізму*. У принципі поліморфізм виявляється в трьох ситуаціях: при перевантаженні функцій, при перевантаженні операторів і при пізнім зв'язуванні. Однак ці терміни нам ще має бути освоїти.

Описуючи принципи об'єктно-орієнтованого програмування, ми оперували поняттям об'єкт. Однак об'єкт — це фізична сутність, що виникає при виконанні програми, тобто сукупність комірок пам'яті, що зберігають дані і код. Для того щоб створити об'єкт, необхідна його схема: які дані він містить, які функції обробляють ці дані, як організований доступ до цих даних і функцій, що називаються *членами об'єкта*. У мові C++ схемою об'єкта називається *клас*.

7.1. ЧЛЕНИ КЛАСУ І КЕРУВАННЯ ДОСТУПОМ

Для оголошення класу в мові C++ призначене ключове слово `class`. Усе, що розташовано між фігурними дужками, що слідують за цим ключовим словом, являє собою *тіло класу*, що складається з його *членів*. Варто пам'ятати, що клас — це логічна схема об'єкта. Отже, виділення пам'яті відбудеться лише тоді, коли об'єкт буде *визначений*. Розглянемо загальний вид оголошення класу.

class *ім'я*{
члени класу;
private:
члени класу;
protected:
члени класу;
public:
члени класу;
} <*список об'єктів*>;

Правила, за якими утворюється клас, нічим не відрізняються від правил, установлених для структур. (У сучасних компіляторах між структурами і класами взагалі немає різниці, тобто ключові слова `struct` і `class` є взаємозамінними.)

Визначати об'єкти відразу після оголошення класів не обов'язково — це можна зробити в придатному місці програми. Як показане вище, тіло класу розділяється на три розділи, позначені наступними ключовими словами (специфікаторами доступу).

```
public    // Відкритий розділ
private   // Закритий розділ
protected // Захищений розділ
```

Ключове слово `public` позначає розділ, у якому розміщаються функції і дані, доступні з будь-якої точки програми, — відкриті члени. Ключове слово `private` оголошує розділ, у якому розташовані закриті члени класу, що є доступними лише функціям-членам самого класу, а також дружнім функціям і класам. Зверніть увагу на те, що функції-члени можуть знаходитися в ніби безіменному розділі. Насправді, за замовчуванням функції і дані, оголошені в такому розділі, є закритими. Зміст специфікатора `protected` буде розкритий пізніше при описі спадкування класів.

Порядок проходження специфікаторів доступу не регламентується — вони можуть повторюватися неодноразово й у будь-якому порядку. Кожен наступний специфікатор припиняє дія попереднього. Як правило, відкриті члени оголошуються останніми, але це стосується питань стилю, а не до синтаксичних правил.

Розглянемо конкретний приклад.

Доступ до члена класу

```
#include <stdio.h>

class TComplex
{
   double Re;
   double Im;
public:
   void print() {printf("Z = %lf + i*%lf \n",Re,Im);}
   void init()  {Re = 1; Im = 2;}
};

int main()
```

```
{
   TComplex Z;
   // Z.Re = 1;     Помилка! Член Re знаходиться в закритому розділі!
   // Z.Im = 2;     Помилка! Член Im знаходиться в закритому розділі!
   Z.init();        // Виклик функції-члена init()
   Z.print();       // Виклик функції-члена print()
   return 0;
}
```

На екран виводиться число

```
Z = 1.000000 + i*2.000000
```

У цій програмі оголошений клас `TComplex`, що описує структуру комплексного числа і дії над ним. У даному випадку дій небагато: задати початкове значення і вивести число на екран. Визначення інших операцій відкладемо на майбутнє. Поки нас цікавить два питання: як здійснюється доступ до членів класу і як створюється об'єкт класу.

Члени `Re` і `Im` (дійсна і уявна частини комплексного числа) у класі `TClass` є закритими. З цієї причини звернутися до них з функції `main()` неможливо. Для того щоб забезпечити доступ до членів `Re` і `Im`, у відкритому розділі класу розміщені функції `init()` і `print()`. Вони є членами класу і, отже, мають прямий доступ до будь-якого іншого членуа класу.

Функції `print()` і `init()` відносяться до різних категорій. Функція `print()` просто виводить на екран значення членів об'єкта `Z`, не змінюючи їх. Такі функції-члени називаються *функціями доступу*.

Функція `init()` привласнює полям `Re` і `Im` задані значення. Такі функції-члени іменуються такими, *що модифікують*. Обидві функції `init()` і `print()` утворюють *інтерфейс* класу.

Зверніть увагу на те, як відбувається звертання до членів класу. Для цього використовується оператор доступу до членів класу «`.`».

```
   Z.init();        // Виклик функції-члена init()
Z.print();       // Виклик функції-члена print()
```

Якщо в програмі оголошений вказівник на об'єкт класу, використовується оператор посилання на члени класу «`->`». Проілюструємо його застосування наступним прикладом.

Посилання на член класу

```
   #include <stdio.h>
#include <malloc.h>

class TComplex
{
   double Re;
   double Im;
public:
   void print() {printf("Z = %lf + i*%lf \n",Re,Im);}
   void init()  {Re = 1; Im = 2;}
};

int main()
{
   TComplex* p = (TComplex*)malloc(sizeof(TComplex));
   // Z.Re = 1;     Помилка! Член Re знаходиться в закритому розділі!
   // Z.Im = 2;     Помилка! Член Im знаходиться в закритому розділі!
   p->init();       // Виклик функції-члена init()
   p->print();      // Виклик функції-члена print()
   return 0;
}
```

Клас не може містити власний об'єкт, оскільки опис об'єкта повинен передувати його створенню, а в оголошенні

```
class TClass
{
  TClass Z;   // Не можна!
  TClass* p;  // Можна!
  ...
};
```

опис класу ще не довершено. Правда, вказівник на об'єкт класу може міститися усередині самого класу.

Крім того, при оголошенні полів класу не можна користатися ключовими словами `auto`, `extern` і `register`.

7.1.1. Структури й об'єднання як різновиди класів

У мові C++ між класами, з одного боку, і структурами й об'єднаннями, з інший, не багато розходжень. Розглянемо деякі з них.

Для початку подамо наш клас `TComplex` у вигляді структури. Для цього досить просто замінити ключове слово `class` словом `struct`.

Структура — найпростіший клас

```
  struct TComplex
{
   double Re;
   double Im;
public:
   void print() {printf("Z = %lf + i*%lf \n",Re,Im);}
   void init()  {Re = 1; Im = 2;}
};
```

На структури поширюються усі властивості класів, навіть зв'язані зі спадкуванням. Таким чином, ключові слова `class` і `struct` цілком еквівалентні. Однак, щоб не виникало плутанини, структурою звичайно називають сукупність даних (без функцій-членів), а класом — структуру з функціями-членами. Єдиною відмінністю між структурами і класами є рівень доступу до їхніх членів, прийнята за замовчуванням: якщо рівень доступу (`public`, `private` чи `protected`) не зазначений явно, усі члени структури вважаються відкритими, а всі члени класу — закритими.

З класами також тісно зв'язані *об'єднання*. Оскільки цей вид структури більш складний і зв'язаний з досить хитромудрим розподілом пам'яті, на нього поширюються деякі обмеження. Зокрема, до об'єднань не можна застосовувати механізм успадкування; вони не можуть містити віртуальні функції, а також статичні змінні і посилання. Крім того, об'єднання не повинні містити об'єкти класів з перевантаженим оператором присвоювання, а також явно визначеними конструкторами і деструктором.

Не варто намагатися реалізувати клас `TClass` у виді об'єднання.

```
  union TComplex
{
   double Re;
   double Im;
public:
   void print() {printf("Z = %lf + i*%lf \n",Re,Im);}
   void init()  {Re = 1; Im = 2;}
};
```

У цьому випадку змінні `Re` і `Im` будуть зберігатися в одній і тій же області пам'яті, перекриваючи один одного. Тому при виклику функції `print()` на екран замість числа `Z = 1.00000 + i*2.000000` буде виведене число `Z = 2.000000 + i*2.000000`.

Цікаво, а скільки пам'яті займає об'єкт класу, структури й об'єднання? Як і випливало очікувати — 17, 17 і 8 байт відповідно. Звідси випливає, що скільки б членів-функцій ні містив клас, на розмір його об'єкта впливає лише кількість і тип його змінних-членів. Розмір функцій-членів не враховується. Крім того, розмір об'єкта «порожнього класу» (тобто класу, що не має ни члена) також не є однозначно визначеним: різні компілятори повертають різні розміри (1 чи 2 байти).

7.1.2. Функції-члени класу, що підставляються

Повернемося до нашого класу `TComplex`. Функції-члени `print()` і `init()` мають дуже простий зміст і невеликі по обсязі. Отже, має сенс зробити їх що підставляються. Як відомо, для цього варто поставити перед їх визначенням ключове слово `inline`.

```
   inline void print() {printf("Z = %lf + i*%lf \n",Re,Im);}
inline void init()  {Re = 1; Im = 2;}
```

Тепер ці функції будуть не викликатися, а підставлятися у відповідне місце програми. Для членів класу все набагато простіше. Якщо визначення функції-члена міститься усередині класу, сама функція автоматично *вважається такою, що підставляється*, або *inline-функцією* (хоча і не *стає* нею автоматично). Ключове слово `inline` для цього вказувати необов'язково, хоча і не заборонено. Зрозуміло, на функцію-член, що підставляються, поширюються ті ж обмеження, що і на звичайні inline-функції, тому, зокрема, не слід розміщати усередині оголошення класу великі функції — це знижує наочність класу і не робить функцію дійсно inline.

7.1.3. Визначення функцій-членів поза класом

Як ми уже відзначили, визначення деяких функцій-членів класу варто розміщати за межами його оголошення, залишаючи всередині лише їх прототипи. Остання вимога обов'язкова! Саме воно робить конкретну функцію членом класу.

Для того щоб зв'язати визначення функції-члена з класом, якому вона належить, перед її ім'ям вказується ім'я класу й оператор дозволу області видимості `::`. Перепишемо нашу програму, виходячи з нових розумінь.

Визначення функції-члена поза класом

```
   #include <stdio.h>
#include <malloc.h>

class TComplex
{
   double Re;
   double Im;
public:
   void print();
   void init();
};

int main()
{
   TComplex* p = (TComplex*)malloc(sizeof(TComplex));
   // Z.Re = 1;     Помилка! Член Re знаходиться в закритому розділі!
   // Z.Im = 2;     Помилка! Член Im знаходиться в закритому розділі!
   p->init();       // Виклик функції-члена init()
   p->print();      // Виклик функції-члена print()
   return 0;
}
 void TComplex::print() {printf("Z = %lf + i*%lf \n",Re,Im);}
void TComplex::init()  {Re = 1; Im = 2;}
```

7.1.4. Вказівник this

При виклику функція-член одержує вказівник на зухвалий об'єкт. Цей вказівник позначається ключовим словом `this`. Отже, вказівник `this` на об'єкт класу `T` має тип `T*`.

Повернемося до програми, у якій створюється об'єкт класу `TComplex`. Явне застосування вказівника `this` у даному випадку зовсім зайве, але наочно демонструє механізм, що дозволяє функції-члену класу з'ясувати, який об'єкт її викликає.

Доступ до члена класу за допомогою вказівника this

```
#include <stdio.h>
#include <malloc.h>

class TComplex
{
    double Re;
    double Im;
public:
    void print();
    void init();
};

int main()
{
    TComplex* p = (TComplex*)malloc(sizeof(TComplex));
    p->init();       // Виклик функції-члена init()
    p->print();      // Виклик функції-члена print()
    return 0;
}
void TComplex::print() {printf("Z = %lf + i*%lf \n",this->Re,this->Im);}
void TComplex::init()  {this->Re = 1; this->Im = 2;}
```

Однак необхідно мати у виді, що змінна `this` — не звичайний вказівник. Зокрема, їй не можна нічого привласнювати і вона не має адреси.

Неправильне використання вказівника this

```
  #include <stdio.h>
#include <malloc.h>

class TComplex
{
   double Re;
   double Im;
public:
   void print();
   void init();
   void noncorrect(TComplex*);
};

int main()
{
   TComplex* p = (TComplex*)malloc(sizeof(TComplex));
   p->init();       // Виклик функції-члена init()
   p->print();      // Виклик функції-члена print()
   TComplex* q = (TComplex*)malloc(sizeof(TComplex));
   q->noncorrect(p);
   return 0;
}
void TComplex::print() {printf("Z = %lf + i*%lf \n",this->Re,this->Im);}
void TComplex::init()  {this->Re = 1; this->Im = 2;}
void TComplex::noncorrect(TComplex* p)
{
   this = p;    // Помилка: вказівнику this нічого не можна привласнювати!
   printf("&this = %p \n",&this); // Помилка: вказівник this не має адреси!
}
```

Вказівник `this` відіграє важливу роль при перевантаженні операторів (зокрема, для перевірки самоприсвоювання). При цьому варто пам'ятати, що вказівник `this` завжди вважається *константним* і ніколи не передається статичним функціям-членам.

7.1.5. Статичні змінні—члени класу

Статичні змінні-члени мають особливі властивості. Вони доступні всім об'єктам класу, не належачи нікому з них окремо. Якби клас не був абстракцією, можна було б сказати, що статична змінна належить класу, а не об'єктам. У реальності це означає, що статична змінна-член класу існує в єдиному екземплярі незалежно від кількості об'єктів. Особливо важно, що статична змінна ініціалізується ще до створення першого об'єкта класу.

Оголошення статичної змінної-члена класу має декілька тонкощів. По-перше, замало оголосити статичну змінна-член — необхідно виділити для неї пам'ять. Для цього її оголошення слід повторити поза тілом класу, в області визначення глобальних змінних. Визначення статичної змінної-члена можна сполучити з її ініціалізацією. По-друге, повторювати ключове слово `static` при визначенні статичної функції не можна! Воно повинно бути присутнім тільки в тілі класу! Розглянемо приклад, у якому змінні `Re` і `Im` оголошені статичними членами класу `TComplex`.

Оголошення статичних членів класу

```
  #include <stdio.h>
#include <malloc.h>

class TComplex
{
   static double Re;
   static double Im;
public:
   void print();
   void init();
};


double TComplex::Re=0.0;
double TComplex::Im=0.0;

int main()
{
   TComplex Z;
```

```
    printf("sizeof = %d \n",sizeof(Z));
    Z.init();       // Виклик функції-члена init()
    Z.print();      // Виклик функції-члена print()
    return 0;
}

void TComplex::print() {printf("Z = %lf + i*%lf \n",Re,Im);}
void TComplex::init()  {Re = 1; Im = 2;}
```

Цікаво, що розмір класу `TComplex` змінився. Тепер він дорівнює лише одному байту (як у зовсім порожнього класу).

Статичні змінні зручно використовувати як лічильник об'єктів, а також як індикатори на зразок «Зайнято» — «Вільно». Об'єкти можуть по черзі змінювати значення статичної змінної, повідомляючи своїм спадкоємцям, що вони виконали свою місію.

Розглянемо програму, у якій створюються і виводяться на екран 25 комплексних чисел. Зверніть увагу на синтаксичну конструкцію, прийняту для звертання до статичної змінної-члена: `TComplex::counter`. Це підкреслює, що статична змінна-член не належить жодному об'єкту. До речі, це не єдина форма звертання до неї. Можна також використовувати вираз `TComplex.counter`. Для того щоб підкреслити, що кожен об'єкт має доступ до статичного змінна-члену, можна оператор `TComplex.counter++` замінити оператором `Z.counter++`.

Використання статичних членів класу

```
   #include <stdio.h>
#include <malloc.h>

class TComplex
{
   double Re;
   double Im;
public:
   static int counter;
   void print();
   void init(double x, double y);
};


int TComplex::counter=0;


int main()
{
   for (int i = 1; i <= 5; i++)
   for (int j = 1; j <= 5; j++)
   {
     TComplex Z;
     Z.init((double)i,(double)j);  // Виклик функції-члена init()
     Z.print();                    // Виклик функції-члена print()
     TComplex::counter++;          // Інкрементація статичної
                                   // змінна-члена
   }
   printf("Counter = %d\n",TComplex::counter);
   return 0;
}

void TComplex::print() {printf("Z = %lf + i*%lf \n",Re,Im);}
void TComplex::init(double x, double y)  {Re = x; Im = y;}
```

7.1.7. Статичні функції-члени класу

Статичними можуть бути не тільки змінні-члени, але і функції-члени класу. У цьому випадку вони втрачають доступ до нестатичних членів класу і можуть звертатися тільки до статичних змінних-членів і функцій. Крім того, вони не можуть бути віртуальними, одержувати вказівник `this`, а також не повинні мати кваліфікаторів `const` і `volatile`.

Розглянемо програму, у якій для висновку значення статичного лічильника застосовується статична функція `TComplex::HowMuch()`.

Використання статичних функцій-членів класу

```
  #include <stdio.h>
#include <malloc.h>

class TComplex
{
   double Re;
   double Im;
public:
   static int counter;
   void print();
   void init(double x, double y);
   static void HowMuch();
};

int TComplex::counter=0;

int main()
{
   for(int i = 1; i <= 5 ; i++)
   for(int j = 1; j <= 5 ; j++)
   {
     TComplex Z1;
     Z1.init((double)i,(double)j);  // Виклик функції-члена init()
     Z1.print();                    // Виклик функції-члена print()
     TComplex::counter++;           // Інкрементація статичного лічильника
   }

   TComplex::HowMuch();             // Виклик статичної функції
   return 0;
}

void TComplex::print() {printf("Z = %lf + i*%lf \n",Re,Im);}
void TComplex::init(double x, double y)  {Re = x; Im = y;}
void TComplex::HowMuch(){printf("Counter = %d\n",TComplex::counter);}
```

Застосування статичних функцій обмежено ініціалізацією статичних змінних, котру необхідно виконати до створення реальних об'єктів. Наприклад, вони лежать в основі корисної ідіоми «іменованих конструкторів», яку ми розглянемо нижче.

7.1.7. Константні функції-члени

Для захисту від модифікації члени класу, як правило, поміщають у закритий розділ і оголошують константними. Однак, якщо надалі нам знадобиться змінити константне поле, прийдеться виконувати приведення типу за допомогою оператора `const_cast`. Утім, існує ще один спосіб захистити поля — розділивши функції-члени на функції доступу і функції, що модифікують. Перші функції не мають права змінювати вміст полів. Для цього них оголошують константними. Друга група функцій може вільно привласнювати полям нові значення.

Наприклад, у наведеному нижче лістингу функція `print()` є константною і не може присвоїти членам `Re` і `Im` жодних значень. Для цього між її заголовком і тілом поставлене ключове слово `const`. Зверніть увагу на те, що функція `change()` теж оголошена константною. Отже, вона також не повинна змінювати зміст полів класу. Однак ми зняли захист з одного з членів класу — `Re` — поставивши перед його оголошенням ключове слово `mutable`. Тепер будь-яка константна функція зможе змінити значення цього поля.

Застосування ключових слів const і mutable

```
  #include <stdio.h>

class TComplex
{
   mutable double Re;
   double Im;
public:
   TComplex():Re(0),Im(0){};
   void change(double x) const {Re = x};
   void print() const { printf("Z = %lf + i*%lf \n",Re,Im);}
}Z;

int main()
{
```

```
    TComplex X;
    X.print();
    X.change(1.0);
    X.print();

    return 0;
}
  Результат приведений нижче.
   Z = 0.000000 + i*0.000000
Z = 1.000000 + i*0.000000
```

7.1.8. Вкладені класи

Мова C++ дозволяє створювати *вкладені класи*. Це дозволяє сховати внутрішній клас від зовнішнього світу, обмеживши його область видимості зовнішнім класом.

Проілюструвати ця властивість досить просто. У попередній програмі перенесемо оголошення класів `TDouble` і `TInteger` усередину класу `TComplex`.

Визначення внутрішніх класів

```
   #include <stdio.h>

class TComplex
{
 class Double
 {
     double x;
  public:
     Double(double y):x(y){printf(" Ctor Double \n");}
 };

 class Integer
 {
        int n;
 public:
     Integer(int m):n(m){printf(" Ctor Integer \n");}
 };

   Double Re;
   Integer Im;
public:
   void print();
   TComplex(Integer k, Double v):Re(v), Im(k){};
};

int main()
{
   TComplex Z(10.0, 20);
   Z.print();

   return 0;
}

void TComplex::print() {printf("Z = %lf + i*%d \n",Re,Im);}
```

Внутрішній (чи локальний) клас можна визначити не тільки усередині класу, але й усередині функції, наприклад усередині функції `main()`.

Визначення локального класу

```
   #include <stdio.h>

int main()
{
class TComplex
{
   double Re;
   int Im;
public:
   void print(){printf("Z = %lf + i*%d \n",Re,Im);};
   TComplex(int k, double v):Re(v), Im(k){};
};
```

```
    TComplex Z(10.0, 20);
    Z.print();

    return 0;
}
```

Оскільки функції не можна визначати усередині інших функцій, їх визначення необхідно занурити усередину локального класу. Тільки під цією оболонкою вони доступні для зовнішньої функції.

У силу обмеженості області видимості локальних класів, їм недоступні інші локальні змінні, оголошені усередині функції, крім статичних і зовнішніх. Крім того, усередині локальних класів неможливо визначити статичні змінні.

7.1.9. Вказівники на членів класу

У деяких ситуаціях можна одержати доступ до функції-члена класу, не вказуючи імен ні об'єкта, ні функції. Механізм, що дозволяє це зробити, ґрунтується на використанні вказівників на член класу. Зрозуміло, щоб викликати функцію, необхідний реальний об'єкт класу, однак доступ до його функції-члену програма одержує за допомогою непрямої адресації. Для цього спочатку необхідно створити об'єкт у динамічній пам'яті й установити на нього вказівник. Потім з'являється вказівник на функцію-член класу. Тепер, передаючи як параметри вказівники на об'єкти і функції-члени, ми можемо викликати кожну функцію-член класу, не називаючи її явно. Природно, її оголошення повинне точно відповідати оголошенню вказівника на функцію-член.

Виклик функції-члена класу через вказівник на неї

```
   #include <stdio.h>
#include <malloc.h>

class TComplex
{
   double Re;
   double Im;
public:
   void print(){printf("Z = %lf + i*%lf \n", Re, Im);}
   void init(){Re = 1; Im = 1;}
};

typedef void (TComplex::*pMem)(); // Оголошення вказівника на функцію-член
void call(TComplex* s, pMem p);

int main()
{
   TComplex* pObj = (TComplex*)malloc(sizeof(TComplex));
   pMem p;
   p= &TComplex::init;
   call(pObj,p);
   p = &TComplex::print;
   call(pObj,p);

   return 0;
}

void call(TComplex* s, pMem p)
{
  (s->*p)();
}
```

Не слід плутати вказівник на функцію-член з вказівником на звичайну функцію. Вказівник на функцію-член задає не адресу, а зсув члена класу щодо адреси об'єкта.

Вказівники можна встановлювати не тільки на функції-члени, але і на дані-члени. Розглянемо як приклад наступну програму.

Використання вказівників на члени класу: перший варіант

```
   #include <stdio.h>
#include <malloc.h>

class TComplex
{
public:
   double Re;
   double Im;
};
```

```
 double TComplex::*pMem;    // Вказівник на член класу, що має тип double

void initRe(TComplex*, double TComplex::*pMem);
void initIm(TComplex*, double TComplex::*pMem);
void printRe(TComplex, double TComplex::*pMem);
void printIm(TComplex, double TComplex::*pMem);

int main()
 {
    TComplex z;
    pMem = &TComplex::Re;
    initRe(&z,pMem);
    pMem = &TComplex::Im;
    initIm(&z,pMem);
    pMem = &TComplex::Re;
    printRe(z,pMem);
    pMem = &TComplex::Im;
    printIm(z,pMem);
    return 0;
 }

 void initRe(TComplex* p, double TComplex::*pMem)
 {
   p->*pMem = 1.0;
 }

 void initIm(TComplex* p, double TComplex::*pMem)
 {
   p->*pMem = 1.0;
 }

 void printRe(TComplex z, double TComplex::*pMem)
 {
   printf(" Re = %lf \n", z.*pMem);
 }

 void printIm(TComplex z, double TComplex::*pMem)
 {
   printf(" Im = %lf \n", z.*pMem);
 }
```

Тут продемонстровані два способи звертання до члена класу через вказівник на нього: прямий і непрямий. Прямий спосіб використовує об'єкт класу: `z.*pMem`, а непрямий — посилання на нього: `p->*pMem`. В обох випадках виробляється розіменування вказівника на член класу `pMem`.

На жаль, вказівники на члени класу не підкоряються правилам арифметики, прийнятим для звичайних вказівників. Саме з цієї причини нам приходиться ініціалізувати кожен член класу `TComplex` окремою функцією — адже в нас немає засобів, що дозволяють переміщатися по членах класу шляхом інкрементації вказівника `pMem`.

Утім, приведену вище програму можна небагато поліпшити, зробивши вказівник на член класу локальним.

Використання вказівників на члени класу: другий варіант

```
   #include <stdio.h>
 #include <malloc.h>

 class TComplex
 {
 public:
    double Re;
    double Im;
 };

void init(TComplex*, double, double);
void print(TComplex);

int main()
 {
    TComplex z;
    init(&z, 1.0, 2.0);
```

```
   print(z);
   return 0;
}

void init(TComplex* p,double x, double y)
{
   double TComplex::*pMem;
   pMem = &TComplex::Re;
   p->*pMem = x;
   pMem = &TComplex::Im;
   p->*pMem = y;
}


void print(TComplex z)
{
   double TComplex::*pRe, TComplex::*pIm;
   pRe = &TComplex::Re;
   pIm = &TComplex::Im;
   printf(" z = %lf + i* %lf\n", z.*pRe, z.*pIm);
}
```

Тепер кількість функцій скорочена вдвічі.

7.2. КОНСТРУКТОРИ І ДЕСТРУКТОР

У попередніх прикладах ми ініціалізували об'єкт класу `TComplex` за допомогою функції `init()`. Утім, це було зайве. У мові C++ для ініціалізації об'єктів призначений механізм, називаний *конструктором*. Це — функція-член, ім'я якої збігається з ім'ям класу. Вона може мати будь-які параметри, що необхідні для ініціалізації полів об'єкта. При цьому конструктор не має ніякого значення, що повертається. Його виклик являє собою визначення об'єкта. Інакше кажучи, визначаючи об'єкт, ви викликаєте конструктор.

Оголошення об'єкта

```
   #include <stdio.h>
#include <malloc.h>

class TComplex
{
   double Re;
   double Im;
public:
   static int counter;
   void print();
   TComplex(double x, double y);
   static void HowMuch();
};

int TComplex::counter=0;

int main()
{
     for(int i = 1; i <= 5 ; i++)
     for(int j = 1; j <= 5 ; j++)
    {
       TComplex Z1(i,j);                    // Виклик конструктора
       Z1.print();                          // Виклик функції-члена print()
       TComplex::counter++;
     }

     TComplex::HowMuch();
   return 0;
}

void TComplex::print() {printf("Z = %lf + i*%lf \n",Re,Im);}
TComplex::TComplex(double x, double y)  {Re = x; Im = y;}
void TComplex::HowMuch(){printf("Counter = %d\n",TComplex::counter);}
```

Такий спосіб визначення найбільш розповсюджений. Однак існує ще одна форма:

```
TComplex Z = TComplex(i,j);
```

Цю конструкцію ми розглянемо пізніше.

7.2.1. Конструктори з одним параметром

Вище ми розглянули варіант конструктора з двома параметрами. Узагалі кажучи, конструктори можуть мати будь-які кількість чи параметрів не мати їх зовсім. Однак якщо конструктор має один параметр, виникає особлива ситуація.

Припустимо, що ми хочемо створити комплексне число з однаковими дійсною і уявною частинами. Природно, для його ініціалізації достатньо одного параметра.

Конструктор з одним параметром

```
   #include <stdio.h>
#include <malloc.h>

class TComplex
{
   double Re;
   double Im;
public:
   void print();
   TComplex(double v);
};


int main()
{
   TComplex X(10);              // Перший варіант
   X.print();
   TComplex Y = TComplex Y(10); // Другий варіант
   TComplex Z = 10;             // Третій варіант

   return 0;
}

void TComplex::print() {printf("Z = %lf + i*%lf \n",Re,Im);}
TComplex::TComplex(double v)  {Re = v; Im = v;}
```

Конструктор класу `TComplex` **має** один параметр. Для нього існує три форми ініціалізації; остання дозволяє виконати перед ініціалізацією неявне перетворення типу.

7.2.2. Список ініціалізації

Найчастіше зручно задавати початкові значення за допомогою списку ініціалізації. Його синтаксична конструкція виглядає в такий спосіб.

ім'я_класу(*тип параметр1, ... тип параметрN*):
член_класу(*параметр1*), ...,
член_класу(*параметрN*) {}

Список ініціалізації — єдиний спосіб задати значення константних змінних-членів. Крім того, така ініціалізація буває більш ефективною. Правда, якщо клас містить велику кількість членів, список може стати занадто довгим, що знизить наочність програми.

Використання списку ініціалізації

```
   #include <stdio.h>

class TComplex
{
   double Re;
   double Im;
public:
   void print();
   TComplex(double v):Re(v), Im(v){};
};

int main()
{
   TComplex Z=10;
   Z.print();

   return 0;
}

void TComplex::print() {printf("Z = %lf + i*%lf \n",Re,Im);}
```

Згадаємо, що клас може містити об'єкти іншого класу. У цьому випадку при ініціалізації елементів списку будуть викликатися відповідні конструктори. Порядок їх виклику визначається списком параметрів (причому в зворотному напрямку), а не списком ініціалізації, хоча правила гарного стилю вимагають не порушувати їхній природний порядок. При цьому порядок перерахування полів у класі значення не має.

Уявимо собі комплексні числа, у яких дійсні частини задаються числами з точкою, що плаває, а уявні — цілими числами. Будемо вважати, що дійсні і цілі числа імітуються фіктивними класами `Double` і `Integer` відповідно.

Порядок перерахування аргументів у списку ініціалізації

```
#include <stdio.h>

class TDouble
{
   double x;
public:
   TDouble(double y):x(y){printf(" Конструктор TDouble \n");}
};

class TInteger
{
   int n;
public:
   TInteger(int m):n(m){printf(" Конструктор TInteger \n");}
};

class TComplex
{
   TDouble Re;
   TInteger Im;
public:
   void print();
   TComplex(TDouble v, TInteger k):Re(v), Im(k){};
};


int main()
{
   TComplex Z(10.0, 20);
   Z.print();

   return 0;
}

void TComplex::print() {printf("Z = %lf + i*%d \n",Re,Im);}
```

Виконання цієї програми супроводжується наступними повідомленнями.

```
Конструктор TInteger
Конструктор TDouble
Z = 10.00000 + i*20
```

Переставимо параметри в конструкторі `TComplex`:

```
TComplex(TInteger k, TDouble v):Re(v), Im(k){}
```

На екрані з'являться такі рядки.

```
Конструктор TDouble
Конструктор TInteger
Z = 10.00000 + i*20
```

7.2.3. Конструктор за замовчуванням

У розглянутих вище прикладах об'єкти класу `TComplex` створювалися за допомогою конструктора, що має один чи два параметри. Задамося питанням: «Що відбудеться, якщо в класі не визначений жодний конструктор?» У цьому випадку об'єкт класу створюється за допомогою *конструктора за замовчуванням*, що, природно, не має параметрів. Він заповнює всі поля глобального об'єкта нулями. Якщо об'єкт є локальним, його поля будуть містити випадкові значення. Усі функції-члени класу, створеного за допомогою конструктора за замовчуванням, працюють у відповідності зі своїм описом.

Конструктор за замовчуванням

```
#include <stdio.h>

class TComplex
```

```
{
   int Re;
   int Im;
public:
   void print(){printf("Z = %d + i*%d \n",Re,Im);};
}Z;

int main()
{

   Z.print();

   return 0;
}
```

У підсумку одержуємо наступний результат.

```
Z = 0.0 + i*0.0
```

Якщо програміст визначить у класі хоча б один конструктор, навіть без параметрів, компілятор не буде генерувати автоматичний конструктор.

7.2.4. Конструктор копіювання

Знову створений об'єкт можна ініціалізувати умістом раніше існував об'єкта. Для цього призначений *конструктор копіювання*, що має наступний вид.

```
ім'я_класу(ім'я_класу&)
```

Пояснимо необхідність цього конструктора на прикладі. Уявимо собі об'єкт класу `TArray`, що є масивом цілих чисел. Він задається двома параметрами — початковою адресою (вказівником) і розміром. У наступній програмі ми створюємо масив `x`, а потім — масив `y`, причому новий масив повинний бути копією старого.

Використання конструктора копіювання

```
   #include <stdio.h>

class TArray
{
   int *p;
   int size;
public:
   TArray(long n, int x);
   TArray(TArray&);
   void view();
};

int main()
{
   TArray x(10,1);
   x.view();
   TArray y = x;
   y.view();
   return 0;
}

TArray::TArray(long n, int x)
{
   size = n;
   p = new int[size];
   for (long i=0; i<size; i++)p[i] = x;
   printf("\nAddress x = %p \n",p);
}

TArray::TArray(TArray& X)
{
   size=X.size;
   p = new int[size];     // Глибоке копіювання
   for (long i=0; i<X.size; i++) p[i] = X.p[i];
   printf("\nAddress y = %p \n",p);
}

void TArray::view()
{
```

```
    for(long i=0; i<size; i++) printf("%d ",p[i]);
}
```

У цій програмі реалізоване *глибоке копіювання*, що має на увазі створення абсолютно самостійної копії раніше існував об'єкта. Зверніть увагу на те, що в конструкторі копіювання поле p не дублюється. Замість цього йому привласнюється адреса знову виділеної пам'яті, у яку копіюються інші елементи масиву. У результаті ми побачимо на екрані наступні рядки.

```
  Адреса x = 007803A0
  1 1 1 1 1 1 1 1 1 1
 Адреса x = 00780340
  1 1 1 1 1 1 1 1 1 1
```

А що відбудеться, якщо програміст не визначив конструктор копіювання? У цьому випадку буде викликаний *конструктор копіювання за замовчуванням*. Він виконує *побітове*, тобто *поверхневе копіювання*, і замість нової копії ми одержимо два вказівники, що посилаються на той самий адресу.

```
  Адреса x = 007803A0
  1 1 1 1 1 1 1 1 1 1
 Адреса x = 007803A0
  1 1 1 1 1 1 1 1 1 1
```

7.2.5. «Іменовані конструктори»

Конструктори, як і звичайні функції, можна перевантажувати, варіюючи типи і кількість параметрів. Однак бувають ситуації, коли об'єкт класу повинний створюватися двома різними способами, що реалізуються конструкторами, що мають однакову кількість і тип параметрів.

Яскравий приклад такої ситуації — ініціалізация комплексного числа. Як відомо, комплексні числа мають дві форми запису: у декартових і полярних координатах. У першому випадку дійсна і мнима частини числа задаються абсцисою й ординатою ($z = x + iy$), а в другому — радіусом і кутом, між радіусом-вектором і віссю абсцис ($z = r\cos(\alpha)+ir\cos(\alpha)$). Обидві форми запису задаються парою дійсних чисел, тому прототипи конструкторів цілком збігаються, і розрізнити їхній виклик неможливо. Для того щоб розв’язати цю дилему, слід скористатися статичними функціями-членами, що викликають закритий конструктор і повертають створений об'єкт класу. Лапки означають, що ці функції насправді конструкторами не є, хоча виконують зовсім аналогічну задачу. Ідея цієї ідіоми полягає в тім, що статичні функції-члени викликаються ще *до* створення об'єкта зазначеного класу. Отже, з їхньою допомогою можна викликати закритий конструктор, передавши йому відповідний набір аргументів — декартові координати або радіус і кут.

Використання «іменованих конструкторів»

```
   #include <stdio.h>
#include <math.h>
#define PI 3.141592

class TComplex
{
public:
   static TComplex Dekart(double, double);
   static TComplex Euler(double, double);
   void view(){printf("Z = %lf +i*%lf \n",Re,Im); }
private:
   double Re;
   double Im;
   TComplex(double x, double y):Re(x),Im(y){}
};

int main()
{
   TComplex objDekart = TComplex::Dekart(1.0,0.0);
   TComplex objEuler  = TComplex::Euler(1.0,PI/2);
   objDekart.view();
   objEuler.view();
   return 0;
}

TComplex TComplex::Dekart(double x, double y)
{
    return TComplex(x,y);
}

TComplex TComplex::Euler(double r, double alpha)
{
```

```
    return TComplex(r*cos(alpha), r*sin(alpha));
}
```

Результат роботи цієї програми приведений нижче.

```
Z = 1.000000 +i*0.000000
Z = 0.000000 +i*1.000000
```

Відзначимо два моменти: 1) конструктор оголошений закритим, тому доступ до нього можливий лише через функції інтерфейсу; 2) інтерфейсні функції є статичними, тому їх можна викликати *до* створення нового об'єкта. Якби функції `Dekart()` і `Euler()` не були статичними, їх виклик став би можливий тільки *після* створення об'єкта. Однак оскільки конструктор оголошений закритим, створити об'єкт було б неможливо. Ключове слово `static` розриває це замкнуте коло.

7.2.7. Деструктор

Деструктор знищує об'єкти класу, звільняючи зайняті ресурси. Ця функція, як і конструктор, не має значення, що повертається, однак, у відмінність він конструктора, вона не має одного параметра. Ім'я деструктора повинне збігатися з ім'ям класу, перед яким поставлена тильда.

`~`*`ім'я_класу`*`() {`*`тіло деструктора`*`}`

Чудовою особливістю деструктора є той факт, що він ніколи (за одним-єдиним виключенням) не викликається явно. Просто, коли об'єкт виходить з області видимості, програма сама викликає деструктор, що виконує запропоновані дії. Якщо програміст не визначив деструктор класу, компілятор згенерує *деструктор за замовчуванням*.

Використання деструктора

```
   #include <iostream.h>

class TClass2
{
public:
   int a;
   TClass2():a(1){cout << "Ctor TClass2" << endl;}
   ~TClass2()    {cout << "Dtor TClass2" << endl;}
};

class TClass1
{
public:
   TClass2 *p;
   TClass1()
   {
     p = new TClass2;
     cout << "Ctor TClass1" << endl;
   }
   ~TClass1()
   {
   delete p;
   cout << "Dtor TClass1" << endl;
 }
};


int main()
{
   TClass1 a;
   return 0;
}
```

Наприклад, у даній програмі в модулі `main()` визначений об'єкт `a` класу `TClass1`. Цей об'єкт містить усередині себе вказівник на об'єкт класу `TClass2`. Отже, першим повинний викликатися конструктор класу `TClass2`, а потім — конструктор класу `TClass1`. Знищення об'єкта `a` відбувається «зсередини»: спочатку знищується об'єкт класу `TClass2`, а потім — об'єкт класу `TClass1`.

```
  Ctor TClass2
 Ctor TClass1
 Ctor TClass2
 Ctor TClass1
```

Якби в об'єкті класу `TClass1` об'єкт класу `TClass2` не створювався, а просто був його членом, деструктори викликалися б у зворотному порядку.

Зворотний порядок викликів деструкторів

```
   #include <iostream.h>

 class TClass2
 {
 public:
    int a;
    TClass2():a(1){cout << "Ctor TClass2" << endl;}
    ~TClass2()    {cout << "Dtor TClass2" << endl;}
 };

 class TClass1
 {
 public:
    TClass2 p;
    TClass1()
    {
      cout << "Ctor TClass1" << endl;
    }
    ~TClass1()
    {
      cout << "Dtor TClass1" << endl;
    }
 };

 int main()
 {
    TClass1 a;
    return 0;
 }
```

Тепер повідомлення мають наступний вид.

```
   Ctor TClass2
 Ctor TClass1
 Ctor TClass1
 Ctor TClass2
```

Розглянемо ситуацію, у якій необхідно явно викликати деструктор.

Явний виклик деструктора

```
  #include <iostream.h>
 #include <new.h>

 class TClass
 {
    public:
    int a;
    TClass() {cout << "Ctor TClass " << endl;}
    ~TClass(){cout << "Dtor TClass " << endl;}
 };

int main()
 {
    char memory[sizeof(TClass)];
    void *p=memory;
    TClass* q = new(p) TClass;
    cout << q << endl;
    q->~TClass();
    return 0;
 }
```

Результат роботи цієї програми виглядає так.

```
  Ctor TClass
  0x0075fdc4
  Dtor TClass
```

Тут використаний так званий *синтаксис розміщення*. Це — різновид оператора `new`, що дозволяє створювати і розміщати об'єкт по заздалегідь заданій адресі. Спочатку ми резервуємо пам'ять для об'єкта — масив `memory`, а потім створюємо об'єкт за допомогою синтаксису розміщення. Саме тому, що ми занадто глибоко втрутилися в механізм розподілу пам'яті для нових об'єктів, нам доведеться самим подбати про її звільнення.

7.2.7. Передача аргументів і повернення значень

Конструктори, конструктори копіювання і деструктор дозволяють продемонструвати механізм передачі аргументів і значень функцій, що повертаються.

Об'єкти, як і звичайні змінні, можуть передаватися за значенням і за посиланням. Як це відбувається, ілюструється наступною програмою.

Передача об'єктів за значенням

```
  #include <stdio.h>
 #include <math.h>

 class TComplex
 {
    double Re;
    double Im;
 public:
    TComplex(double x, double y):Re(x),Im(y){ printf("Ctor\n");}
    TComplex(TComplex& z){ Re = z.Re; Im = z.Im; printf("Copy ctor\n");}
    ~TComplex(){printf("Dtor\n");}
    double getRe() { return Re;}
    double getIm() { return Im;}
 };

 double modul(TComplex);

 int main()
 {
    printf("Start main\n");
    TComplex z(1,1);
    printf("Модуль z = %lf\n",modul(z));
    printf("End modul\n");
    printf("End main\n");
    return 0;
 }

 double modul(TComplex z)
 {
    printf("Start modul\n");
    return sqrt(z.getRe()*z.getRe() + z.getIm()*z.getIm());
 }
```

На екрані з'являться наступні рядки.

```
  Start main
 Ctor
 Copy ctor
 Start modul
 Dtor
 End modul
  Модуль z = 1.414214
 End main
 Dtor
```

Отже, після запуску головного модуля створюється об'єкт `z` класу `TComplex`. Про це свідчить повідомлення від конструктора — `Ctor`. Потім за допомогою конструктора копіювання створюється дублікат об'єкта `z`. Конструктор копіювання сповіщає про це, виводячи рядок `Copy ctor`. Потім починається виконання функції `modul()`, виводиться результат і знищується тимчасова копія параметра. Про це говорить рядок `Dtor`, що виводиться на екран після виконання оператора `return`. Отже, на екрані з'являється результат обчислень. Виконання функції `main()` кінцю добігає кінця, і після виконання оператора `return 0` викликається деструктор класу `TComplex`, що знищує об'єкт `z`.

Основний висновок, якому необхідно зробити, — при передачі об'єкта за значенням його копія створюється *конструктором копіювання*.

Досить поставити після імені класу `TComplex` у прототипі і заголовку функції `modul()` символ амперсанду, що означає передачу параметра за посиланням, — і конструктор копіювання викликатися не буде. На екрані з'явиться наступне повідомлення.

```
   Start main
 Ctor
 Start modul
 End modul
```

```
Модуль z = 1.414214
End main
```

Розглянемо програму, що підсумовує два комплексних числа. У цьому випадку функція `sum()` повертає копію локального об'єкта, що зберігає результат. Після передачі ця копія й об'єкт знищуються.

Повернення об'єкта

```
   #include <stdio.h>

class TComplex
{
   double Re;
   double Im;
public:
   static int counter;
   TComplex(double x, double y):Re(x),Im(y)
     { counter++;printf("Ctor %d\n",counter); }
   TComplex(TComplex& z)
     { Re = z.Re; Im = z.Im; counter++; printf("Copy %d\n",counter);}
   ~TComplex(){printf("Dtor %d\n",counter);counter--;}
   double getRe() { return Re;}
   double getIm() { return Im;}
   void putRe(double x) { Re =x;}
   void putIm(double y) { Im = y;}
};

TComplex sum(TComplex,TComplex);
int TComplex::counter = 0;

int main()
{
   printf("Start main\n");
   TComplex z(1,1), w(2,2), u(0,0);
   u = sum(z,w);
   printf("End sum\n");
   printf("w + u = %lf + i*%lf\n",u.getRe(),u.getIm());
   printf("End main\n");
   return 0;
}

TComplex sum(TComplex u, TComplex v)
{
   printf("Start \n");
   TComplex w(0,0);
   w.putRe(u.getRe() + v.getRe());
   w.putIm(u.getIm() + v.getIm());
   return w;
}
```

От як виглядають повідомлення від конструкторів, конструкторів копіювання і деструкторів (коментарі додані для більшої ясності).

```
   Start main            //Начало функції main()
Ctor 1                 // Створюється об'єкт z
Ctor 2                 // Створюється об'єкт w
Ctor 3                 // Створюється об'єкт u
Copy 4                 // Створюється копія об'єкта z
Copy 5                 // Створюється копія об'єкта w
Start sum              // Починається виконання функції sum()
Ctor 7                 // Створюється локальний об'єкт s
Copy 7                 // Створюється копія локального об'єкта
Dtor 7                 // Знищується копія локального об'єкта
Dtor 7                 // Знищується копія локального об'єкта
Dtor 5                 // Знищується копія об'єкта w
Dtor 4                 // Знищується копія об'єкта z
End sum                // Виконання функції sum() довершене
w + u = 3.000000 + i*3.000000   // Результат
Dtor 3                 // Знищення об'єкта u
Dtor 2                 // Знищення об'єкта w
Dtor 1                 // Знищення об'єкта z
```

Отже, відзначимо, що конструктор копіювання викликається в наступних ситуаціях.

1. При ініціалізації об'єкта під час оголошення.
2. При передачі об'єкта як параметр функції.
3. При поверненні об'єкта як значення, що повертається

7.3. ДРУЖНІ ФУНКЦІЇ І КЛАСИ

У попередній програмі, коли нам знадобилося скласти два комплексних числа, довелося застосувати чотири інтерфейсні функції: `getRe()`, `getIm()`, `putRe()`, `putIm()`. Програма стала б простіше, якби функція `sum()` мала прямий доступ до полів класу `TComplex`. Для цього можна оголосити цю функцію дружньою, використовуючи ключове слово `friend`.

Використання дружніх функцій

```
  #include <stdio.h>

class TComplex
{
    friend TComplex sum(TComplex, TComplex);
    double Re;
    double Im;
public:
    static int counter;
    TComplex(double x, double y):Re(x),Im(y)
        { counter++;printf("Ctor %d\n",counter);}
    TComplex(TComplex& z)
        { Re = z.Re; Im = z.Im; counter++; printf("Copy %d\n",counter);}
    ~TComplex(){printf("Dtor %d\n",counter);counter--;}
    void print() {printf("Z = %lf + i*%lf\n",Re,Im);}
};

TComplex sum(TComplex,TComplex);

int TComplex::counter = 0;

int main()
{
    printf("Start main\n");
    TComplex z(1,1), w(2,2), u(0,0);
    u = sum(z,w);
    printf("End sum\n");
    u.print();
    printf("End main\n");
    return 0;
}

TComplex sum(TComplex u, TComplex v)
{
    printf("Start \n");
    TComplex w(0,0);
    w.Re= u.Re + v.Re;
    w.Im=u.Im + v.Im;
    return w;
}
```

Дружня функція має всі права члена класу. Отже, їй доступні всі поля — як відкриті, так і закриті. Для того щоб оголосити функцію дружньої, ключове слово `friend` слід помістити в оголошення класу. (Клас сам оголошує своїх друзів!)

Розділимо клас `TComplex` на два: `TNumber` і `TFunction`. У першому класі залишимо змінні-члени, а до другого віднесемо усі функції.

Використання дружніх функцій-членів

```
  #include <stdio.h>

class TNumber;

class TFunction
{
public:
    void print(TNumber z);
```

```
};

class TNumber
{
    double Re;
    double Im;
public:
    static int counter;
    TNumber(double x, double y):Re(x),Im(y)
        { counter++;printf("Ctor %d\n",counter);}
    TNumber(TNumber& z)
        { Re = z.Re; Im = z.Im; counter++; printf("Copy %d\n",counter);}
    ~TNumber(){printf("Dtor %d\n",counter);counter--;}
    friend void TFunction::print(TNumber);
};


int TNumber::counter = 0;

int main()
{
    printf("Start main\n");
    TNumber z(1,1);
    TFunction a;
    a.print(z);
    printf("End main\n");
    return 0;
}

void TFunction::print(TNumber z)
{
  printf("z = %lf + i*%lf\n", z.Re, z.Im);
}
```

У результаті одержуємо наступні повідомлення.

```
 Start main
Ctor 1
Copy 2
z = 1.000000 + i*1.000000
Dtor 2
End main
Dtor 1
```

Зверніть увагу на те, що функція `print()`, що належить класу `TFunction`, одержує параметр типу `TNumber`. Отже, цей тип повинний бути визначений раніше. Однак клас `TNumber` оголошує дружню функцію `print()`, що належить класу `TFunction`. Для розв'язання цієї проблеми в мові C++ передбачений механізм *неповного оголошення класу*.

Працюючи з дружніми функціями, потрібно враховувати наступні обмеження.

1. Вони не успадковуються.
2. Вони не можуть мати специфікатори `static` і `extern`.
3. У класі, функція-член якого є дружньою до не цілком оголошеного класу, слід розміщати тільки прототип. Реалізація функції повинна знаходитися після повного оголошення класу.

Коли усі функції деякого класу є дружніми стосовно іншого класу, можна оголосити весь клас дружнім. Наприклад, попередню програму можна переробити в такий спосіб.

Використання дружніх класів

```
  #include <stdio.h>

class TNumber;                  // Неповне оголошення

 TFunction
 {
 public:
    void print(TNumber z);     // Апеляція до не цілком оголошеного класу
 };

 class TNumber
 {
```

```
    double Re;
    double Im;
    public:
      static int counter;
      TNumber(double x, double y):Re(x),Im(y)
        { counter++;printf("Ctor %d\n",counter);}
      TNumber(TNumber& z)
        { Re = z.Re; Im = z.Im; counter++; printf("Copy %d\n",counter);}
      ~TNumber(){printf("Dtor %d\n",counter);counter--;}
      friend class TFunction;    // Оголошення дружнього класу
    };


int TNumber::counter = 0;

int main()
 {
    printf("Start main\n");
    TNumber z(1,1);
    TFunction a;
    a.print(z);
    printf("End main\n");
    return 0;
 }

 void TFunction::print(TNumber z)
 {
    printf("z = %lf + i*%lf\n", z.Re, z.Im);
 }
```

Варто мати на увазі, що члени класу `TFunction` просто мають доступ до членів класу `TNumber`, але члени класу `TNumber` від цього не стають членами класу `TFunction`.

Іноді два класи настільки тісно зв'язані, що їм необхідний повний доступ до всіх членів. У цьому випадку їх оголошують взаємно дружніми.

Використання взаємно дружних класів

```
   #include <stdio.h>

 class TNumber;

 class TFunction
 {
    void print(TNumber z);
    friend class TNumber;
 };

 class TNumber
 {
    double Re;
    double Im;
 public:
    static int counter;
    TNumber(double x, double y):Re(x),Im(y)
      { counter++;printf("Ctor %d\n",counter);}
    TNumber(TNumber& z)
      { Re = z.Re; Im = z.Im; counter++; printf("Copy %d\n",counter);}
    ~TNumber(){printf("Dtor %d\n",counter);counter--;}
    void sum(TNumber y, TFunction z)
      {Re = Re + y.Re; Im = Im + y.Im; z.print(*this);}
    friend class TFunction;
 };


 int TNumber::counter = 0;

 int main()
 {
    printf("Start main\n");
```

```
    TNumber z(1,1),x(1,1);
    TFunction a;
    z.sum(z,a);
    printf("End main\n");
    return 0;
}

void TFunction::print(TNumber z)
{
 printf("z = %lf + i*%lf\n", z.Re, z.Im);
}
```

У ході виконання цієї програми на екрані з'являться наступні рядки.

```
Start main
Ctor 1
Ctor 2
Copy 3
Copy 4
z = 2.000000 + i*2.000000
Dtor 4
Dtor 3
End main
Dtor 2
Dtor 1
```

Як бачимо, функція `sum()`, що є членом класу `TNumber`, одержує вільний доступ до функції `print()`, оголошеної в закритому розділі класу `TFunction`. Якби класи `TNumber` і `TFunction` не були взаємно дружніми, це було б неможливо.

7.4. РЕЗЮМЕ

- В основу *об'єктно-орієнтованого* програмування покладене поняття *об'єкта* і принципи *інкапсуляції*, *прихування інформації*, *абстракції даних*, *успадкування* і *поліморфізму*.
- Основна ідея об'єктно-орієнтованого програмування полягає в тім, що всі обчислення відбуваються усередині об'єктів, що обмінюються між собою повідомленнями.
- Відповідно до принципу інкапсуляції *об'єкт — це об'єднання даних і алгоритмів, що обробляють ці дані*.
- В об'єктно-орієнтованих мовах існує можливість *перевантажувати* операції і функції, що дозволяє об'єкту самому конкретизувати їхній зміст у ході виконання програми. »Що дозволяє спростити програмування. Наприклад, якщо перевантажити оператор +, то операція `a + b` може мати різний сенс. Якщо ці об'єкти — числа, застосовується звичайна операція додавання, якщо матриці — матрична (конечно, для цього необхідно визначити цю операцію). У цьому (і не тільки) полягає сутність *поліморфізму*. У принципі поліморфізм виявляється в трьох ситуаціях: при перевантаженні функцій, при перевантаженні операторів і при пізнім зв'язуванні.
- Об'єкт — це фізична сутність, що виникає при виконанні програми, тобто сукупність комірок пам'яті, що зберігають дані і код. Для того щоб створити об'єкт, необхідна його схема: які дані він містить, які функції обробляють ці дані, як організований доступ до цим даних і функцій, що називаються *членами об'єкта*. У мові C++ схемою об'єкта називається *клас*.
- Для оголошення класу в мові C++ призначене ключове слово `class`. Усе, що розташовано між фігурними дужками, що випливають за цим ключовим словом, являє собою *тіло класу*, що складає з його *членів*. Варто пам'ятати, що клас — це логічна схема об'єкта. Отже, виділення пам'яті відбудеться лише тоді, коли об'єкт буде *визначений*. Оголошення класу виглядає так:

```
class                                                    ім'я{
      члени                                              класу;
 private:
      члени                                              класу;
  protected:
      члени                                              класу;
 public:
      члени                                              класу;
 } <список об'єктів>;
```

- Тіло класу розділяється на три розділи, позначені наступними ключовими словами (специфікаторами доступу).

```
public     // Відкритий розділ
private    // Закритий розділ
```

```
protected // Захищений розділ
```

- Ключове слово `public` позначає розділ, у якому розміщаються функції і дані, доступні з будь-якої точки програми, — відкриті члени. Ключове слово `private` оголошує розділ, у якому розташовані закриті члени класу, доступні лише функціям-членам самого класу, а також дружнім функціям і класам. Зверніть увагу на те, що функції-члени можуть знаходитися в ніби безіменному розділі. Насправді, за замовчуванням функції і дані, оголошені в такому розділі, є закритими. Зміст специфікатора `protected` буде розкритий пізніше при описі спадкування класів.
- З класами також тісно зв'язані *об'єднання*. Оскільки цей вид структури більш складний і зв'язаний з досить хитромудрим розподілом пам'яті, на нього поширюються деякі обмеження. Зокрема, до об'єднань не можна застосовувати механізм успадкування; вони не можуть містити віртуальні функції, а також статичні змінні і посилання. Крім того, об'єднання не повинні містити об'єкти класів з перевантаженим оператором присвоювання, а також явно визначеними конструкторами і деструктором.
- Порядок специфікаторів доступу не регламентується — вони можуть повторюватися неодноразово й у будь-якому порядку. Кожен наступний специфікатор припиняє дія попереднього. Як правило, відкриті члени оголошуються останніми, але це відноситься до питань стилю, а не до синтаксичних правил.
- Функції-члени, що не змінюють поля класу, називаються *функціями-членами доступу*.
- Функції-члени, що змінюють поля класу, називаються *функціями-членами, що модифікують*.
- Відкриті функції члени утворюють *інтерфейс* класу.
- Для звертання до членів класу використовується оператор доступу до членів класу «`.`».

```
Z.init();        // Виклик функції-члена init()
Z.print();       // Виклик функції-члена print()
```

- Якщо в програмі оголошений вказівник на об'єкт класу, використовується оператор посилання на члени класу «`->`».
- При оголошенні полів класу не можна користатися ключовими словами `auto`, `extern` і `register`.
- Якщо визначення функції-члена міститься усередині класу, сама функція автоматично *вважається що* підставляється (хоча і не *стає* нею автоматично). Ключове слово `inline` для цього вказувати необов'язково, хоча і не заборонено.
- При виклику функція-член одержує вказівник на об'єкт, що зробив виклик. Цей вказівник позначається ключовим словом `this`. Отже, вказівник `this` на об'єкт класу `T` має тип `T*`.
- Вказівник `this` завжди вважається *константним* і ніколи не передається статичним функціям-членам.
- Статичні змінні-члени мають особливі властивості. Вони доступні всім об'єктам класу, не належачи нікому з них окремо. Якби клас не був абстракцією, можна було б сказати, що статична змінна належить класу, а не об'єктам. У реальності це означає, що статична змінна-член класу існує в єдиному екземплярі незалежно від кількості об'єктів. Особливо важно, що статична змінна ініціалізується ще до створення першого об'єкта класу.
- Оголошення статичної змінної члену класу має декілька тонкощів. По-перше, мало оголосити статичну змінна-член — необхідно виділити для неї пам'ять. Для цього її оголошення варто повторити поза тілом класу, в області визначення глобальних змінних. Визначення статичної змінна-члена можна сполучити з її ініціалізацією. По-друге, повторювати ключове слово `static` при визначенні статичної функції не можна! Воно повинно бути присутнім тільки в тілі класу! Розглянемо приклад, у якому змінні `Re` і `Im` оголошені статичними членами класу `TComplex`.
- Статичними можуть бути не тільки змінні, але і функції-члени класу. У цьому випадку вони втрачають доступ до нестатичних членів класу і можуть звертатися тільки до статичних змінна-членів і функцій. Крім того, вони не можуть бути віртуальними, одержувати вказівник `this`, а також не повинні мати кваліфікаторів `const` і `volatile`.
- Для захисту від модифікації члени класу, як правило, поміщають у закритий розділ і оголошують константними. Однак, якщо надалі нам знадобиться змінити константне поле, прийдеться виконувати приведення типу за допомогою оператора `const_cast`. Утім, існує ще один спосіб захистити поля — розділивши функції-члени на функції доступу і функції, що модифікують. Перші функції не мають права змінювати вміст полів. Для цього них оголошують константними. Друга група функцій може вільно привласнювати полям нові значення.
- Якщо функція-член класу оголошена константно, вона не повинна змінювати зміст полів класу. Однак якщо зняти захист з члену класу, поставивши перед його оголошенням ключове слово `mutable`, то будь-яка константна функція зможе змінити значення цього поля.
- Мова C++ дозволяє створювати *вкладені класи*. Це дозволяє сховати внутрішній клас від зовнішнього світу, обмеживши його область видимості зовнішнім класом.

- Внутрішній (чи локальний) клас можна визначити не тільки усередині класу, але й усередині функції, наприклад усередині функції `main()`.
- Оскільки функції не можна визначати усередині інших функцій, за необхідності їхні визначення можна занурити усередину локального класу. Тільки під цією оболонкою вони доступні зовнішньої функції.
- У силу обмеженості області видимості локальних класів, їм недоступні інші локальні змінні, оголошені усередині функції, крім статичних і зовнішніх. Крім того, усередині локальних класів неможливо визначити статичні змінні.
- У деяких ситуаціях можна одержати доступ до функції-члена класу, не вказуючи імен ні об'єкта, ні функції. Механізм, що дозволяє це зробити, ґрунтується на використанні *вказівників на член класу*. Зрозуміло, щоб викликати функцію, необхідний реальний об'єкт класу, однак доступ до його функції-члена програма одержує за допомогою непрямої адресації. Для цього спочатку необхідно створити об'єкт у динамічній пам'яті й установити на нього вказівник. Потім з'являється вказівник на функцію-член класу. Тепер, передаючи як параметри вказівники на об'єкти і функції-члени, ми можемо викликати кожну функція-член класу, не називаючи ее явно. Природно, її оголошення повинне точно відповідати оголошенню вказівника на функцію-член.
- Прямий спосіб доступу до вказівника на член класу використовує об'єкт класу: `z.*pMem`, а непрямий — посилання на нього: `p->*pMem`. В обох випадках виробляється розіменування вказівника на член класу `pMem`.
- На жаль, вказівники на члени класу не підкоряються правилам арифметики, прийнятим для звичайних вказівників. Саме з цієї причини нам приходиться ініціалізувати кожен член класу `TComplex` окремою функцією — адже в нас немає засобів, що дозволяють переміщатися по членах класу шляхом інкрементації вказівника `pMem`.
- У мові C++ для ініціалізації об'єктів призначений механізм, називаний *конструктором*. Це — функція-член, ім'я якої збігається з ім'ям класу. Вона може мати будь-які параметри, що необхідні для ініціалізації полів об'єкта. При цьому конструктор не має ніякого значення, що повертається. Його виклик являє собою визначення об'єкта. Інакше кажучи, визначаючи об'єкт, ви викликаєте конструктор.
- Конструктори можуть мати будь-які кількість чи параметрів не мати їх зовсім. Однак якщо конструктор має один параметр, виникає особлива ситуація: для нього існує три форми ініціалізації:

```
TComplex X(10);                // Перший варіант
TComplex Y = TComplex Y(10); // Другий варіант
TComplex Z = 10;               // Третій варіант
```

- Найчастіше зручно задавати початкові значення за допомогою списку ініціалізації. Його синтаксична конструкція виглядає в такий спосіб.

```
ім'я_класу(тип параметр1, ... тип параметрN):
          член_класу(параметр1), ...,
          член_класу(параметрN) {}
```

Список ініціалізації — єдиний спосіб задати значення константних змінна-членів.

- Якщо в класі не визначений жодний конструктор, то об'єкт класу створюється за допомогою *конструктора за замовчуванням*, або *автоматичний конструктор*, що, природно, не має параметрів. Він заповнює всі поля глобального об'єкта нулями. Якщо об'єкт є локальним, його поля будуть містити випадкові значення. Усі функції-члени класу, створеного за допомогою конструктора за замовчуванням, працюють у відповідності зі своїм описом.
- Якщо програміст визначить у класі хоча б один конструктор, навіть без параметрів, компілятор не буде генерувати автоматичний конструктор.
- Знову створений об'єкт можна ініціалізувати змістом об'єкта, що уже існує. Для цього призначений *конструктор копіювання*, що має наступний вид.

```
ім'я_класу(ім'я_класу&)
```

- Конструктор копіювання викликається при ініціалізації об'єкта під час оголошення., при передачі об'єкта як параметр функції і при поверненні об'єкта як значення, що повертається.
- *Глибоке копіювання* означає створення абсолютно самостійної копії об’єкта, що раніше існував раніше існував об'єкта (з іншою адресою, але тим же змістом).
- Якщо програміст не визначив конструктор копіювання, то буде викликаний *конструктор копіювання за замовчуванням*. Він виконує *побітове*, тобто *поверхневе копіювання*, і замість нової копії ми одержимо два вказівники, що посилаються на той самий адресу.
- Конструктори, як і звичайні функції, можна перевантажувати, варіюючи типи і кількість параметрів. Однак бувають ситуації, коли об'єкт класу повинний створюватися двома різними способами, що реалізуються конструкторами, що мають однакову кількість і тип параметрів. У уьому випадку використовується ідіома «іменованих конструкторів», що складається із статичних функцій-членів, що викликають закритий конструктор і повертають створений об'єкт класу. Ідея цієї ідіоми полягає в

тім, що статичні функції-члени викликаються ще *до* створення об'єкта зазначеного класу. Отже, з їхньою допомогою можна викликати закритий конструктор, передавши йому відповідний набір аргументів.

- Відзначимо два моменти: 1) конструктор оголошений закритим, тому доступ до нього можливий лише через функції інтерфейсу; 2) інтерфейсні функції є статичними, тому їх можна викликати *до* створення нового об'єкта. Якби функції не були статичними, їх виклик став би можливий тільки *після* створення об'єкта. Однак оскільки конструктор оголошений закритим, створити об'єкт було б неможливо. Ключове слово `static` розриває це замкнуте коло.
- *Деструктор* знищує об'єкти класу, звільняючи зайняті ресурси. Ця функція, як і конструктор, не має значення, що повертається, однак, у відмінність він конструктора, вона не має одного параметра. Ім'я деструктора повинне збігатися з ім'ям класу, перед яким поставлена тильда.

  `~`*`ім'я_класу`*`()` `{`*`тіло деструктора`*`}`
- Особливістю деструктора є той факт, що він ніколи (за одним-єдиним виключенням) не викликається явно. Просто, коли об'єкт виходить з області видимості, програма сама викликає деструктор, що виконує запропоновані дії. Якщо програміст не визначив деструктор класу, компілятор згенерує *деструктор за замовчуванням*.
- *Синтаксис розміщення* — це різновид оператора `new`, що дозволяє створювати і розміщати об'єкт по заздалегідь заданій адресі. Спочатку ми резервуємо пам'ять для об'єкта , а потім створюємо об'єкт за допомогою синтаксису розміщення. Саме тому, що ми занадто глибоко втрутилися в механізм розподілу пам'яті для нових об'єктів, нам доведеться самим подбати про її звільнення, явно викликавши деструктор.
- Для того щоб функція мала прямий доступ до полів класу, її можна оголосити функцію дружньої, використовуючи ключове слово `friend`.
- Дружня функція має всі права члена класу. Для неї відкриті всі поля — як відкриті, так і закриті. Для того щоб оголосити функцію дружньої, ключове слово `friend` слід помістити в оголошення класу. (Клас сам оголошує своїх друзів!)
- Працюючи з дружніми функціями, потрібно враховувати наступні обмеження: вони не успадковуються, вони не можуть мати специфікатори `static` і `extern`; у класі, функція-член якого є дружньої до не цілком оголошеного класу, слід розміщати лише прототип. Реалізація функції повинна знаходитися після повного оголошення класу.
- Коли усі функції деякого класу є дружніми стосовно іншого класу, можна оголосити весь клас дружнім.
- Що таке неповне оголошення класів?
- Іноді два класи настільки тісно зв'язані, що їм необхідний повний доступ до всіх членів. У цьому випадку їх оголошують взаємно дружніми.

7.5. КОНТРОЛЬНІ ПИТАННЯ

1. Які поняття і принципи покладено в основу об'єктно-орієнтованого програмування?
2. У чому полягає основна ідея об'єктно-орієнтованого програмування полягає?
3. Сформулюйте принцип інкапсуляції.
4. Опишіть сутність поліморфізму.
5. Що таке клас і об'єкт?
6. Як описується клас у мові C++?
7. З яких розділів складається тіло класу?
8. Що означає ключове слово `public`?
9. Що означає ключове слово `private`?
10. Опишіть об'єднання та обмеження, що на них накладаються.
11. Як діють специфікатори доступу?
12. Які функції-члени називаються функціями-членами доступу?
13. Які функції-члени називаються функціями-членами, що модифікують?
14. З яких функцій складається інтерфейс класу?
15. Як звернутися до членів класу?
16. Які ключові слова не можна використовувати при оголошенні полів класу?
17. Що являє собой вказівник `this`?
18. Опишіть особливості статичних змінних-членів.
19. Опишіть особливості статичних функцій-членів.
20. Як зняти захист з члену класу?
21. Як оголосити вкладені класи? Опишіть особливості.
22. Як використовується вказівник на функцію-член?
23. Опишіть прямий і непрямий способи доступу до вказівника на член класу.
24. Що таке конструктор?

25. Що таке список ініціалізації?
26. Що таке *конструктора за замовчуванням*, або автоматичний конструктор? У яких ситуаціях від генерується, а у яких ні?
27. Опишіть процес ініціалізації об’єкта.
28. У яких ситуаціях викликається конструктор копіювання?
29. Що собою являє г*либоке копіювання*?
30. Яке копіювання здійснює конструктор копіювання за замовчуванням?
31. Опишіть ідіому «іменованих конструкторів».
32. Що таке деструктор? Які особливості його виклику?
33. Що таке *синтаксис розміщення*?
34. Які функції називаються дружньоми відносно класу.
35. Які обмеження накладаються на використання дружніх функцій?
36. Які класи називаються дружніми?
37. Які класи називаються взаємно дружніми?