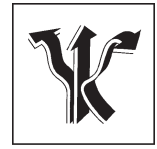
МІЖРЕГІОНАЛЬНА
АКАДЕМІЯ УПРАВЛІННЯ ПЕРСОНАЛОМ



О. В. ГАЛКІН, М. М. ВЕРЕС

МОВА ПРОГРАМУВАННЯ С++

Конспект лекцій

Київ
ДП «Видавничий дім «Персонал»
2017

Рецензенти: *М. Г. Глибовець,* д-р фіз.-мат. наук, проф.
В. Б. Звариdчук, канд. фіз.-мат. наук, доц.
В. П. Шевченко, канд. фіз.-мат. наук, доц.

Схвалено Вченою радою Міжрегіональної Академії управління персоналом (протокол № 1 від 26.01.11)

Галкін О. В., Верес М. М.

Мова програмування С++: конспект лекцій / О. В. Галкін, М. М. Верес. — К.: ДП "Вид. дім "Персонал", 2017. — 260 с. — Бібліогр.: с. 249.

ISBN 978-617-02-0194-2

У конспекті лекцій розглядаються всі основні можливості мови С++ і їх застосування при розробці об'єктно-орієнтованих програм. Дається короткий опис бібліотек мови С++, необхідних для створення типових програм. Систематизовано викладаються основні поняття й описуються можливості мови С++.

Для студентів вищих навчальних закладів, які вивчають програмування на мові С++.

ISBN 978-617-02-0194-2

ВСТУП

Мова C++ була створена на початку 80-х років XX ст. співробітником фірми Bell Laboratories Берном Страуструпом, як розширення мови С. До початку офіційної стандартизації мова розвивалася здебільшого силами Б. Страуструпа у відповідь на запити програмістського співтовариства. У 1998 р. був ратифікований міжнародний стандарт мови C++: ISO/IEC 14882:1998 "Standard for the C++ Programming Language"; після прийняття технічних виправлень до стандарту у 2003 р. — сучасна версія цього стандарту — ISO/IEC 14882:2003.

Ідея створення нової мови бере початок від мови моделювання Сімула (Simula). Вона мала низку можливостей, що були корисні для розробки великого програмного забезпечення, але працювала занадто повільно. У той самий час мова BCPL досить швидка, але занадто близька до мов низького рівня й не підходить для розробки великого програмного забезпечення. Б. Страуструп почав працювати у Bell Laboratories над задачами теорії черг (у додатку до моделювання телефонних викликів). Він вирішив доповнити мову С (спадкоємець BCPL) можливостями, наявними у мові Сімула. Мова С, яка є базовою мовою системи UNIX, швидка та багатофункціональна. Б. Страуструп додав їй можливість роботи із класами й об'єктами. У результаті практичні задачі моделювання виявилися доступними для розв'язання як з погляду часу розроблення (завдяки використанню Сімула-подібних класів), так і з погляду часу обчислень (завдяки швидкодії С).

Нововведеннями C++ порівняно із С є:

- підтримка об'єктно-орієнтованого програмування;
- підтримка узагальненого програмування через шаблони;
- додаткові типи даних;
- виключення;
- простір імен;
- вбудовані функції;
- перевантаження операторів;
- перевантаження імен функцій;

- посилання й оператори керування вільно розподіленою пам'яттю;
- доповнення до стандартної бібліотеки.

Отже, можна сказати, що мова C++ багато у чому є надмножиною мови *C*:

Варто також відзначити, що C++ має ряд недоліків, деякі з них успадковані від *C*:

- синтаксис, що провокує помилки, наприклад, операція присвоювання позначається як = , а операція порівняння як ==, їх легко переплутати; *if* (*x* = 0) {оператори}. Тут в умовному операторі помилково написане присвоювання замість порівняння. У результаті, замість того, щоб зрівняти поточне значення *x* з нулем, програма присвоїть *x* нульове значення;
- макроси (*#define*) є потужним, але небезпечним засобом. Вони збережені у C++, незважаючи на те, що в них немає необхідності завдяки шаблонам і вбудованим функціям. В успадкованих стандартних *C*-бібліотеках багато потенційно небезпечних макросів;
- деякі перетворення типів неінтуїтивні. Зокрема, операція над беззнаковим і знаковим числами видає беззнаковий результат;
- необхідність записувати *break* у кожному розгалужені оператора switch і можливість послідовного виконання кількох розгалужень при його відсутності провокує помилки через пропуск *break*. Ця сама особливість дає змогу робити сумнівні "трюки", що базуються на вибірковому незастосуванні *break*, і ускладнює розуміння коду.

Однак, незважаючи на свої недоліки, C++ є найпоширенішою мовою програмування для операційних систем Windows і Unix. Також, незважаючи на декларовану досконаліть мов C# і Java, витиснути C++ ним так і не вдалося.

Представлений нижче конспект лекцій є узагальненням курсу з об'єктно-орієнтованого програмування на базі мови C++, що протягом кількох років читався авторами студентам факультету кібернетики Київського національного університету.

ОСНОВНІ ТИПИ ДАНИХ МОВИ С++

Лекція 1. ТИПИ ДАНИХ У С++

У С++ є набір вбудованих типів даних для подання цілих і дійсних чисел, символів, а також тип даних "символьний масив", що служить для зберігання символьних рядків.

Тип *char* використовується для зберігання окремих символів та невеликих цілих чисел і займає 1 байт.

Типи *short*, *int* і *long* призначені для представлення цілих чисел. Вони розрізняються тільки діапазоном значень, які можуть приймати числа, а конкретні розміри перелічених типів залежать від реалізації.

Тип *short* займає половину машинного слова, *int* — одне слово, *long* — одне або два слова. У 32-бітних системах *int* і *long*, як правило, одного розміру.

Типи *float*, *double* і *long double* призначені для чисел із плаваючою крапкою й розрізняються точністю представлення (кількістю значущих розрядів) і діапазоном. Тип *float* (одинарна точність) займає одне машинне слово, *double* (подвійна точність) — два, а l*ong double* (розширена точність) — три слова.

Тип *bool* має два значення — *true* і *false* — і займає 1 біт.

Char, *short*, *int* і *long* разом складають цілі типи, які, в свою чергу, можуть бути знаковими (*signed*) і беззнаковими (*unsigned*). У знакових типах самий лівий біт використовується для зберігання знака (0 — плюс, 1 — мінус), а біти, що залишилися, містять значення. У беззнакових типах всі біти використовуються для значення. 8-бітовий тип *signed char* може представляти значення від -128 до 127, а *unsigned char* — від 0 до 255.

Коли у програмі зустрічається деяке число, наприклад 1, то це число називається літералом, або літеральною константою. Літерали цілих типів можна записати у десятковому, восьмеричному й шістнадцятиричному вигляді. Наприклад, число 20, представлене десятковим, восьмеричним і шістнадцятиричним літералами, має вигляд:

- *20 // десятковий;*
- *024 // восьмеричний;*
- *0x14 // шістнадцятиричний.*

Якщо літерал починається з 0, він трактується як восьмеричний, якщо з 0*x* або 0*X*, то як шістнадцятиричний.

За замовчуванням всі цілі літерали мають тип *signed int*. Можна явно визначити цілий літерал таким, що має тип *long*, приписавши наприкінці числа літеру *L* (використовується як прописна *L*, так і рядкова *l*). Буква *U* (або *u*) наприкінці визначає літерал як *unsigned int*, а дві літери — *UL* або *LU* — як тип *unsigned long*.

Наприклад:

128u 1024UL 1L 8Lu.

Літерали, що представляють дійсні числа, можуть бути записані як з десятковою крапкою, так і в науковій (експонентній) нотації. За замовчуванням вони мають тип *double*. Для явної вказівки типу *float* потрібно використовувати суфікс *F* або *f*, а для *long double* — *L* або *l*, але тільки у випадку запису з десятковою крапкою.

Наприклад:

3.14159F 0/1f 12.345L 0.0 3el 1.0 E-3E 2. 1.0L.

Спеціальні символи (табуляція, повернення каретки) записуються як *escape*-послідовності. Визначено наступні послідовності (вони починаються із символу зворотної косої риски):

- новий рядок *n*;
- горизонтальна табуляція *t*;
- забій *b*;
- вертикальна табуляція *v*;
- повернення каретки *r*;
- прогін аркуша *f*;
- дзвінок *a*;
- зворотна коса риска \\;
- питання \?;
- одиночні лапки \';
- подвійні лапки \";

- *escape* — послідовність загального вигляду має форму \ooo, де ooo — від одного до трьох восьмеричних цифр. Це число є кодом символу.

Наприклад:

- *\7 (дзвінок);*
- *\14 (новий рядок);*
- *\0 (null);*
- *\062 ('2').*

Змінні

Змінна, або об'єкт — це іменована область пам'яті, до якої ми маємо доступ із програми; туди можна записувати значення а потім їх використовувати. Кожна змінна C++ має певний тип, що характеризує розмір і розташування в області пам'яті, діапазон значень, який вона може зберігати, і набір операцій, які застосовуються до цієї змінної.

Наприклад:

int student_count;
double salary;
bool on_loan;
string street_address;
char delimiter.

Початкове значення може бути задане прямо в операторі визначення змінної. У C++ припустимі дві форми ініціалізації змінної — явна, з використанням оператора присвоювання:

int ival = 1024; string project = "Fantasia 2000";

і неявна, із завданням початкового значення у дужках:

int ival (1024); string project ("Fantasia 2000").

Обидва визначення еквівалентні.

Вбудовані типи даних мають спеціальний синтаксис для завдання нульового значення:

// ival одержує значення 0, a dval — 0.0
int ival = int();
double dval = double().

Зі змінною асоціюються дві величини:

1) власне значення, або *r*-значення (від *read value* — значення для читання), що зберігається у цій області пам'яті й притаманне як змінній, так і літералу;

2) значення адреси області пам'яті, що асоційована зі змінної, або *l*-значення (від *location value* — значення місця розташування) — місце, де зберігається *r*-значення, притаманне тільки об'єкту.

Наприклад, у виразі

ch = ch –'0';

змінна *ch* перебуває і ліворуч, і праворуч від символу операції присвоювання.

Праворуч розташоване значення для читання (*ch* і символьний літерал '0'): асоційовані зі змінної дані зчитуються з відповідної області пам'яті.

Ліворуч — значення місця розташування: в область пам'яті, асоційовану зі змінної *ch*, міститься результат віднімання.

Ім'я змінної, або ідентифікатор, може складатися з латинських літер, цифр і символу підкреслення. Прописні й малі літери в іменах розрізняються. Мова C++ не обмежує довжину ідентифікатора, однак користуватися занадто довгими іменами типу незручно. Ключові слова не можна використовувати як ідентифікатори.

Покажчики

Покажчик — це об'єкт, що містить адресу іншого об'єкта і дає змогу побічно маніпулювати цим об'єктом. Найчастіше покажчики використовуються для роботи з динамічно створеними об'єктами для побудови зв'язаних структур даних, таких як зв'язані списки й ієрархічні дерева, і для передачі у функції великих об'єктів — масивів і об'єктів класів.

Кожний покажчик асоціюється з деяким типом даних, причому їх внутрішнє подання не залежить від внутрішнього типу: і розмір пам'яті, що займає об'єкт типу покажчик, і діапазон значень у них однаковий. Різниця полягає в тому, як компілятор сприймає об'єкт, що адресується. Покажчики на різні типи можуть мати те саме значення, але область пам'яті, де розміщуються відповідні типи, може бути різною: покажчик на *int*, що містить значення

адреси 1000, вказує на область пам'яті 1000–1003 (у 32-бітній системі); покажчик на *double*, що містить значення адреси 1000, вказує на область пам'яті 1000–1007 (у 32-бітній системі).

Наприклад:

```
int *ip1, *ip2;
complex<double> *cp;
string *pstring;
vector<int> *pvec;
double *dp.
```

Розглянемо приклади використання покажчиків:

```
int ival = 1024;
//pi ініціалізовано нульовою адресою
int *pi = 0;
// pi2 ініціалізовано адресою ival
int *pi2 = &ival;
// правильно: pi і pi2 містять адресу ival
pi = pi2;
// pi2 містить нульову адресу
pi2 = 0;
// помилка: pi не може приймати значення int pi = ival;
double dval;
double *ps = &dval; // помилки компіляції, неприпустиме присвоювання типів даних: int* //<== double* pi = pd pi = &dval;
//void* може містити адреси будь-якого типу
void *pv = pi;
pv = pd;
// непряме присвоювання змінній ival значення ival2 *pi = ival2;
// непряме використання змінної ival як rvalue і lvalue
*pi = abs(*pi); // ival = abs(ival);
*pi = *pi + 1; // ival = ival + 1;
// C++ допускає використання покажчика на покажчик
int **ppi = &pi;
int *pi2 = *ppi;
cout << "Значення ival\n" << "явне значення: " << ival << "\n"
<< "непряма адресація: " << *pi << "\n"
<< "двічі непряма адресація: " << **ppi << endl.
```

Покажчики можуть бути використані в арифметичних виразах:

```
int i, j, k;
int *pi = &i; // i = i + 2;
*pi = *pi + 2; // збільшення адреси, що міститься в pi, на
2pi = pi + 2.
```

Специфікатор const

Специфікатор const використовується для оголошення константи:

Наприклад:

```
const int bufSize = 512;
// помилка: неініціалізована константа
const double pi.
```

Можна також використовувати покажчики на константи

```
const double *pc = 0;
const double minWage = 9.60;        // правильно: не можемо змінювати minWage за допомогою pc;
pc = &minWage;
double dval = 3.14;                 // правильно: не можемо змінювати minWage за допомогою pc, хоча dval і не константа;
pc = &dval;                         // правильно;
dval = 3.14159;                     //правильно;
*pc = 3.14159.                      // помилка.
```

Існують і константні покажчики

```
int errNumb = 0;
int *const currErr = &errNumb.
```

У цьому випадку *curErr* — константний покажчик на неконстантний об'єкт. Це значить, що ми не можемо присвоїти йому адресу іншого об'єкта, хоча сам об'єкт допускає модифікацію. Спроба присвоїти значення константному покажчику викличе помилку компіляції:

```
curErr = &myErNumb;                 // помилка.
```

Посилання

Посилання іноді називають псевдонімом. Воно використовується для завдання об'єкту додаткового імені. Посилання дає можливість побічно маніпулювати об'єктом так само, як це робиться за допомогою покажчика.

Посилання позначається оператором взяття адреси (&) перед іменем змінної. Посилання повинно бути ініціалізовано.

Наприклад:

```
int ival = 1024;
//refVal — посилання на ival
int &refVal = ival;
//помилка: посилання повинно бути ініціалізовано
int &refVal2.
```

Якщо ми визначаємо посилання в одній інструкції через кому, перед кожним об'єктом типу посилання повинен стояти амперсант (&) — оператор взяття адреси (такий самий, як і для покажчиків).

Наприклад:

```
// визначено два об'єкти типу int
int ival = 1024, ival2 = 2048;
// визначено одне посилання й один об'єкт
int &rval = ival, rval2 = ival2;
// визначено один об'єкт, один покажчик і одне посилання
int inal3 = 1024, *pi = ival3, &ri = ival3;
// визначено два посилання
int &rval3 = ival3, &rval4 = ival2.
```

Константне посилання може бути ініціалізовано об'єктом іншого типу (якщо існує можливість перетворення одного типу в інший), а також безадресною величиною, такою як літеральна константа.

Наприклад:

```
double dval = 3.14159;          // правильно тільки для
                                // константних посилань;
const int &ir = 1024;
const int &ir2 = dval;
const double &dr = dval + 1.0.
```

Лекція 2. СКЛАДОВІ ТИПИ ДАНИХ

Клас string

Необхідно підключити бібліотеку:

#include<string>,

Оголошення рядка

string st="aaa\n";

string st("aaa\n").

Функції для роботи з рядками:

1) ***size()*** — довжина рядка
   st.size ();
2) додавання рядків
   string s1="Hello1";
   string s2="Hello2";
   string s3=s1+s2; *//"Hello1Hello2";*
3) ***find()*** — пошук у рядку
   string s1="Hello1";
   int pos=s1.find("H"); *//pos=0;*
   if(pos==string::npos) *// npos-символ не знайдений*
   ... else ..;
4) ***find_first_of()*** — пошук першого входження символу в рядок (використовується аналогічно ***find()***);
5) ***substr ()*** — вирізка підстроки
   string s2=s1.substr(поч_поз, кін_поз);
6) ***erase()*** — видалення підстроки
   s1.erase(поч_поз, до-у символів);
7) ***insert ()*** — вставка підстроки
   s1.insert(поч_поз, підстрока);
8) ***replace()*** — заміна підстроки
   s1.replace(поч_поз, довжина_замін_стр, підстр);
9) ***compare()*** — порівняння
   s1.compare(s2);
10) ***c_str()*** — перетворення *string в const char*;*
11) ***assign()*** копіювання
    рядок_куди_копіюють.assign (рядок_з_якої_копіюють, кількість_символів);

12) ***swap()*** — змінює місцями значення
swap(s1,s2).

Вбудований строковий тип

Для використання потрібно підключити бібліотеку:
#include<cstring>.
Ініціалізація:
char a="Hello"*.
Функції для роботи з рядками:
1) ***int strlen(const char*)*** — довжина рядка;
2) ***int strcmp(const char*, const char*)*** — порівняння рядків;
3) ***char* strcpy(char*, const char*)*** — копіювання рядків.
Наприклад:

```
const char* a="Hello\n";
int main(){
int len=0;
while(*a++) ++len;
a= a-len-1;
cout<<len<<" "<<a<<endl;
return 0;
}.
```

Перерахування

Оголошення перерахування:
enum modes{input=1,output,append}.
Наприклад:

```
modes om=input;
cout<<om;                    //відповідь 1.
```

Одержати рядки *input*, *output*, *append* не можна.
В арифметичних виразах *enum* перетворюється в *int*.

Масиви

Оголошення:
int a[константа або константний вираз, кількість елементів масиву].

Наприклад:

int a[3];

для багатомірних масивів.

int a[3][4][5].

Ініціалізація

```
int a[3]={0,1,2};
int a[2][3]={0,1,2,3,4,5};
int a[]={0,1,2};
int *a[]={&a1,&a2,&a3};          //масив покажчиків;
int &a[]={a,b,c};                //error — масиви посилань
                                 неприпустимі.
```

Взаємозв'язок масивів і покажчиків.

Наприклад:

```
int a[3]={0,1,2};
cout<< a;                        // одержуємо 0;
cout<<*(a+1);                    //одержуємо 1;
&a[1];                           //адреса другого елемента.
```

Примітка. Не варто плутати вираз **a+1* та **(a+1)*.

Клас vector

Для використання потрібно підключити бібліотеку:

#include< vector>.

Оголошення має вигляд:

vector<mun> vec(n).

Наприклад:

vector<int> vec(4).

Використання аналогічно вбудованого масиву:

vec[1] — другий елемент вектора.

Функція ***push_back()*** заносить у вектор значення.

Наприклад:

```
vector<int> vec(2);
vec.push_back(10);
vec.push_back(15);
(у вектор занесені значення 10 і 15).
```

Клас complex

Для використання потрібно підключити бібліотеку:

#include<complex>.

Наприклад:

```
complex<int> p(1,3);          //1+3i
complex<int> p1(2,4);
complex<int> p2=p+p1;         //додавання комплексних чисел.
```

У класі ***complex*** визначені операції **+,-,\,*.**

Директива typedef

typedef – специфікатор типів.

Наприклад:

```
typedef double wages;
typedef vector<int> vec;
typedef int *p.
```

Кваліфікатор volatile

Ціль *volatile* – повідомити компілятора про те, що той не може визначити, хто і як може змінити значення, і тому компілятор не повинен виконувати оптимізацію коду.

Наприклад:

```
volatile int a.
```

Вираз

Вираз складається з одного або більше операндів і операцій.

Наприклад: *a*b.*

Операції бувають унарні та бінарні.

Арифметичні операції:

****, +,-,\,%(остача від ділення), ++, --.***

Логічні операції:

!(логічне не), <, >, <=, >=, ==(порівняння на рівність), !=(не дорівнює), &&(логічне та), || (логічне або).

Операція присвоєння

Операція присвоєння має вигляд = .

Наприклад:

int a=3;

val=jval=0.

Умовний вираз

Умовний вираз у С++ має вигляд:

(умова)? Вираз_якщо_умова_істина: вираз_якщо_умова_хибна.

Наприклад:

(a>0)?b=3:b=4.

Оператор sizeof

Оператор *sizeof()* — повертає розмір об'єкта в байтах.

Наприклад:

int a;

cout<<sizeof(a); *//одержуємо розмір a в байтах.*

Оператори new і delete

Оператор *new* — динамічне виділення пам'яті в купі.

Наприклад:

int p=new int;*

створення в купі комірки, в яку може бути записане число цілого типу, а в *p* міститься адреса цієї комірки;

*int *p=new int(100);*

ініціалізація значенням *100*;

int p1=new int [n];*

створення динамічного масиву з *n*-елементів, при цьому *n* не обов'язково повинно бути константою. Працювати з таким масивом можна також як і з вбудованим масивом;

*int (*p2)[100]=new int [n][100];*

створення двомірного масиву.

Оператор *delete* — видалення елемента з купи.

```
delete p;
delete [] p1;                    //видалення масиву.
```

Питання. Як видалити двомірний масив?

Оператор кома

Наприклад:

int val=(ia!=0)?ix=3, a=ix:ia=10, b=a.

Результатом першої частини виразу буде *ix*, а другої *a*. Результат оператора кома — результат самого правого виразу.

Побітові оператори

Існують наступні побітові оператори:

~ — побітове не : ***~вираз;***

<< — зміщення вліво: ***вираз 1<< вираз 2;***

>> — зміщення вправо: ***вираз 1>> вираз 2;***

& — побітове І: ***вираз 1*** **&** ***вираз 2;***

| — побітове АБО: ***вираз1 | вираз2***

&= — побітове І із присвоєнням: **вираз1&=вираз2;**

^= — побітове виключне АБО із присвоєнням: вираз**1^=вираз2;**

|= — побітове АБО із присвоєнням: вираз**1|=вираз2;**

<<= — зміщення вліво із присвоєнням: **вираз1<<=вираз2;**

>>= — зміщення вправо із присвоєнням: **вираз1>>=вираз2.**

Наприклад:

```
unsigned int a=0;          // всі біти рівні 0;
a=1<<27;                   // в 27 біті 1;
a|=1<<27;                  // всі біти без зміни, а в 27
                              біті 1;
bool b=a&(1<<27);          //побітове І.
```

Клас bitset

Для використання класу *bitset* підключаємо бібліотеку:

#include<bitset>.

Оголошення:

bitset<32> a; *//змінна а має 32 біта.*

Методи класу bitset

test(pos) — повертає *true*, якщо біт *pos* дорівнює 1;
any() — повертає *true*, якщо хоча б один біт дорівнює 1;
none() — повертає *true*, якщо жоден біт не дорівнює 1;
count() — кількість бітів рівних 1;
size() — загальна кількість бітів;
[pos] — доступ до біта *pos* (*a*[*pos*]);
flip() — змінити значення всіх бітів;
flip(pos) — зміна значення біта *pos*;
set() — вставити всі біти в 1;
set(pos) — вставити біт *pos* в 1;
reset() — вставити всі біти в 0;
reset(pos) — вставити біт *pos* в 0;
to_string() — перетворення в *string*;
to_long() — перетворити в *unsigned long*.

Структури

Структура оголошується у такий спосіб:
struct my_struct{
int a;
double b;}.
Структуру можна оголосити в альтернативний спосіб:
typedef struct{
int a;
double b;} my_struct.
Приклад використання:

```
int main(){
my_struct s1;
s1.a=3;
s1.b=4.5;
my_struct* s2;
```

```
s 2-2->a=3;                    //(*s2).a=3;
s 2-2->b=4.5;                  //(*s2).b=4.5;
...
return 0;
}.
```

Перетворення типів

У С++ існують явні й неявні перетворення типів.

Явні перетворення типів виконуються за допомогою оператора ***static_cast<тип, до якого перетворюється >.***

Наприклад:

```
double a=3.56;
int val=static_cast<int>(a).
```

Оператор ***const_cast<тип, до якого перетворюється>*** здійснює перетворення константного типу в неконстантний.

Наприклад:

```
const char* a;
char* b=const_cast<char*>(a).
```

Оператор ***reinterpret_cast<тип до якого відбувається перетворення>*** працює із внутрішнім представленням об'єктів. Відповідальним за наслідки такого перетворення є програміст.

Наприклад:

```
complex<int>* pcompl;
char* p=reinterpret_cast<char*>(pcompl).
```

Оператор ***dynamic_cast<тип, до якого здійснюється перетворення>*** застосовується при ідентифікації типу під час виконання.

Існує застаріла форма явного перетворення: ***(тип) вираз***.

Наприклад:

```
double a=23.6;
int val=(int) a.
```

Неявне перетворення типів

С++ визначає набір стандартних неявних перетворень типів у наступних випадках.

1. Арифметичне перетворення з операндами різних типів.
Усі операнди приводяться до найбільшого типу.
Наприклад:

```
int a=3;
double b=3.45;
a+b;                    //a приводиться до double.
```

При арифметичних перетвореннях виконуються наступні правила:

- типи завжди приводяться до типу, що здатний забезпечити найбільший діапазон значень при найбільшій точності;
- будь-який арифметичний вираз, який включає в себе цілі операнди типів, що менші за *int*, перед обчисленням завжди приводяться до *int*.

Ієрархія правил перетворення

Якщо один з операндів має тип *long double*, то інші операнди виразу приводяться до нього.

Якщо у виразі максимальний тип операндів *double*, то інші типи приводяться до нього.

У випадку цілочисельних операндів всі типи приводяться до *int*. Але якщо є хоча б один операнд із типом *unsigned long*, то інші операнди приводяться до цього самого типу.

Якщо максимальний тип *long*, то всі операнди приводяться до цього типу, але *unsigned int* перетворюється в *long* тільки у тому випадку, якщо він здатний умістити в себе *unsigned int*.

2. Присвоєння значення виразу одного типу іншому.
У цьому випадку результуючим є тип об'єкта, якому значення присвоюється.
Наприклад:

```
int* p=0;          // 0 типу int присвоюється
                   значенню типу int*.
```

3. Передача функції аргументу, тип якого відрізняється від типу відповідного формального параметру.

Тип фактичного аргументу приводиться до типу формального параметра.

Наприклад:

```
void f(int );
double a=3.2;
f(a);                                  // a приводиться до типу int.
```

4. Повернення значення з функції, тип якого не збігається з типом результату, що повертається, заданим в оголошенні функції.

Тип значення, що повертається фактично, приводиться до оголошеного.

Наприклад:

```
double f(int a,int b);
{return a+b;}.
```

Лекція 3. ІНСТРУКЦІЇ

Порожня інструкція *";"*

Наприклад:

int a=23;.

Інструкції оголошення

Інструкція оголошення може бути простою.

Наприклад:

int val.

Також інструкція оголошення може бути складною.

Наприклад:

vector<string>::iterator iter=text.begin(),
end=text.end().

Умовні інструкції

Інструкція *if*

Синтаксис має вигляд:

if (умова) { інструкції, якщо умова true; }
else {інструкції, якщо умова false; }.

Наприклад:

1) *if (a>b) a=3; else a=4;*
2) *if (val=3) val++;*
3) проблема *else*

if (a>b) a=4;
if (b>c) b=3;
else c=10;//питання: до якого if належить else. У даному випадку до другого if, але це не очевидно.

Інструкція *switch*

Синтаксис має вигляд:

switch (вираз){
case константний_вираз_1: { список_операторів_1; }
case константний_вираз_2: { список_операторів_2; }
…

case константний_вираз_n: { список_операторів_n; }
[default: оператори].
}.

Наприклад:

```
int main(){
char ch;
int a_count, e_count, i_count, o_count, u_count;
while(cin>>ch)
switch (ch) {
    case 'a': case 'A': ++a_count;break; // якщо ch дорівнює
                                         'a', або 'A'
    case 'e': case 'E': ++e_count;break;
    case 'i': case 'I': ++i_count;break;
    case 'o': case 'O': ++o_count;break;
    case 'u': case 'U': ++u_count;break;
default: cout<<"The letter is not vowel";
break;
}
return 0;
}.
```

Інструкція *for*

Синтаксис циклу *for* такий:

for (інструкція ініціалізації; умова; вираз)
{інструкції;}

Порядок обчислень такий:

1. Виконується *інструкція ініціалізації*. Ця інструкція виконується тільки один раз.

2. Перевіряється *умова*, якщо вона *true*, то виконуються *інструкції* в циклі. Якщо найперше обчислення умови дає *false*, то *інструкції* виконуватися не будуть.

3. Виконується *вираз*.

Наприклад:

```
1) int s=0, a=1;
for(int i=0; i<100; i++)
```

*{ a=a*i; s=s+a; };*

2) *int i=0, s=0, a=1;*
for(; i<100; i++)
*{ a=a*i; s=s+a;};*

3) *int i=0, s=0, a=1;*
for(; i<100;)
*{ a=a*i; s=s+a; i++;};*

4) нескінченний цикл
for(; ;)
{ інструкції;}

вихід з такого циклу можна здійснити, наприклад, за допомогою інструкції *break*;

5) *for (int i=0, a=1; i<100; i++, j++)*
*{ a=a*i; s=s+a+j; }.*

Інструкція *while*

Синтаксис циклу *while* такий:
while (умова)
{ інструкції;}.
Цикл виконується поки умова *true*.
Наприклад:

int i=0,s=0,a=1;
while(i<100)
*{ a=a*i; s=s+a; i++;}.*

Інструкція *do while*

Синтаксис циклу *do while* такий:
do
{ інструкції; }
while (умова).
Інструкції виконуються доти, поки умова *true*.
Наприклад:

int i=0, s=0, a=1;

```
do {a=a*i; s=s+a; i++;}
while(i<100).
```

Інструкція *break*

Інструкція *break* припиняє роботу циклів *for, while, do while* і блоку *switch*. Використання інструкції *break* у блоці *if*, що не міститься у циклі, або *switch* – є синтаксичною помилкою.

Наприклад:

```
int i=0,s=0,a=1;
for (; ;)
{ if (i<100) break;
   else{
   a=a*i;
   s=s+a;
   i++; }
}.
```

сюди передається керування після спрацьовування інструкції *break*.

Інструкція *continue*

Інструкція *continue* завершує поточну ітерацію циклу й передає керування на обчислення умови, після чого цикл може тривати.

Наприклад:

```
int i=0,s=0,a=1;
while(i<100)
{ i++;
if (i==50)
continue; // керування передається на обчислення умови
i<100, оператори, які //знаходяться нижче не виконуються.
a=a*i;
s=s+a;} .
```

Інструкція *goto*

Синтаксис

***goto мітка*.**

Інструкція *goto* здійснює безумовний перехід до іншої інструкції всередині цієї самої функції.

Наприклад:

```
void f(int a)
{
...
goto lab1;
...
lab1: ...;
}.
```

Не можна ставити *мітку* безпосередньо перед закриваючою фігурною дужкою.

Наприклад:

```
{
...
lab1:}                          // error.
```

Потрібно писати наступним чином

```
{
...
lab1: ;
}.
```

Лекція 4. ФУНКЦІЇ

Функція — операція, визначена користувачем. Функція визначається у такий спосіб:

int max (int, int);

int — тип значення, що повертається; *max* — ім'я функції вибране користувачем; *(int, int)* — типи формальних параметрів; *max (int, int)* — сигнатура функції.

Реалізація функції має вигляд:

```
int max (int a,int b)
{//тіло функції
if (a>b) return a;
else return b;
}.
```

Виклик функції має вигляд:

```
int main()
{
int c, d;
cin>>c,d;
int f=max(c,d);
cout<<f<<endl;
return 0;
}.
```

Існує два можливих способи обробки викликів функції:

1 спосіб — *inline int max(int a,int b){…}*.

У цьому випадку в місце виклику функції підставляється тіло функції.

2 спосіб — *int max(int a, int b) {…}*.

У цьому випадку відбувається звичайний виклик функції, що приводить до передачі керування цій функції під час її виконання. Робота функції завершується *return*.

Зауваження

Функція повинна бути оголошена до моменту її виклику.

Як правило функція оголошується у заголовному файлі *.h*
Наприклад:

```
----------max.h----------
int max(int,int);
----------max.cpp----------
int max(int a,int b)
{ if (a>b) return a;
else return b;}
----------main.cpp----------
#include<iostream>
#include "max.h"
using namespace std;
int main()
{ int c=10,d=20;
cout<<max(c,d);
return 0; }.
```

В оголошенні функції описується її інтерфейс *int max(int,int)*.
Функція, яка нічого не повертає, визначається як *void*.
Наприклад:

void f(int a){...}.

У функціях, в яких тип значення, що повертається — *void*, використовувати *return* не обов'язково.

Прототип функції

Прототип функції описує її інтерфейс і складається з типу значення, що повертається, імені та списку параметрів.
Наприклад:

int max(int, int).

Розглянемо всі ці параметри.

Список параметрів функції

Список параметрів функції не може бути опущений.
Наприклад, наступні оголошення еквівалентні:

int f();
int f(void);

Після типу може стояти параметр, але при цьому не можна використовувати скорочений запис.

Наприклад:

```
int max(int a, int b);          // правильно;
int max(int a,b);               // помилка.
```

Імена параметрів не можуть повторюватися:

```
int max(int a, double a);       //помилка.
```

Імена в оголошенні функції можна використовувати в її тілі:

```
void f(int a)
{...
a=4;
...}.
```

Імена параметрів в оголошенні і визначенні однієї й тієї самої функції можуть не збігатися:

```
int max(int a, int b);
int max(int c, int d)
{...}.
```

C++ здійснює перевірку типів формальних параметрів у випадку їх не співпадіння з типами фактичних параметрів і намагається здійснити неявне перетворення типів, якщо це неможливо — видає помилку.

Наприклад:

```
int f(int c,int d) {...};
int main(){
f("Hello","world");          //error неможливо перетворити char* до int;
f(34.5, 34);                 //відбудеться перетворення типу double до int у першому аргументі;
f(34);                       //error відсутній другий аргумент.
```

Передача аргументів у функцію

Функції використовують пам'ять зі стека програми. Деяка область стека виділяється функцією й залишається з нею до за-

кінчення її роботи. По завершенню роботи функції пам'ять звільняється. Кожному параметру функції виділяється місце у певній області. Під час виклику функції пам'ять ініціалізується значеннями фактичних аргументів.

Стандартний спосіб передачі аргументів є копіювання їх значень, тобто передача за значенням. При цьому аргументи не змінюють свого значення.

Наприклад:

```
void swap(int a, int b)
{
int c=a; a=b; b=c;
}
int main(){
int c=3, d=4;
swap(c, d);
cout<<c<<d;
return 0;}.
```

На екрані одержимо знову значення *3* та *4*.

Параметри посилання

При такій передачі аргументів функція маніпулює локальними копіями аргументів. Вона одержує *lvalue* своїх аргументів і може їх змінювати.

Наприклад:

```
void swap(int& a, int& b){
int c=a; a=b; b=c;}
int main(){
int c=3,d=4;
swap(c,d);
cout<<c<<d;
return 0;}.
```

На екрані одержимо *4* та *3*, як і передбачалось.

Передачу параметрів за посиланням зручно використовувати у випадку передачі у функцію великих об'єктів. У такий спосіб можна уникнути непотрібного копіювання.

Наприклад:

```
struct my_struct{
double a[200];
...;
void f(const my_struct& sobj){...};
int main(){
my_struct table[1000];
for(int i=0; i<1000;i++)
f(table[i]);
...
return 0;}.
```

Модифікатор *const* використовується для того, щоб гарантувати неможливість зміни об'єкта *sobj*.

Параметри покажчики

У функціях можна використовувати параметри покажчики.

Наприклад:

```
void swap(int* a, int* b){
int c=*a; *a=*b; *b=c;}
int main(){
int c=3,d=4;
swap(&c,&d);
cout<<c<<d;
return 0;}.
```

На екрані одержимо *4* та *3*.

У цьому випадку використання покажчиків привело до того самого ефекту, що й використання посилань, однак у випадку покажчиків виклик менш явний.

Посилання vs покажчики

Покажчик варто використовувати, якщо параметр повинен вказувати на різні об'єкти під час виконання функції.

Однак при застосуванні покажчика слід пам'ятати, що перед його використанням необхідно перевірити, чи не дорівнює він 0.

Наприклад:

```
struct my_struct{...};
void f(my_struct* ps){
if(ps!=0)                    //звертаємося до об'єкта
}.
```

У випадку використання параметрів-посилань цього робити не потрібно.

Найважливіша сфера застосування посилань — це ефективна реалізація перевантажених операцій.

Наприклад:

```
struct matrix{
int a[100][100];
...};
matrix operator+(matrix* a, matrix* b)
{matrix c;
...
                             //необхідні операції
...
return  c;}
int main(){
matrix a,b,c;
c=&a+&b;                     //не явний виклик
...
return 0;}.
```

У випадку використання параметрів посилань маємо:

```
matrix operator+(const matrix& a, const matrix& b)
{matrix c;
...
                             //необхідні операції
...
return  c;}
int main(){
matrix a,b,c;
c=a+b;                       //явний виклик operator+
...
return 0;}.
```

Параметри масиви

У C++ масиви ніколи не передаються за значенням, а тільки як покажчик на нульовий елемент. Наступні оголошення функції еквівалентні:

void f(int* a);
void f(int a[]);
void f(int a[10]).

Передача масивів як покажчиків має наступні особливості:

1. Зміна значення аргументу всередині функції змінює об'єкт, що передається, а не його копію. Якщо така дія небажана, то можна вказати функцію так, що вона не буде змінювати значення:

void f(const int a[10]).

2. Розмір масиву не є частиною типу параметра, тому функція не знає реального розміру масиву. Бажано вживати наступне оголошення:

void f(int a[], int size); *// size — розмір масиву.*

Інший спосіб повідомити функції розмір масиву — оголосити параметр як посилання.

Наприклад:

```
void f(int (&a)[10]){...;
int main(){
int b[5];
int c[10];
f(b);                          //error масив b не є масивом
                               10 елементів;
f(c);                          //правильно;
return 0;}.
```

Параметром може бути багатомірний масив. Для такого параметра повинні бути задані праві межі всіх вимірів, крім першого.

Наприклад:

int f(int a[][10], int rowsize).

Еквівалентне оголошення має вигляд:

*int f(int (*a)[10]).*

Багатомірний масив передається як покажчик на його нульовий елемент. Як і у випадку одномірного масиву, перший вимір багатомірного масиву не впливає на тип параметра.

Так, параметрами можуть виступати класи.

Наприклад:

void f(complex<int>a).

Значення параметрів за замовчуванням

Для переданих параметрів можна вказувати значення за замовчуванням.

Наприклад:

```
void f(int a=10, int b=20){
cout<<a<<b;}
int main(){
int c=30,d=40;
f(c,d);                    //на екрані 30 і 40
f();                       //на екрані 10 і 20
f(c);                      //на екрані 30 і 20
return 0;}.
```

Параметри за замовчуванням повинні бути розміщені наприкінці.

Наприклад:

void f(int a, int b=10).

Однак наступна комбінація помилкова:

```
int f(int a, int b=10, int c=20);
int f(int a, int b, int c=20).
```

Три крапки — "…"

Якщо не можна перелічити типи й число можливих аргументів, то варто використовувати "**…**":

void f(int a,…);

void f(…).

Наступні оголошення НЕ еквівалентні:

void f();

void f(…).

Наступні виклики коректні:

f(a,b,c);

f(a);
f().

Розглянемо приклад використання "**...**":

```
#include<iostream>
#include<stdarg>
using namespace std;
void f(int a,...){
   va_list ap;
   int p;
   va_start (ap,a);
   while(p=va_arg(ap,int))!=0) //продовжуємо вибирати
                               значення зі стеку доки не зу-
                               стрінеться 0;
   cout<<p<<endl;
   va_end(ap);}
int main() {
int k=2,s=100,s1=200,s3=0;
f(k,s,s1,s3);
return 0;}.
```

Повернення значення

Для повернення значення з функції використовується інструкція ***return***. Ця інструкція вживається у двох формах:

return;

return expression.

Перша форма використовується для функцій з типом значення ***void***, що повертається. Використовувати ***return*** у цьому випадку можна, якщо потрібно примусово завершити роботу функції.

Наприклад:

```
void (int s, int l, int d)
{
if(a==l)
 return;                          //завершення роботи функції;
...
}.
```

У другій формі інструкції ***return*** вказується значення, що функція повинна повернути. Це значення може бути виразом:

*return a*f(b).*

Якщо тип значення, що повертається, не точно відповідає зазначеному в оголошенні, то застосовується неявне перетворення типів.

Можливі три форми повернення значення функції.

1 — повернення за значенням.

У цьому випадку функція одержує копію результату обчислень.

Наприклад:

```
int grow(double* x){
int b;
...
return b;
}.
```

2 — повернення за покажчиком.

Наприклад:

```
int* grow(int a)
{ int* b=&a;
return b;}.
```

У цьому випадку повертається покажчик на об'єкт.

3 — повернення за посиланням.

При поверненні за посиланням, що викликає, функція одержує ***lvalue*** для відповідної змінної. Тому функція може модифікувати відповідний об'єкт.

Наприклад:

```
int& grow(int a){
int* b=&a;
return *b;}.
```

Повернення за посиланням ефективне у випадку повернення великого об'єкта.

При оголошенні функції, що повертає посилання, варто пам'ятати про можливі помилки:

1. Повернення посилання на локальний об'єкт, тривалість життя якого обмежено часом виконання функції. По завершенні

роботи функції такому посиланню відповідає область пам'яті, яка містить невизначене значення.

Наприклад:

```
int& grow(int a, int b){
int c=10;
if(a>b) c=20;
return c;}.
```

2. Функція повертає ***lvalue*** і будь-яка його модифікація змінює сам об'єкт.

Наприклад:

```
int& func ( int a) { return a;}
int main()
{
int k=10;
func(10)++;
return 0;
}.
```

Для запобігання несанкціонованої модифікації тип об'єкта, що повертається, варто використовувати ***const***.

Наприклад:

```
const int& func(int a){...}.
```

Параметри й об'єкти, що повертаються, vs-глобальні об'єкти

Різні функції у програмі можуть взаємодіяти між собою за допомогою двох механізмів: перший — використання глобальних об'єктів, другий — передача параметрів і повернення значень. Механізм глобальних об'єктів краще не використовувати.

Рекурсія

С++ допускає рекурсію. Наприклад, розглянемо обчислення *n*!:

```
int fact(int n){
if (n>0) return n*fact( n-1);
else return 1;}.
```

Директива лінкування *extern “C”*

У деяких ситуаціях необхідно використовувати функції, написані на інших мовах, наприклад, на ***“C”***. Для вказівки компіляторові, що функція написана на іншій мові, використовується директива ***extern***.

Наприклад:

```
extern “C” void exit(int);
extern “C”{
int f(int);
int g(int);
}
extern “C”{
#include<cmath>
}.
```

Фігурні дужки варто вказувати після директиви ***extern “C”***. Вони не обмежують області видимості.

У цьому випадку вважається, що всі функції, які оголошені у заголовному файлі ***cmath.h***, написані на ***C***.

Директива ***extern*** не може з’являтися в тілі функції.

Наприклад:

```
int main(){
extern “C” double sqrt(double); //error
...
return 0;}
```

Коректним буде таке оголошення:

```
extern “C” double sqrt(double);
int main(){
...}.
```

Зазвичай ця директива міститься у заголовний файл (*.h*).

Єдина зовнішня мова, підтримку якої гарантує компілятор ***C++*** — це ***C***. Але деякі мови підтримують й інші мови, наприклад **gcc** підтримує мову ***Ada***. У цьому випадку директива лінкування має вигляд: ***extern “Ada”***.

Функція main

При запуску програми їй можна дати деякі параметри.

Наприклад:

myprog.exe -d -o -f data.txt

У цьому випадку функція ***main()*** програми *myprog* повинна мати вигляд:

int main(int argc, char argv[]){...}.*

Результатом такого виклику отримаємо:

argc=5;
argv[0]=“myprog”;
argv[1]=“-d”;
argv[2]=“-o”;
argv[3]=“-f”;
argv[4]=“data.txt”.

Розбір параметрів командного рядка можна вести за допомогою операторів ***if*** або ***switch***.

Лекція 5. ФУНКЦІЇ ВВОДУ-ВИВОДУ

Функція printf

Функція ***printf()*** є функцією стандартного виводу. За допомогою цієї функції можна вивести на екран монітора рядок символів, число, значення змінної...

Функція ***printf()*** має прототип у файлі *stdio.h*

int printf(char *керуючий рядок, ...).

У випадку успіху функція **printf()** повертає число виведених символів.

Керуючий рядок містить два типи інформації: символи, які безпосередньо виводяться на екран, і специфікатори формату, що визначають, як виводити аргументи.

Функція ***printf()*** — функція форматованого виводу. Це означає, що в параметрах функції необхідно вказати формат даних, які будуть виводитися. Формат даних вказується специфікаторами формату. Специфікатор формату починається із символу %.

Специфікатори формату:

%c — символ;

%d — ціле десяткове число;

%e — десяткове число у вигляді x.xxe+xx;

%f — десяткове число з плаваючою комою xx.xxxx;

%g ***%f*** або ***%e***, — що коротше;

%o — восьмиричне число;

%s — рядок символів;

%u — без знакове десяткове число;

%x — шістнадцятиричне число;

%% — символ %;

%p — покажчик.

Крім того, до команд формату можуть бути застосовані модифікатори *l* і *h*.

%ld — друк long int;

%hu — друк short unsigned;

%Lf — друк long double.

У специфікаторі формату після символу **%** може бути вказана точність (кількість цифр після коми). Точність задається таким чином: **%.*n*<код формату >**, де ***n*** — число цифр після коми, а **<код формату >** — один із наведених кодів.

Наприклад:

якщо є змінна *x=10.3563* типу float і необхідно вивести її значення з точністю до 3-х цифр після коми, то ми повинні написати:

printf(“змінна x = %.3f”,x).

Результат:

змінна x = 10.356.

Ви також можете вказати мінімальну ширину поля, що відводиться для друку. Якщо рядок або число більше вказаної ширини поля, то рядок або число друкується повністю.

Наприклад:

printf(“%5d”,20);

результат буде:

20.

Варто помітити, що число *20* надрукувалося не із самого початку рядка. Якщо ви хочете, щоб невикористане місце поля заповнювалися нулями, то потрібно поставити перед визначенням ширини поля символ 0.

Наприклад:

printf(“%05d”,20);

Результат:

00020.

Крім специфікаторів формату даних у керуючому рядку можуть перебувати керуючі символи:

f — нова сторінка;

n — новий рядок;

r — повернення каретки;

t — горизонтальна табуляція;

v — вертикальна табуляція;

“ — подвійні лапки;

’ — апостроф;

\ — зворотна коса риска;

a — сигнал;
N — восьмерична константа;
xN — шістнадцятирична константа;
? — знак питання.

Розглянемо приклади

Приклад 1:

```
#include <stdio.h>
void main()
{
int a,b,c;                    //Оголошення змінних a, b, c;
a=5;
b=6;
c=9;
printf("a=%d, b=%d, c=%d",a,b,c);
}.
```

Результат роботи програми:
a=5, b=6, c=9.

Приклад 2:

```
#include <stdio.h>
void main()
{
float x,y,z;
x=10.5;
y=130.67;
z=54;
printf("Координати об'єкта: x:%.2f, y:%.2f, z:%.2f", x, y, z);
}.
```

Результат роботи програми:
Координати об'єкта: *x:10.50, y:130.67, z:54.00.*

Приклад 3:

```
#include <stdio.h>
void main()
{
```

```
int x;
x=5;
printf("x=%d", x*2);
}.
```

Результат роботи програми:

x=10.

Приклад 4:

```
#include <stdio.h>
void main()
{
printf(""Текст у лапках"");
printf("%nВміст кисню: 100%%");
}
```

Результат роботи програми:

"Текст у лапках"
Вміст кисню: 100%.

Приклад 5:

```
#include <stdio.h>
void main()
{
int a;
a=11;                          // 11 у десятковій дорівнює
                               b у шістнадцятиричній;
printf(" a-dec=%d, a-hex=%X",a,a);
}.
```

Результат роботи програми:

a-dec=11, a-hex=b

Приклад 6:

```
#include <stdio.h>
void main()
{
printf("Доброго дня!n");       //після друку буде перехід
                               на новий рядок — n;
printf("Мене звати Павло.");   //це буде надруковано на
                               новому рядку.
}.
```

Результат роботи програми:

Доброго дня!
Мене звати Павло.

Функція scanf

Функція ***scanf()*** — функція форматованого вводу. За її допомогою ви можете вводити дані зі стандартного пристрою вводу (клавіатури). Даними, що вводяться, можуть бути цілі числа, числа із плаваючої комою, символи, рядки й покажчики.

Функція ***scanf()*** має наступний прототип у файлі *stdio.h:*

int scanf(char *керуючий рядок).

Функція повертає число змінних, яким було присвоєне значення.

Керуючий рядок містить три види символів: специфікатори формату, пробіли й інші символи. Специфікатори формату починаються із символу **%**.

Специфікатори формату:

%c — читання символу;
%d — читання десяткового цілого;
%e — читання числа типу *float* (плаваюча кома);
%h — читання *short int*;
%o — читання восьмеричного числа;
%s — читання рядка;
%x — читання шістнадцятиричного числа;
%p — читання покажчика.

При введенні рядка за допомогою функції ***scanf()*** (специфікатор формату ***%s***), рядок вводитися до першого пробілу, тобто якщо ввести рядок “Привіт світ!” з використанням функції ***scanf()***, отримаємо:

char str[80] — масив на 80 символів;

scanf(“%s”,str) — після введення результуючий рядок, що буде зберігатися у масиві *str*, буде складатися з одного слова “Привіт”. Функція вводить до першого пробілу! Якщо необхідно вводити рядок із пробілами, то використовуйте функцію

char *gets(char *buf).

За допомогою функції ***gets()*** можна вводити повноцінні рядки. Функція ***gets()*** читає символи з клавіатури до появи символу нового рядка ***(n)***. Сам символ нового рядка з'являється, коли ви натискаєте клавішу ***enter***. Функція повертає покажчик на ***buf*** — буфер (пам'ять), де зберігаються введені строки.

Наприклад:

```
include <stdio.h>
void main()
{
char buffer[100];          // масив (буфер), де зберігається введений рядок;
gets(buffer);              // вводимо рядок і натискаємо enter;
printf("%s",buffer);       // вивід введеного рядка на екран.
}.
```

Для введення даних за допомогою функції ***scanf()***, як параметри потрібно передавати адреси змінних, а не самі змінні.

Наприклад:

```
#include <stdio.h>
void main(void)
{
int x;
printf("Введіть змінну x:");
scanf("%d",&x);
printf("Змінна x=%d",x);
}.
```

Роздільниками між двома числами, що вводяться, є символи пробілу, табуляції або нового рядка. Знак * після **%** і перед кодом формату (специфікатором формату) дає команду прочитати дані зазначеного типу, але не присвоїти це значення, розглянемо:

scanf("%d%*c%d",&i,&j) — при введенні 50+20 відбудеться присвоєння змінній *i* — значення 50, змінній *j* — значення 20, а символ + буде прочитаний і проігнорований.

У команді формату може бути вказана найбільша ширина поля, яка підлягає зчитуванню, розглянемо:

scanf("%5s",str) — вказує на необхідність прочитати з потоку вводу перші 5 символів. При введенні 1234567890ABC масив *str* буде містити тільки 12345, інші символи будуть проігноровані. Роздільники: пробіл, символ табуляції й символ нового рядка — при введенні символу сприймаються як і всі інші символи.

Однієї з потужних особливостей функції ***scanf()*** є можливість задавати множину пошуку ***(scanset)***. Множина пошуку визначає набір символів, з якими будуть порівнюватися прочитані функцією ***scanf()*** символи. Функція ***scanf()*** читає символи доти, поки вони зустрічаються у множині пошуку. Якщо символ, що введений, не знайдений у множині пошуку, функція ***scanf()*** переходить до наступного специфікатора формату. Множина пошуку визначається списком символів, вкладених у квадратні дужки. Перед відкриваючою дужкою ставиться знак **%**.

Наприклад:

```
include <stdio.h>
void main()
{
char str1[10], str2[10];
scanf("%[0123456789]%s", str1, str2);
printf("n%sn%s", str1, str2);
}.
```

Введемо набір символів:

12345abcdefg456.

На екрані програма видасть:

12345

abcdefg456.

При визначенні множини пошуку можна також використовувати символ "дефіс" для визначення проміжків, а також максимальну ширину поля вводу.

```
scanf("%10[ A-Z1-5], str1).
```

Можна також визначити символи, які не входять у множину пошуку. Перед першим з цих символів ставиться знак ^. Множина символів розрізняє рядкові й прописні букви.

Розглянемо приклади

Приклад 1:

```
#include <stdio.h>
void main(void)
{
int x, y;
printf("nКалькулятор:");
scanf("%d+%d", &x, &y);
printf("n%d+%d=%d", x, y, x+y);
}
```

Приклад 2:

```
#include <stdio.h>
void main()
{
char name[5];
printf("nВведіть ваш логін (не більше 5 символів):");
scanf("%5s", name);
printf("nВи ввели %s", name);
}.
```

Приклад 3:

```
#include <stdio.h>
void main()
{
char bal;
printf("Ваша оцінка 2,3,4,5:");
scanf("%[2345]", &bal);
printf("nОцінка %c", bal);
}.
```

Покажчики на функції

Нехай є функція:

int max(int a, int b){…}.

Покажчик на функцію оголошується у такий спосіб:

int (*pf) (int, int).

У цьому разі ***pf*** оголошений як покажчик на функцію. Покажчик здатний адресувати функцію зі співпадаючою сигнатурою.

Ініціалізація й присвоювання

Для того, щоб ***pf*** вказував на функцію ***max***, необхідно написати:

pf=max;

інший варіант:

int (*pf) (int, int) =max.

Операція присвоювання відбудеться успішно, якщо тип значення, що повертається, а також типи параметрів у точності співпадуть. Інакше — помилка.

Можна також оголосити:

int (*pf) (int, int) =0;

pf=max.

Виклик

Покажчик на функцію застосовується для виклику функції, на яку він вказує.

Наприклад:

```
int max(int a, int b){...};
int (*pf) (int, int)=max;
void main(){
int k=20, l=10;
int s=max(k,l);
int s1=pf(k,l);                    //те ж саме, що й max
int s2=(*pf)(k,l);
...}.
```

Масиви покажчиків на функції

Масив покажчиків на функції оголошується у такий спосіб:

int (*pf[10]) (int,int).

Можна також використовувати директиву:

typedef int (*pf[10]) (int,int);

pf func_mas[10].

Ініціалізація масиву покажчиків на функції має вигляд:

```
int max(int a, int b){...};
int min(int c, int d){...};
typedef int (*pf)(int,int);
pf func_mas[2]={max,min};
```

або альтернативне присвоювання

```
pf (*func_mas)[2]={max,min}.
```

Виклик буде мати вигляд:

```
void main(){
int k=10;
int l=20;
int s=func_mas[0](k,l);          //виклик max;
int s1=func_mas[1](k,l);         //виклик min;
```

альтернативний виклик має вигляд

```
int s=((*func_mas)[0])(k,l);     //виклик max;
...
```

Наприклад:

```
int max(int a, int b){...};
int min(int c, int d){...};
typedef int (*pf)(int,int);
pf func_mas[2]={max,min};
int calc(int a, int b, pf func)
{ return func(a,b);}
int main(){
int k=10, l=20;
for(int i=0;I<2;i++)
cout<<calc(k,l,func_mas[i])<<endl;
return 0;}.
```

Покажчик на функцію може бути також і типом значення, що повертається.

Наприклад:

```
int ((*ff)(int))(int,int),
```

де *ff* — оголошена як функція, що має один параметр типу *int* і повертає покажчик на функцію типу *int (*)(int,int)*;

Наприклад:

```
int max(int a, int b){...};
int min(int c, int d){...};
typedef int (*pf)(int,int);
pf func_mas[2]={max,min};
pf func(int n){
return func_mas[n];}
```

```
pf ((*ff)(int))=func;
void main(){
int k=10, l=20;
pf func1=ff(0);
cout<<func1(k,l)<<endl;}.
```

Покажчики на функції оголошені як extern "C"

У C++ можна оголошувати покажчики функції, що написані на інших мовах:

```
extern "C" int (*pf)(int,int).
```

Наприклад:

```
extern "C" int max(int,int);
extern "C" int (*pf)(int,int)=max;
void main(){
...
cout<<pf(10,20);}.
```

Варто зазначити, що функція ***max*** повинна бути оголошена як ***extern "C"***.

```
int (*pf1)(int,int)=0;
extern "C" int max(int,int);
extern "C" int (*pf)(int,int)=max;
void main(){
pf1=pf;                          //error pf1 і pf мають різні типи;
}.
```

Розглянемо оголошення

extern "C" int f1(int (*pf)(int,int)),

де ***f1*** — є *C*-функцією з одним параметром ***pf*** — покажчиком на *C*-функцію.

У наступному прикладі маємо C++ функцію, що як параметр має *C*-функцію

```
extern "C" typedef void FC(int);
void f2(FC* f3).
```

Лекція 6. ДИРЕКТИВИ ПРЕПРОЦЕСОРА

Директивами препроцесора називаються рядки, які починаються із символу #, за якими йде ідентифікатор, що називається ім'ям директиви.

Розглянемо основні директиви.

1. Директива **#include**.

Використовується для підключення файлів. Включення у програму файла, що підключається, дає той самий результат, як при копіюванні цього файла у кожний файл програми. Зазвичай файли, що підключаються, закінчуються на *.h* .

Директива **#include** має такі три модифікації:

а) **#include<FILE>** – ця модифікація використовується для підключення системних файлів. При її виконанні здійснюється пошук файла з ім'ям **FILE** у списку зазначених заздалегідь каталогів, а потім у стандартному списку каталогів.

б) **#include "FILE"** – ця модифікація застосовується для підключення файлів користувача. Спочатку пошук файла з ім'ям **FILE** здійснюється у поточному каталозі, а потім у списку каталогів системних файлів.

в) **#include ANYTHING ELSE** – ця модифікація називається директивою **#include**, що обчислюється. Рядок **ANYTHING ELSE** перевіряється на наявність відповідного макросу, значення якого потім заміняє його назву. Одержаний у результаті рядок має відповідати одній з розглянутих модифікацій.

2. Директива **#ifndef**.

Одноразово підключені файли. Ця директива використовується для усунення ситуації з повторним підключенням того самого файла.

Розглянемо приклад:

```
#ifndef FILE_FOO_SEEN
#define FILE_FOO_SEEN
...                                  //визначення користувача;
#endif.
```

Макрос *FILE_FOO_SEEN* вказує на те, що файл вже один раз був підключений. Якщо у підключеному файлі міститься директива **#include**, що вказує на файл або макрос, який у директиві **#ifndef** вже визначений, то цей файл повністю ігнорується.

3. Директива **#pragma once**.

Ця директива вказує препроцесору, що даний файл повинен бути підключений не більше одного разу.

4. Директива **#import**.

Ця директива використовується для підключення файла не більше одного разу.

5. Директива **#define**.

Ця директива використовується для визначення макросів. Макрос — це тип скорочення, який можна заздалегідь визначити й використовувати надалі.

Наприклад:

```
#define BUF 100
...
int main(){
int c=BUF;
...}.
```

Існують також макроси з аргументами.

Наприклад:

```
#define MIN(X,Y) ((X)<(Y)?(X):(Y))
...
int main(){
int c=MIN(1,2);
...}.
```

При визначенні аргументів дужки повинні закриватися, а кома не повинна завершувати аргумент. Однак не існує обмежень на використання квадратних або кутових дужок.

Наприклад:

macro(array[x=y,x+1]).

У цьому випадку макросу передається 2 аргументи: *'array[x=y'* і *'x+1]'*.

Аргументи можуть містити посилання до інших макросів як з аргументами, так і без них.

Наприклад, макрос *MIN(MIN(a,b),c)* заміняється наступним текстом *((((a)<(b)?(a):(b)))<(c)?(((a)<(b)?(a):(b))):(c))*.

Існують наперед визначені макроси.

6. **_FILE_** — цей макрос замінюється на ім'я поточного вихідного файла у формі строкової константи С.

7. **_LINE_**— цей макрос замінюється на номер поточного рядка у формі десяткової цілої константи.

8. **_DATE_** — цей макрос замінюється на строкову константу, яка вказує дату запуску препроцесора, вона має наступний формат виводу: *Jan 22 1995*.

9. **_TIME_** — цей макрос замінюється на строкову константу, що вказує час запуску препроцесора.

10. **_STDC_** — цей макрос замінюється на константу зі значенням **1** для вказівки, що компілятор підтримує стандарт ANSI**.**

11. **_STDC_VERSION_** — цей макрос замінюється на номер версії стандарту С.

12. **_GNUC_** — цей макрос визначений тоді, коли використовується GNU C.

13. **_GNUC_MINOR_** — цей макрос містить додатковий номер версії компілятора.

14. **_GNUG_** — цей макрос визначений, якщо компіляція відбувається у мові С++.

15. **_BASE_FILE_** — цей макрос замінюється на ім'я базового вихідного файла у формі константи С.

16. **_INCLUDE_LEVEL_** — цей макрос замінюється на десяткову цілу константу, що вказує рівень вкладеності файлів, що підключаються.

17. **_VERSION_** — цей макрос указує номер версії GNU C.

18. **_OPTIMIZE_** — цей макрос визначається в оптимізуючих компіляторах. Якщо він визначений, то це приводить до створення у підключених файлах GNU альтернативних макровизначень для деяких функцій із системних бібліотек.

19. **_CHAR_UNSIGNED_** — цей макрос визначається тоді й тільки тоді, коли тип *char* є беззнаковим.

20. **_REGISTER_PREFIX_** — цей макрос замінюється на рядок, що описує префікс, що додається до позначення регістрів процесора в асемблерному коді.

21. **Стрінгіфікація.** Стрінгіфікація означає перетворення фрагмента коду в строкову константу, яка містить текст цього фрагмента коду.

Наприклад:

```
#define WARN_IF(EXP)
do{
if(EXP) cout<<“warning”<<#EXP;}
while(0);
int main(){
WARN_IF(10);//на екрані буде warning 10
return 0;}.
```

22. **Об’єднання.** Об’єднання — це з’єднання строкових констант в одну. При визначенні макросу перевіряється наявність операторів ‘##’ у його тілі. При виклику макросу й після підстановки аргументів усі оператори ‘##’, а також усі пробіли поруч з ними віддаляються.

Наприклад:

```
#define COMMAND(NAME){#name,name##_command}
struct command commands[]={COMMAND(quit),COMMAN
D(help),..};
```

Це те саме, що

```
struct command commands[]={
{“quit”,quit_command},{“help”,help_command},
...
```

23. **Видалення макросів.** Видалити макрос — це скасувати його визначення. Це здійснюється за допомогою директиви ***#undef***.

Наприклад:

```
#define FOO 2
```

```
...
int x=FOO;                    //x=2
#undef FOO
...
int x=FOO;                    //error.
```

24. **Рекурсивні макроси.** Рекурсивні макроси — це макроси, у визначенні яких використовується ім'я самого макросу.

Стандарт ANSI C не розглядає рекурсивний виклик макросу як виклик.

Наприклад:

```
#define foo (4+foo).
```

У цьому випадку, дотримуючись звичайних правил, кожне посилання на *foo* замінюється на значення *(4+foo)* доти, поки це не призведе до помилки препроцесора. Однак при використанні макросів виклик завершиться після отримання результату *(4+foo)*.

25. **Умови.** У C препроцесорі існують умовні директиви ***#if***, ***#ifdef***,***#ifndef.***

Наприклад:

```
#if EXPERSSION
   text_if_true
#else
   text_if_else
#endif.
```

Якщо необхідно перевірити більше ніж одну умову, то можна використовувати директиву ***#elif***.

Наприклад:

```
#if X==1
...
#elif X==2.
```

Текст після ***elif*** включається, коли умова ***#if*** хибна, а ***#elif*** істинна.

```
...
#else
```

...

#endif.

Директива ***#if*** може застосовуватися разом із ***#define***.

Наприклад:

#if defined(MACROS).

У цьому випадку значення буде істинним, якщо *MACROS* визначений як макрос.

#ifdef NAME еквівалентно *#if defined(NAME)*.

26. **Твердження.**

Твердження виглядає у такий спосіб:

#PREDICATE(ANSWER).

У цьому записі твердженням є **PREDICATE**, а відповідь на нього **ANSWER**. Значенням **ANSWER** може бути будь-яка послідовність слів.

Перевірка, чи є **ANSWER** твердження **PREDICATE**, виглядає таким чином:

#if #PREDICATE(ANSWER).

Лекція 7. ОБЛАСТЬ ВИДИМОСТІ Й ЧАС ЖИТТЯ

У С++ визначено 4 області видимості:

1. Локальна область видимості.
2. Глобальна область видимості.
3. Область видимості простору імен.
4. Область видимості класу.

Локальна область видимості

Локальна область видимості — це частина вихідного тексту, що міститься у визначенні функції.

Наприклад:

```
int func(int a, int b){
// I рівень
while(a<b){
// II рівень
int mid;
...}
...}.
```

Функція *func* містить два рівні. Змінну *mid* видно тільки всередині циклу. Розв'язання дозволу імен здійснюється у такий спосіб.

1. Переглядається та область видимості, де воно зустрілося, якщо ім'я знайдене, то його дозволено використовувати.

2. Переглядається область, яка містить область, згадану в п. 1.

Така послідовність може призвести до того, що наступне оголошення затінює попереднє.

Наприклад:

```
int low;
int func(){
...
int low;                    //локальне оголошення low
                            затінює глобальне;
...}.
```

Глобальна область видимості

Час життя глобального об'єкта починається з моменту запуску програми й закінчується з її завершенням. Глобальний об'єкт може бути визначений тільки один раз.

Глобальний об'єкт без явної ініціалізації гарантовано отримає нульове значення.

Інструкція **extern.**

Інструкція **extern** дозволяє оголосити об'єкт, не визначаючи його.

Наприклад:

extern int i.

Ця інструкція говорить про те, що у програмі є визначення *int i.*

Оголошення **extern** з ініціалізацією вважається визначенням.

Наприклад:

extern const int i=3.

Ключове слово **extern** може бути вказано при оголошенні функції, це значить, що функція визначена в іншому місці:

extern int f(int,int).

Проблема, що виникає з можливості оголошувати об'єкт у різних файлах — ймовірність невідповідності оголошень.

Наприклад:

```
----------------------------a.cpp -----------------------------
int add(unsigned char a){...};
----------------------------b.cpp -----------------------------
extern int add(char);
...
char s='e';
int d=add(s);                          //error.
```

Локальні об'єкти

Оголошення змінної в локальній області видимості вводить локальний об'єкт. Існує 3 види локальних об'єктів:

1. Автоматичний об'єкт.
2. Реєстровий об'єкт.
3. Статичний об'єкт.

Автоматичний об'єкт

Об'єкт, що розміщується в пам'яті під час виклику функції, в якій він визначений, називається автоматичним.

Не ініціалізований автоматичний об'єкт містить випадкове значення. Час життя об'єкта закінчується із завершенням роботи функції.

Реєстровий об'єкт

Автоматичний об'єкт, що використовується багато разів, можна оголосити реєстровим.

Наприклад:

```
for(register int i=0;i<10;i++){...}.
```

Реєстровий об'єкт завантажується в машинні регістри. Якщо це неможливо, об'єкт залишиться в основній пам'яті.

Статичний об'єкт

Всередині функції можна оголосити об'єкт, значення якого буде зберігатися між викликами функції. Оголошується такий об'єкт за допомогою ключового слова *static: static int i=0.*

Наприклад:

```
int fact(int n){
static int depth=1;
cout<<"глибина: "<<depth++<<endl;
if (n==0) return 1;
else return n*fact( n-1);
}.
```

Шаблон auto_ptr

Шаблон ***auto_ptr*** – зручний механізм маніпуляції динамічними об'єктами, що створюються оператором ***new***.

Об'єкт ***auto_ptr*** ініціалізується адресою динамічного об'єкта, створеного за допомогою ***new***. Такий об'єкт знищується, коли закінчується час його життя.

Наприклад:

```
#include <memory>
auto_ptr<int> pi(new int (1024)).
```

Шаблон ***auto_ptr*** підтримує поняття ***володіння***:

auto_ptr<int> pi2(pi).

Спочатку відповідальним за комірку пам'яті була змінна *pi,* потім стала *pi2*.

Функція ***get()*** повертає внутрішній покажчик, що використовується в об'єкті ***auto_ptr***.

Функція ***reset()*** присвоює значення внутрішньому покажчику.

Наприклад:

```
auto_ptr<int> p1;                    //внутрішній покажчик 0
if(p1.get()!=0)
*p1=1024;
else
p1.reset(new int(1024)).
```

Функція ***release()*** гарантує, що кілька визначених покажчиків не є власниками того самого об'єкта. Ця функція повертає адресу об'єкта, як це робить ***get()***, але також передає володіння їм: *auto_ptr<int> p2(p1.release()).*

Простір імен

Проблему засмічення області видимості глобального простору імен дозволяє вирішити введення користувачем власного простору імен.

Простір імен визначається таким чином:

```
namespace myname{
int a;
int b;
int func(int a){...};}.
```

Використовувати змінні ***a***, ***b*** можна у такий спосіб:

```
int main(){
myname:: a=10;
myname:: b=20;
int c=myname::func(myname::a);
...},
```

де **::** — оператор розв'язання області видимості. Цей оператор може бути використаний для звернення до елемента глобального простору імен.

Наприклад:

```
int max=100;
void func(int max){
if(max<2) ...;
if(::max>10) ...;
...
```

Вкладені простори імен

Вкладені простори імен визначаються таким чином:

```
namespace myname1{
int a;
    namespace myname2{
    int b;
    }
}.
```

Визначені простори можна використовувати у такий спосіб:

```
int main(){
myname1::a=10;
myname1:myname2::b=20;
...
}.
```

У процесі розв'язання області видимості імен вкладені простори поводяться як вкладені блоки.

Правило одного визначення й простір імен

Визначення простору імен може складатися з розрізнених частин і розміщатися у різних файлах.

Наприклад:

```
----------a.h----------
namespace myname{
extern void inverse(int);
}
----------a.cpp----------
namespace myname{
void inverse(int a){...}
}.
```

Безіменні простори імен

Буває необхідно визначити об'єкт, функцію тощо, щоб вони були видимими тільки на невеликій ділянці програми. Це можна зробити, використовуючи безіменні простори імен:

```
----------a.cpp----------
namespace {
void swap(int& a, int& b){...}
}
int func(int a,int b){
...
swap(a,b);
}.
```

Функцію *swap* видно тільки у файлі *a.cpp*. Ім'я *swap* можна вживати у некваліфікованій формі.

У мові С аналогом безіменного простору імен є ключове слово ***static***:

```
----------a.cpp----------
static void swap(int& a, int& b){...;}.
```

Псевдоніми простору імен

Псевдонім простору імен використовується для визначення короткого синоніма імені простору.

Наприклад:

```
namespace myname1{
int a;
namespace myname2{
int b;
}
};
namespace myname3=myname1::myname2;
int main(){
myname3::a=30;
...
}.
```

Using-оголошення

Using-оголошення вводить імена в область видимості.

Наприклад:

```
namespace myname{
void func(int a){...};
}
int main(){
using myname::func;
int c=10;
func(c);
...
};
namespace myname{
int bi=16,bj=15,bk=20;
}
int bj=0;
void main(){
using myname::bi;
++bi;                          //myname::bi=17;
using myname::bj;
++bj;                          //myname::bj=16;
int bk;
using myname::bk;              //error повторне визначення
                               bk;
...
```

Using-директива

Якщо бібліотека досить велика, то краще замість ***using*-оголошення** використовувати ***using*-директиви.**

Наприклад:

```
namespace myname{
int a;}
using namespace myname;
int main(){
a=30;
```

```
...}
namespace myname{
int bi=16,bj=15,bk=20;
}
int bj=0;
int main(){
using namespace myname;
++bi;                          //myname::bi=17;
++bj;                          //error неоднозначність;
++::bj;                        //bj=1;
++myname::bj;                  //myname::bj=16;
int bk=90;
bk++;                          //bk=98;
...
```

Можливі також такі помилки:

```
namespace myname1{
void swap(int& a,int& b){...};
}
namespace myname2{
void swap(int& a,int& b){...} ;
}
using namespace myname1;
using namespace myname2;
int main(){
int k=10;
int l=30;
swap(k,l);                     //error, невидомо, яку з
                               функцій swap треба викли-
                               кати у цьому місті;
...
```

Стандартний простір імен std

Усі компоненти стандартної бібліотеки C++ перебувають у стандартному просторі імен *std*. Усе, що оголошено в стандартному заголовному файлі, перебуває у просторі імен *std*.

Лекція 8. ПЕРЕВАНТАЖЕНІ ФУНКЦІЇ

Перевантаження дозволяє мати кілька однойменних функцій, що виконують схожі операції над аргументами.

Наприклад:

int max(int,int);
double max(double,double).

Якщо в деякій області видимості ім'я функції оголошене кілька разів, то друге оголошення інтерпретується компілятором наступним чином.

1. Якщо списки параметрів двох функцій розрізняються числом або типами, то функції вважаються перевантаженими:

void print(int a){…};
void print(double a){…}.

2. Якщо списки параметрів і тип значення, що повертається в оголошеннях двох функцій є однаковими, то оголошення вважається повторним. Імена параметрів до уваги не приймаються:

void print(int a){…};
void print(int b){…}.

3. Якщо списки параметрів двох функцій однакові, але типи значень, що повертаються, різні, то друге оголошення вважається помилковим:

unsigned int max(int a, int b){…};
int max(int a, int b){…}; *//error.*

4. Якщо списки параметрів двох функцій відрізняються тільки значеннями за замовчуванням аргументів, то оголошення вважається повторним:

int max(int a,int b){…};
int max(int a, int b=0){…}.

Ключове слово **typedef** створює тільки альтернативне ім'я для існуючого типу даних, новий тип не створюється.

Кваліфікатори **const** і **volatile** при порівнянні до уваги не беруться:

void func(int);
void func(const int); *//повторне оголошення.*

Однак, якщо кваліфікатор **const** або **volatile** застосовується до параметра покажчика або відсилки, то при порівнянні він враховується:

void f(int*);
void f(const int*),
а також
void f(int&)
void f(const int&); *//це різні функції.*

Директива extern "C" і перевантажені функції

У директиві **extern "C"** дозволяється задавати тільки одну з двох перевантажених функцій.

Наприклад:

```
extern "C" void print(const char*);
extern "C" void print(int);          //error.
```

Однак наступний приклад працює коректно:

```
extern "C" int max(int,int);
extern double max(double, double).
```

Покажчики на перевантажені функції

Можна повідомляти покажчик на одну з множини перевантажених функцій.

Наприклад:

```
extern void ff(int);
extern void ff(double);
void (*pf)(int)=&ff.
```

У цьому випадку буде вибрана функція *void ff(int)*. Тобто компілятор шукає у множині перевантажених функцій ту, яка має тип і список параметрів такий, як і функція, на яку вказує покажчик.

Якщо такої функції немає, то видається помилка.

Дозвіл перевантаження

1. Ідентифікується множина перевантажених функцій, які будуть розглядатися при певному виклику. Ті функції, що ввійшли у цю множину, називаються кандидатами.

2. Вибираються функції, до типів формальних аргументів яких можливо привести фактичні параметри, тобто компілятор ідентифікує й ранжує перетворення. Ранжування дає один із наступних результатів:

а) точна відповідність.

Тип формального аргументу точно відповідає типу формального параметра.

Наприклад:

```
void print(unsigned int);
void print(const char*);
void print(char);
```

наступний виклик дає точну відповідність

```
print('a');                    //викликає print(char);
```

б) відповідність із перетворенням типу.

Тип фактичного аргументу не відповідає типу формального параметра, але може бути перетворений до нього;

в) відсутність перетворення.

Знаходиться функція, що найкраще відповідає виклику.

Найбільш відповідна функція

Найбільш відповідною функцією буде функція, для якої виконуються умови:

1. Перетворення застосовані до аргументів не гірше перетворень, необхідних для виклику будь-якої відповідної функції.

2. Хоча б для одного аргументу застосоване перетворення краще, ніж для того самого аргументу в будь-якій іншій відповідній функції.

Рангом ланцюжка перетворень є ранг найгіршого перетворення, що міститься у ньому.

Перетворення ранжуються наступним чином:

1. Точна відповідність краще підвищення типу.

2. Підвищення типу краще стандартного перетворення.

3. Стандартне перетворення краще перетворення, визначеного користувачем.

Точна відповідність

До точної відповідності відноситься:

а) точна відповідність аргументів. Наприклад, перерахований тип точно відповідає визначеним у ньому елементам:

```
enum stat {Fail,Pass};
extern void ff(stat);
extern void ff(int);
int main(){
ff(Pass);                    //точна відповідність;
...;
```

б) перетворення **lvalue->rvalue**.

Під **lvalue** розуміється об'єкт, що відповідає наступним умовам:

- можна одержати адресу об'єкта;
- можна одержати значення об'єкта.

Це значення легко модифікувати (за винятком модифікатора **const**).

Наприклад:

```
int f(int);
int main(){
int lval=5;
int d=f(lval);               //точна відповідність;
...};
```

в) перетворення масиву в покажчик.

Наприклад:

```
int a[3];
void f(int*);
int main(){
f(a);                        //точна відповідність;
...};
```

г) перетворення функції в покажчик;

д) перетворення кваліфікаторів.

Це перетворення полягає у додаванні кваліфікаторів **const** і **volatile**.

Наприклад:

```
int a[3]={23,24,25};
```

```
int* pi=a;
void f(const int*);
int main(){
f(pi);                          //точна відповідність;
...}.
```

Точну відповідність можна встановити примусово, скориставшись явним приведенням типів.

Підвищення типу

Підвищення типу здійснюється так:

- фактичний аргумент типу **char, unsigned char, short** підвищується до типу **int**;
- фактичний аргумент **unsigned short** підвищується до типу **int** (якщо розмір **int** більше, ніж розмір **short**, і до типу **unsigned int** в іншому випадку);
- **float-> double;**
- **enum->int; unsigned int; long; unsigned long;**
- **bool->int.**

Наприклад:

```
extern void print(unsigned int);
extern void print(int);
extern void print(char);
int main(){
unsigned char u;
print(u);                       //print(int); для и потрібне
                                підвищення типу;
...}.
```

Стандартне перетворення

До стандартного перетворення відносяться:

1. Цілочисельні перетворення: цілочисельний тип <-> цілочисельний тип (крім перетворень, які були віднесені до підвищення типу).

2. Тип із плаваючою крапкою <-> тип із плаваючою крапкою (крім перетворень, які були віднесені до підвищення типу).

3. Цілочисельний тип <-> тип із плаваючою крапкою.

4. Перетворення покажчиків:

- 0-0->покажчик;
- покажчик->void*.

5. Перетворення будь-якого цілочисельного типу, типу з плаваючою крапкою, перерахованого типу або покажчика в bool.

Наприклад:

```
extern void f(unsigned int);
extern void f(float);
int main(){
f('a');
f(0);
f(2L);                          //всі виклики неоднозначні
f(3.14);
f(true);
...
}.
```

Посилання

Якщо формальний параметр посилання, то можливі наступні випадки:

1. Фактичний параметр підходить у якості ініціалізатора параметра-посилання. Тоді у цьому випадку маємо точну відповідність.

Наприклад:

```
void swap(int&, int&);
int main(){
int l1,l2;
swap(l1,l2);
...
}.
```

2. Фактичний аргумент не може ініціалізувати параметр-відсилку. У цьому випадку точної відповідності немає й видається помилка.

Приклад 1:

```
void f(double&);
int main(){
int a;
f(a);                          //error;
...
}.
```

Приклад 2:

```
struct my_struct{...};
void take(my_struct&);
my_struct myfunc();
int mian(){
take(my_func());               //error;
...
}.
```

Тимчасова змінна не може ініціалізувати посилання, тому повертається значення, яке повинно бути типу *const my_struct*.

Аргументи зі значеннями за замовчуванням

Наявність аргументів за замовчуванням дає можливість розширити множину підходящих функцій.

Наприклад:

```
extern void ff(int);
extern ff(long, int=0);
int main(){
ff(2L);                        //ff(long,0);
ff(0,0)                        //ff(long,int);
ff(0);                         //ff(int);
ff(3.14);                      //error, неоднозначність;
...
}.
```

Лекція 9. ШАБЛОНИ ФУНКЦІЙ

Оголошення й визначення шаблона функції завжди починається із ключового слова **template**.

Приклад 1:

```
template<class Type>          //можна замість слова class використовувати typename
Type max(Type a, Type b){
if(a>b) return a;
else return b;}.
```

Приклад 2:

```
template<class Type, int size>
Type min(Type (&array)[size]);
...
template<class Type,int size>
Type min(const Type (&array)[size])
{
Type min_val=array[0];
for(int i=0;i<size;i++)
if(array[i]<min_val)
min_val=array[i];
return min_val;
}.
```

Приклад 3:

```
typedef Type;
template<class Type>
Type min(Type a,Type b){...};     //ім'я Type затінює оголошене в typedef.
```

Приклад 4:

```
template<class Type>
Type min(Type a, Type b){
...
typedef Type;                     //error, таке ім'я вже існує
...}.
```

Приклад 5:

```
template<class Type,class Type>
```

```
Type min(Type,Type);          //error повторне оголошення імені Type.
```

Приклад 6:

```
template<class T,class U, class F>
T min(U a,F b){...};
```

Приклад 7:

```
template<typename T,U>
inline T func(U s){...};          //error
template<typename T,typename U>
inline T func(U s){...};          //правильно.
```

Конкретизація шаблону функції

Шаблон функції описує, як побудувати конкретні функції, якщо задано множину фактичних типів або значень.

Процес конструювання називається конкретизацією шаблону.

Приклад 1:

```
template<class Type,int size>
Type min(const Type (&array)[size])
{
Type min_val=array[0];
for(int i=0;i<size;i++)
if(array[i]<min_val)
min_val=array[i];
return min_val;
}
int main(){
int a[]={2,3,4,5,6,10};
double b[]={2.3,23.67,12.5};
int min_v=min(a);          //замість абстрактного типу Type підставляється int; size=6;
int min_v2=min(b);         //замість абстрактного типу Type підставляється double; size=3;
...
}.
```

Процес визначення типів і значень аргументів шаблонів за відомими аргументами функцій називається виведенням аргументів шаблону.

Приклад 2:

```
template<typename Type, int size>
Type min(Type (&array)[size]){...};
...
int (*pf)(int (&) [10])=&min; //у цьому випадку відбувається виведення аргументів
                              шаблону;
Type=int, size=10.
```

Приклад 3:

```
template<class T>
int max(T a,T b){...};
int main(){
int a=10;
double b=20.1;
max(a,b);//error
...
}
template<class T>
T func(Type* array, int size){....};
int main(){
int a[]={1,2,3};
func(a,3);                    //у цьому випадку масив приводиться до типу
                              покажчика, а потім виводиться тип T;
...
}.
```

Явне завдання аргументів шаблону

У деяких ситуаціях вивести аргументи шаблону неможливо. У цих випадках необхідно явно задавати типи таких аргументів.

Приклад 1:

```
template<class T>
```

```
T min(T a, T b){...};
int main(){
double a=10,b=30;
int s=min<int>(a,b);                    //T=int
...
}.
```

Приклад 2:

```
template<class T, class U, class F>
T func(U a, F b){...};
int main(){
int a=10;
double g=20;
int d=func<int,double,float>(a,g); //T=int;U=double;F=float
...
```

Приклад 3:

```
template<class T, class U, class F>
T func(U a, F b){...};
int main(){
int a=10;
double g=20;
int d=func<int,float>( a, g);     //T=int;  U=int;  F=float;
                                  опускати можна тільки
                                  хвостові аргументи;
...
}.
```

Приклад 4:

```
template<class T1, class T2, class T3>
T1 sum(T2 op1, T3 op2){...};
void func(int (*pf)(int,char));
void func(double (*pf)(float,float));
int main(){
func(&sum);                              //error, яка з func;
func(&sum<double,float,float>);
...
}.
```

Моделі компіляції шаблонів

Існує 2 моделі компіляції шаблонів.

1. Модель компіляції із включенням.

У цій моделі визначення шаблону включається у кожний файл, де цей шаблон конкретизується. Звичайне визначення міститься у заголовному файлі.

2. Модель компіляції з поділом.

Відповідно до цієї моделі оголошення шаблонів функцій міститься у заголовному файлі, а визначення — у файл із вихідним текстом програми.

Приклад (компіляція із включенням):

```
----------a.h----------
template<class T>
T min(T a, T b){...};
----------a.cpp ----------
#include “a.h”
int i,j;
double d=min(i,j).
```

Приклад (компіляція з поділом):

```
----------a.h----------
template<class T>
T min(T a, T b){...};
----------a.cpp----------
export template<class T>
T min(T a,T b){...};
----------d.cpp ----------
#include “a.h”
int i.j
double d=min(i,j).
```

Явні оголошення конкретизації

Явні оголошення конкретизації шаблону необхідні для усунення проблем у моделі компіляції із включенням (характерно для деяких компіляторів).

Наприклад:

```
template<typename T>
T sum(T a, T b){...};
template int sum<int>(int, int); // явне оголошення конкретизації.
```

Визначення шаблону повинне перебувати у тому самому файлі, де і явне оголошення, інакше буде помилка.

Явна спеціалізація шаблону

Не завжди вдається написати шаблон функції, що підходив би до всіх можливих типів. Тому іноді використовується явна спеціалізація шаблону.

Приклад 1:

```
template<class T>
T max(T a, T b){...};
template<> int max<int>(int a, int b){...};
                                    //явна спеціалізація шаблону;
int main(){
int s=1,d=2;
max(s,d);                           //викликається явна спеціалізація шаблону;
double f,g;
max(f,g);                           // у шаблоні T=double;
...
}.
```

Приклад 2:

```
template<class T1,class T2, class T3>
T1 sum(T2 a, T2 b){...};
template<>int sum<int, char>sum(char f, char g)
{...};
                                    //явна спеціалізація шаблону.
```

Можна оголошувати явну спеціалізацію шаблону функції, не визначаючи її:

```
template<>int sum<int, int>sum(int f,int g).
```

Перевантаження шаблонів функцій

Шаблон функції може бути перевантажений.

Розглянемо приклад:

```
template<class T>
T min(int,int);
template<class T>
T min(T, int);
template<class T>
T min(T,T);
int main(){
double s=2.3;
int a=34;
int l=min(s,200);          //викликається min(T,int);
int d=min(a,300);          //неоднозначність;
...}.
```

Алгоритм дозволу перевантаження при наявності шаблонів такий:

1. Побудова множини функцій-кандидатів.

Розглядаються шаблони функцій з таким ім'ям, що й викликана. Якщо аргументи шаблонів виведені з фактичних аргументів функції успішно, то в множину функцій кандидатів включається або конкретизований шаблон, або спеціалізація шаблону для виведених аргументів, якщо вона існує.

2. Побудова множини підходящих функцій.

У множині функцій-кандидатів залишаються тільки функції, які можна викликати з даними фактичними аргументами.

3. Ранжирування перетворень типів:

а) якщо є тільки одна функція, то вона й викликається;

б) якщо виклик неоднозначний, видалити із множини підходящих функцій, конкретизованих із шаблонів.

4. Дозвіл перевантаження: розглядаються тільки звичайні функції:

а) якщо є тільки одна функція, то вона й викликається;

б) в іншому випадку виклик неоднозначний.

Наприклад:

```
template<class Type>
```

```
Type max(Type,Type);
double max(double,double);
int max(){
int a;
float b;
double s;
max(0,a);                    //обидва аргументи int виклик max(Type,Type);
max(0.25,s);                 //виклик double max(double, double);
max(0,b);                    // виклик double max(double, double);
...}.
```

Дозвіл імен у визначеннях шаблонів

Розглянемо приклад:

```
void print(const char*);     //це оголошення тут не потрібно;

template<typename T>
T min(T a,T b){
T min_val;
print(min_val);              //невідомо який print потрібний;
...;
void print (int);
int main(){
int a=3,b=5;
min(a,b);                    //буде викликаний print(int);
...}.
```

Простір імен і шаблони функцій

Якщо шаблон функції визначений у деякому просторі імен, то його явні специфікації повинні бути визначені у тому самому просторі імен.

Наприклад:

```
namespace myname{
template<class T>
T max(T a,T b){...};
}
namespace myname{
template<> int max<int>(int d, int g){...};
}
int main(){
using myname::max;
double a=2.3,b=4.5;
max(a,b);                    //викликається max(T,T).
...
```

ОБ'ЄКТНЕ ПРОГРАМУВАННЯ

Лекція 10. КЛАСИ

Визначення класу складається із 2 частин: заголовка й тіла.
Наприклад:

```
class Myclass{
private:
   int a;
   int b;
};
```

Різниця між модифікаторами private і public

Розглянемо приклади.
Приклад 1:

```
class A{
public:
int s;
};
int main(){
A aobj;
aobj.s=23;                    //правильно;
A* pobj=new A;
pobj->s=56;                   //правильно;
...
```

Приклад 2:

```
class B{
private:
int s;
};
int main(){
B bobj;
bobj.s=22;                    //error;
B* bobj=new B;
bobj->s=34;                   //error;
...
```

Визначення класу

class G{
int s;
}
еквівалентно наступному оголошенню
class G{
private:
int s;
}.

Структури (*struct*) еквівалентні класам, за винятком одного нюансу.

Визначення структури

struct G{
int s;}
еквівалентно наступному визначенню
struct G{
public:
int s;
}.

Атрибути класу не можуть бути ініціалізовані в місці визначення, за винятком статичних атрибутів:

```
class H{
int s=3;//error
double f=23.6;          //error
static int j=25;        //правильно
}.
```

Функції-члени класу

Крім даних-членів у класі можуть бути функції-члени.
Наприклад:

```
class Myclass{
private:
int a;
```

```
public:
void func(int g){
a=g;
}
}.
```

Виклик функції-члена здійснюється у такий спосіб:

```
int main(){
Myclass obj;
obj.func(34);
...
Myclass pobj=new Myclass;
pobj->func(34);
...
```

Функція-член класу може бути визначена поза класом.

Наприклад:

```
class Myclass{
private:
int a;
public:
void func(int);
};
void Myclass::func(int g){a=g}.
```

Друзі класу

Деяку функцію можна оголосити другом класу, у цьому випадку вона одержить доступ до закритих членів класу.

Наприклад:

```
class Myclass{
private:
int s;
}
void func (const Myclass& m){
m.s=100;                          //error член s закритий
                                  член-класу;
}
```

```
class Myclass;
void func(const Myclass&);
class Myclass{
private:
friend void func(const Myclass&);
int s;}
void func(const Myclass& m){
m.s=300;                    //правильно (однак у MVS
                            6.0 не працює);
...
```

Функції-члени з кваліфікаторами const і volatile

Функція-член класу може бути оголошена з кваліфікатором **const**. Це означає, що вона не може змінювати дані члени класу.

Розглянемо приклад.

Приклад 1:

```
class Myclass{
private:
int s;
public:
void func(int d) const {
s=d;                        //error, константна функ-
                            ція-член класу не може
                             змінювати атрибут s;
...
}.
```

Приклад 2:

```
class Myclass{
private:
int s;
public:
int get() const{ return s;}     //правильно;
...}.
```

Приклад 3:

```
class Myclass{
```

```
private:
int s;
public:
int get(){return s;}
void set(int a){s=a;}
}
int main(){
const Myclass obj;
int t=obj.get();                    //правильно;
obj.set(100);                       //error, функція-член set
                                    неконстантна;
...}.
```

Оголошення mutable

Щоб дозволити модифікацію члена класу, що належить константному об'єкту, його необхідно оголосити змінюваним.

Розглянемо приклад:

```
class Myclass{
private:
int s;
mutable int h;
public:
int get() const {
s++;                                //error;
h++;                                //правильно;
return s;}
void set(int d){s=d;}
void set(int w) const {h=w;}
}

int main(){
Myclass t;
t.get();
const Myclass t1;
```

```
t1.set(30);
return 0;
}.
```

Неявний покажчик this

Покажчик *this* адресує об'єкт, для якого викликана функція-член.

Розглянемо приклади:

Приклад 1:

```
class Myclass{
private:
int s;
public:
void set(int s){
this->s=s;                    //this — це покажчик;
}
}.
```

Приклад 2:

```
class Myclass{
   private:
      int s;
   public:
      void set(int d){s=d;}
      bool operator>(const Myclass& m){
            if(this->s>m.s) return true;
            return false;}
      Myclass& operator=(int d){
            s=d;
            return *this;}        //повертає об'єкт цього
                                  самого класу;
};
int main(){
Myclass a,b;
a.set(10);
```

```
b.set(29);
b=545;
if (a>b) cout<<“a>b”<<endl;
else cout<<“a<b”<<endl;
return 0;
}.
```

Статичні члени класу

Іноді потрібно, щоб усі об’єкти деякого класу мали доступ до єдиного глобального об’єкта.

У цій ситуації прийнятним рішенням є статичний член класу, що поводиться як глобальний об’єкт, що належить своєму класу. Статичний член класу належить класу, але не об’єкту. Щоб зробити член статичним, необхідно помістити на початку його оголошення в тілі класу ключове слово **static**.

Наприклад:

```
class Account{
private:
static double Rate;
double amount;
...}.
```

У загальному випадку статичний член ініціалізується поза класом (але може і всередині класу).

Наприклад:

```
class Account{
private:
static double Rate;
...}.
double Account::Rate=2...
```

Ініціалізацію статичних членів не варто розміщувати у заголовних файлах. Приклад правильної ініціалізації:

```
----------a.h----------
class Account{
private:
```

```
static int a;
...}
-----------------------------a.cpp ----------------------------
#include "a.h"
int Account::a=2.4.
```

Статичний член класу доступний функції-члену того самого класу:

```
class A{
private:
static int a;
double b;
public:
int f(){ return a*b;}
...}.
```

Функції, що не є членами класу, можуть звертатися до статичних членів двома способами:

1-й спосіб — за допомогою операторів доступу:

```
class Account{
private:
static double a;
friend void f(Account& ac){
double c =ac.a*20;
...};
```

2-й спосіб — кваліфікація імені члена ім'ям класу:

```
class Account{
private:
static double a;
friend void f(Account& ac){
double c =Account::a*20;
...}.
```

Статичні члени можуть використовуватися способами що неприпустимі для нестатичних членів.

1. Статичний член можна оголошувати як об'єкт свого ж класу:

```
class Bar{
private:
```

```
static Bar m1;                  //правильно;
Bar* m2;                        //правильно;
Bar m3;                         //error;
...}.
```

2. Статичний член може виступати у ролі аргументу за замовчуванням, що заборонено для нестатичного:

```
class Bar{
private:
int a;
static int b;
public:
int mem(int c=a){...};          //error;
int mem1(int c=b){...}          //правильно;
...}.
```

Статичні функції-члени

Крім статичних даних-членів існують також статичні функції-члени. Статична функція-член також належить класу, а не об'єкту.

Наприклад:

```
class A{
private:
int a;
static int b;
public:
static void f(int);
...}.
```

Статична функція-член класу не може використовувати нестатичні атрибути класу.

Наприклад:

```
class A{
private:
int a;
static int b;
public:
static void f(int c){
```

```
a=c;                    //error, a — нестатичний
                        атрибут;
b=c;                    //правильно, b — статич-
                        ний атрибут;
...}
void f1(int c){
a=c;                    //правильно;
b=c;                    //правильно;
...
}.
```

Покажчик на член класу

При роботі зі звичайними функціями можна використовувати покажчики на функції:

```
int (*pf1) ();
int f();
pf1=f.
```

Однак таке присвоєння є помилковим у випадку функції-члена класу.

Наприклад:

```
class Bar{
...
public:
int f();
...
int (*pf)();
pf=&Bar::f;             //error, порушення типіза-
                        ції.
```

Покажчик на функцію-члена класу можна оголосити таким чином:

```
int(Bar::*pf)();
pf=&Bar::f;             //правильно.
```

Для оголошення покажчика на функцію-члена класу можна використовувати **typedef**.

Наприклад:

```
typedef int(Bar::*pf)();
pf pfunc=&Bar::f;
```

До покажчиків на члени класу можна звертатися тільки за допомогою конкретного об'єкта або покажчика на об'єкт класу.

Наприклад:

```
int main(){
Bar obj;
(obj.*pfunc)();
Bar* pobj;
(pobj->*pfunc)();
...
```

Запишемо приклад з використанням **typedef**:

```
typedef int (Bar::*pf)();
class Bar{
private:
int a;
...
public:
int func(pf pfunc){
...
(this->*pfunc)();
...} }.
```

Покажчики на статичні члени класу

Покажчики на статичні члени класу — це звичайні покажчики.

Наприклад:

```
class Bar{
public:
static int func();
...}
int (*pf)();
pf=&Bar::func;                    //правильно;
int (Bar::*pf1)();
```

```
pf1=&Bar::func;          //error, pf1 — це покажчик
                         на функцію-члена класу.
```

Об'єднання

Об'єднання — це особливий вид класу. Дані члени в ньому зберігаються таким чином, що перекривають один одного. Всі члени розміщуються, починаючи з однакової адреси. Для об'єднання приділяється стільки пам'яті, скільки необхідно для зберігання найбільшого його члена.

Наприклад:

```
union Val{
    char value;
    int ival;
    double dval;
};
```

Об'єкт типу *Val* буде дорівнювати розміру *dval*, як найбільшого за розміром члена об'єднання. Члени об'єднання можна оголошувати відкритими, закритими або захищеними.

Наприклад:

```
union myunion{
private:
    int val;
public:
    double dval;
    int func(){...;}
};
```

Об'єднання поводиться так само, як і клас.

Звернення до членів об'єднання здійснюється так само, як і до членів класу.

Наприклад:

```
union myunion{
public:
int val;
double dval;
}
```

```
    int main(){
  myunion mobj;
    mobj.val=200;
    cout<<mobj.dval<<endl;      //повинне бути 200;
                                //однак Visual studio 6.0
                                ототожнює class і union
return 0;}.
```

Існують також анонімні об'єднання.

Наприклад:

```
  class myclass{
  public:
    int a;
    union{
      int val;
      double dval;
    }
  }.
```

До даних анонімного об'єднання звертатися можливо у тій самій області видимості, де воно визначено:

```
  int main(){
  myclass mobj;
  mobj.a=10;
  mobj.val=300;
  return 0;}.
```

Бітові поля

Для зберігання заданого числа бітів можна оголошувати член класу як бітове поле. Після ідентифікатора бітового поля вказується двокрапка, а потім константний вираз, що задає кількість бітів.

Наприклад:

```
  class myclass{
  public:
  unsigned int mybit:1;
  ...}.
```

Бітове поле повинне мати цілий тип даних. Доступ до бітового поля здійснюється також, як і до інших членів класу:

```
void func(){
myclass m;
m.mybit=1;
}.
```

Розв'язання дозволу імен в області видимості класу

Клас можна розглядати як простір імен. Розв'язання дозволу імені проводиться за допомогою оператора **::**

Наприклад:

```
int val;
class myclass{
private:
   int val;
public:
   int func(int val){
   int a=val;                    //локальний параметр val;
   int b=::val;                  //глобальна змінна val;
   int c=this->val;              //член класу;
   int d=myclass::val;           //член класу;
...
}.
```

Локальні класи

Клас, визначений всередині тіла функції, називається локальним. Він видимий тільки у тій області видимості, де він визначений. До члена такого класу не можна звертатися ззовні локальної області видимості, яка містить його визначення. Функції-члени локального класу повинні визначатися всередині самого класу.

Наприклад:

```
void func(){
class Bar{
   public:
```

```
    int a;
    int b;}
    Bar bobj;
    bobj.a=10;
...
}.
```

Вкладені класи

Клас, оголошений всередині іншого класу, називається вкладеним. Визначення вкладеного класу може перебувати у кожній із секцій **public, protected, private**.

Наприклад:

```
class Node{...};
class List{
   public:
   class Node{
     private:
     int data;
     Node* next;
     public:
     int get(){return data;}
     Node* get(){return next;}
     };
   void func(){
  Node* p;                          //локальний Node;
   }
}

int main(){List* lobj;
List::Node* nobj;
int c=nobj.get();
...
}.
```

Клас, що містить інші класи, можна оголосити другом вкладеного класу:

```
class List{
public:
   class Node{
   friend class List;
...
}
};
```

Вкладений клас *Node* має доступ до закритих членів класу ***List***, що містить його, а клас *List* доступу до членів вкладеного класу *Node* не має.

Наприклад:

```
class List{
    private:
    int s;
    public:
    class Node{
        private:
        int d;
        public:
        void get(){
        int h=s; }                          //правильно;
    };
    void get(){
    int t=d; };                             //error;
};
```

Лекція 11. КОНСТРУКТОРИ ТА ДЕСТРУКТОРИ КЛАСУ

Конструктори класу

Конструктор — визначена проектувальником функція (можливо перевантажена), що автоматично застосовується до кожного об'єкта класу перед його використанням.

Ім'я конструктора класу збігається з іменем класу.

Наприклад:

```
class Account{
private:
    char* name;
    double balance;
public:
    Account();
...}.
```

Єдине синтаксичне обмеження, яке накладається на конструктор, полягає у тому, що він не повинен мати тип значення, який повертається, навіть тип **void**.

Наприклад, помилкові оголошення:

```
void Account::Account();//error
Account* Account::Account(const char* pc){..}; //error.
```

В одному класі може бути кілька конструкторів.

Наприклад:

```
class myclass{
private:
   int a;
   double b;
public:
   myclass(){                    //конструктор за замовчуванням;
   a=0;
   b=0.0;}
   myclass(int _a, double _b=0){ //конструктор з параметрами;
   a=_a; b=_b;} }.
```

Об'єкт класу *myclass*, що ініціалізується конструктором, можна оголосити у такий спосіб:

```
myclass mobj;                    //викликається конструктор за замовчуванням;
myclass mobj1(3,4.5);            //викликається конструктор myclass(int _a, double _b=0);
myclass mobj2(3);                // викликається конструктор myclass(int _a, double _b=0).
```

Якщо в класі не оголошено жодного конструктора, то створюється неініціалізований об'єкт.

Наприклад:

```
class myclass{
private:
        int a;
public:
...}
int main(){
myclass obj;                     //об'єкт неініціалізовано;
...}.
```

Однак, якщо в класі був оголошений хоча б один конструктор, то об'єкт повинен бути обов'язково ініціалізований.

Наприклад:

```
    class myclass{
    private:
       int a;
    public:
       myclass(int s){...};
}
int main(){
myclass obj;                     //error, відсутній конструктор за замовчуванням;
...}.
```

Існує альтернативний синтаксис: список ініціалізації членів, у якому через кому вказуються імена й початкові значення:

```
class myclass{
```

```
private:
    int a;
    double b;
public:
    myclass():a(0),b(0.0){};
    myclass(int s,double f):a(s),b(f){...};
};
```

Цей синтаксис більш коректний.

Конструктор не можна повідомляти із ключовими словами **const** і **volatile**.

Обмеження прав на створення об'єкта

Можна обмежити або заборонити деякі форми створення об'єктів, якщо помістити конструктор у закриту або захищену секцію:

```
class myclass{
private:
        int a;
        myclass();
public:
...
};
```

Копіювальний конструктор

Ініціалізація об'єкта іншим об'єктом того самого класу називається почленною ініціалізацією за замовчуванням. Копіювання одного об'єкта в іншій виконується шляхом послідовного копіювання кожного нестатичного члена. Таку послідовність дій можна змінити, оголосивши копіювальний конструктор.

Наприклад:

```
class myclass{
private:
    int a;
    double b;
public:
    myclass(const myclass& obj):a(obj.a),b(obj.b){...}; }  //ко-
піювальний конструктор.
```

Деструктор класу

Деструктор — це спеціальна зумовлена користувачем функція-член, що автоматично викликається, коли об'єкт виходить за межі області видимості, або коли до покажчика на об'єкт застосовується операція **delete**. Ім'я цієї функції починається з позначки ~.

Наприклад:

```
class myclass{
private:
    int* a;
    double b;
public:
    myclass(int d, double s):a(new int(d)),b(s){};
    ~myclass(){                    //деструктор класу myclass
    delete a;
    b=0.0;}
};
```

У принципі деструктор може реалізовувати будь-яку операцію, а не тільки звільнення ресурсів.

Деструктор не може бути перевантажений.

Явний виклик деструктора

Іноді деструктор для деякого об'єкта можна викликати явно.

Наприклад:

```
class myclass{
private:
...
public:
    ~myclass(){...;}
int main(){
int* a=new int[10];
myclass* pt=new (a) myclass;
pt->~myclass();                //можна було б застосувати delete pt;
...
```

Лекція 12. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ

Перевантаження операторів

Більшість операторів у C++ можуть бути перевантажені. Перевантажувати можна такі оператори:

+, -, *, /, %, ^, &, |, ~, !, , , =, <, >, <=, >=, ++, --, <<, >>, ==, !=, &&, ||, +=, -, =, /=, %=, ^=, &=, |=, *=, <<=, >>=, [], (), ->, ->*, new, new[], delete, delete[].

Оператори, які не можна перевантажувати:

::, .*, . , ?:

Розглянемо деякі оператори, які можуть бути перевантажені:

operator==

Цей оператор може бути членом класу, але його можна також оголошувати поза класом.

а) operator== є членом класу.

У цьому випадку його перевантаження виглядає так:

```
class myclass{
private:
        int a;
public:
        int get(){return a;};
        bool operator==(const myclass& obj){
        if(a==obj.a) return true;
        else return false;}
...
```

Виклик даного оператора буде виглядати таким чином:

```
int main(){
myclass a,b;
if(a==b) ...
...};
```

б) operator== не є членом класу.

Тоді його перевантаження буде мати вигляд:

```
bool operator==(const myclass& s, const myclass& s1)
{
```

```
if(s.get()==s1.get()) return true;
else return false;
};
```

в) operator== є другом класу.

Тоді маємо

```
bool operator==(const myclass& s,const myclass& s1)
{
if(s.a==s1.s) return true;
else return false;
};
class myclass{
private:
    int a;
public:
    friend bool operator==(const myclass&,const myclass&);
...}
}.
```

operator=

Оператор присвоєння можна перевантажувати тільки в класі. Наприклад:

```
class myclass{
private:
    int a;
public:
    myclass& operator=(const myclass& obj){
    a=obj.a;
    return *this;}
}.
```

operator[]

Оператор індексування **operator[]** визначається для класів. Наприклад:

```
class myclass{
private:
```

```
        int* a;
    public:
        myclass(int d)
        {
        a=new int[d];
        ...}
        int& operator[](int index){
        return a[index];
    }
    }.
```

Використання цього оператора може бути наступним:

```
    int main(){
    myclass obj(34);
    int c=obj[5];
    obj[7]=56;
    ...
    return 0;
    }.
```

Розглянемо приклад створення багатомірних масивів об'єктів.

```
class array_mas{
private:
    int* a;
    int size;
public:
    array_mas():size(10){
    a=new int[size];}
    int& operator[](int value){ return a[value];}
    ~array_mas(){ size=0; delete [] a;}
};

class array{
private:
    array_mas* ma;
    int size;
public:
```

```
    array(int s):size(s){
    ma=new array_mas[size];}
    array_mas& operator[](int value){return ma[value];}
    ~array(){size=0; delete [] ma;}
}
int main(){
array obj(10);
obj[1][1]=10;
...}.
```

operator()

Оператор виклику функції **"()"** може бути перевантажений для об'єктів типу клас. Якщо визначено клас, який містить деяку операцію, то для її виклику перевантажується відповідний параметр. Перевантажений оператор **operator()** може бути оголошений як функція-член з довільним числом параметрів.

Наприклад:

```
class myclass{
private:
    double f;
public:
    double operator(double a, double b){
    f=a;
    double res=f+b;
    return res;}
};
int main(){
myclass obj;
double t=obj(2.3,6.7);//t дорівнює 9.
```

operator ->

Оператор **"->"**, що дає доступ до членів класу, може перевантажуватися для об'єктів класу. Оператор доступу до членів класу — це унарний оператор, тому параметри йому не передаються.

Перевантажений оператор **"->"** повинен повертати або покажчик на тип класу, або об'єкт класу. Якщо вертається покажчик, то до нього застосовується семантика вбудованого оператора **"->"**.

Наприклад:

```
class A{
private:
    A* ptr;
public:
    void f(){ cout<<"call function f from a"<<endl;  } };

class myclass{
private:
    A* p;
public:
    myclass(){  p=new A;  }
    void f(){ cout<<"call function f from myclass"<<endl;  }
    A& operator*(){ cout<<"call operator*"<<endl; return *p;}
    A* operator->(){ cout<<"call operator->"<<endl; return p;}
};
int main(){
myclass* obj =new myclass;
obj->f();
(*obj).f();
myclass obj1;
obj 1-1->f();
(*obj1).f();
return 0;}.
```

operator ++ та operator --

Ці оператори можуть перевантажуватися у префіксній і постфіксній формах.

Такі оператори можуть перевантажуватися для об'єктів класу.

а) префіксна форма:

```
class myclass{
private:
    int s;
```

```
public:
        myclass():s(0){};
        myclass& operator++(){    //можна повертати посилання не тільки на myclass
        s+=2;
        return *this;}
        int get(){return s;}
};
```

Виклик цього оператора може мати вигляд:

```
int main(){
myclass obj;
cout<<obj.get()<<endl;
++obj;
cout<<obj.get()<<endl;
return 0;};
```

б) постфіксна форма:

```
class myclass{
private:
    int s;
public:
    myclass():s(0){};
    myclass& operator++(int g){
    if(g==0) s+=2;
    else s+=g;
    return *this;}
    int get(){return s;}
};
```

Виклик такого оператора може мати вигляд:

```
int main(){
myclass obj;
obj.operator++(10);
obj++;
return 0;}.
```

Аналогічно для оператора **декремента**.

Перевантажені оператори інкремента й декремента можна повідомляти друзям класу.

```
class myclass{
public:
        int s;
        myclass():s(0){};
        friend myclass& operator++(myclass&,int);
        int get(){return s;}
};
myclass& operator++(myclass& obj,int g){
        if(g==0) obj.s+=2;
        else obj.s+=g;
        return obj;};
int main(){
myclass obj;
operator++(obj,10);
obj++;
return 0;}.
```

operator *, opeator +, operator -, operator /

Як приклад розглянемо оператор додавання. Інші перевантажуються аналогічно. Оператор додавання може бути перевантажений у класі, поза класом або оголошений другом класу:

а) оператор додавання як функція-член класу.

Наприклад:

```
class myclass{
private:
    int s;
public:
    myclass(int d):s(d){}:
    myclass operator+(const myclass& obj){ //вказується
                                    тільки один параметр
    return myclass(s+obj.s);}};
int main(){
myclass obj1(10);
myclass obj2(20);
```

```
myclass obj3=obj1+obj2;
...};
```

б) оператор додавання, оголошений поза класом.

Наприклад:

```
class myclass{
private:
    int s;
public:
    myclass(int d):s(d){};
    int get() const {return s;}};
myclass operator+(const myclass& s1, const myclass& s2) // у цьому випадку треба вказувати два параметри
    {return myclass(s1.get()+s2.get()); };
int main(){
myclass obj1(10);
myclass obj2(20);
myclass obj3=obj1+obj2;
...};
```

в) оператор додавання, як друг класу.

Наприклад:

```
class myclass{
private:
    int s;
public:
    myclass(int d):s(d){};
    friend myclass operator+(const myclass&, const myclass&);
    int get() const {return s;}};
    myclass operator+(const myclass& s1, const myclass& s2)
    { return myclass(s1.get()+s2.get()); };
int main(){
myclass obj1(10);
myclass obj2(20);
myclass obj3=obj1+obj2;
...}.
```

operator ,

Оператор кома може бути перевантажений як у класі, так і поза класом:

а) перевантаження у класі.

Приклад 1:

```
class myclass{
public:
    int s;
    myclass(int d):s(d){};
    void operator,(myclass& obj){ obj.s=s;};
    int get() const {return s;}};
int main(){
myclass obj1(10);
myclass obj2(20);
obj1,obj2;
...}.
```

Приклад 2:

```
class myclass{
public:
    int s;
    myclass(int d):s(d){};
    myclass& operator,(myclass& obj){
    obj.s=s;
    return obj;};
    int get() const {return s;};
    };
int main(){
myclass obj1(10);
myclass obj2(20);
obj1,obj2;
...
};
```

б) перевантаження поза класом.

Приклад:

```
class myclass{
public:
    int s;
```

```
        myclass(int d):s(d){};
        int get() const {return s;};
    };
    void operator,(const myclass& obj,myclass& obj1)
    {
    cout<<“ok”<<endl;
    obj1.s=obj.s;
    };
    int main(){
    myclass obj1(10);
    myclass obj2(20);
    obj1,obj2;
    ...};
```

в) оператор кома як друг класу:

```
class myclass{
public:
        int s;
        myclass(int d):s(d){};
        friend void operator,(const myclass&, myclass&)
        int get() const {return s;};
};

void operator, (const myclass& obj,myclass& obj1)
{
cout<<“ok”<<endl;
obj1.s=obj.s;
};
int main(){
myclass obj1(10);
myclass obj2(20);
obj1,obj2;
...}.
```

operator<< і operator>>

Оператори **operator<<** і **operator>>** можуть бути перевантажені тільки поза класом, або можуть бути друзями класу. Розглянемо приклади.

Приклад 1:

```
class myclass{
public:
    int s;
...;
ostream& operator<< (ostream& os, const myclass& obj){
    os<<obj.s<<endl;
    return os;}.
```

Виклик оператора буде мати вигляд:

```
int main(){
myclass obj;
cout<<obj;
...
```

Приклад 2:

```
class myclass{
public:
    int s;
    ...;
istream& operator>> (istream& is, const myclass& obj){
    is>>obj.sl;
    return is;}.
```

Відповідно виклик оператора буде мати вигляд:

```
int main(){
myclass obj;
cin>>obj.
```

Оператори new і delete

За замовчуванням у стандартній бібліотеці C++ для виділення пам'яті об'єкта класу з купи й звільнення зайнятої їм пам'яті визначені відповідно глобальні оператори **new()** і **delete()**. Але можна реалізувати власну стратегію керування пам'яттю за допомогою перевантаження операторів **new()** і **delete()**.

Оператор new

Оператор **new()** може бути перевантажений тільки в класі. Визначення перевантаженого оператора **new()** має вигляд:

```
class myclass{
public:
void* operator new(size_t);
...}.
```

Тип *size_t* визначено у заголовному файлі *cstddef.h*. Параметр *size_t* оператора *new()* автоматично ініціалізується значенням, рівним розміру *myclass* у байтах.

За допомогою оператора розв'язання доступу глобальної області видимості можна викликати глобальний *new()*, навіть якщо в класі *myclass* визначена перевантажена версія цього оператора:

```
myclass* ps=::new myclass.
```

Наприклад:

```
class myclass{
private:
        class Node{
public:
int data;
        Node* next;
        }
        Node* first;
        int size;
public:
        void* operator new(size_t);
        void print();
}

void* myclass::operator new(size_t s){
myclass* p=::new myclass; //::new — це виклик опертора new з глобального простору імен
p-p->size=10;
Node* firstel=::new Node;
firstel->data=0;
firstel->next=0;
Node* cur=firstel;
for(int i=1; i< p-p->size; i++){
cur->next=::new Node;
```

```
cur =cur->next;
cur->data=i;
cur->next=0;}
p-p->first=firstel;
return p;
};

void myclass::print()
{
Node* cur=first;
while(cur!=0)
{
cout<<cur->data<<endl;
cur=cur->next;
}
};
int main(){
myclass* p=new myclass;
p-p->print();
return 0;}.
```

Варто помітити, що перевантажений оператор *new()* є **статичним** оператором класу.

Оператор delete

Оператор **delete()** можна перевантажити тільки в класі. Перевантажений оператор **delete()**, як і оператор **new()**, є **статичним** оператором класу. Перевантаження оператора **delete()** виглядає так:

```
class myclass{
public:
void operator delete(void*).
```

Інший варіант перевантаження оператора **delete()** має вигляд:

```
class myclass{
public:
void operator delete(void*, size_t).
```

Розглянемо приклад перевантаження оператора **delete()** (розглянемо визначений раніше клас *myclass*):

```
class myclass{
private:
        class Node{
        public:
        int data;
        Node* next;
        }
        Node* first;
        int size;
public:
        void* operator new(size_t);
        void print();
        void operator delete(void*, size_t);
}

void myclass::operator delete(void*p, size_t size)
{
myclass* mp=(myclass*) p;
Node* firstel=mp->first;
Node* p1;
while(firstel!=0)
{p1=firstel;
firstel=firstel->next;
::delete p1;}// ::delete — це виклик оператору delete з глобально-
го простору імен
};

int main(){
myclass* p=new myclass;
p-p->print();
delete p;
return 0;}.
```

Використовуваний оператор **delete()** повинен відповідати тому оператору **new()**, за допомогою якого була виділена об'єкту

пам'ять. Якби в класі *myclass* не був перевантажений оператор *delete()*, то в *main()* був би викликаний стандартний оператор *delete()*.

Оператори new[] і delete[]

У класі можна перевантажувати також оператори **new[]** і **delete[]** для роботи з масивами.

Перевантаження оператора **new[]** виглядає таким чином:

class myclass{
public:
void* operator new[](size_t);
...}.

Коли за допомогою **new[]** створюється масив об'єктів типу класу, компілятор перевіряє, чи визначений у класі оператор **new[]**. Якщо так, то для виділення пам'яті під масив викликається саме він.

Перевантаження оператора **delete[]** виглядає таким чином:

class myclass{
public:
void operator delete[](void*).

Інший варіант

void operator delete[](void*, size_t);
}.

Оператори **new[]** і **delete[]** є статичними операторами класу.

Наприклад:

```
class Massive{
private:
        int a;
public:
        Massive(int c=0):a(0){};
        static void* operator new[](size_t );
        static void operator delete[](void*,size_t);
}

void* Massive::operator new[](size_t size){
```

```
cout<<“our new[]”<<endl;
cout<<“size ”<<(size/sizeof(Massive))<<endl;
int n= size/sizeof(Massive);
Massive* first=::new Massive[n];
return first;};
Void Massive::operator delete[](void* first,size_t size){
cout<<“our delete[]”<<endl;
Massive* first1=reinterpret_cast<Massive*>(first);
::delete []first1;};
int main(){
Massive* obj=new Massive[5];
delete []obj;
return 0;}.
```

Оператори розміщення new() і delete()

Оператор розміщення **new()** може бути перевантажений за умови, що всі оголошення мають різні списки параметрів. Перший параметр повинен бути обов’язково **size_t**. Оператори розміщення теж є статичними операторами класу.

Наприклад:

```
class myclass{
private:
    class Node{
    public:
        int data;
        Node* next;
    };
    Node* first;
    int size;
public:
    void* operator new(size_t,int);
    void operator delete(void*,size_t);
};

void* myclass::operator new(size_t size, int a){
```

```
myclass* p=::new myclass;
p-p->size=a;
Node* firstel=::new Node;
firstel->data=0;
firstel->next=0;
Node* cur=firstel;
for(int i=1; i< p-p->size; i++){
cur->next=::new Node;
cur =cur->next;
cur->data=i;
cur->next=0;}
p-p->first=firstel;
return p;};

void myclass::operator delete(void* p,size_t s)
{ myclass* mp=(myclass*) p;
Node* firstel=mp->first;
Node* p1;
while(firstel!=0)
{ p1=firstel;
firstel=firstel->next;
    ::delete p1;}
};
int main(){
myclass* p=new (12) myclass;
delete p;
return 0;
}.
```

Існують також оператори розміщення new[] і delete[].

ВИЗНАЧЕНІ КОРИСТУВАЧЕМ ПЕРЕТВОРЕННЯ

Конвертер

Конвертер — це особливий випадок функції-члена, що реалізує визначене користувачем перетворення об'єкта у деякий інший тип. Конвертер визначається всередині класу.

Конвертер визначається наступним чином:

class myclass{
public:
operator type().

Розглянемо кілька прикладів.

Приклад 1:

```
class myclass{
private:
    double a;
public:
    ...
    operator double(){return a;}};
    double f(double b){return b};
}
int main(){
myclass obj;
cout<<f(obj)<<endl;
return 0;}.
```

Приклад 2:

```
class B{
...}
class A{
private:
    B obj;
public:
    operator B(){return obj;}
...}.
```

Конструктор як конвертер

Набір конструкторів класу, що приймають єдиний параметр, наприклад, *myclass(int)* класу *myclass*, визначає множину неявних перетворень у значеннях типу *myclass*. Тобто конструктор *myclass(int)* перетворює значення типу *int* у значення типу *myclass*.

Однак можна заборонити використовувати конструктор як конвертер за допомогою ключового слова **explicit**.

Наприклад:

```
class myclass{
private:
        double a;
public:
        myclass(double s):a(s){};
...}
void f(myclass g){...};
int main(){
double r;
f(r);                                   //правильно;
...}
class myclass{
private:
        double a;
public:
        explicit myclass(double s):a(s){};
...}
void f(myclass g){...};
int main(){
double r;
f(r);                                   //error, конструктор має
                                        оголошення explicit;
...}.
```

Лекція 13. ШАБЛОНИ КЛАСІВ

C++ дозволяє крім шаблонів функцій визначати також шаблони класів. Розглянемо приклад визначення шаблону класу:

```
template<class Type>
class Queue{
public:
        Queue();
        ~Queue();
        Type& remove();
        void add(const Type&);
        bool is_empt();
        ...
private:
        Type s;
};
```

Використання такого шаблону класу може бути наступне:

```
int main(){
Queue<int> q;
Queue<vector<double>> vs;
...
```

Визначення шаблону можна також задавати у вигляді:

template<class T,class U>
class Test{…}.

Коли параметр не є типом параметра шаблону, то маємо звичайне оголошення:

```
template<class Type,int size>
class Buffer{...}.
```

У наступному прикладі:

```
typedef double Type;
template<class Type>
class Queue{
private:
Type item;
...}.
```

Об'єкт *item* не обов'язково буде типу *double*.

Ім'я параметра шаблону може зустрічатися тільки один раз.

Параметрами шаблону можуть бути аргументи за замовчуванням.

Наприклад:

```
template<class Type=string, int size>
class Buffer{....}
template <class Type, int size=1024>
class Test{...}.
```

Конкретизація шаблону класу

У визначенні шаблону вказується як будувати індивідуальні класи, якщо вказані один або кілька типів чи значень.

Розглянемо приклади конкретизації шаблонів. Нехай є шаблон класу:

```
template<class Type>
class Queue{
public:
    Queue():front(0),back(0){};
    ~Queue();
    Type& remove();
    void add(const Type& );
private:
    Queueitem<Type> first;
    }.
```

Приклади конкретизації.

Приклад1:

```
int main(){
Queue<int> d;
Queue<string> h;
Queue<vector<int> v;
...}.
```

Приклад 2:

```
template<class Type>
void func(const Queue<Type>& q, Type fd)
{...                                    //шаблон функції;
```

```
int main(){
Queue<int> g;
int d=10;
func(g,d);
...}.
```

Приклад 3:

```
template<class Type>
void func(const Queue<int>& q, Type fd)
{...                                    //шаблон функції;
int main(){
Queue<int> g;
int d=10;
func(g,d);
...}.
```

Тут варто зазначити, що шаблон класу у функції *func* не конкретизується при компіляції, хоча вказаний тип *int*. Він конкретизується тільки при виклику функції.

Аргументи шаблону для параметрів констант

Параметр шаблону класу може й не бути типом. На такі аргументи накладаються деякі обмеження.

Наприклад:

```
template<int h, int w>
class myclass{
public:
myclass():a(h),b(w){};
...
private:
int a;
int b;
}.
```

Вираз, з яким зв'язаний параметр, що не є типом, повинен бути константним, тобто такий, що обчислюється під час компіляції.

Приклад 1:

```
int main(){
myclass<30,40> q;          //правильно 30 і 40 константи.
```

Приклад 2:

```
template<int* ptr>
class Buffer{...};
int main(){
Buffer<new int[25]> d;          //error, не можна обчислити під час компіляції;
...}.
```

Між типом аргументу шаблону й типом параметра-константи припустимі деякі перетворення.

1. Перетворення **lvalue**, що включають: перетворення **lvalue** в **rvalue**, масиву в покажчик та функції в покажчик.

Наприклад:

```
template<int* p>
class Buffer{...}
int main(){
int a[10];
Buffer<a> d.
```

2. Перетворення кваліфікаторів.

Наприклад:

```
template<int* p>
class Buffer{...}
int main(){
int a;
Buffer<&a> d;
...}.
```

3. Підвищення типів.

Наприклад:

```
template<int h, int w>
class myclass{...;
int main(){
const short sh=20;
const short dh=30;
myclass<sh,dh> fg;
...
}.
```

4. Перетворення цілочисельних типів.

Наприклад:

```
template<unsigned int size>
class buffer{...};
int main(){
buffer<1024> bf;          //перетворення з int в
                          unsigned int;
...}.
```

Функції-члени шаблонів класів

При визначенні функції-члена шаблону поза визначенням самого шаблону варто застосовувати спеціальний синтаксис.

Наприклад:

```
template<class Type>
class Queue{
public:
Queue();
private:
...}
template<class Type>
Queue<Type>::Queue(){...}.
```

Оголошення друзів у шаблонах класів

У шаблоні класу можуть бути присутні три види оголошень друзів.

1. Звичайний (не шаблонний) клас-друг або функція-друг.

Наприклад:

```
class myclass{
void func(); };
template<class Type>
class Queueitem{
friend class myclass1;
friend void myclass::func();
```

2. Зв'язаний дружній шаблон класу або функції.

Наприклад:

```
template<class Type>
class myclass1{...};
```

```
template<class Type>
void func(myclass1<Type>&);
template<class Type>
class Queue{
void f();
...};
template<class Type>
class Queueitem{
friend class myclass1<Type>;
friend void func<Type>(myclass1<Type>&);
friend void Queue<Type>::f();
...}.
```

3. Незв'язаний дружній шаблон класу.

Наприклад:

```
template<class Type>
class Queueitem{
template<class U>
friend class myclass1;
template<class U>
friend void func (myclass1<U>&);
template<class U>
friend class Queue<U>::f();
...
```

Статичні члени шаблонів класу

У шаблоні класу можуть бути оголошені статичні дані-члени.

Наприклад:

```
template<class Type>
class Queue{
private:
static Queueitem<Type> q;
static const int a.
```

Ініціалізація таких членів буде виглядати так:

```
template<class Type>
Queueitem<Type> Queue<Type>:: q=...;
template <class Type>
const int Queue<Type>::a=0;
```

Вкладені типи шаблонів класів

У шаблоні класу можна визначити вкладені шаблони.

Наприклад:

```
template<class Type>
class Queue{
...
private:
class Queueitem{
public:
Queueitem(Type val):item(val){....}
Type item;
Queueitem* next;};
...}.
```

Шаблон *Queueitem* конкретизується тим самим типом, що й шаблон *Queue*. Вкладені класи шаблонів так само є шаблонами.

Варто зазначити, що вкладений у шаблон клас конкретизується тільки у тому випадку, якщо він використовується у контексті, де потрібен повний тип класу.

Шаблони-члени

Шаблон функції або класу може бути членом звичайного класу або шаблону класу.

Наприклад:

```
template<class T>
class Queue{
private:
template<class Type>
class C{
Type member;
T mem;};
public:
template<class U>
void func(U a){...}
...
int main(){
Queue<int> q;
double b=2.3;
```

```
Queue<int> :: func(b);
...}.
```

Спеціалізації шаблонів класів

Шаблони класів можна явно спеціалізувати.

Наприклад:

```
template<class Type>
class Queue{
private:
Type item;
Type item1;
public:
Type max(){...};
...}.
```

Явна спеціалізація шаблону:

```
template<> double Queue<double>::max()
{...};

int main(){
Queue<int> q;
q.max();                    //викликається max, конкретизована із шаблону;
Queue<double> s;
s.max();                    //викликається явна спеціалізація функції max, що має пріоритет перед функцією max, конкретизованої із шаблону;
...
}.
```

Аналогічно можна оголошувати явні спеціалізації шаблонів класів.

Часткова спеціалізація шаблону

Шаблон класу може бути спеціалізований частково.

Наприклад:

Шаблон класу має вигляд:

```
template<class T, int Rows>
class Matrix{
    T* matr;
public:
    Matrix(){
        matr=new T[Rows];
        for(int i=0;i<Rows; i++)
        cin>>matr[i];  }
    void print(){
        for(int i=0;i<Rows; i++)
        cout<<matr[i];  }
    ~Matrix(){ delete [] matr; }
};
```

Частково спеціалізований шаблон має вигляд:

```
template<int Rows>
class Matrix<float,Rows>{
    float* matr;
public:
    Matrix(){
        matr=new float[Rows];
        for(int i=0;i<Rows; i++)
        cin>>matr[i];
        cout<<"ok"<<endl;  }
    void print(){
        for(int i=0;i<Rows; i++)
        cout<<matr[i];  }
    ~Matrix(){ delete [] matr; }
};

int main(){
    Matrix<float,5> d;
d.print();
return 0;}.
```

Однак подібний механізм не працює в Microsoft Visual Studio 6.0.

Можливий альтернативний спосіб може мати вигляд (в Інтернеті можна знайти інші способи):

```
template<class T, int Rows>
class Matrix{
    template<class U>
    class Matrix_common{
    U* matr;
public:
    Matrix_common(){
        matr=new U[Rows];
        for(int i=0;i<Rows; i++)
        cin>>matr[i];                    }
    void print(){
        for(int i=0;i<Rows; i++)
        cout<<matr[i];  }
};
template<> class Matrix_common<float>{
    float* matr;
public:
    Matrix_common<float>(){
        matr=new float[Rows];
        for(int i=0;i<Rows; i++)
        cin>>matr[i]; };
    void print(){
        for(int i=0;i<Rows; i++)
        cout<<matr[i];  };
};
Matrix_common<T>* p;
public:
    Matrix(){ p=new Matrix_common<T>; }
    void print(){ p-p->print(); }
    ~Matrix(){  delete p; }
    };
int main(){
Matrix<float,5> d;
d.print();
return 0;
}.
```

Лекція 14. ВИКЛЮЧЕННЯ

Виключення — це аномальна поведінка, яку програма може виявити під час виконання, наприклад, ділення на 0, вихід за межі масиву та ін. Такі виключення порушують нормальний хід роботи програми і на них потрібно реагувати.

Генерація виключень

Генерація виключень відбувається за допомогою інструкції **throw**.

Розглянемо приклад **throw** конструктора:

```
class ListOnEmpty{
private:
    string error;
public:
    ListOnEmpty(string s):error(s){};
    String getMessage(){return error;}
};
class List{
private:
class Node{
        int data;
        Node* next;
        }
Node* first;
int del(int k) throw (ListOnEmpty){
    if (first==0) throw ListOnEmpty("spisok pust"); //генерація
                                        виключення
    else{
    Node* cur=first;
    for(int i=0;i<k;i++)
    cur=cur->next;
    Node* p=cur->next;
    cur->next=cur->next->next;
    int c= p-p->data;
```

```
    delete p;
    }
    return c;
}.
```

Try і Catch блоки

Виключення можна перехоплювати безпосередньо у тій самій функції, де відбулась його генерація, або перекласти обробку цього виключення на функцію, що викликається (як це було зроблено вище). Обробка виключень здійснюється за допомогою **try** і **catch** блоків.

Наприклад:

```
int del(int k){
    try{
    if (first==0) throw ListOnEmpty("spisok pust");        //ге-
                                                  нерація виключення
    else{
    Node* cur=first;
    for(int i=0;i<k;i++)
    cur=cur->next;
    Node* p=cur->next;
    cur->next=cur->next->next;
    int c= p-p->data;
    delete p;
    }
return c;
}
catch(ListOnEmpty e){                  // більш коректним є запис
                                       ListOnEmpty& e
cerr<<e.getMessage();}
catch(Exception e){...}
}.
```

Пошук **catch**-оброблювача здійснюється у такий спосіб. Коли вираз **throw** перебуває у **try** блоці, всі асоційовані з ним **catch**

блоки досліджуються з погляду на те, чи можуть вони обробити виключення. Якщо оброблювач знайдений, то виключення обробляється, а якщо ні, то пошук триває у функції, що викликається. Якщо у програмі немає **catch**-оброблювача здатного обробити виключення, то воно залишається не обробленим і керування передається функції **terminate()** зі стандартної бібліотеки **C++**. За замовчуванням **terminate()** активізує функцію **abort()**, що аномально завершує програму. Дії, які виконуються функцією **terminate()**, можна перевизначити.

Повторна генерація виключення

Може виявитися так, що в одному виклику **catch** блоку не вдалося повністю обробити виключення, і виконавши коригувальні дії, **catch**-оброблювач може вирішити, що подальшу обробку варто доручити функції, розташованій вище у ланцюжку викликів. Передати виключення іншому **catch**-оброблювачу можна за допомогою повторної генерації виключення.

Для цієї мети використовується конструкція **throw**, яка знову генерує об'єкт-виключення. Варто пам'ятати, що при повторній генерації новий об'єкт-виключення не створюється. Це має значення у тому випадку, коли **catch**-оброблювач модифікує об'єкт, перш ніж згенерувати виключення повторно. Це означає, що у **catch** блок об'єкт-виключення потрібно передавати за посиланням.

Наприклад:

```
class ListOnEmpty{
private:
    string error;
    int count;
public:
    ListOnEmpty(string s):error(s),count(0){};
    String getMessage(){return error;}
    void incr(){count++;}
    int show(){return count;}
};
class List{
```

```
    int del(int k){
    try{
    if (first==0) throw ListOnEmpty("spisok pust");
                                        //генерування виключення
    else{
    Node* cur=first;
    for(int i=0;i<k;i++)
    cur=cur->next;
    Node* p=cur->next;
    cur->next=cur->next->next;
    int c= p-p->data;
    delete p;}
    return c;}
    catch(ListOnEmpty& e){
    cerr<<e.getMessage();
    if (e.show()<5){
    e.incr();
    return;
}
};
catch(Exception e){...}
}
...}.
```

Перехоплення всіх виключень

Іноді функції потрібно виконати певну дію до того, як вона завершить обробку виключення, навіть незважаючи на те, що обробити його вона не може. Для цього можна використовувати конструкцію, що дозволяє перехопити всі виключення.

catch(…){…}.

Наприклад:

```
int del(int k){
    try{
    if (first==0) throw ListOnEmpty("spisok pust");//генерування виключення
```

```
        else{
        Node* cur=first;
        for(int i=0;i<k;i++)
        cur=cur->next;
        Node* p=cur->next;
        cur->next=cur->next->next;
        int c= p-p->data;
        delete p;
        }
    return c;
    }
    catch(ListOnEmpty& e){
    cerr<<e.getMessage();
    if (e.show()<5){
    e.incr();
    return;
    } };
    catch(Exception e){...}
    catch(...){//Перехоплюємо решту виключень

    Node* cur=first;
    while (first->next!=0)
    { delete first;
    first=cur->next;
    cur=first;}
    delete first;
    }
    }.
```

Специфікація виключень

Специфікація виключень дає змогу визначити в оголошенні функції всі виключення, які вона може генерувати. При цьому гарантується, що інші виключення функція генерувати не може.

Наприклад:

int del(int) throw(ListOnEmpty,Exception1).

Якщо в оголошенні функції присутня специфікація виключень, то при повторному оголошенні цієї самої функції повинні бути перераховані такі самі типи.

Наприклад:

```
extern int foo(int=0) throw(myException);
extern int foo(int param);          //error опущена специфікація виключень.
```

Виключення генерується тільки під час виконання програми. Під час компіляції невідомо, яке виключення зустрінеться. Тому порушення специфікації може бути виявлено тільки під час виконання програми.

Якщо функція генерує виключення не зазначене у специфікації, то викликається функція **unexpected()**, а та, у свою чергу, викликає **terminated()**. Дії, які виконуються функцією **unexpected()**, можна перевизначити.

Порожня специфікація вказує, що функція не генерує ніяких виключень.

Наприклад:

```
extern void myfunc() throw().
```

Між типом генерованого виключення і типом виключення, зазначеного у специфікації, не дозволяється проводити ніяких перетворень.

Наприклад:

```
int conv(int param) throw(string){
...
if (....)
throw "help";                   //error, неможливо конвертувати const char* в string
}.
```

Специфікації виключень і покажчики на функції

Специфікацію виключень можна задавати і при оголошенні покажчика на функцію.

Наприклад:

```
void (*pfunc) (int) throw(Exception).
```

Існують обмеження на тип покажчика, що використовується як ініціалізатор або розміщений у правій частині присвоювання. Обмеження на покажчик-ініціалізатор повинні бути такими самими або суворішими, ніж на покажчик, що ініціалізується.

Розглянемо приклади:

```
void f1(int) throw(Exception1);
void f2() throw();
void f3(int) throw(string,Exception1);
void (*pf1)(int) throw(Exception1)=&f1;  //правильно, обмеження, що накладаються на специфікації виключень pf1 і f1, однакові;
void (*pf2)() throw(string)=&f2;  //правильно, обмеження, що накладаються на специфікацію виключень f2 суворіші, ніж на pf2;
void (*pf3)(int) throw(string)=&f3;//error, обмеження, що накладаються на специфікацію виключень f3 менш суворі, ніж для pf3.
```

Лекція 15. КОНТЕЙНЕРНІ ТИПИ

Клас vector

Для використання класу **vector** необхідно підключити заголовний файл **#include<vector>**.

Приклад використання вектора:

```
#include<vector>
#include<iostream>
int main(){
vector<int> ivec;
cout<<“razmer:”<<ivec.size();
cout<<“emkost:”<<ivec.capacity();//кількість елементів, що може вмістити в себе контейнер;
for(int i=0;i<30;i++)
ivec.push_back(i);
cout<<ivec[10];
return 0;}
```

Можна створювати багатомірні вектори.

Наприклад:

```
typedef vector<int> vec;
vector<vec> ivec;
vec vec1;
int main(){
for(int i=0; i<30;i++){
for(int j=0;j<30;j++)
vec1.push_back(j);
ivec.push_back(vec1);}
cout << ivec[3][4];
return 0;}.
```

Клас list

Для використання списку необхідно підключити файл **#include<list>**.

Заповнення списку

```
#include<list>
#include<iostream>
int main(){
list<int> ilist;
if(ilist.empty()!=true)          //проблеми;
int i;
while(cin>>i)
ilist.push_back(i);              //додавання у кінець списку;
while(cin>>i)
ilist.push_front(i);             //додавання на початок
                                 списку;
//вставка елемента перед елементом зі значенням res
int res=3;
list<int>::iterator iter=find(ilist.begin(),ilist.end(),res);
ilist.insert(iter,ilist.begin(),ilist.end());
//видалення елемента res
res=5;
list<int>::iterator iter=find(ilist.begin().ilist.end(),res);
ilist.erase(iter);
//видалення діапазону елементів між res1 і res2
list<int>::iterator iter1=find(ilist.begin().ilist.end(),res1);
list<int>::iterator iter2=find(ilist.begin().ilist.end(),res2);
ilist.erase(iter1,iter2);
...
}.
```

У стандартну бібліотеку C++ входить також клас **deque**, для використання якого необхідно підключити файл **#include <deque>**.

Клас map

Для використання цього класу необхідно підключити файл **#include<map>**.

Розглянемо приклад використання класу **map**:

```
include<map>
```

```
include<string>
int main(){
map<string,int> map1;
map1[string("sssss")]=1;
//вставка в map здійснюється у такий спосіб
map1.insert(value_type(string(dddd),1));
//пошук елементів відображення
  int count;
if(map1.count("sssss"))
  count=map1["sssss"];          //функція count повертає
                                число елементів з даним
                                ключем
  count=0;
map<string,int>::iterator iter=map1.find("ddddd");
if(iter!=map1.end())
    count=(*iter).second; }.
```

Функція *find* повертає ітератор, що вказує на елемент, якщо ключ знайдений та ітератор **end()** в іншому випадку. Значенням ітератора є покажчик на об'єкт типу **pair**, в якому **first** містить ключ, а **second** — значення.

Лекція 16. СПАДКУВАННЯ

Спадкування

Будь-яке поняття не існує ізольовано, воно існує у взаємозв'язку з іншими поняттями й потужність такого поняття визначається наявністю зв'язків.

Клас служить для представлення поняття. Для такого представлення потрібно мати мовні засоби опису зв'язків.

Поняття похідного класу і підтримуючі його засоби служать для представлення ієрархічних зв'язків, тобто для вираження спільності між класами. Наприклад, поняття кола й трикутника пов'язані між собою, тому що вони представляють поняття фігури, тобто містять більш загальне поняття. Таким чином класи кола і трикутник потрібно представити так, щоб було очевидно, що в них є загальний клас — фігура. Розглянемо поняття похідного класу. У C++ клас спадкоємець визначається у такий спосіб:

class B:public A{…}
class B:private A{…}
class B:protected A{…}.

У випадку кола й трикутника маємо:

class Ring:public Shape{…}
class Triangle:public Shape{…}.

Розглянемо приклад спадкування. Припустимо, що необхідно написати програму обліку службовців деякої фірми. Маємо клас *employee(службовець)*:

```
class employee{
private:
    char* name;
    short age;
    short department;
    int salary;
public:
    employee* next;
...}.
```

Поле *next* необхідно, щоб зв'язати записи про службовців одного відділу.

Визначимо також клас *manager (керуючий відділом)*:

```
class manager{
private:
    employee* emp;          //запис employee для керуючого;
    employee* group;        //підлеглий колектив;
...
```

Так як керуючий теж службовець, то запис *employee* зберігається у члені *emp* об'єкта *manager*.

У даній реалізації покажчик на *manager* не є покажчиком на *employee*, а хотілося б.

Інакше кажучи, використовувати *manager* замість *employee* не можна.

Рішення є — *manager* повинен бути *employee* з деякою додатковою інформацією. Тобто у цьому випадку можна пов'язати ці поняття, використовуючи спадкування:

```
class manager: public employee{
private:
    employee* group;
...}.
```

Тоді можна написати

```
void spisok(){
manager m1,m2;
employee  e1, e2;
employee* elist;
elist=&m1;
m1.next=&e1;
e1.next=&m2;
m2.next=&e2;
e2.next=0;
...
}.
```

Тобто покажчик на *manager* можна присвоїти покажчику, який має тип *employee* неявно. Зворотне перетворення може бути тільки явним.

Наприклад:

```
void f(){
manager mm;
employee* pe=&mm;          //правильно;
employee ee;
manager* pm=&ee;           //error;
pm=(manager*) pe;          //правильно, але відповідальність за коректність несе програміст;
...
```

Поліморфізм

Поліморфізм — це можливість неявного присвоєння покажчика на базовий клас покажчика, на клас спадкоємець.

Наприклад:

```
class A{...};
class B:public A{...};
int main(){
A* pa=new B;               //поліморфізм.
```

Зворотне перетворення можливо тільки явно

```
B* pb=new A;               //error;
A* paa=new A;
B* pb=(B*) paa;
...
}.
```

Атрибути класу, функції-члени класу спадкоємця

Розглянемо наступний приклад:

```
class A{
private:
    int a;
protected:
```

```
    int c;
public:
    void print(){
        cout<<"this is class A";}
        void f(){...}
} ;

class B:public A{
private:
    int b;
public:
    void print(){
        cout<<"this is class B";}
        void f1(){b=10; c=30;
        a=20;                          //error
}
};
int main(){
A* pa = new A;
pa->f();                               //викликається f() із класу A;
pa->print();                           //викликається print із класу A;

B* pb=new B;
pb->f1();                              //викликається f1() із класу B;
pb->print();                           //викликається print() із класу B;

A* pa=new B;
pa->f();                               //викликається f() із класу A;
pa->f1();                              //error, функція f1() відсутня у класі A;
pa->print();                           //викликається print із класу A, а хотілося б print із класу B;
```

```
B ob;
A ab=ob;
ab->print();                    //викликається print із класу A;
...
return 0;}.
```

Конструктори й деструктори

Якщо у похідному класі є конструктор, то в цьому конструкторі повинен викликатися конструктор базового класу. Розглянемо приклад:

```
class A{
private:
    int a;
public:
    A(int s):a(s){};
...
};

class B:public A{
private:
    int b;
public:
    B(int s, int g):A(s),b(g){};
...
}.
```

Рівні доступу: private, public, protected

Для атрибутів класів і функцій-членів класу можливі такі рівні доступу: **public, protected, private**. Розглянемо наступні приклади.

Приклад 1:

```
class A{
private:
    int a;
    void f(){...};
```

```
public:
    int a1;
    void f1(){...}
...};

class B: public A{
private:
    int b;
public:
    void f2(){
    b=10;                    //правильно;
    a=20;                    //error, атрибут a має рівень доступу private;
    f();                     //error, функція-член має рівень доступу private;
    f1();                    //правильно, функція-член має рівень доступу public;
...}
};

int main(){
A* pa=new B;
pa->f1();                    //правильно;
pa->f();                     //error, f() має рівень доступу private;
...
```

Приклад 2:

```
class A{
private:
    int a;
protected:
    int a1;
    void f();
public:
    int a2;
    void f1();
```

```
...}

class B{
private:
    int b;
public:
    void f1(){
    a=10;                          //error;
    a1=20;                         //правильно, a1 має рівень
                                   доступу protected;
    a2=30;                         //правильно;
    f();                           //правильно, f() має рівень
                                   доступу protected;
    f1();                          //правильно;
...}
};

int main(){
A* pa=new A;
pa->a=10;                          //error, a має рівень досту-
                                   пу private;
pa->a1=20;                         //error, a1 має рівень до-
                                   ступу protected;
pa->a2=30;                         //правильно, a2 має рівень
                                   доступу public;
pa->f();                           //error, f() має рівень до-
                                   ступу protected;
pa->f1();                          //правильно;
...}.
```

Віртуальні функції

За замовчуванням функції-члени не є віртуальними. Якщо функція-член віртуальна, то при зверненні до неї викликається функція, визначена в динамічному типі об'єкта класу.

Розглянемо приклад оголошення віртуальної функції-члена класу:

```
class A{
private:
    int a;
public:
    virtual void f();
...};
void A::f(){...}.
```

Для оголошення віртуальної функції використовується ключове слово **virtual**.

Розглянемо приклад використання механізму віртуальних функцій:

```
class A{
private:
    int a;
public:
    void f(){...};
    virtual void f1(){...};
...}
class B: public A{
private:
    int b;
public:
    void f(){...};
    void f1(){...};
...}

int main(){
A* pa=new B;
pa->f();                    //викликається f() із класу A;
pa->f1();                   //викликається f1() із класу B;
B bobj;
A aobj=bobj;
aobj.f();                   //викликається f() із класу A;
aobj.f1();                  //викликається f1() із класу A;
...}.
```

Чисто віртуальні функції

Функція, в якій відсутня реалізація, називається чисто віртуальною функцією, а клас, що містить чисто віртуальні функції, називається абстрактним. Головна особливість абстрактних класів — неможливість створення об'єктів таких класів.

Розглянемо приклад:

```
class A{
private:
    int a;
public:
    virtual void f()=0;          //"=0" означає, що дана
                                 функція не має реалізації;
    virtual void f1(){...};
};
class B: public A{
private:
    int b;
public:
    void f(){...}
    void f1(){...}
...}

int main(){
A* pa=new A;                     //error, не можна створю-
                                 вати об'єкти абстрактних
                                 класів;
A* pa=new B;                     //правильно;
A pa;//error
pa->f();                         //викликається f() із класу B;
...
```

Статичні функції-члени класу не можуть бути віртуальними. Наприклад:

```
class A{
public:
    virtual static void f();     //error;
};
```

Статичний виклик віртуальної функції

Для скасування механізму віртуалізації можна використовувати статичний виклик віртуальної функції.

Наприклад:

```
class A{
private:
    int a;
public:
    virtual void f(){a=10;}
...

class B: public A{
private:
    int b;
public:
    void f(){b=30;}
...}
int main(){
A* pa=new B;
pa->f();                    //викликається f() із класу B;
pa->A::f();                 //викликається f() із класу
                            A статичний виклик віртуальної функції;
...
```

Віртуальні функції й аргументи за замовчуванням.

Розглянемо наступний приклад:

```
class Base{
public:
    virtual void print(int a=1000){
        cout<<a<<endl;}
};
class Derived: public Base{
public:
    virtual void print(int a=2000){
        cout<<a<<endl;}
```

```
};

int main(){
Derived* d=new Derived;
Base* b=d;
d-d->print();                    //на екрані 2000;
b-b->print();                    //на екрані 1000;
...
```

Таким чином, параметри, що передані у функцію, визначаються не під час виконання, а під час компіляції.

Доступ до членів базового класу

Всередині похідного класу до членів базового класу можна звертатися безпосередньо.

Приклад 1:

```
class A{
private:
    int a;
public:
    void f(){...}
}
class B: public A{
private:
    int b;
public:
    void g(){
    f();                         //функція базового класу;
...
}
```

Приклад 2:

```
class A{
private:
    int a;
public:
```

```
    void f();};
class B: public A{
private:
    int a;
public:
    void f();
    void g(){
...
    f();                    //вживання імені f() дозволяється на користь f() із класу B;
    a=100;                  //вживання імені a дозволяється на користь a із класу B;
};
}.
```

Варто зазначити, що у прикладі 2 функції *f()* з *A* и *f()* з *B* не є перевантаженими.

Приклад 3:

```
class A{
private:
    int a;
public:
    void f();};
class B: public A{
private:
    int a;
public:
    void f();
    void g(){
...
    A::f();                 //вживання імені f() дозволяється на користь f() із класу A;
    A::a=100;               //вживання імені a дозволяється на користь a із класу A;
```

```
};
};
```

Якщо похідний клас хоче отримати доступ до закритих членів базового класу, то він повинен бути оголошений другом базового класу:

```
class A{
    friend class B;
    private:
        int a;
...};
class B: public A{
public:
    void f(){
    a=10;                          //правильно, клас B друг
                                   класу A;
};
}.
```

Віртуальні функції у похідному класі

Оголошувати функцію віртуальною у похідному класі потрібно в тому випадку, якщо передбачається, що похідний клас може мати спадкоємців, в яких ця функція теж буде перевизначена.

Розглянемо приклад:

```
class A{
private:
    int a;
public:
    virtual void f();
};
class B:public A{
private:
    int a;
public:
    virtual void f();              //передбачається, що дана
                                   функція може бути перевиз-
```

```
                                    значена в класах спадкоєм-
                                    цях, класу B;
};
```
Розглянемо інший приклад:
```
class A{
private:
    int a;
public:
    virtual void f();
};
class B:public A{
public:
    void f();                       //фактично заборона на
                                    перевизначення функції f()
                                    у класах спадкоємцях B;
};
```

Віртуальні функції вводу/виводу

Розглянемо реалізацію виводу у класі-спадкоємці:
```
class A{
private:
    int a;
public:
    virtual ostream& print(ostream& os) const { os<<a<<endl;
    return os;}
};

class B: public A{
private:
    int b;
    public:
    virtual ostream& print(osteram& os) const{
        A::print(os);
        os<<b<<endl;
        return os;}
};
```

```
inline ostream& operator<<(ostream& os, const A& obj){
    return obj.print(os);};
int main(){
B bobj;
cout<<bobj<<endl;
return 0;}.
```

Віртуальні деструктори

Розглянемо наступний приклад:

```
class A{
...
    ~A(){...}
};
class B:public A{
...
    ~B(){...}
};
int main(){
A* pa=new B;
...
delete pa;                    //викликається деструктор ~A(), а потрібно, щоб ~B();
...}.
```

Вихід — використання віртуальних деструкторів.

```
class A{
...
    virtual ~A(){...}
};
class B:public A{
...
    ~B(){...}
};
int main(){
A* pa=new B;
...
```

```
delete pa;                    //викликається деструктор ~B();
...};
```

Майже віртуальний оператор new

Розглянемо створення копії об'єкта в купі.

```
class A{
private:
...
public:
    A(const A&);
};
int main(){
A* pa =new A;
//створимо копію об'єкта
A* pa1=new A(pa);
...};
```

Розглянемо більш складну задачу.

```
class A{
public:
    virtual A* clone()=0;
...};
class B: public A{
private:
...
public:
    B(const B& obj){...};
    virtual*clone(){
    return new(*this);}
};
int main(){
A*pa=new;
A*pa1=pa->clone();
B*pb=new;
B*pb1=static_cast<B*>(pa->clone());
...};
```

Відкрите, закрите і захищене спадкування

1. Відкрите спадкування.

У цьому випадку синтаксис має вигляд:

class B:public A{…}.

Цей тип спадкування був докладно розглянутий у попередній лекції.

2. Закрите спадкування.

Синтаксис у цьому випадку має вигляд:

class B:private A{…}.

Розглянемо приклад:

```
class A{
private:
    int a;
public:
    void f(){a=10;}
};

class B:private A{
private:
    int b;
public:
    void g() { f(); }
};

int main(){
A* pa=new B;                    //error поліморфізм у цьому випадку не працює;
B* pb=new B;
pb->f();                        //error, спадкування закрите;
pb->g();                        //правильно;
return 0;
}.
```

Якщо необхідно відкрити доступ до деяких членів базового класу, то можна використовувати директиву **using**.

Наприклад:

```
class A{
private:
    int a;
public:
    void f(){a=10;} };

class B:private A{
private:
    int b;
public:
    using A::f;
    void g() { f(); }
};

int main(){
B* pb=new B;
pb->f();                          // правильно;
return 0;
}.
```

У такий спосіб у випадку закритого спадкування всі члени базового класу з модифікаторами **public** і **protected** стають **private** членами у похідному класі.

3. Захищене спадкування.

Синтаксис у цьому випадку має вигляд:

class B:protected A{…}.

У випадку захищеного спадкування відкриті члени базового класу стають захищеними у похідному класі.

Розглянемо приклад:

```
class A{
private:
    int a;
public:
    void f(){a=10;}
};
```

```
class B:protected A{
private:
    int b;
public:
    void g() { f(); }
};

int main(){
A* pa=new B;                    //error поліморфізм у цьому
                                випадку не працює;

B* pb=new B;
pb->f();                        //правильно, спадкування
                                захищене;
pb->g();                        //правильно;
return 0.
}.
```

Лекція 17. МНОЖИННЕ СПАДКУВАННЯ

Синтаксис множинного спадкування має вигляд:
class A: public B, public C, private D, protected G{…}.
Однак, наступний запис є хибним:
class A: public B, C, private D, protected G{…} //error.
Для кожного з перелічених базових класів повинен бути зазначений рівень доступу ***public, private, protected***.
Розглянемо приклад множинного спадкування.

```
class A{
private:
    int a;
public:
    void f();
    virtual void g();
};
class B{
private:
    int b;
public:
    void h();
    virtual void k();
};

class C:public A,public B{
private:
    int c;
public:
    void s();
    void g();
    void k();};
int main(){
A* pa=new C;
pa->f();                    //викликається f() із класу A;
pa->g();                    //correct;
pa->k();                    //error;
```

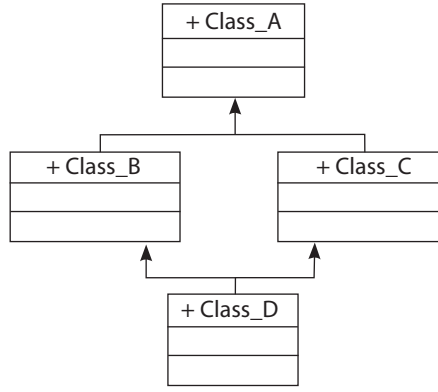
pa->h(); *//error;*
Аналогічно
B pb=new C.*

Конструктори й множинне спадкування

Розглянемо наступний приклад:

```
class A{
...
public:
    A(int c);
};
class B{
...
public:
    B(int h);
};
class C:public A, public B{
private:
    int c;
public:
    C(int s, int k, int l):A(s),B(k),c(l){...}
}.
```

Розглянемо наступний приклад:



Відповідний код має вигляд:

```
class A{
private:
    int a;
public:
    void f();}
class B:public A{
private:
    int b;
public:
    void g();
}

class C:public A{
private:
    int c;
public:
    void h();
}
class D:public B,public C{
private:
    int d;
public:
    void k();}

int main(){
D* pd=new D;
pd->f();                    //error, незрозуміло, яка f()
                            повинна бути викликана;
pd->h();                    //правильно;
pd->g();                    //правильно;
pd->k();                    //правильно;
...}.
```

Однак, наступний код до помилки не приводить

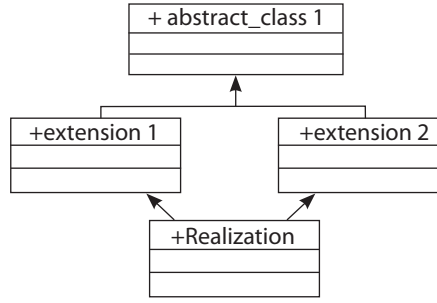
```
class A{
```

```
private:
    int a;
public:
    virtual void f();}
class C:public A{...};
class B:public A{...};
class D:public B,public C{
...
public:
    void f();
};
int main(){
D* pd=new D;
pd->f();                          //правильно;
...}.
```

Коли потрібно використовувати множинне спадкування

Виправдано використовувати множинне спадкування для **розширення інтерфейсу**.

Приклад:

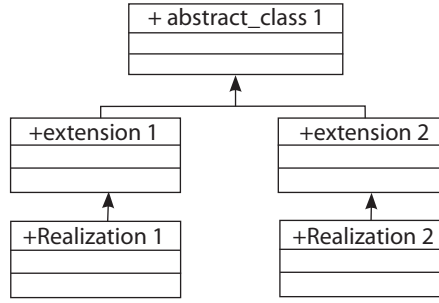


Код відповідний наведеній діаграмі класів, має вигляд:

```
class abstract_class1{
public:
```

```
    virtual void func1()=0;
    virtual int func2(int,int)=0;
};
class extension1:public abstract_class1{
public:
    virtual void func3()=0;
};
class extension2:public abstract_class1{
public:
    virtual void func4()=0;
};
class realization: public extension1,public extension2{
private:
...
public:
    virtual void func1(){...};
    virtual int func2(int a,int b){...};
    virtual void func3(){...};
    virtual void func4(){...};
}.
```

Однак більш оптимальним буде інше рішення, яке відображає два можливих шляхи розвитку проекту:



Область видимості класу при множинному спадкуванні

Розглянемо наступний приклад:

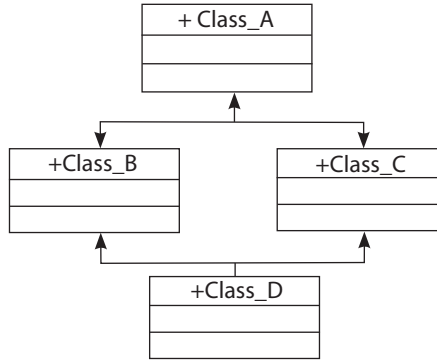
```
class A{
public:
    void f(){...};};
class B{
private:
    bool f(){return true;}
public:
    void g(){...};};
class C:public B{...};
class D:public C,public A{
public:
    void s(){f();}          //error, неоднозначність, в області видимості класу D є дві функції з ім'ям f;
};

int main(){
D d;
d.f();                      //error, неоднозначність в області видимості класу D є дві функції з однаковим ім'ям.
```

Дозволити неоднозначність можна у такий спосіб:

```
d.A::f();
return 0;
}.
```

Лекція 18. ВІРТУАЛЬНЕ СПАДКУВАННЯ

Розглянемо інший варіант рішення проблеми з неоднозначним викликом функцій класу *A* в ситуації, описаній вище:



Для рішення проблеми використовуємо віртуальне спадкування. У випадку віртуального спадкування успадковується тільки один підоб'єкт базового класу, незалежно від того, скільки разів базовий клас зустрічається в ієрархії. Синтаксис у випадку віртуального спадкування має вигляд:

class B: public virtual A{…};

або

class B: virtual public A{…}.

Наприклад:

```
class A{
private:
    int a;
public:
    void f(){a=10;}
};
class B:virtual public A{
private:
    int b;
public:
    void g(){b=20;}
};
```

```
class C:virtual public A{
private:
    int c;
public:
    void h(){c=400;}
};
class D:public B,public C{...};
int main(){
D* pd=new D;
pd->f();                          // правильно
pd-> g();                         // правильно
pd->h();                          // правильно
...}.
```

Семантика ініціалізації

Розглянемо наступний приклад:

```
class A{
private:
    int a;
public:
    A(int s):a(s){};
...;
class B:virtual public A{
private:
    int b;
public:
    B(int s,int s1):A(s),b(s1){};
...};

class C:virtual public A{
private:
    int c;
public:
    C(int s, int s1):A(s),c(s1){};
...};
```

```
class D:public B, public C{
private:
        int d;
public:
        D(int s, int s1, int s3):B(s,s1),C(s,s3){};      //error, немає підходящого конструктора для A;
...;
int main(){
D obj;
...}.
```

У випадку невіртуального спадкування похідний клас здатний явно ініціалізувати тільки свої безпосередні базові класи. Так, класу *D*, що успадковується від *A* (у випадку невіртуального спадкування), не дозволяється прямо викликати конструктор класу *A*. Однак у випадку віртуального спадкування клас *D* може явно викликати конструктор *A*. Тобто наступний запис є коректним:

```
class D:public B, public C{
private:
    int d;
public:
D(int s, int s1, int s3):A(s),B(s1,s1),C(s3,s3){};      // правильно.
```

У цьому випадку виклики конструктора класу *A* у класах *B* та *C* блокуються. Таким чином, параметр *a* у класі *A* буде проініціалізовано значенням *s*.

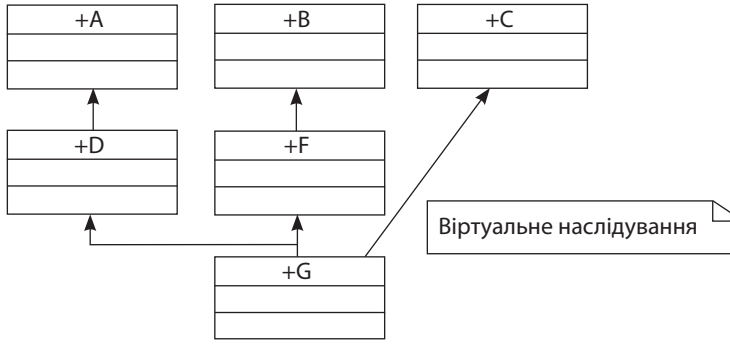
Якщо класи *B* і *C* виступають у ролі проміжних класів, то їх конструктори можна зробити захищеними, тобто

```
class B:virtual public A{
private:
    int b;
protected:
    B(int s,int s1):A(s),b(s1){};
...};
```

Аналогічно й для класу *C*.

Порядок виклику конструкторів і деструкторів при віртуальному спадкуванні

Віртуальні базові класи завжди конструюються перед невіртуальними, незалежно від їх розташування в ієрархії. Розглянемо приклад:



Код для цієї діаграми буде мати вигляд:

```
class A{
public:
    A(){ cout<<“A”<<endl;  }};

class B{
public:
    B(){cout<<“B”<<endl;}};

class C{
public:
    C(){cout<<“C”<<endl;}};

class D:public A{
public:
D(){cout<<“D”<<endl;}};

class F:public B{
public:
```

```
    F(){cout<<“F”<<endl;}};
class G:public D, public F,virtual public C{
public:
    G(){cout<<“G”<<endl;}};
int main(){
G obj;
return 0;
}
```

Порядок виклику конструкторів буде наступним:

C(); A(); D(); B(); F(); G().

Відповідно деструктори будуть викликані у зворотному порядку.

Видимість членів віртуального базового класу

Розглянемо кілька прикладів.

Приклад 1:

```
class A{
private:
    int a;
public:
    void f(){a=10;}
};

class B{
private:
    int b;
public:
    void f(){b=20;}
};

class C:public A,virtual public B{
...
public:
    void f(){
```

```
    void g(){f();}                    // правильно, викликається
                                      f() із класу C;
};
```

Приклад 2:

```
class A{
public:
    void f(){cout<<“f”<<endl;}};

class B:virtual public A{
public:
    void f(){cout<<“f”<<endl;}};

class C:public B,virtual public A{
public:
    void g(){f();}};                  // правильно, викликається
                                      f() із класу B;

int main(){
C obj;
obj.g();
return 0;
};
```

Лекція 19. ВИКОРИСТАННЯ СПАДКУВАННЯ В С++

Оператор dynamic_cast

Оператор **dynamic_cast** можна використовувати для перетворення покажчика на об'єкт типу класу в покажчик на тип класу з тієї самої ієрархії. Приведення типів за допомогою **dynamic_cast** здійснюється під час виконання програми. У випадку неможливості перетворення **dynamic_cast** завершується невдало й повертає значення 0.

Наприклад:

```
class A{
private:
    int a;
public:
    void func(){a=10;}};
class B:public A{
public:
    void func(){...};
};
int main(){
A* pa=new B;
B* pb=dynamic_cast<B*>(pa);
if(pb)
pb->func();                    // правильно;
 else{                         // перетворення закінчилося невдачею;
...
}.
```

При використанні посилань перевірити успішність перетворення не можна, тому що нульових посилань не буває.

```
int main(){
A obj;
A& obj1=obj;
B& pb=dynamic_cast<B&>(obj1);
```

```
if(pb){...}                    // у такий спосіб перевірити
                               //успішність перетворення
                               неможливо.
...}.
```

Оператор typeid

Оператор **typeid** дозволяє з'ясувати фактичний тип виразу. Якщо вираз належить типу класу й цей клас містить хоча б одну віртуальну функцію-член, то тип результату може не збігатися з типом самого виразу.

Розглянемо приклади.

Приклад 1:

```
#include<iostream>
#include<typeinfo>
using namespace std;
class A{
private:
    int a;
public:
    void func(){a=10;}
};

class B:public A{
public:
    void func(){...};};

int main(){
A* pa=new B;
cout<<typeid(pa).name()<<endl;  //на екрані class A *
return 0; }
```

Приклад 2:

```
class A{
private:
    int a;
public:
```

```
    void func(){a=10;}};

class B:public A{
public:
    void func(){...};};

int main(){
B bobj;
A& re=bobj;
cout<<typeid(re).name()<<endl;
return 0; }.
```

Результати повернені оператором **typeid** можна порівнювати.

Приклад 3:

```
class A{
private:
    int a;
public:
    virual void func(){a=10;}};

class B:public A{
public:
    void func(){...};
    void func(){...};};

int main(){
A* pa=new B;
A& ps=*pa;
if(typeid(pa)==typeid(A*))        //true;
if(typeid(pa)==typeid(B*))        //false;
if(typeid(pa)==typeid(A))         //false;
if(typeid(pa)==typeid(B))         //false.
```

У результаті порівняння

if(typeid(pa)==typeid(A))*

отримуємо *true*, тому що поведінка *typeid(pa)* не підкоряється механізму віртуалізації.

При вживанні виразу **pa* в операторі *typeid* результат буде містити тип об'єкта, на який вказує *pa*:

```
typeid(*pa)==typeid(B)          //true
typeid(*pa)==typeid(A)          //false.
```

Оператор *typeid* можна використовувати з посиланнями:

```
typeid(ps)==typeid(B)           //true
typeid(ps)==typeid(A)           //false
typeid(&ps)==typeid(A*)         //true
typeid(&ps)==typeid(B*)         //false.
```

Клас type_info

Точне визначення класу **type_info** залежить від реалізації, але деякі риси залишаються незмінними. Приблизно клас **type_info** має вигляд:

```
class type_info{
private:
    type_info(const type_info&);
    type_info& operator=(const type_info&);
public:
    virtual ~type_info();
    int operator==(const type_info&);
    int operator!=(const type_info&);
    const char* name() const;}
```

Отже, конструктор і оператор присвоювання є закритими членами класу **type_info**, тобто користувач не може створювати об'єкти класу **type_info**. Єдиний спосіб створити об'єкт класу **type_info** — скористатися оператором **typeid**. Клас **type_info** може бути розширений шляхом оголошення класу спадкоємця від **type_info**.

Виключення й спадкування

Раніше для кожного виключення створювався клас. Однак можна визначати ієрархію класів, що описує виключення.

Розглянемо приклад:

```
class Exception{...};
class popOnEmpty:public Exception{...};
class pushOnFull:public Exception{...};
```

Таким чином, *Exception* базовий клас виключень, а класи *popOnEmpty* і *pushOnFull* є уточненням *Exception*.

Обробка виключення типу класу

Основна особливість обробки таких виключень полягає в тому, що оброблювач **(catch-блок)** для виключення *Exception* повинен стояти після оброблювачів для виключень *popOnEmpty* і *pushOnFull*.

Наприклад:

```
class Stack{
...
public:
    push(int value) throw (pushOnFull){
        if(full()) throw pushOnFull();
...}.
};

int main(){
try{
Stack s;
s.push(10);
...
};
catch(pushOnFull e){...};
catch(popOnEmpty e){...};
catch(Exception e){...};          //оброблювач Exception роз-
                                  міщується останнім;
...
}.
```

У класах виключеннях можна описувати віртуальні функції. Робота з ними здійснюється як звичайно.

Розкручування стека й виклик деструкторів

При генерації виключення відбувається розкручування стека — пошук підходящого **catch-оброблювача**. При цьому розкручування стека починається з функції, яка згенерувала виключення й рухається нагору по ланцюжку вкладених викликів. Під час розкручування стека відбувається аномальні виходи з переглянутих функцій. Якщо функція захопила деякий ресурс, то він у такому випадку не звільняється. Розглянемо приклад, що дозволяє вирішити цю проблему:

```
class PTR{
public:
    PTR(){ptr=new int [size];}
    ~PTR(){delete [] ptr;}
private:
    int* ptr;
};
void f(int param){
    PTR localptr;
...
    mathFunc(param);
...
}.
```

Перед виходом з функції *f* з необробленим виключенням процес розкручування стека знищить всі об'єкти типу класів. Тобто дії по захопленню й звільненню ресурсів відбуваються у конструкторі й деструкторі відповідно.

Специфікації виключень

Специфікацію виключень можна робити для функцій-членів класів:

```
class A{
public:
    virtual void f() const throw();
...;
    void A::f() const throw(){...
```

Специфікації в оголошенні функції та її реалізацій повинні збігатися.

Якщо функція перевизначена в похідному класі, то специфікація повинна накладати обмеження не менші, ніж у базовому, тобто

```
class A{
public:
    virtual void f() throw(exp1,exp2);
...;
    class B:public A{
public:
    void f() throw(exp1);          // правильно;
...
```

Функція може генерувати виключення у вигляді об'єкта класу, що відкрито успадковує базовий:

```
class popOnEmpty:public Exception{...};
void f() throw (Exception).
```

Функція *f()* може генерувати виключення *popOnEmpty.*

Конструктори й функціональні try блоки

Розглянемо приклад:

```
class Test{
private:
        int _a;
public:
        Test(int a):_a( a-f()){...};
}.
```

Припустимо, що *f()* може генерувати виключення. Тоді **try** блок буде мати вигляд:

```
Test(int a)
try
:_a( a-f()){...}
catch(...){...}.
```

Ієрархія класів виключень у стандартній бібліотеці C++

Базовий клас у стандартній бібліотеці називається **exception** (визначений в **exception.h**):

```
namespace std{
class exception{
public:
        exception() throw();
        exception(const exception&)throw();
        exceptionj& operator=(const exception&) throw();
        virtual ~exception() throw();
        virtual char* what() const throw();
};
}.
```

Існують й інші класи для повідомлення про помилки, спадкоємці **exception**. Усі помилки можна розділити на 2 категорії: логічні й помилки під час виконання.

Для логічних помилок є наступні класи:

```
class logic_error: public exception{...};
class invalid_argument: public exception{...};
class out_of range: public exception{...};
class length_error: public exception{...};
class domain_error: public exception{...}.
```

Для помилок під час виконання маємо:

```
class runtime_error: public exception{...};
class range_error: public exception{...};
class overflow_error: public exception{...};
class underflow_error: public exception{...}.
```

Для використання даних класів потрібно підключити бібліотеку **stdexcept.h.**

Дозвіл перевантаження й спадкування

Очевидно, що спадкування впливає на дозвіл перевантаження.

Приклад 1:

```
namespace NS{
class A{...};}
class B:public NS::A{...};
int main(){
```

```
B b1;
display(b1);
return 0;}.
```

Кандидатами для виклику **display** будуть не тільки ті функції, що видно в місці виклику, а й ті, що оголошені у просторі імен, в яких оголошені базовий клас *A* і його спадкоємець *B*, тому що аргумент функції **display** в місці виклику має тип *B*.

Таким чином, якщо при виклику звичайної функції заданий аргумент, що є об'єктом класу, посилання або покажчик на об'єкт класу, то множина функцій кандидатів буде:

1. Функції, що видно в місці виклику.
2. Функції, оголошені в тих просторах імен, де визначений тип класу або його базові класи.
3. Функції, що є друзями цього класу або його базових класів.

Приклад 2:

```
class A{
public:
    int f(string);
...};
class B:public A{
public:
    int f(int);
  ...};
int main(){
B b1;
b1.f("Hello");                    // error.
```

У цьому випадку функція *int f(int)* затінює функцію *int f(string)*.

Виправити ситуацію можна у такий спосіб:

```
class B:public A{
...
using A::f;
int f(int);
...};
int main(){
```

```
B b1;
b1.f("Hello");                    // правильно, перевантаження дозволяється на користь int f(string);
...}.
```

Розглянемо вибір найбільш підходящої функції. Спадкування, мабуть, впливає на вибір найбільш підходящої функції. У класах можуть бути перетворення, визначені користувачем. Перетворення, визначені користувачем, бувають 2 видів: **конвертер** і **конструктор** (без ключового слова **explicit**).

Конвертери можуть успадковуватися.

Приклад 3:

```
class A{
public:
    operator char*();
...;

class B:public A{
public:
    void display(const A&);
...;
    extern void display(const char*);
int main(){
B b1;
display(b1);                      // неоднозначність;
...}.
```

Таким чином, при перевантаженні варто враховувати, що наступні неявні перетворення мають той самий ранг, що й стандартні перетворення:

1. Перетворення аргументу типу похідного класу в тип кожного з базових класів.

2. Перетворення покажчика на тип похідного класу в покажчик на тип базового.

3. Ініціалізація посилання на тип базового класу за допомогою **lvalue** типу похідного класу.

Приклад 4:

```
class A{
public:
    operator char*();
...;
class B:public A{...};
    extern void f(const A&);
extern void f(const char*);

int main(){
B b1;
f(b1);                          // викликається void f(const
                                A&), тому що стандартне
                                перетворення краще визна-
                                ченого користувачем;
...}.
```

Приклад 5:

```
class A{...};
class B:public A{...};
class C:public B{...};
extern void f(const A&);
extern void f(const B&);
int main(){
C c1;
f(c1);                          // викликається f(const B&),
                                тому що приведення до кла-
                                су B краще чим до A;
...}.
```

МОВА МОДЕЛЮВАННЯ UML

Лекція 20. ВВЕДЕННЯ У ПРОЦЕС МОДЕЛЮВАННЯ

Центральним елементом діяльності, що веде до створення першокласного програмного забезпечення, є моделювання. Моделі дозволяють нам наочно продемонструвати бажану структуру і поведінку системи. Вони також необхідні для візуалізації й управління її архітектурою. Моделі допомагають досягти кращого розуміння створюваної нами системи, що часто призводить до її спрощення та можливості повторного використання. Нарешті, моделі потрібні для мінімізації ризику.

Моделювання дає змогу вирішити чотири різні задачі:

- візуалізувати систему в її поточному або бажаному для нас стані;
- визначити структуру або поведінку системи;
- отримати шаблон, що дозволяє потім сконструювати систему;
- документувати прийняті рішення, використовуючи отримані моделі.

Моделювати складну систему необхідно, оскільки в іншому випадку ми не можемо сприйняти її як єдине ціле.

Принципи моделювання

Виділяють 4 основні принципи моделювання.

Перший принцип: вибір моделі впливає на підхід до розв'язання проблеми і на те, як буде виглядати цей розв'язок. Правильно вибрана модель висвітить проблеми розробки та дасть змогу проникнути у саму суть завдання, що при іншому підході було б неможливо. Неправильна модель заведе у глухий кут, оскільки увага буде загострюватися на несуттєвих питаннях.

Другий принцип: кожна модель може бути втілена з різним ступенем абстракції.

Найкращою моделлю буде та, яка дає можливість вибрати рівень деталізації залежно від того, хто і з якою метою на неї

дивиться. Для аналітика або кінцевого користувача найбільший інтерес представляє питання "що", а для розробника — "як". В обох випадках необхідно розглядати систему на різних рівнях деталізації у різний час.

Третій принцип: кращі моделі — ті, що ближче до реальності.

Найкращий варіант, якщо моделі будуть в усьому співвідноситися з реальністю, а там, де зв'язок слабшає, повинно бути зрозуміло, у чому полягає відмінність і що з цього випливає. Оскільки модель завжди спрощує реальність, завдання у тому, щоб це спрощення не спричинило будь-які суттєві втрати.

Якщо говорити про програмне забезпечення, то можна сказати, що проблема структурного аналізу — це невідповідність прийнятої в ньому моделі та моделі системного проекту. Якщо такий розрив не буде усунено, то поведінка створеної системи з часом почне все більше відрізнятися від задуманого. При об'єктно-орієнтованому підході можна об'єднати всі незалежні представлення системи в єдине семантичне ціле.

Четвертий принцип полягає в тому, що не можна обмежуватися створенням тільки однієї моделі. Найкращий підхід при розробці будь-якої нетривіальної системи — використовувати сукупність кількох незалежних моделей.

Для розуміння архітектури об'єктно-орієнтованої системи потрібно кілька взаємодоповнюючих видів:

- з точки зору прецедентів, або варіантів використання (щоб виявити вимоги до системи);
- з точки зору проектування (щоб побудувати словник предметної області та області розв'язків);
- з точки зору процесів (щоб змоделювати розподіл процесів та потоків у системі);
- з точки зору реалізації, що дає змогу розглянути фізичну реалізацію системи;
- з точки зору розгортання, що допомагає зосередитися на питаннях системного проектування.

Кожен із зазначених видів має велику кількість структурних та поведінкових аспектів, які у своїй сукупності становлять детальну модель програмної системи.

Залежно від природи системи деякі моделі можуть бути важливіше інших. Так, при створенні систем для обробки великих обсягів даних більш важливі моделі, з точки зору статичного проектування. У додатках, орієнтованих на інтерактивну роботу користувача, на перший план виходить представлення з точки зору статичних та динамічних прецедентів. У системах реального часу найбільш істотними будуть уявлення з точки зору динамічних процесів. Нарешті, у розподілених системах, таких як Web-додатки, основну увагу потрібно приділяти моделям реалізації та розгортання.

Об'єктне моделювання

При розробці програмного забезпечення існує кілька підходів до моделювання. Найважливіші з них — алгоритмічний і об'єктно-орієнтований.

Алгоритмічний метод є традиційним підходом до створення програмного забезпечення. Основним будівельним блоком є процедура або функція, а увага приділяється, насамперед, питанню передачі управління і декомпозиції великих алгоритмів на менші. Нічого поганого у цьому немає, якщо не брати до уваги, що системи не дуже легко адаптуються. При зміні вимог або збільшенні розміру програми (що відбувається нерідко) супроводжувати їх стає складніше.

Найбільш сучасним підходом до розробки програмного забезпечення є **об'єктно-орієнтований**. Тут в якості основного будівельного блоку виступає об'єкт або клас. У самому загальному сенсі об'єкт — це сутність зі словника предметної області або розв'язку, а клас є описом множини однотипних об'єктів. Кожен об'єкт має ідентичність (його можна поіменувати або якось інакше відрізнити від інших об'єктів), стан (зазвичай з об'єктом бувають пов'язані деякі дані) і поведінку (з ним можна щось робити чи він сам може щось робити з іншими об'єктами).

Розглянемо простий приклад трирівневої архітектури білінгової системи, що складається з інтерфейсу користувача, програмного забезпечення проміжного шару і бази даних. Інтерфейс містить конкретні об'єкти — кнопки, меню і діалогові вікна. База даних також складається з конкретних об'єктів (таблиць), що представля-

ють сутності предметної області: клієнтів, продуктів і замовлень. Програми проміжного шару включають такі об'єкти, як транзакції і бізнес-правила, а також більш абстрактні представлення сутностей предметної області (клієнтів, продуктів і замовлень).

Об'єктно-орієнтований підхід до розробки програмного забезпечення є переважаючим тому, що він продемонстрував свою корисність при побудові систем у самих різних областях будь-якого розміру та складності. Крім того, більшість сучасних мов програмування, інструментальних засобів і операційних систем є в тій чи іншій мірі об'єктно-орієнтованими, а це дає вагомі підстави судити про світ у термінах об'єктів. Об'єктно-орієнтовані методи розробки лягли в основу ідеології побудови систем з окремих компонентів. Прикладом могуть слугувати такі технології, як JavaBeans і COM +.

Введення в мову UML

UML — це мова для візуалізації, специфікації, конструювання та документування артефактів програмних систем.

Використання UML дає змогу вирішити проблему: явна модель полегшує спілкування. Деякі особливості системи найкраще моделювати у вигляді тексту, інші — графічно. Насправді в усіх цікавих системах існують структури, які неможливо представити за допомогою лише тільки мови програмування. UML — це графічна мова.

UML — це не просто набір графічних символів. За кожним із них стоїть добре визначена семантика. Це означає, що модель, написана одним розробником, може бути однозначно інтерпретована іншим, або навіть інструментальною програмою.

UML не є мовою візуального програмування, але моделі, створені за її допомогою, можуть бути безпосередньо переведені на різні мови програмування. Інакше кажучи, UML-модель можна відобразити на такі мови, як Java, C ++, Visual Basic, на таблиці реляційної бази даних або об'єкти об'єктно-орієнтованої бази даних. Ті поняття, які краще передавати графічно, представляються в UML; ті ж, які краще описувати у текстовому вигляді, виражаються за допомогою мови програмування.

UML — це мова документування

Компанія, що випускає програмні засоби, крім виконуваного коду виробляє також інші артефакти, і серед них такі:

- вимоги до системи;
- архітектуру;
- проект;
- вихідний код;
- проектні плани;
- тести;
- прототипи;
- версії та ін.

Залежно від прийнятої методики розробки одне виконання робіт проводиться більш формально ніж інше. Згадані артефакти — це не просто складові проекту; вони необхідні для управління, для оцінки результату, а також як засіб спілкування між членами колективу під час розробки системи і після її розгортання.

UML дає можливість вирішити проблему документування системної архітектури і всіх її частин, пропонує мову для формулювання вимог до системи і визначення тестів і, нарешті, надає засоби для моделювання робіт на етапі планування проекту та управління версіями.

Концептуальна модель UML

Основа розуміння UML — це його концептуальна модель, яка включає в себе три складові: основні будівельні блоки мови, правила їх поєднання і деякі загальні для всієї мови механізми.

Будівельні блоки UML

Словник мови UML включає три види будівельних блоків:

- сутності;
- відносини;
- діаграми.

Сутності — це абстракції, які є основними елементами моделі. Відносини пов'язують різні сутності; діаграми групують сукупності сутностей, що представляють інтерес.

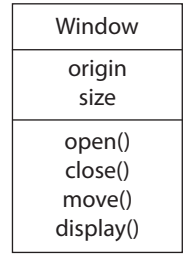
В UML є чотири типи сутностей:

- структурні;
- поведінкові;
- що групуються;
- анотаційні.

Сутності є основними об'єктно-орієнтованими блоками мови. З їх допомогою можна створювати коректні моделі.

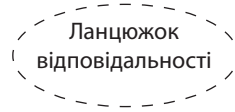
Структурні сутності — це іменники в моделях на мові UML. Як правило, вони є статичними частинами моделі, що відповідають концептуальним або фізичним елементам системи. Існує сім різновидів структурних сутностей.

Клас (Class) — це опис сукупності об'єктів із загальними атрибутами, операціями, відносинами і семантикою. Клас реалізує один або кілька інтерфейсів. Графічно клас зображується у вигляді прямокутника, в якому, зазвичай, записані його ім'я, атрибути та операції (див. рисунок).

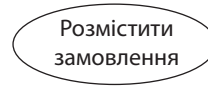


Інтерфейс (Interface) — це сукупність операцій, які визначають сервіс (набір послуг), що надається класом або компонентом. Таким чином, інтерфейс описує доступну ззовні поведінку елемента. Інтерфейс може представляти поведінку класу чи компонента повністю або частково; він визначає тільки специфікації операцій (сигнатури), але ніколи — їх реалізації. Графічно інтерфейс зображується у вигляді кола, під яким пишеться його ім'я. Інтерфейс рідко існує сам по собі, зазвичай він приєднується до класу або компоненти, що його реалізує.

Кооперація (Collaboration) визначає взаємодію; вона є сукупністю ролей та інших елементів, які працюючи спільно, роблять деякий кооперативний ефект, що не зводиться до простої суми доданків. Кооперація має як структурний, так і поведінковий аспекти. Один і той самий клас може брати участь у кількох кооперaціях, таким чином, вони є реалізацією зразків поведінки, що формують систему. Графічно кооперація зображується у вигляді еліпса, обмеженого пунктирною лінією, в якому вказано тільки ім'я.



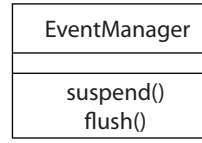
Прецедент (Use case) — це опис послідовності виконуваних системою дій, яка виробляє результат, значущий для якогось певного **актора (Actor)**. Прецедент застосовується для структурування поведінкових сутностей моделі. Прецеденти реалізуються за допомогою кооперації. Графічно прецедент зображується у вигляді обмеженого безперервною лінією еліпса, в якому вказано тільки ім'я.



Три інші сутності — активні класи, компоненти та вузли — подібні до класів: вони описують сукупності об'єктів із загальними атрибутами, операціями, відносинами і семантикою. Однак вони в достатній мірі відрізняються один від одного і від класів і, з огляду на їх важливість при моделюванні певних аспектів об'єктно-орієнтованих систем, заслуговують спеціального розгляду.

Активним класом (Active class) називається клас, об'єкти якого залучені в один або кілька процесів, нитки (Threads), і тому можуть ініціювати керуючий вплив. Активний клас в усьому подібний до звичайного класу за винятком того, що його об'єкти являють собою елементи, діяльність яких здійснюється

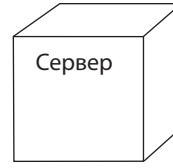
одночасно з діяльністю інших елементів. Графічно активний клас зображується так само, як простий клас, але обмежується прямокутником, що малюється жирною лінією і включає ім'я, атрибути та операції.



Два елементи — компоненти і вузли — також мають свої особливості. Вони відповідають фізичним сутностям системи, тоді як п'ять попередніх — концептуальним і логічним сутностям.

Компонент (Component) — це фізична складова системи, що замінюється, яка відповідає деякому набору інтерфейсів і забезпечує його реалізацію. У системі можна зустріти різні види компонентів, що встановлюються, такі як COM + або JavaBeans, а також компоненти, що є артефактами процесу розробки, наприклад файли вихідного коду. Компонент, як правило, являє собою фізичну упаковку логічних елементів, таких як класи, інтерфейси та кооперації. Графічно компонент зображується у вигляді прямокутника з вкладками, що містить, зазвичай, тільки ім'я.

Вузол (Node) — це елемент реальної (фізичної) системи, який існує під час функціонування програмного комплексу і являє собою обчислювальний ресурс, що володіє як мінімум деяким об'ємом пам'яті, а часто ще й здатністю обробки. Сукупність компонентів може розміщуватися у вузлі, а також мігрувати з одного вузла на інший. Графічно вузол зображується у вигляді куба та містить тільки ім'я.

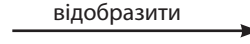


Ці сім базових елементів — класи, інтерфейси, кооперації, прецеденти, активні класи, компоненти та вузли — є основними

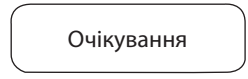
структурними сутностями, які можуть бути включені в модель UML. Існують також різновиди цих сутностей: актори, сигнали, утиліти (види класів), процеси і нитки (види активних класів), додатки, документи, файли, бібліотеки, сторінки і таблиці (види компонентів).

Поведінкові сутності (Behavioral things) є динамічними складовими моделі UML. Це дієслова мови: вони описують поведінку моделі у часі і просторі. Існує всього два основних типи поведінкових сутностей.

Взаємодія (Interaction) — це поведінка, суть якої полягає в обміні повідомленнями (Messages) між об'єктами у рамках конкретного контексту для досягнення певної мети. За допомогою взаємодії можна описати як окрему операцію, так і поведінку сукупності об'єктів. Взаємодія передбачає ряд інших елементів, таких як повідомлення, послідовності дій (поведінка, ініційоване повідомленням) та зв'язку (між об'єктами). Графічно повідомлення зображуються у вигляді стрілки, над якою майже завжди пишеться ім'я відповідної операції.



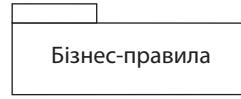
Автомат (State machine) — це алгоритм поведінки, що визначає послідовність станів, через які об'єкт або взаємодія проходить протягом свого життєвого циклу у відповідь на різні події, а також реакції на ці події. За допомогою автомата можна описати поведінку окремого класу або кооперації класів. З автоматом пов'язана низка інших елементів: стан, переходи (з одного стану в інший), події (суть, які ініціюють переходи) і види дій (реакція на перехід). Графічно стан зображується у вигляді прямокутника із закругленими кутами, що містить ім'я і, можливо, підстани.



Ці два елементи — взаємодія і автомат — є основними поведінковими сутностями, що входять у модель UML. Семантично

вони часто бувають пов'язані з різними структурними елементами, насамперед — класами, кооперацією і об'єктами.

Сутності, що групують, є організуючими складовими моделі UML. Це блоки, на які можна розкласти модель. Є тільки одна первинна сутність, що групує, — пакет.



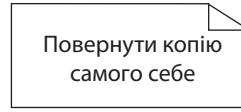
Пакети (Packages) представляють собою універсальний механізм організації елементів у групи. У пакет можна помістити структурні, поведінкові та інші сутності, що групуються. На відміну від компонентів, що існують під час роботи програми, пакети носять суто концептуальний характер, тобто існують тільки під час розробки.

Зображується пакет у вигляді папки із закладкою, що містить, як правило, тільки ім'я й іноді — вміст.

Пакети — це основні сутності, що групуються, за допомогою яких можна організувати модель UML. Існують також варіації пакетів, наприклад каркаси (Frameworks), моделі та підсистеми.

Анотаційні суті — пояснювальні частини моделі UML. Це коментарі для додаткового опису, роз'яснення або зауваження до будь-якого елемента моделі. Є тільки один базовий тип анотаційних елементів — примітка (Note).

Примітка — це просто символ для зображення коментарів або обмежень, приєднаних до елемента або групи елементів. Графічно примітка зображується у вигляді прямокутника із загнутим краєм, що містить текстовий або графічний коментар.



Цей елемент є основною анотаційною сутністю, яку можна включати в модель UML. Найчастіше примітки використовуються, щоб забезпечити діаграми коментарями або обмеженнями, які можна виразити у вигляді неформального або формаль-

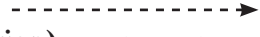
ного тексту. Існують варіації цього елемента, наприклад вимоги, де описують якусь бажану зовнішню поведінку відносно моделі.

У мові UML визначено чотири типи відносин:

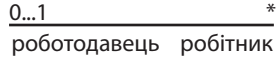
- залежність;
- асоціація;
- узагальнення;
- реалізація.

Ці відношення є основними зв'язуючими будівельними блоками в UML і застосовуються для створення коректних моделей.

Залежність (Dependency) — це семантичне відношення між двома сутностями, при якому зміна однієї з них, незалежної, може вплинути на семантику іншої, залежної. Графічно залежність зображується у вигляді прямої пунктирною лінії, часто зі стрілкою, яка може містити позначку.

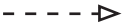


Асоціація (Association) — структурне відношення, що описує сукупність зв'язків; зв'язок — це з'єднання між об'єктами. Різновидом асоціації є агрегація і композиція — так називають структурні відношення між цілим і його частинами. Графічно асоціація зображується у вигляді прямої лінії (іноді завершується стрілкою або містить мітку), поруч з якою можуть бути присутніми додаткові позначення, наприклад, кратність та імена ролей.



Узагальнення (Generalization) — це відношення "спеціалізація/узагальнення", при якому об'єкт спеціалізованого елемента (нащадок) може бути підставлений замість об'єкта узагальненого елемента (одного з батьків). Таким чином, нащадок (Child) успадковує структуру і поведінку свого батька (Parent). Графічно відношення узагальнення зображується у вигляді лінії з незафарбованою стрілкою, що вказує на батька. Нарешті, реалізація (Realization) — це семантичне відношення між класифікаторами, при якому один класифікатор визначає "контракт", а інший гарантує його виконання.

Відносини реалізації зустрічаються у двох випадках: по-перше, між інтерфейсами і реалізуючими їх класами або компонентами, а по-друге, між прецедентами і реалізуючими їх кооперація-ми. Відношення реалізації зображується у вигляді пунктирної лінії з не зафарбованою стрілкою, як щось середнє між відносинами узагальнення і залежності.



Чотири описаних елемента є основними типами відносин, які можна включати до моделі UML. Існують також їх варіації, наприклад, уточнення (Refinement), трасування (Trace), включення та розширення (для залежностей). Діаграма в UML — це графічне представлення набору елементів, що зображується у вигляді зв'язаного графа з вершинами (сутностями) і ребрами (відносинами). Діаграми малюють для візуалізації системи з різних точок зору. Діаграма — у деякому сенсі одна з проекцій системи. Як правило, за винятком найбільш тривіальних випадків, діаграми дають згорнуте подання елементів, з яких складена система. Один і той самий елемент може бути присутнім у всіх діаграмах, або тільки у кількох (найпоширеніший варіант), або не бути присутнім у жодній (дуже рідко). Теоретично діаграми можуть містити будь-які комбінації сутностей і відносин. На практиці, однак, застосовується порівняно невелика кількість типових комбінацій.

Таким чином, в UML виділяють дев'ять типів діаграм:

- діаграми класів;
- діаграми об'єктів;
- діаграми прецедентів;
- діаграми послідовностей;
- діаграми кооперації;
- діаграми станів;
- діаграми дій;
- діаграми компонентів;
- діаграми розгортання.

Лекція 21. ДІАГРАМА КЛАСІВ

Центральне місце у проектуванні займає розробка логічної моделі системи у вигляді діаграми класів. Нотація класів у мові UML проста та інтуїтивно зрозуміла. Нотація UML надає широкі можливості для відображення додаткової інформації (абстрактні операції та класи, стереотипи, загальні та приватні методи, деталізовані інтерфейси, параметризовані класи). При цьому можливе використання графічних зображень для асоціацій та їх специфічних властивостей, таких як відношення агрегації, коли складовими класу можуть виступати інші класи.

Діаграма класів (class diagram) служить для представлення статичної структури моделі системи у термінології класів об'єктно-орієнтованого програмування. Діаграма класів може відображати, зокрема, різні взаємозв'язки між окремими сутностями предметної області, такими як об'єкти і підсистеми, а також описує їх внутрішню структуру і типи відносин. На цій діаграмі не вказується інформація про тимчасові аспекти функціонування системи. Діаграма класів є деякий граф, вершинами якого є елементи типу "класифікатор", які пов'язані різними типами структурних відносин.

Слід зауважити, що діаграма класів може також містити інтерфейси, пакети, відносини і навіть окремі екземпляри, такі як об'єкти і зв'язки. Коли говорять про дану діаграму, мають на увазі статичну структурну модель проектованої системи. Тому діаграму класів прийнято вважати графічним представленням таких структурних взаємозв'язків логічної моделі системи, які не залежать або інваріантні від часу.

Діаграма класів складається з множини елементів, які в сукупності відображають декларативні знання про предметну область. Ці знання інтерпретуються в базових поняттях мови UML, таких як класи, інтерфейси і відносини між ними та їх складовими компонентами. При цьому окремі компоненти цієї діаграми можуть утворювати пакети для представлення більш загальної моделі системи. Якщо діаграма класів є частиною деякого пакета, то її компоненти повинні відповідати елементам цього пакета, включаючи можливі посилання на елементи з інших пакетів.

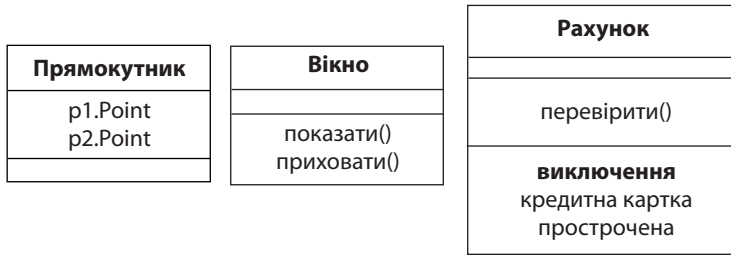
У загальному випадку пакет статичної структурної моделі може бути представлений у вигляді однієї або кількох діаграм класів.

Клас

Клас (class) у мові UML служить для позначення множини об'єктів, які володіють однаковою структурою, поведінкою і відносинами з об'єктами інших класів. Графічно клас зображується у вигляді прямокутника, що додатково може бути розділений горизонтальними лініями на розділи або секції. У цих розділах можуть зазначатися назва класу, атрибути (змінні) та операції (методи):

а *б* *в*

Обов'язковим позначенням елементу класу є його ім'я. На початкових етапах розробки діаграми окремі класи можуть позначатися простим прямокутником із зазначенням тільки імені відповідного класу (*а*). По мірі опрацювання окремих компонентів діаграми опису класів доповнюються атрибутами (*б*) і операціями (*в*).



а *б* *в*

Навіть, якщо секція атрибутів і операцій є порожньою, у позначенні класу вона виділяється горизонтальною лінією, щоб відразу відрізнити клас від інших елементів мови UML. Приклади графічного зображення класів на діаграмі класів наведено на рисунку.

Ім'я класу має бути унікальним у межах пакета, який описується деякою сукупністю діаграм класів (можливо однією діаграмою). Воно вказується у першій верхній секції прямокутника. На додаток до загального правила найменування елементів мови UML ім'я класу записується по центру секції імені напівжирним шрифтом і повинно починатися з великої літери. Рекомендується в якості імен класів використовувати іменники, записані з практичних міркувань без пробілів. У першій секції позначення класу можуть знаходитися посилання на стандартні шаблони або абстрактні класи, від яких утворено даний клас і, відповідно, від яких він успадковує властивості та методи. У цій секції може наводитися інформація про розробника даного класу і статус стану розробки, а також можуть записуватися інші загальні властивості цього класу, що мають відношення до інших класів діаграми або стандартних елементів мови UML.

Клас може не мати примірників або об'єктів. У цьому випадку він називається абстрактним класом, а для позначення його імені використовується курсив. У мові UML прийнята спільна угода про те, що будь-який текст, що відноситься до абстрактного елементу, записується курсивом.

У деяких випадках необхідно явно вказати, до якого пакета належить той або інший клас. Для цієї мети використовується спеціальний символ розділювач — подвійна двокрапка "**::**". Синтаксис рядка імені класу в цьому випадку буде такий **<Ім'я_пакета>:: <Ім'я_класу>**. Іншими словами, перед ім'ям класу має бути явно вказано назву пакета, до якого його слід віднести. Наприклад, якщо визначено пакет з ім'ям "Банк", то клас "Рахунок" у цьому банку може бути записаний у вигляді: "Банк:: Рахунок".

У другій зверху секції прямокутника класу записуються його атрибути (attributes) або властивості. У мові UML прийнята

певна стандартизація запису атрибутів класу, яка підпорядковується деяким синтаксичним правилам. Кожному атрибуту класу відповідає окремий рядок тексту, який складається з квантора видимості атрибута, назви ознаки, його кратності, типу значень атрибута і його можливого вихідного значення:

<квантор видимості> <ім'я атрибута> [кратність]:
<тип атрибута> = <початкове значення> (рядок-властивість).

Квантор видимості може приймати одне з трьох можливих значень і, відповідно, відображатися за допомогою спеціальних символів:

• символ **"+"** — атрибут з областю видимості типу загальнодоступний (public). Атрибут з цією областю видимості доступний або видимий з будь-якого іншого класу пакета, в якому визначена діаграма;

• символ "#" — атрибут з областю видимості типу захищений (protected). Атрибут з цією областю видимості недоступний або невидимий для всіх класів, за винятком підкласів даного класу;

• знак "-" позначає атрибут з областю видимості типу закритий (private). Атрибут з цією областю видимості недоступний або невидимий для всіх класів без винятку.

Квантор видимості може бути опущений. Його відсутність означає, що видимість ознаки не вказується.

Оскільки мова UML інваріантна щодо реалізації своїх конструкцій у конкретних мовах програмування, семантика окремих кванторів видимості не є строго фіксованою. Значення цих кванторів повинні додатково уточнюватися пояснювальним текстом на природній мові або угодою з використання відповідних програмно-залежних синтаксичних конструкцій.

Назва атрибута представляє собою рядок тексту. Вона використовується ідентифікатор відповідного атрибута і тому повинна бути унікальною в межах певного класу. Назва атрибута — це єдиний обов'язковий елемент його синтаксичного позначення.

Кратність атрибута характеризує загальну кількість конкретних атрибутів даного типу, що входять до складу окремого класу. У загальному випадку кратність записується у формі рядка тексту в квадратних дужках після імені відповідного атрибута:

[нижня_межа1 … верхня_межа1, нижня_межа2 … верхня_межа2, …, нижня_межаk … верхня_межаk],

де нижня_межа і верхня_межа є позитивними цілими числами, кожна пара яких служить для позначення окремого замкнутого інтервалу цілих чисел. Як верхня_межа використовуватися спеціальний символ "*", який означає довільне позитивне ціле число. Інакше кажучи, це означає необмежену зверху значення кратності відповідного атрибута.

Як приклад розглянемо наступні варіанти завдання кратності атрибутів:

• [0 ... 1] означає, що кратність атрибута може приймати значення 0 або 1. При цьому 0 означає відсутність значення для даного атрибута;

• [0 ...*] означає, що кратність атрибута може приймати будь-яке позитивне ціле значення більше або рівне 0. Ця кратність може бути записана коротше у вигляді простого символу [*];

• [1.: *] означає, що кратність атрибута може приймати будь-яке позитивне ціле значення більше або рівне 1;

• [1 ... 5] означає, що кратність атрибута може приймати будь-яке значення з чисел: 1, 2, 3, 4, 5;

• [1 ... 3, 5, 7] означає, що кратність атрибута може приймати будь-яке значення з чисел: 1, 2, 3, 5, 7;

• [1 ... 3,7 ... 10] означає, що кратність атрибута може приймати будь-яке значення з чисел: 1, 2, 3, 7, 8, 9, 10;

• [1 ... 3,7 ...*] означає, що кратність атрибута може приймати будь-яке значення з чисел: 1, 2, 3, а також будь-яке позитивне ціле значення більше або рівне 7.

• якщо кратність ознаки не вказана, то за замовчуванням приймається її значення рівне 1 ... 1, тобто в точності 1.

Тип атрибута є виразом, семантика якого визначається мовою специфікації відповідної моделі. У нотації UML тип атри-

бута, який передбачається використовувати для реалізації цієї моделі, іноді визначається залежно від мови програмування.

Можна навести такі приклади завдання імен і типів атрибутів класів:

• колір: Color — тут колір є ім'ям атрибута (ім'ям типу цього атрибута. Вказаний запис може визначати RGB-модель (червона, зелена, синя) традиційно використовується для представлення кольору. У цьому випадку ім'я типу Color як раз і характеризує семантичну конструкцію, яка застосовується у більшості мов програмування для представлення кольору;

• ім'я_співробітника [1 ... 2]: String — тут імя_ співробітника є ім'ям атрибута, який служить для представлення інформації про ім'я, а можливо, і по батькові конкретного співробітника. Тип атрибута String (Строка) вказує на той факт, що окреме значення імені представляє собою рядок тексту з одного або двох слів (наприклад, "Кирило" або "Дмитро Іванович"). Оскільки в багатьох мовах програмування існує тип даних String, використання відповідного англомовного терміна не викликає непорозуміння у більшості програмістів. Однак, хоча в мові UML всі терміни даються в англомовному поданні, використання в якості типу атрибута Рядок у даній ситуації не виключається і визначається тільки міркуваннями зручності.

Початкове значення служить для завдання деякого початкового значення для відповідного атрибута у момент створення окремого екземпляра класу. Тут необхідно дотримуватися правила належності значення типу конкретного атрибута. Якщо початкове значення не вказано, то значення відповідної ознаки не визначено на момент створення нового екземпляра класу.

Як приклади вихідних значень атрибутів можна навести наступні доповнені варіанти завдання атрибутів:

• колір: Color = (255, 0, 0) — у RGB-моделі кольору це відповідає чистому червоному кольору як вихідного значення для цьо-го атрибута;

• ім'я_ співробітника [1 ... 2]: String = Іван Іванович.

При завданні атрибутів можуть бути використані дві додаткові синтаксичні конструкції — це підкреслення рядка атрибута і пояснювальний текст у фігурних дужках.

Підкреслення рядка атрибута означає, що відповідний атрибут може приймати підмножину значень з деякої області значень атрибута, яка визначається його типом.

Наприклад, якщо певний атрибут задано у вигляді **форма: Прямокутник**, то це буде означати, що всі об'єкти цього класу можуть мати кілька різних форм, кожна з яких є прямокутником.

У третій зверху секції прямокутника записуються операції або методи класу. Операція (operation) — це деякий сервіс, що надає кожен екземпляр класу на вимогу. Сукупність операцій характеризує функціональний аспект поведінки класу. Запис операцій класу в мові UML також стандартизований і підпорядковується визначеним синтаксичним правилам. При цьому кожній операції класу відповідає окремий рядок, який складається з квантора видимості операції, імені операції, виразу типу, що повертається операцією значення, і, можливо, рядка-властивості певної операції:

<квантор видимості> <ім'я операції> (список параметрів): <вираз типу значення, що повертається> (рядок-властивість).

Квантор видимості, як і у випадку атрибутів класу, може приймати одне з трьох можливих значень і, відповідно, відображатися за допомогою спеціального символу:

• символ "+" означає операцію з областю видимості типу загальнодоступний (public);

• символ "#" означає операцію з областю видимості типу захищений (protected);

• символ "-" використовується для позначення операції з областю видимості типу закритий (private).

Квантор видимості для операції може бути опущений. Його відсутність означає, що видимість операції не вказується.

Конкретними мовами програмування можуть бути визначені додаткові квантори видимості. У цьому випадку подібні до-

повнення є розширенням базової нотації. Вони вимагають відповідних пояснень у формі тексту рідною мовою або у вигляді рядка-властивості.

Назва операції — це рядок тексту. Вона використовується як ідентифікатор відповідної операції і тому повинна бути унікальною в межах певного класу. Назва атрибута — це єдиний обов'язковий елемент синтаксичного позначення операції.

Список параметрів є переліком розділених комою формальних параметрів, кожен з яких може бути представлений у наступному вигляді:

<вид параметра> <ім'я параметра>: <вираз типу> = <значення параметра за замовчуванням > (Властивість).

Тут вид параметра — одне з ключових слів in, out або inout зі значенням in за замовчуванням, у випадку, якщо вид параметра не вказується. Ім'я параметра — ідентифікатор відповідного формального параметра. Вираз типу є залежною від конкретної мови програмування специфікацією типу значення, що повертається для відповідного формального параметра. Нарешті, значення за замовчуванням у загальному випадку являє собою вираз для значення формального параметра, синтаксис якого залежить від конкретної мови програмування і підпорядковується прийнятим у ньому обмеженням.

Рядок-властивість служить для вказівки значень властивостей, які можуть бути застосовані до даного елемента. Рядок-властивість не є обов'язковим, він може бути відсутнім, якщо ніякі властивості не специфіковані.

Для підвищення продуктивності системи одні операції можуть виконуватися паралельно або одночасно, а інші — тільки послідовно. У цьому випадку для вказівки паралельності виконання операції використовується рядок-властивість виду "(concurrency = ім'я)", де ім'я може приймати одне з наступних значень: послідовна (sequential), паралельна (concurrent), під охороною (guarded). При цьому дотримуються наступної семантики для даних значень:

• послідовна (sequential) — для цієї операції необхідно забезпечити її єдине виконання в системі, одночасне виконання інших операцій може призвести до помилок або порушень цілісності об'єктів класу;

• паралельна (concurrent) — ця операція в силу своїх особливостей може виконуватися паралельно з іншими операціями в системі, при цьому паралельність повинна підтримуватися на рівні реалізації моделі;

• під охороною (guarded) — всі звернення до цієї операції повинні бути строго впорядковані у часі з метою збереження цілісності об'єктів даного класу, при цьому можуть бути прийняті додаткові заходи з контролю виняткових ситуацій на етапі її виконання.

Поява сигнатури операції на самому верхньому рівні оголошує цю операцію на весь клас, при цьому ця операція успадковується всіма нащадками даного класу. Якщо в деякому класі операція не виконується (тобто деякий метод не застосовується), то така операція може бути позначена як абстрактна (abstract). Інший спосіб показати абстрактний характер операції — записати її сигнатуру курсивом.

Як приклади запису операцій можна навести наступні позначення окремих операцій:

• + створити () — може позначати абстрактну операцію по створенню окремого об'єкта класу, що є загальнодоступною і не містить формальних параметрів. Ця операція не повертає ніякого значення після свого виконання;

• + намалювати (форма: Багатокутник = прямокутник, колір_заповненя: Color = (O, O, 255)) — може позначати операцію відображення на екрані монітора прямокутної області синього кольору, якщо не вказуються інші значення як аргументи даної операції;

• вивести_повідомлення ():{"Помилка ділення на нуль") — зміст цієї операції не потребує пояснення, оскільки він міститься в рядку-властивості операції. Дане повідомлення може з'явитися на екрані монітора у разі спроби поділу деякого числа на нуль, що неприпустимо.

Відносини між класами

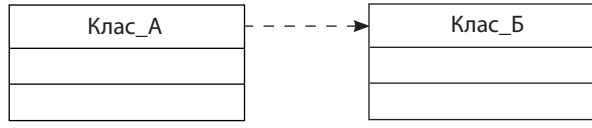
Крім внутрішнього представлення або структури класів на відповідній діаграмі вказуються різні відносини між класами. При цьому сукупність типів таких відносин фіксована в мові UML і зумовлена семантикою цих типів відносин. Базовими відносинами або зв'язками в мові UML є:

- відношення залежності (dependency relationship);
- відношення асоціації (association relationship);
- відношення узагальнення (generalization relationship);
- відношення реалізації (realization relationship).

Відношення залежності

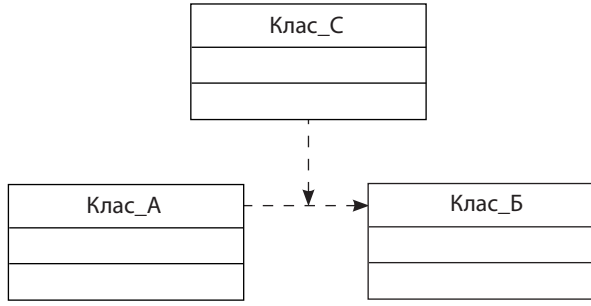
Відношення залежності у загальному випадку вказує на деяке семантичне відношення між двома елементами моделі або двома множинами таких елементів, які не є відношенням асоціації, узагальнення або реалізації. Відношення залежності використовується у такій ситуації, коли деяка зміна одного елемента моделі може вимагати зміни іншого залежного від нього елемента моделі.

Відношення залежності графічно зображується пунктирною лінією між відповідними елементами із стрілкою на одному з її кінців ("--->" або "<---"). На діаграмі класів дане відношення пов'язує окремі класи між собою, при цьому стрілка спрямована від класу-клієнта залежності до незалежного класу або класу-джерела:



Як клас-клієнт і клас-джерело залежності можуть виступати цілі множини елементів моделі. У цьому випадку одна лінія зі стрілкою, що виходить від джерела залежності, розщеплюється в деякій точці на кілька окремих ліній, кожна з яких має окрему стрілку для класу-клієнта. Наприклад, якщо функціонування

Класу_С залежить від особливостей реалізації Класу_А і Класу_Б, то ця залежність може бути зображена у такий спосіб:



Стрілка може позначатися необов'язковим, але стандартним ключовим словом у лапках, і необов'язковим індивідуальним ім'ям. Для відношення залежності визначені ключові слова, які позначають деякі спеціальні види залежностей. Ці ключові слова (стереотипи) записуються у лапках поруч зі стрілкою, що відповідає даній залежності. Приклади стереотипів для відношення залежності такі:

• "access" — служить для позначення доступності відкритих атрибутів і операцій класу-джерела для класів-клієнтів;

• "bind" — клас-клієнт може використовувати певний шаблон для своєї подальшої параметризації;

• "derive" — атрибути класу-клієнта можуть бути обчислені за атрибутами класу-джерела;

• "import" — відкриті атрибути та операції класу-джерела стають частиною класу-клієнта, так якби вони були оголошені безпосередньо в ньому;

• "refine" — вказує, що клас-клієнт служить уточненням класу-джерела в силу причин історичного характеру, коли з'являється додаткова інформація в ході роботи над проектом.

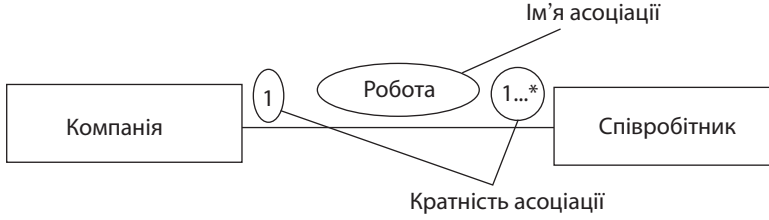
Відношення залежності є найбільш загальною формою відносин у мові UML. Усі інші види розглянутих відносин можна вважати окремим випадком цього відношення.

Відношення асоціації

Відношення асоціації відповідає наявності деяких відносин між класами. Це відношення позначається суцільною лінією з додатковими спеціальними символами, які характеризують окремі властивості конкретної асоціації. В якості додаткових спеціальних символів можуть використовуватися назва асоціації, а також імена і кратність класів-ролей асоціації. Назва асоціації не є обов'язковим елементом її позначення. Якщо вона задана, то записується з великої літери поруч з лінією відповідної асоціації.

Найбільш простий випадок даного відношення — бінарна асоціація. Вона пов'язує в точності два класи і, як виняток, може пов'язувати клас із самим собою. Для бінарної асоціації на діаграмі може бути зазначений порядок проходження класів з використанням трикутника у формі стрілки поруч з ім'ям цієї асоціації. Напрямок цієї стрілки вказує на порядок класів, один з яких є першим (з боку трикутника), а інший — другим (з боку вершини трикутника). Відсутність даної стрілки поруч з ім'ям асоціації означає, що порядок проходження класів у даному відношенні не визначений.

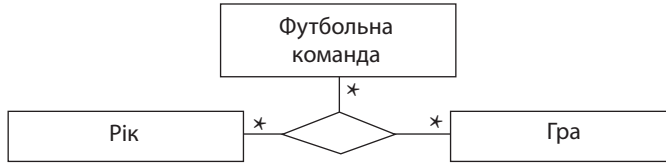
Як простий приклад ставлення бінарної асоціації розглянемо відношення між двома класами — класом "Компанія" і класом "Співробітник";



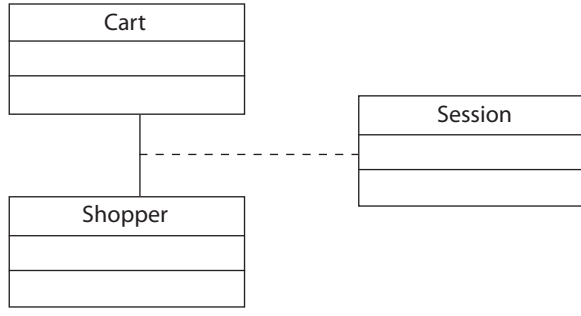
Тернарна асоціація та асоціації більш високої арность у загальному випадку називаються *N*-арною асоціацією (читається — "Ен-арна асоціація"). Така асоціація пов'язує деяким відношенням три і більше класів, при цьому один клас може брати участь в асоціації більше ніж один раз. Клас асоціації має певну роль у відповідному відношенні, що може бути явно вказано на

діаграмі. Кожен екземпляр *N*-арної асоціації являє собою *N*-арний кортеж значень об'єктів з відповідних класів. Бінарна асоціація є окремим випадком *N*-арної асоціації, коли значення $N = 2$, і має своє власне позначення.

N-арна асоціація графічно позначається ромбом, від якого ведуть лінії до символів класів даної асоціації. У цьому випадку ромб з'єднується із символами відповідних класів суцільними лініями. Зазвичай лінії проводяться від вершин ромба або від середини його сторін. Назва *N*-арної асоціації записується поруч з ромбом відповідної асоціації:



Порядок класів у *N*-арній асоціації, на відміну від порядку множин у відношенні, на діаграмі не фіксується. Деякий клас може бути приєднаний до ромбу пунктирною лінією. Це означає, що даний клас забезпечує підтримку властивостей відповідної *N*-арної асоціації, а сама *N*-арна асоціація має атрибути, операції та/або асоціації. Інакше кажучи, така асоціація, у свою чергу, є класом з відповідним позначенням у вигляді прямокутника, а також є самостійним елементом мови UML — асоціацією-класом (Association Class). *N*-арна асоціація не може містити символ агрегації ні для якої зі своїх ролей.

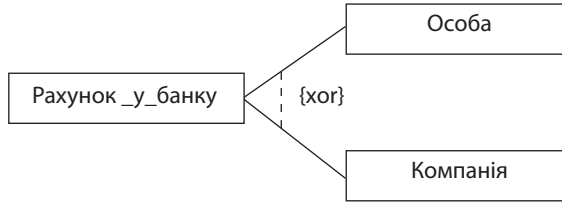


Наступний елемент позначень — кратність окремих класів, які є кінцями асоціації. Кратність окремого класу позначається у вигляді інтервалу цілих чисел, аналогічно кратності атрибутів і операцій класів.

Так, для розглянутого раніше прикладу кратність "1" для класу "Компанія" означає, що кожен працівник може працювати тільки в одній компанії. Кратність "1 ...*" для класу "Співробітник" означає, що в кожній компанії можуть працювати співробітники, загальна кількість яких заздалегідь невідома і нічим не обмежена. Зауважимо, що замість кратності "1 ...*" записати тільки символ "*" не можна, оскільки останній означає кратність "0 ...*". Для даного прикладу це означало б, що окремі компанії можуть зовсім не мати співробітників у своєму штаті.

Приватним випадком відношення асоціації є так звана виключна асоціація (Xor-association). Семантика цієї асоціації вказує на той факт, що з кількох потенційно можливих варіантів цієї асоціації у кожний момент часу може використовуватися тільки один її примірник. На діаграмі класів виключна асоціація зображується пунктирною лінією, що сполучає дві і більше асоціації, поряд з якою записується рядок-обмеження "(Xor)".

Наприклад, рахунок у банку може бути відкритий для клієнта, в якості якого може виступати фізична особа або компанія, що зображується за допомогою виключної асоціації.

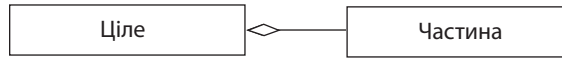


Відношення агрегації

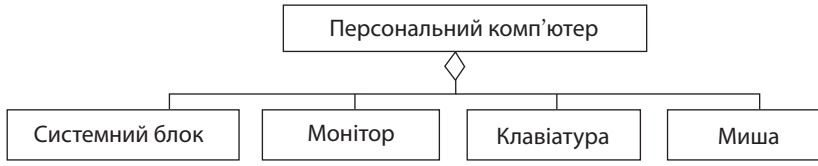
Відношення агрегації має місце між кількома класами у тому випадку, якщо один із класів являє собою деяку сутність, що включає в себе в якості складових інші сутності.

Це відношення має фундаментальне значення для опису структури складних систем, оскільки застосовується для представлення системних взаємозв'язків типу "частина-ціле". Розкриваючи внутрішню структуру системи, відношення агрегації показує, з яких компонентів складається система і як вони пов'язані між собою. З точки зору моделі, окремі частини системи можуть виступати як у вигляді елементів, так і у вигляді підсистем, які, у свою чергу, теж можуть утворювати складові компоненти або підсистеми. Це відношення по своїй суті описує декомпозицію або розбиття складної системи на більш прості складові, що також можуть бути піддані декомпозиції, якщо в цьому виникне необхідність у подальшому.

Очевидно, що розглянутий у такому аспекті поділ системи на складові являє собою деяку ієрархію її компонентів, однак ця ієрархія принципово відрізняється від ієрархії, породжуваної відношенням узагальнення. Відмінність полягає в тому, що частини системи ніяк не зобов'язані наслідувати її властивості та поведінку, оскільки є цілком самостійними сутностями. Більше того, частини цілого володіють власними атрибутами і операціями, які суттєво відрізняються від атрибутів і операцій цілого. Графічно відношення агрегації зображується суцільною лінією, один з кінців якої являє собою незафарбований всередині ромб. Цей ромб вказує на той з класів, який являє собою "ціле". Інші класи є його "частинами".



Ще одним прикладом відношення агрегації може служити відомий кожному з читачів поділ персонального комп'ютера на складові: системний блок, монітор, клавіатуру і мишу.

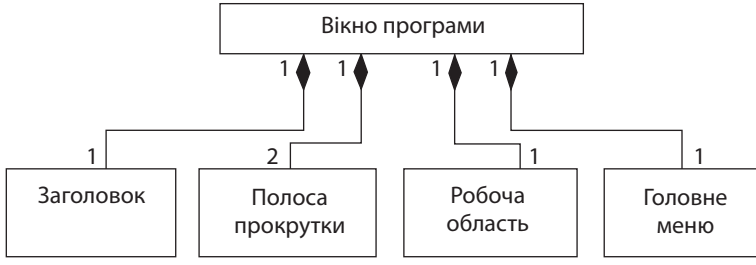


Відношення композиції

Відношення композиції, як вже зазначалося, є окремим випадком відношення агрегації. Це відношення служить для виділення спеціальної форми відношення "частина-ціле", при якому складові у деякому розумінні знаходяться всередині цілого. Специфіка взаємозв'язків між ними полягає у тому, що частини не можуть виступати у відриві від цілого, тобто зі знищенням цілого знищуються і всі його складові.

Графічно відношення композиції зображається суцільною лінією, один з кінців якої являє собою зафарбований всередині ромб. Цей ромб вказує на той з класів, який являє собою клас-композицію або "ціле". Інші класи є його "частинами".

В якості додаткових позначень для відношення композиції та агрегації можуть використовуватися додаткові позначення, що застосовуються для відношення асоціації, а саме: вказівка кратності класу асоціації та імені даної асоціації, які не є обов'язковими. Стосовно до описаного вище прикладу класу "Вікно_програми" його діаграма класів може мати такий вигляд:

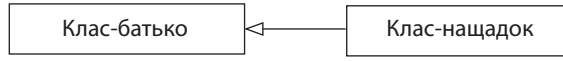


Відношення узагальнення

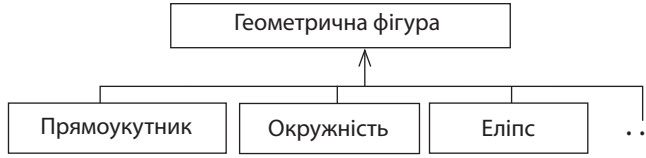
Відношення узагальнення є звичайним відношенням між більш загальним елементом (батьком) і більш приватним або спеціальним елементом (дочірнім або нащадком). Дане відно-

шення може використовуватися для представлення взаємозв'язків між пакетами, класами, варіантами використання та іншими елементами мови UML.

Стосовно діаграми класів це відношення описує ієрархічну будову класів та успадкування їх властивостей та поведінки. При цьому передбачається, що клас-нащадок має всі властивості та поведінку класу-батька, а також має особисті властивості та поведінку, які відсутні у класі-батька. На діаграмах відношення узагальнення позначається суцільною лінією з трикутною стрілкою на одному з кінців. Стрілка вказує на більш загальний клас (клас-батько або суперклас), а її відсутність — на більш спеціальний клас (клас-нащадок або підклас).



З метою спрощення позначень на діаграмі класів сукупність ліній, що позначають одне й те саме відношення узагальнення, можуть бути об'єднані в одну лінію. При цьому окремі лінії сходяться до одної та об'єднуються стрілкою, що має з ними спільну точку перетину.



Поруч зі стрілкою узагальнення може розміщуватися рядок тексту, що вказує на деякі додаткові властивості цього відношення. Цей текст буде відноситись до всіх ліній узагальнення, які йдуть до класів-нащадків. Інакше кажучи, зазначена властивість стосується всіх підкласів даного відношення. При цьому текст слід розглядати як обмеження, і тоді він записується у фігурних дужках.

Як обмеження можуть бути використані наступні ключові слова мови UML:

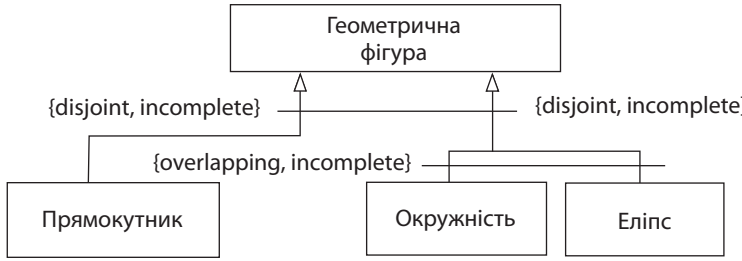
• **(complete)** — означає, що в даному відношенні узагальнення специфіковані всі класи-нащадки, інших класів-нащадків у

даного класу-батька бути не може. Приклад, клас Кліент_банку є батьком для двох класів: Фізична_особа і Компанія, інших класів-нащадків він не має. На відповідній діаграмі класів це можна вказати явно, записавши поруч з лінією узагальнення цей рядок-обмеження;

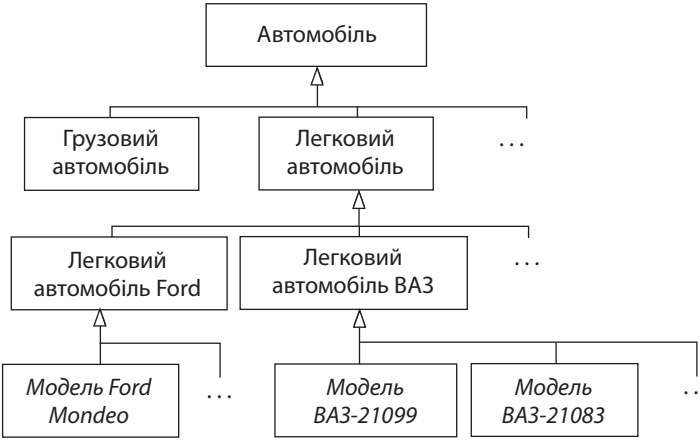
• **(disjoint)** — означає, що класи-нащадки не можуть містити об'єктів, які одночасно є екземплярами двох або більше класів. У наведеному вище прикладі це умова також виконується, оскільки передбачається, що ніяка конкретна фізична особа не може бути одночасно і конкретною компанією. У цьому випадку поряд з лінією узагальнення можна записати цей рядок-обмеження;

• **(incomplete)** — означає випадок, протилежний першому. Передбачається, з одного боку, що на діаграмі зазначено не всі класи-нащадки. У подальшому можливо заповнити їх перелік, не змінюючи вже побудовану діаграму. Приклад, діаграма класу "Автомобіль", для якої зазначення всіх без винятку моделей автомобілів можна порівняти зі створенням відповідного каталогу. З іншого боку, для окремої задачі, такої як розробка системи продажу автомобілів конкретних моделей, у цьому немає необхідності. Але вказати неповноту структури класів-нащадків все ж таки слід;

• **(overlapping)** — означає, що окремі екземпляри класів-нащадків можуть відноситися до кількох класів. Приклад, клас "Багатокутник" є класом-батьком для класу "Прямокутник" і класу "Ромб". Однак існує окремий клас "Квадрат", екземпляри якого одночасно є об'єктами перших двох класів. Цілком природно таку ситуацію вказати явно за допомогою цього рядка-обмеження.



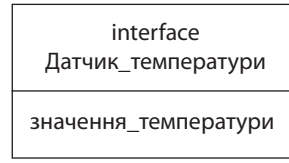
Як приклад розглянемо ієрархію вкладеності класів для абстрактного класу “Автомобіль”. Як неважко помітити, відношення між окремими класами на цих рисунках є саме відношення узагальнення, яке в мові UML має спеціальне графічне позначення. З урахуванням цієї графічної нотації, фрагмент семантичної мережі для подання ієрархії класу “Автомобіль” може бути представлений у вигляді наступної діаграми класів.



У цьому прикладі всі класи верхніх рівнів є абстрактними, тобто не можуть бути представлені своїми екземплярами. Саме тому їх імена записані курсивом. На відміну від них класи нижнього рівня є конкретними, оскільки можуть бути представлені своїми екземплярами, в якості яких виступають виготовлені автомобілі відповідної моделі з унікальним ім’ям.

Інтерфейси

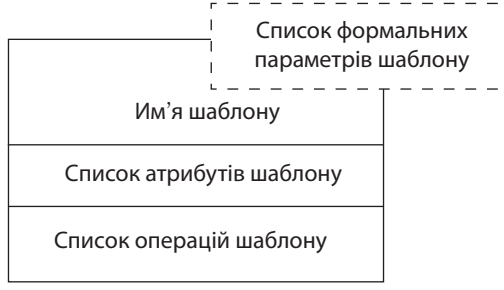
Інтерфейси є елементами діаграми варіантів використання. Однак, при побудові діаграми класів окремі інтерфейси можуть уточнюватися. У даному випадку для їх зображення використовується спеціальний графічний символ — прямокутник класу з ключовим словом або стереотипом “interface”. При цьому секція атрибутів у прямокутника відсутня, а вказується тільки секція операцій.



Шаблони або параметризовані класи

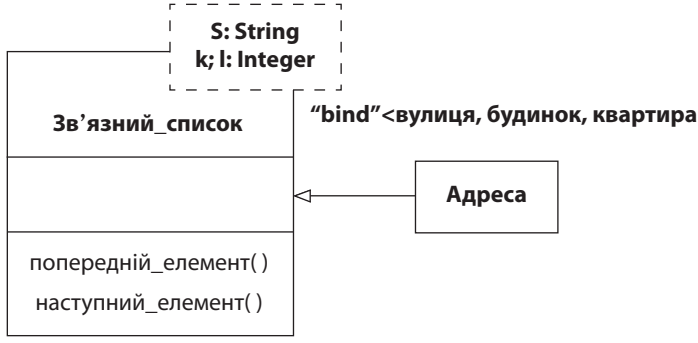
Шаблон (template) або параметризований клас (parametrized class) призначений для позначення такого класу, який має один (або більше) нефіксований формальний параметр. Він визначає ціле сімейство або множину класів, кожен з яких можна отримати зв'язуванням цих параметрів з дійсними значеннями. Зазвичай параметрами шаблонів служать типи атрибутів класів, такі як цілі числа, перерахування, масив рядків та ін. У більш складному випадку формальні параметри можуть представляти і операції класу.

Графічно шаблон зображується у вигляді прямокутника, на верхньому правому куті якого приєднано маленький прямокутник з пунктирних ліній. Великий прямокутник може бути розділений на секції відповідно до позначенн для класу. У верхньому прямокутнику вказується список формальних параметрів для тих класів, які можуть бути отримані на основі цього шаблону. У верхній секції шаблону записується його ім'я за правилами запису імен для класів.



Шаблон не може бути безпосередньо використаний як клас, оскільки містить невизначені параметри. Найчастіше як шаблон

виступає деякий суперклас, параметри якого уточнюються в його класах-нащадках. Очевидно, у цьому випадку між ними існує відношення залежності з ключовим словом “bind”, коли клас-клієнт може використовувати певний шаблон для своєї подальшої параметризації. У більш окремому випадку між шаблоном і класом, що формується від нього, має місце відношення узагальнення з наслідуванням властивостей шаблону. У прикладі нижче відзначений той факт, що клас “Адреса” може бути отриманий із шаблону Зв’язний_список на основі актуалізації формальних параметрів “S, k, l” фактичними атрибутами “вулиця, будинок, квартира”.



Лекція 22. ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ (ДІАГРАМА ПРЕЦЕДЕНТІВ)

Першою зазвичай будується модель у формі так званої діаграми варіантів використання (use case diagram), яка описує функціональне призначення системи або, інакше кажучи, те, що система буде робити у процесі свого функціонування.

Розробка діаграми варіантів використання переслідує такі цілі:

• визначити спільні кордони і контекст предметної області, що моделюється на початкових етапах проектування системи;

• сформулювати загальні вимоги до функціональної поведінки проектованої системи;

• розробити вихідну концептуальну модель системи для її подальшої деталізації у формі логічних і фізичних моделей;

• підготувати вихідну документацію для взаємодії розробників системи з її замовниками та користувачами.

Суть цієї діаграми полягає у наступному: проектована система представляється у вигляді множини сутностей або акторів, що взаємодіють із системою за допомогою так званих варіантів використання. При цьому актором (actor) або дійовою особою називається будь-яка сутність, що взаємодіє із системою ззовні. Це може бути людина, технічний пристрій, програма або будь-яка інша система, що може служити джерелом впливу на змодельовану систему так, як визначить розробник. У свою чергу, варіант використання (use case) служить для опису сервісів, які система надає акторові. Інакше кажучи, кожен варіант використання визначає деякий набір дій, що чиниться системою при діалозі з актором. При цьому нічого не говориться про те, яким чином буде реалізовано взаємодія акторів із системою.

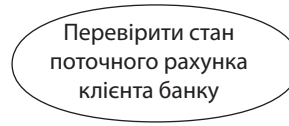
У самому загальному випадку діаграма варіантів використання являє собою граф спеціального виду, який є графічною нотацією для представлення конкретних варіантів використання, акторів, можливо деяких інтерфейсів, і відносин між цими елементами. При цьому окремі компоненти діаграми можуть бути вкладеними у прямокутник, що позначає проектова-

ну систему загалом. Слід зазначити, що відносинами цього графа можуть бути тільки деякі фіксовані типи взаємозв'язків між акторами і варіантами використання, які у сукупності описують сервіси або функціональні вимоги до системи, що моделюється.

Варіант використання

Конструкція або стандартний елемент мови UML варіант використання застосовується для характеристики загальних особливостей поведінки системи або будь-якої іншої сутності предметної області без розгляду внутрішньої структури цієї сутності. Кожен варіант використання визначає послідовність дій, які повинні бути виконані проектованою системою при взаємодії її з відповідним актором. Діаграма варіантів може доповнюватися пояснювальним текстом, який розкриває зміст або семантику складових її компонентів. Такий пояснювальний текст отримав назву примітки або сценарію.

Окремий варіант використання позначається на діаграмі еліпсом, всередині якого міститься його коротка назва або ім'я у формі дієслова з пояснювальними словами.



Мета варіанта використання полягає у тому, щоб визначити закінчений аспект, або фрагмент поведінки деякої сутності без розкриття внутрішньої структури цієї сутності. Такою сутністю може виступати вихідна система або будь-який інший елемент моделі, що володіє власною поведінкою, подібно підсистемі або класу в моделі системи.

Кожен варіант використання відповідає окремому сервісу, який надає змодельована сутність або система за запитом користувача (актора), тобто визначає спосіб застосування цієї сутності. Сервіс, який ініціалізується за запитом користувача, представляє собою закінчену послідовність дій. Це означає, що після

того, як система закінчить обробку запиту користувача, вона повинна повернутися у початковий стан, в якому готова до виконання таких запитів.

Варіанти використання описують не тільки взаємодію між користувачами і сутністю, а й реакцію сутності на отримання окремих повідомлень від користувачів і сприйняття цих пові-домлень за межами сутності. Варіанти використання можуть включати в себе опис особливостей способів реалізації сервісу і різних виняткових ситуацій, таких як коректна обробка помилок системи.

Із системно-інформаційної точки зору, варіанти використання можуть застосовуватися як для специфікації зовнішніх вимог до проектованої системи, так і для специфікації функціональної поведінки вже існуючої системи. Крім цього, варіанти використання неявно встановлюють вимоги, що визначають, як користувачі повинні взаємодіяти із системою, щоб мати можливість коректно працювати з наданими цією системою сервісами.

Актор

Актор являє собою будь-яку зовнішню, відносно системи, що моделюється, сутність, яка взаємодіє із системою і використовує її функціональні можливості для досягнення певної мети або розв'язання окремих завдань. При цьому актори служать для позначення узгодженої множини ролей, які можуть грати користувачі в процесі взаємодії з проектованою системою. Кожен актор може розглядатися як якась окрема роль щодо конкретного варіанта використання. Стандартним графічним позначенням актора на діаграмах є фігурка "чоловічка", під якою записується конкретне ім'я актора.

У деяких випадках актор може позначатися у вигляді прямокутника класу з ключовим словом "актор" і звичайними складовими елементами класу. Імена акторів повинні записуватися великими літерами і слідувати рекомендаціям використання

імен для типів і класів моделі. При цьому символ окремого актора пов'язує відповідний опис актора з конкретним ім'ям. Імена абстрактних акторів, як й інших абстрактних елементів мови UML, рекомендується позначати курсивом.

Так як у загальному випадку актор завжди знаходиться поза системою, його внутрішня структура ніяк не визначається. Для актора має значення тільки його зовнішнє уявлення, тобто те, як він сприймається з боку системи. Актори взаємодіють із системою за допомогою передачі та прийому повідомлень від варіантів використання. Повідомлення — це запит актором сервісу від системи та отримання цього сервісу. Ця взаємодія може бути виражена за допомогою асоціацій між окремими акторами та варіантами використання або класами. Крім цього, з акторами можуть бути пов'язані інтерфейси, які визначають, яким чином інші елементи моделі взаємодіють з цими акторами.

Інтерфейс

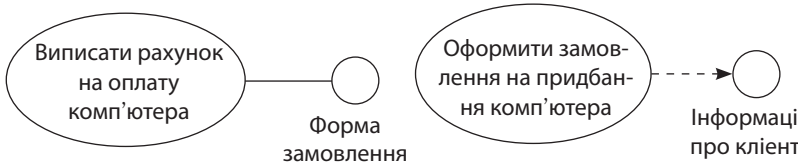
Інтерфейс служить для специфікації параметрів моделі, які видимі ззовні без вказівки їх внутрішньої структури. У мові UML інтерфейс є класифікатором і характеризує лише обмежену частину поведінки змодельованої сутності. Стосовно діаграм варіантів використання, інтерфейси визначають сукупність операцій, які забезпечують необхідний набір сервісів або функціональності для акторів. Інтерфейси не можуть містити ні атрибутів, ні станів, ні спрямованих асоціацій. Вони містять тільки операції без вказівки особливостей їх реалізації. Формально інтерфейс еквівалентний абстрактному класу без атрибутів і методів з наявністю тільки абстрактних операцій.

На діаграмі варіантів використання інтерфейс зображується у вигляді маленького кола, поряд з яким записується його ім'я (*a*). Ім'я може бути іменником, що характеризує відповідну інформацію або сервіс (наприклад, "датчик", "сирена", "відеокамера"), але частіше — це рядок тексту (наприклад, "запит до бази даних", "форма вводу", "пристрій подачі звукового сигналу"). Якщо ім'я записується англійською, то вона повинна почина-

тися з великої літери I (наприклад, ISecureInformation, ISensor) (*б*).

а *б*

Графічний символ окремого інтерфейсу може з'єднуватися на діаграмі суцільною лінією з тим варіантом використання, який його підтримує. Суцільна лінія у цьому випадку вказує на той факт, що пов'язаний з інтерфейсом варіант використання повинен реалізовувати всі операції, необхідні для цього інтерфейсу, а можливо і більше (*а*). Крім цього, інтерфейси можуть з'єднуватися з варіантами використання пунктирною лінією зі стрілкою (*б*). Це означає, що варіант використання призначений для специфікації тільки того сервісу, який необхідний для реалізації цього інтерфейсу.



а *б*

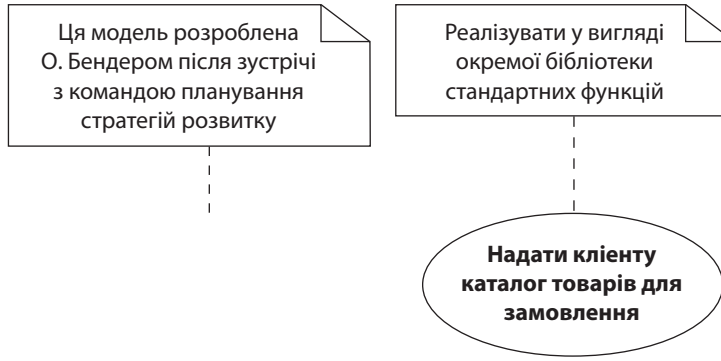
Важливість інтерфейсів полягає в тому, що вони визначають стикувальні вузли в проектованої системи, що абсолютно необхідно для організації колективної роботи над проектом. Більш того, специфікація інтерфейсів сприяє "безболісної" модифікації вже існуючої системи при переході на нові технологічні рішення.

Примітки

Примітки (notes) у мові UML призначені для включення у модель довільної текстової інформації, що має безпосереднє відношення до контексту проекту, що розробляється. Такою ін-

формацією можуть бути коментарі розробника (наприклад, дата і версія розробки діаграми або її окремих компонентів), обмеження (наприклад, на значення окремих зв'язків або екземпляри сутностей) і помічені значення. Стосовно діаграм варіантів використання примітки можуть носити саму загальну інформацію, що відноситься до загального контексту системи.

Графічно примітки позначаються прямокутником із "загнутим" верхнім правим куточком. Всередині прямокутника міститься текст примітки. Примітка може відноситися до будь-якого елемента діаграми, в цьому випадку їх з'єднує пунктирна лінія. Якщо примітка відноситься до кількох елементів, то від неї проводяться, відповідно, кілька ліній.



Відносини на діаграмі варіантів використання

Між компонентами діаграми варіантів використання можуть існувати різні відносини, які описують взаємодію екземплярів одних акторів і варіантів використання з екземплярами інших акторів і варіантів. Один актор може взаємодіяти з кількома варіантами використання. У цьому випадку актор звертається до кількох сервісів даної системи. У свою чергу один варіант використання може взаємодіяти з кількома акторами, надаючи для всіх них свій сервіс. Слід зауважити, що два варіанти використання, визначені для однієї і тієї самої сутності, не можуть взає-

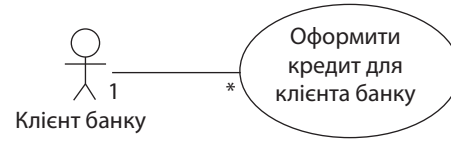
модіяти один з одним, бо кожен з них самостійно описує закінчений варіант використання цієї сутності.

У мові UML є кілька стандартних видів відносин між акторами і варіантами використання:

- відношення асоціації (association relationship);
- відношення розширення (extend relationship);
- відношення узагальнення (generalization relationship);
- відношення включення (include relationship).

Відношення асоціації

Відношення асоціації служить для позначення специфічної ролі актора в окремому варіанті використання. Інакше кажучи, асоціація специфікує семантичні особливості взаємодії акторів і варіантів використання в графічній моделі системи. Таким чином, це відношення встановлює, яку конкретну роль грає актор при взаємодії з екземпляром варіанта використання. На діаграмі варіантів використання, так само як і на інших діаграмах, ставлення асоціації позначається суцільною лінією між актором і варіантом використання. Ця лінія може мати додаткові умовні позначення, наприклад, такі як ім'я та кратність.



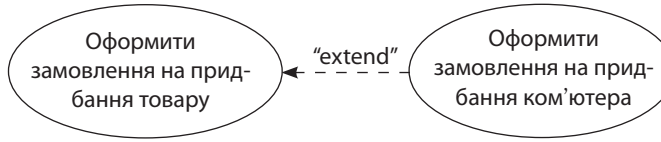
Кратність (multiplicity) асоціації вказується поруч із позначенням компонента діаграми, який є учасником цієї асоціації. Кратність характеризує загальну кількість конкретних екземплярів цього компонента, що можуть виступати як елементи цієї асоціації.

Тут кратність "*" означає, що кожен окремий клієнт банку може оформити для себе кілька кредитів, при цьому їх загальна кількість заздалегідь невідома і нічим не обмежується. При цьому деякі клієнти можуть зовсім не мати оформлених на своє ім'я кредитів (варіант значення 0).

Відношення розширення

Відношення розширення визначає взаємозв'язок екземплярів окремого варіанта використання з більш загальним варіантом, властивості якого визначаються на основі способу спільного об'єднання цих екземплярів. У метамоделі відношення розширення є спрямованим і зазначає, що відносно окремих прикладів деякого варіанта використання повинні бути виконані конкретні умови, визначені для розширення даного варіанта використання. Так, якщо має місце відношення розширення від варіанта використання *A* до варіанта використання *B*, то це означає, що властивості екземпляру варіанта використання *B* можуть бути доповнені завдяки наявності властивостей у розширеного варіанта використання *A*.

Відношення розширення між варіантами використання позначається пунктирною лінією зі стрілкою (варіант ставлення залежності), спрямованої від того варіанта використання, який є розширенням для вихідного варіанта використання. Пунктирна лінія зі стрілкою позначається ключовим словом "extend" ("розширює").

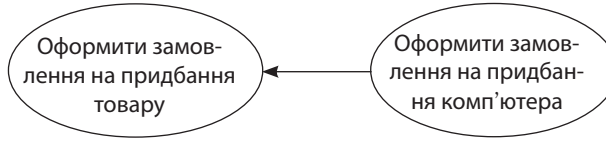


Відношення розширення відзначає той факт, що один із варіантів використання може приєднувати до своєї поведінки деяку додаткову поведінку, визначену для іншого варіанта використання.

Відношення узагальнення

Відношення узагальнення служить для зазначення того факту, що деякий варіант використання *A* може бути узагальнений до варіанта використання *B*. У цьому випадку варіант *A* буде спеціалізацією варіанта *B*. При цьому *B* називається батьком відносно *A*, а варіант *A* — нащадком відносно варіанта використання *B*. Слід підкреслити, що нащадок успадковує всі власти-

вості та поведінку свого батька, а також може бути доповнений новими властивостями і особливостями поведінки.



Відношення узагальнення між варіантами використання застосовується в тому випадку, коли необхідно відзначити, що дочірні варіанти використання володіють усіма атрибутами і особливостями поведінки батьківських варіантів. При цьому дочірні варіанти використання беруть участь в усіх відношеннях батьківських варіантів. У свою чергу, дочірні варіанти можуть наділятися новими властивостями поведінки, які відсутні у батьківських варіантів використання, а також уточнювати або модифікувати успадковані від них властивості поведінки.

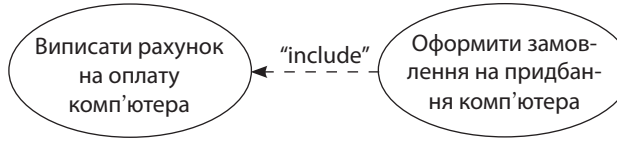
Між окремими акторами також може існувати відношення узагальнення. Це відношення є спрямованим і вказує на факт спеціалізації одних акторів щодо інших. Наприклад, відношення узагальнення від актора *A* до актора *B* відзначає той факт, що кожен екземпляр актора *A* є водночас примірником актора *B* і володіє всіма його властивостями. У цьому випадку актор *B* є батьком відносно актора *A*, а актор *A*, відповідно, нащадком актора *B*. При цьому актор *A* має здатність грати таку саму множину ролей, що і актор *B*:

Відношення включення

Відношення включення між двома варіантами використання вказує, що деяка задана поведінка для одного варіанта використання включається як складний компонент у послідовність поведінки іншого варіанта використання. Це відношення є спря-

мованим бінарним відношенням у тому сенсі, що пара екземплярів варіантів використання завжди впорядкована відносно включення.

Відношення включення, спрямоване від варіанта використання *A* до варіанта використання *B*, вказує, що кожен екземпляр варіанта *A* включає в себе функціональні властивості, задані для варіанта *B*. Ці властивості спеціалізують поведінку відповідного варіанта *A* на даній діаграмі. Графічно це відношення позначається пунктирною лінією зі стрілкою (варіант відношення залежності), спрямованої від базового варіанта використання до того, що включається. Пунктирна лінія зі стрілкою позначається ключовим словом "include" ("включає"):



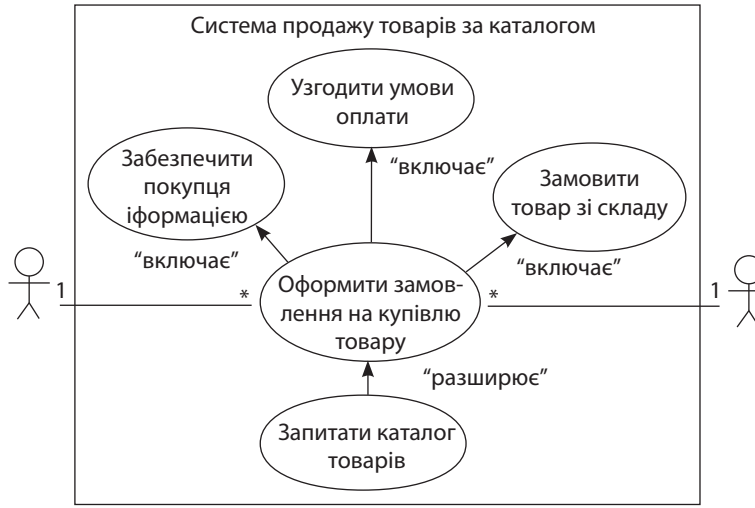
Приклад побудови діаграми варіантів використання

Як приклад розглянемо процес моделювання системи продажу товарів за каталогом, що може бути використана при створенні відповідних інформаційних систем.

Акторами цієї системи можуть виступати два суб'єкти, один з яких є продавцем, а другий — покупцем. Кожен з цих акторів взаємодіє з розглянутою системою продажу товарів за каталогом і є її користувачем, тобто вони обидва звертаються до відповідного сервісу "Оформити замовлення на купівлю товару".

Значення вказаних на цій діаграмі кратності відображають загальні правила або логіку оформлення замовлень на купівлю товарів. Згідно з цими правилами, з одного боку, продавець може брати участь в оформленні кількох замовлень, у той самий час кожне замовлення може бути оформлено тільки одним продавцем, який несе відповідальність за коректність його оформлення і, у зв'язку з цим, буде мати агентську винагороду за його оформлення. З іншого боку, кожен покупець може оформляти на себе кілька замовлень, але, в той самий час, кожне замовлення повинно бути оформлене на єдиного покупця, до якого переходять права власності на товар після його оплати.

На наступному етапі розробки цієї діаграми варіант використання "Оформити замовлення на купівлю товару" може бути уточнений на основі введення у розгляд чотирьох додаткових варіантів використання. Це випливає з більш детального аналізу процесу продажу товарів, що дає змогу виділити в якості окремих сервісів такі дії. Продаж товарів за каталогом припускає наявність самостійного інформаційного об'єкта — каталогу товарів, що в деякому розумінні не залежить від реалізації сервісу з обслуговування покупців.



Наведена діаграма варіантів використання, у свою чергу, може бути деталізована далі з метою більш глибокого уточнення вимог і конкретизації деталей її подальшої реалізації, що пред'являються до системи. З одного боку, деталізація може бути виконана на основі встановлення додаткових відносин типу "узагальнення-спеціалізація" для вже наявних компонентів діаграми варіантів використання. Так, у рамках цієї системи продажу товарів може мати самостійне значення і специфічні особливості окрема категорія товарів — комп'ютери. У цьому випадку діаграма може бути доповнена варіантом використання "Оформити замовлення на купівлю комп'ютера" і акторами "Покупець комп'ютера" і "Продавець комп'ютерів", які пов'язані з відповідними компонентами діаграми відношення узагальнення.

Лекція 23. ДІАГРАМА СТАНІВ

Для більшості фізичних систем, крім самих простих і тривіальних, статичних уявлень зовсім недостатньо для моделювання процесів функціонування подібних систем як в цілому, так і їх окремих підсистем та елементів.

Для моделювання поведінки на логічному рівні в мові UML можуть використовуватися відразу кілька канонічних діаграм: станів, діяльності, послідовності та кооперації, кожна з яких фіксує увагу на окремому аспекті функціонування системи. На відміну від інших діаграм, діаграма станів описує процес зміни станів тільки одного класу, а точніше — одного екземпляру певного класу, тобто моделює всі можливі зміни у стані конкретного об'єкта. При цьому зміна стану об'єкта може бути викликана зовнішніми впливами з боку інших об'єктів чи ззовні. Саме для опису реакції об'єкта на подібні зовнішні впливи і використовуються діаграми станів.

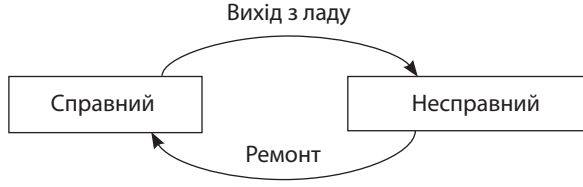
Головне призначення цієї діаграми — описати можливі послідовності станів і переходів, які в сукупності характеризують поведінку елемента моделі протягом його життєвого циклу. Діаграма станів представляє динамічну поведінку сутностей, на основі специфікації їх реакції на сприйняття деяких конкретних подій. Системи, які реагують на зовнішні дії від інших систем або від користувачів, іноді називають реактивними. Якщо такі дії ініціюються в довільні випадкові моменти часу, то говорять про асинхронну поведінку моделі.

Автомат

Автомат (state machine) у мові UML є деякий формалізм для моделювання поведінки елементів моделі та системи загалом. У метамоделі UML автомат, з одного боку, є пакетом, в якому визначено множину понять, необхідних для подання поведінки сутності, що моделюється у вигляді дискретного простору з кінцевим числом станів і переходів. З іншого боку, автомат описує поведінку окремого об'єкта у формі послідовності станів, які

охоплюють всі етапи його життєвого циклу, починаючи від створення об'єкта і закінчуючи його знищенням. Кожна діаграма станів представляє деякий автомат.

Найпростішим прикладом візуального представлення станів і переходів на основі формалізму автоматів може служити справність технічного пристрою, такого як комп'ютер. У цьому випадку вводяться у розгляд два найзагальніших стани: "справний" і "несправний", а також два переходи: "вихід з ладу" і "ремонт". Графічно ця інформація представлена у вигляді такої схеми станів комп'ютера:



Основними поняттями, що входять до формалізму автомата, є стан і перехід. Головна відмінність між ними полягає у тому, що тривалість перебування системи в окремому стані суттєво перевищує час, який витрачається на перехід з одного стану в інший. Передбачається, що час переходу з одного стану в інший дорівнює нулю (якщо додатково нічого не сказано). Інакше кажучи, перехід об'єкта зі стану в стан відбувається миттєво.

Формалізм звичайного автомата заснований на виконанні таких обов'язкових умов:

• автомат не запам'ятовує історію переміщення зі стану в стан;

• у кожний момент часу автомат може перебувати в одному і тільки в одному зі своїх станів;

• хоча процес зміни станів автомата відбувається у часі, але концепція часу не входить до формалізму автомата;

• кількість станів автомата повинно бути обов'язково кінцевим (у мові UML розглядаються тільки кінцеві автомати), і всі вони повинні бути специфіковані певним чином;

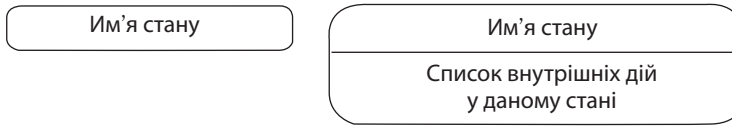
• граф автомата не повинен містити ізольованих станів і переходів;

• автомат не повинен містити конфліктуючих переходів, тобто таких переходів з одного і того самого стану, коли об'єкт водночас може перейти у два і більше наступних стани (крім випадку паралельних підавтоматів).

Стан

У мові UML під станом розуміється абстрактний метаклас, що використовується для моделювання окремої ситуації, протягом якої має місце виконання деякої умови. Стан може бути задано у вигляді набору конкретних значень атрибутів класу або об'єкта, при цьому зміна їх окремих значень буде відображати зміну стану модельованого класу або об'єкта.

Не кожен атрибут класу може характеризувати його стан. Як правило, мають значення тільки такі властивості елементів системи, які відображають динамічний або функціональний аспект її поведінки. Стан на діаграмі зображується прямокутником з округленими вершинами.



а *б*

Назва стану

Ім'я стану представляє собою рядок тексту, що розкриває змістовний сенс цього стану. Назва завжди записується з великої літери. Оскільки стан системи є складовою процесу її функціонування, рекомендується в якості імені використовувати дієслова у теперішньому часі (дзвенить, друкує, очікує) або відповідні дієприкметники (зайнятий, вільний, передано, отримано).

Список внутрішніх дій

Ця секція містить перелік внутрішніх дій чи діяльностей, які виконуються у процесі знаходження модельованого елемента в

даному стані. Кожна з дій записується у вигляді окремого рядка і має такий вигляд:

<мітка-дії '/' вираз-дії>.

Мітка дії вказує на обставини або умови, за яких буде виконуватися діяльність, визначена виразом дії. При цьому вираз дії може використовувати будь-які атрибути і зв'язки, які належать області імен або контексту модельованого об'єкта. Якщо список виразів дії порожній, то роздільник у вигляді похилої риски '/' може не вказуватися.

Перелік міток дії має фіксовані значення у мові UML, які не можуть бути використані як імена подій. Ці значення такі:

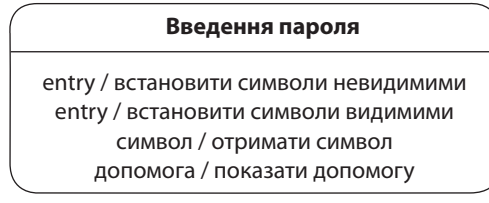
• entry — ця позначка вказує на дію, специфіковану наступним за нею виразом дії, що виконується в момент входу в даний стан (вхідна дія);

• exit — ця позначка вказує на дію, специфіковану наступним за нею виразом дії, що виконується в момент виходу з цього стану (вихідна дія);

• do — ця позначка специфікує виконувану діяльність ("do activity"), яка виконується протягом усього часу, поки об'єкт знаходиться в даному стані, або до тих пір, поки не закінчиться обчислення, специфіковане наступним за нею виразом дії. При завершенні дії генерується відповідний результат;

• include — ця позначка використовується для звернення до підавтомату, при цьому наступний за нею вираз дії містить ім'я цього підавтомату.

Як приклад стану розглянемо ситуацію введення пароля користувача при аутентифікації входу в деяку програмну систему. У цьому випадку список внутрішніх дій в цьому стані не порожній і включає 4 окремих дії, перші дві з яких стандартні та описані вище, а дві останні визначаються своєю специфікацією.



Початковий стан

Початковий стан — це окремий випадок стану, який не містить ніяких внутрішніх дій (псевдостан). У цьому стані знаходиться об'єкт за замовчуванням у початковий момент часу. Він служить для вказівки на діаграмі станів графічної області, від якої починається процес зміни станів. Графічно початковий стан у мові UML позначається у вигляді зафарбованою кола.

початковий стан
а

кінцевий стан
б

Кінцевий стан

Кінцевий (фінальний) стан — це окремий випадок стану, який також не містить ніяких внутрішніх дій (псевдостанів). У цьому стані буде знаходитися об'єкт за замовчуванням після завершення роботи автомата у кінцевий момент часу. Він служить для вказівки на діаграмі графічної області, в якій завершується процес зміни станів або життєвий цикл цього об'єкта.

Перехід

Простий перехід (simple transition) — це відношення між двома послідовними станами, яке вказує на факт зміни одного стану іншим. Перебування модельованого об'єкта у першому стані може супроводжуватися виконанням певних дій, а перехід у другий стан буде можливий після завершення цих дій, а також після задоволення деяких додаткових умов.

Перехід здійснюється при настанні певної події: закінчення виконання діяльності (do activity), отримання об'єктом повідомлення або прийом сигналу. На переході зазначається ім'я події. Крім того, на переході можуть зазначатися дії, здійснювані об'єктом у відповідь на зовнішні події при переході з одного стану в інший . Спрацьовування переходу може залежати не тільки від настання певної події, а й від виконання певної умови

(вартової умови). Об'єкт перейде з одного стану в інший у тому випадку, якщо відбулася вказана подія і вартова умова прийняла значення "істина".

На діаграмі станів перехід зображується суцільною лінією зі стрілкою, яка спрямована у цільовий стан. Кожен перехід позначений рядком тексту, що має наступний загальний формат:

<сигнатура події >'['< вартова умова> ']' <вираз дії>.

При цьому сигнатура події описує деякі подія з необхідними аргументами:

<ім'я події>'('<список параметрів, розділених комами >')'.

Подія

Подія — це специфікація деякого факту, що має місце в просторі і часі. У мові UML події відіграють роль стимулів, які ініціюють переходи з одних станів в інші. Як події можна розглядати сигнали, виклики, закінчення фіксованих проміжків часу або моменти закінчення виконання певних дій. Назва події ідентифікує кожен окремий перехід на діаграмі станів і може містити рядок тексту, що починається з малої літери.

Вартова умова

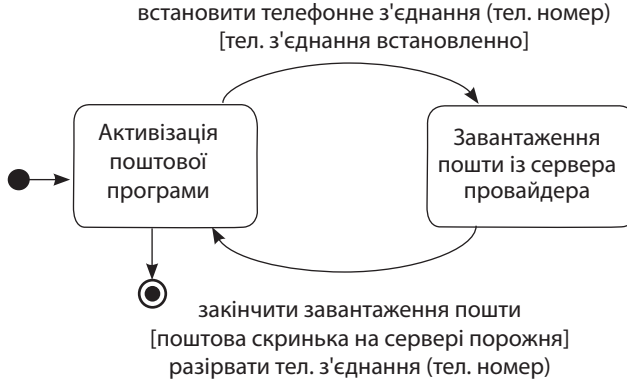
Вартова умова (guard condition), якщо воно є, завжди записується у прямих дужках після події-тригера і є деяким булевським виразом.

Якщо вартова умова приймає значення "істина", то відповідний перехід може спрацювати, внаслідок чого об'єкт перейде у цільовий стан. Якщо ж вартова умова приймає значення "хиба", то перехід не може спрацювати, і при відсутності інших переходів об'єкт не може перейти у цільовий стан з цього переходу. Однак обчислення істинності вартової умови відбувається тільки після виникнення асоційованої з ним події-тригера, яка ініціює відповідний перехід.

Прикладом події-тригера може служити розрив телефонного з'єднання з провайдером Інтернет-послуг після закінчення

завантаження електронної пошти клієнтською поштовою програмою (при віддаленому доступі до Інтернету). У цьому випадку вартова умова є не що інше, як відповідь на питання: “Чи пустять поштову скриньку клієнта на сервері провайдера?”. У разі позитивної відповіді “істина”, слід відключити з’єднання з провайдером, що і робить автоматично поштова програма-клієнт. У випадку негативної відповіді “хиба”, мають залишатись у стані завантаження пошти і не розривати телефонне з’єднання.

Однак за необхідності отримати нову пошту, користувач повинен встановити телефонне з’єднання з провайдером, що і показано на діаграмі верхнім переходом. Інакше кажучи, користувач ініціює подію-тригер “встановити телефонне з’єднання”. Як параметр цієї події виступає конкретний номер телефону модемного пулу провайдера. Далі йде перевірка вартової умови “телефонне з’єднання встановлено”, яке слід розуміти як питання. Тільки у випадку позитивної відповіді “так”, тобто “істина”, відбувається перехід поштового програми-клієнта зі стану “активізація поштової програми” у стан “завантаження пошти з сервера провайдера”.



Другий тригерний перехід на діаграмі ініціює автоматичний розрив телефонного з’єднання з провайдером після закінчення завантаження пошти на комп’ютер користувача. У цьому випад-

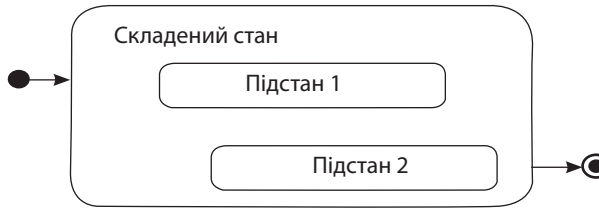
ку подія-тригер “закінчити завантаження пошти” відбувається після перевірки вартової умови “поштова скринька на сервері порожня”, що також слід розуміти у формі питання. При позитивній відповіді на це питання (вся пошта завантажена або її просто немає в ящику) поштова програма припиняє завантаження пошти і переходить у стан активізації. У разі ж негативної відповіді завантаження пошти буде продовжено.

Вираз дії

Вираз дії (action expression) виконується тільки у тому випадку, коли перехід спрацьовує. Це — атомарна операція (просте обчислення), що виконується після спрацьовування відповідного переходу до початку будь-яких дій у цільовому стані. Атомарність дії означає, що вона не може бути перервана ніякою іншою дією доти, поки не закінчиться її виконання. Ця дія може впливати як на сам об’єкт, так і на його оточення, якщо це з очевидністю випливає з контексту моделі. Вираз записується після знака “/” у рядку тексту, приєднаного до відповідного переходу.

Складений стан і підстани

Складений стан (composite state) — такий складний стан, що складається з інших вкладених у нього станів. Останні будуть виступати відносно першого як підстани (substate). Хоча між ними має місце відношення композиції, графічно всі вершини діаграми, які відповідають вкладеним станам, зображуються всередині символу складного стану.



Складений стан може містити два чи більше паралельних підавтомати або кілька послідовних підстанів.

Послідовні підстани

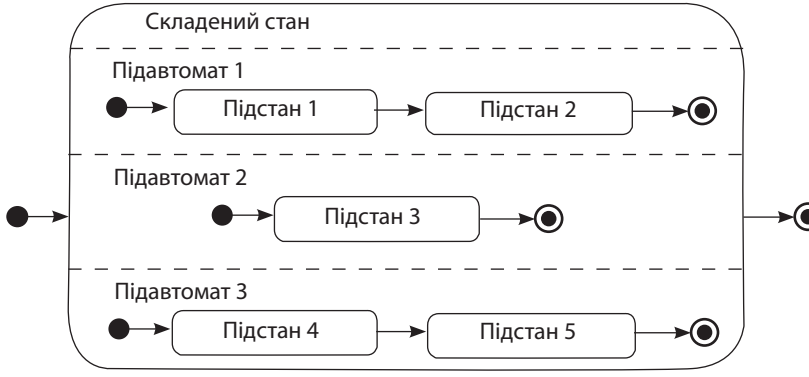
Послідовні підстани (sequential substates) використовуються для моделювання такої поведінки об'єкта, під час якої в кожний момент часу об'єкт може знаходитися тільки в одному підстані. Поведінка об'єкта у цьому випадку — це послідовна зміна підстанів, починаючи від початкового і закінчуючи кінцевим підстаном.

Для прикладу розглянемо в якості модельованого об'єкта звичайний телефонний апарат. Він може перебувати в різних станах, одним з яких є стан дозвону до абонента.

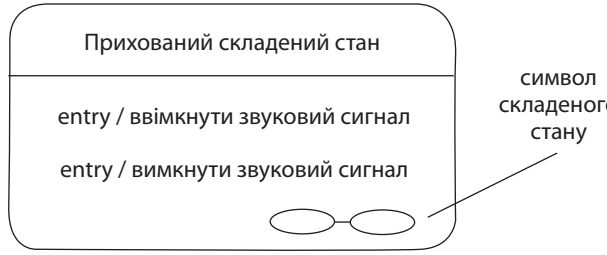
Паралельні підстани

Паралельні підстани (concurrent substates) дозволяють специфікувати два і більше підавтомати, які можуть виконуватися паралельно всередині складеної події. Кожен із підавтоматів займає деяку область (регіон) всередині складеного стану, яка відокремлюється від решти горизонтальною пунктирною лінією. Якщо на діаграмі станів є складений стан з вкладеними паралельними підстанами, то об'єкт може одночасно перебувати в кожному з цих підстанів.

Окремі паралельні підстани можуть, у свою чергу, складатися з кількох послідовних підстанів. У цьому випадку за визначенням об'єкт може знаходитися тільки в одному з послідовних підстанів підавтомата.

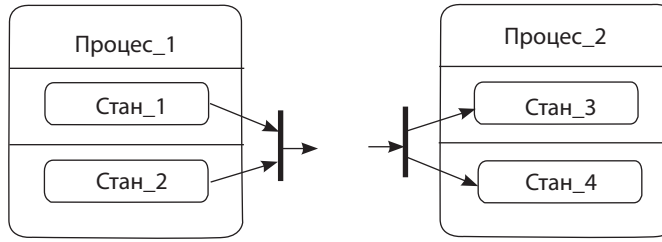


У деяких випадках бажано приховати внутрішню структуру складеного стану. Наприклад, окремий підавтомат, що специфікує складений стан, може бути настільки великим за масштабом, що його візуалізація утруднить загальне уявлення діаграми станів. У подібній ситуації допускається не розкривати на вихідній діаграмі станів дані складеного стану, а вказати у правому нижньому кутку спеціальний символ-піктограму.



Складні переходи. Переходи між паралельними станами

В окремих випадках перехід може мати кілька станів-джерел і кілька цільових станів. Такий перехід отримав спеціальну назву — паралельний перехід. Введення у розгляд паралельного переходу зумовлено необхідністю синхронізувати та/або розділити окремі підпроцеси на паралельні нитки без специфікації додаткової синхронізації у паралельних підавтоматах.

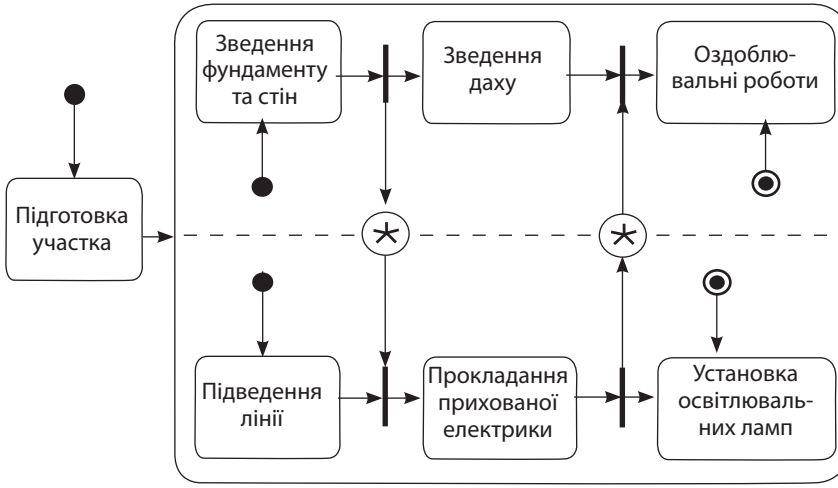


Синхронізуючі стани

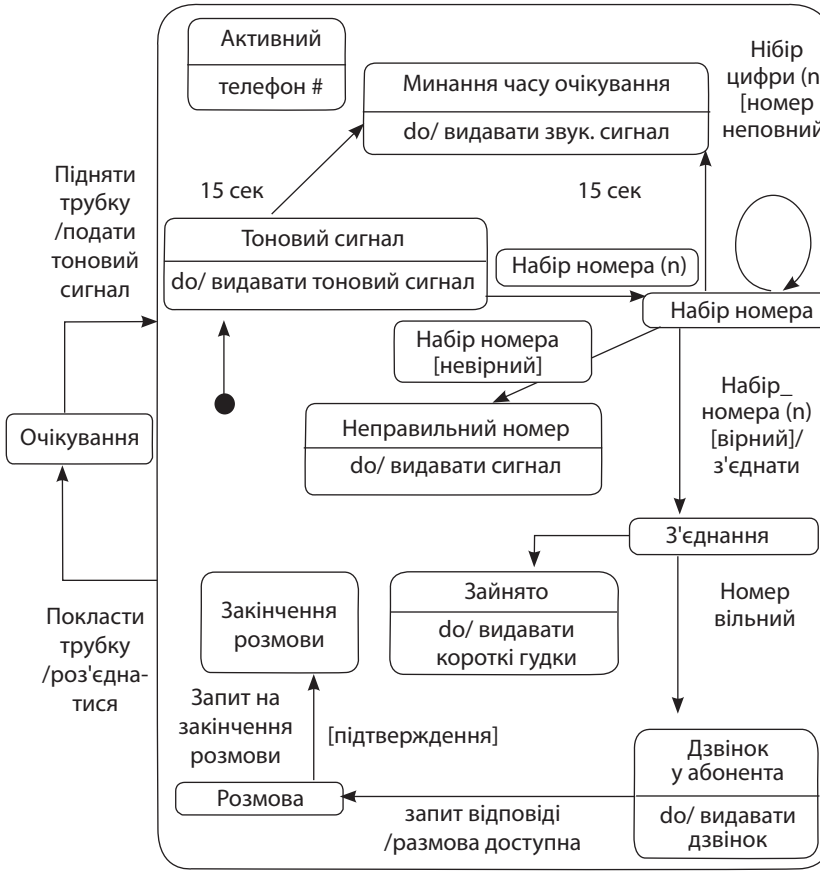
В окремих випадках може виникнути необхідність врахувати в моделі синхронізацію настання окремих подій. Для цієї мети в мові UML є спеціальний псевдостан, який називається синхронізуючим станом.

Синхронізуючий стан (synch state) позначається невеликим колом, всередині якого знаходиться символ зірочки "*". Він використовується спільно з переходом-з'єднанням або переходом-розгалуженням для того, щоб явно вказати події в інших підавтоматах, які безпосередньо впливають на поведінку цього підавтомата.

Для ілюстрації використання синхронізуючих станів розглянемо спрощену ситуацію з моделюванням процесу будівництва будинку.



Розглянемо, як приклад, діаграму станів, що є прикладом моделювання поведінки конкретного об'єкта — процесу функціонування телефонного апарата.


Активний
телефон #
Минання часу очікування
do/ видавати звук. сигнал
Нібір цифри (n) [номер неповний]
15 сек
15 сек
Підняти трубку /подати тоновий сигнал
Тоновий сигнал
do/ видавати тоновий сигнал
Набір номера (n)
Набір номера
Набір номера [невірний]
Неправильний номер
do/ видавати сигнал
Набір_ номера (n) [вірний]/ з'єднати
Очікування
З'єднання
Закінчення розмови
Зайнято
do/ видавати короткі гудки
Номер вільний
Покласти трубку /роз'єдна-тися
Запит на закінчення розмови
[підтвердження]
Дзвінок у абонента
do/ видавати дзвінок
Розмова
запит відповіді /размова доступна

Лекція 24. ДІАГРАМА ПОСЛІДОВНОСТІ

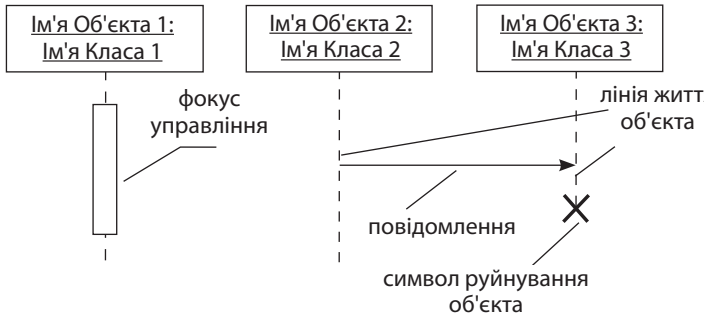
У мові UML взаємодія елементів розглядається в інформаційному аспекті їх взаємодії, тобто взаємодіючи об'єкти обмінюються між собою деякою інформацією. При цьому інформація набуває форми закінчених повідомлень. Інакше кажучи, хоча повідомлення і має інформаційний зміст, воно набуває додаткової властивості впливати на свого одержувача.

Взаємодію об'єктів можна розглядати у часі, і тоді для подання часових особливостей передачі та прийому повідомлень між об'єктами використовується діаграма послідовності.

Об'єкти

На діаграмі послідовності зображуються виключно ті об'єкти, які безпосередньо беруть участь у взаємодії і не показують можливі статичні асоціації з іншими об'єктами. Для діаграми послідовності ключовим моментом є саме динаміка взаємодії об'єктів у часі.

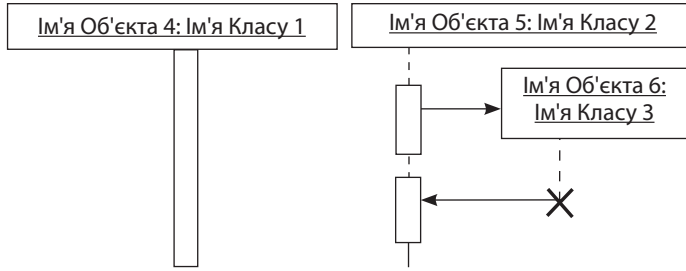
Діаграма послідовності має два виміри. Один вимір — зліва направо у вигляді вертикальних ліній, кожна з яких зображує лінію життя окремого об'єкта, який бере участь у взаємодії.



Крайнім ліворуч на діаграмі зображується об'єкт, який є ініціатором взаємодії. Правіше зображується інший об'єкт, який безпосередньо взаємодіє з першим. Таким чином, всі об'єкти на діаграмі послідовності утворюють деякий порядок, який визначається ступенем активності цих об'єктів при взаємодії один з одним.

Другий вимір діаграми послідовності — вертикальна тимчасова вісь, спрямована зверху вниз. Початковому моменту часу відповідає верхня частина діаграми. При цьому взаємодії об'єктів реалізуються за допомогою повідомлень, які надсилаються одними об'єктами іншим. Повідомлення зображуються у вигляді горизонтальних стрілок з ім'ям повідомлення та утворюють порядок з часу свого виникнення.

Не обов'язково створювати всі об'єкти у початковий момент часу. Окремі об'єкти в системі можуть створюватися в міру необхідності, істотно економлячи ресурси системи і підвищуючи її продуктивність. У цьому випадку прямокутник такого об'єкта зображується не у верхній частині діаграми послідовності, а в тій її частині, що відповідає моменту створення об'єкта.

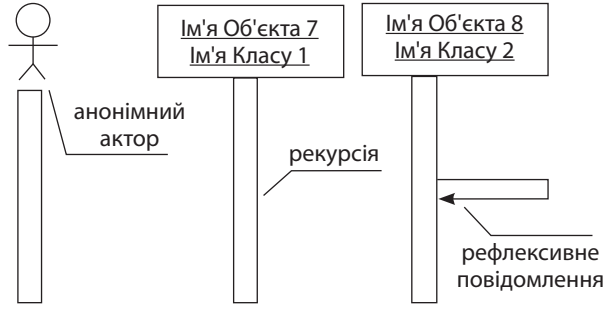


Фокус керування

У процесі функціонування об'єктно-орієнтованих систем, з одного боку, одні об'єкти можуть перебувати в активному стані, безпосередньо виконуючи певні дії, або в стані пасивного очікування повідомлень від інших об'єктів. Щоб явно виділити таку активність об'єктів, у мові UML застосовується спеціальне поняття, що отримало назву фокуса керування (focus of control). Фокус керування зображується у формі витягнутого вузького прямокутника.

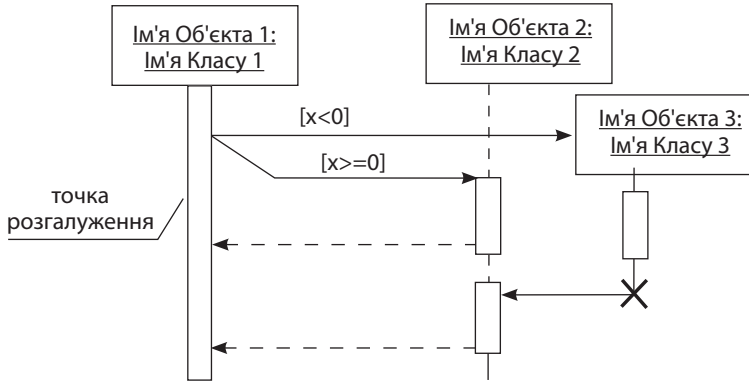
З іншого боку, періоди активності об'єкта можуть чергуватися з періодами його пасивності чи очікування. У цьому випадку у такого об'єкта є кілька фокусів керування.

В окремих випадках ініціатором взаємодії у системі може бути актор або зовнішній користувач. У цьому випадку актор зображується на діаграмі послідовності найпершим об'єктом зліва зі своїм фокусом керування.



Повідомлення

Кожна взаємодія описується сукупністю повідомлень, якими об'єкти обмінюються між собою. У цьому сенсі повідомлення (message) — це закінчений фрагмент інформації, який відправляється одним об'єктом іншому. При цьому прийом повідомлення ініціює виконання певних дій, спрямованих на вирішення окремої задачі тим об'єктом, якому це повідомлення надіслано. Розглянемо приклад:



Стереотипи повідомлень

У мові UML передбачені деякі стандартні дії, що виконуються у відповідь на отримання відповідного повідомлення. Вони можуть бути явно зазначені на діаграмі послідовності у формі стереотипу поряд з повідомленням, до якого вони відносяться. У цьому випадку вони записуються в лапках. Використовуються такі позначення для моделювання дій:

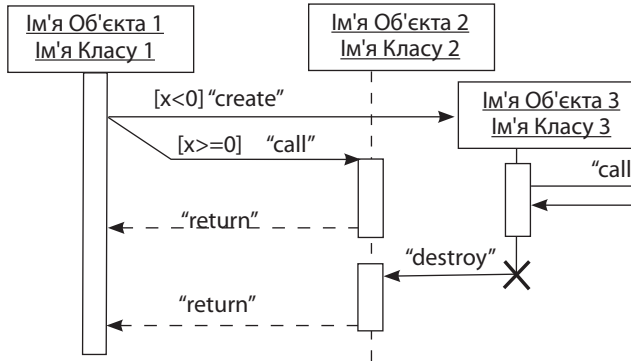
• “call” (викликати) — повідомлення, що вимагає виклику операції або процедури об’єкта, який приймає. Якщо повідомлення з цим стереотипом рефлексивне, то воно ініціює локальний виклик операції в об’єкта, який послав це повідомлення;

• “return” (повернути) — повідомлення, що повертає значення виконаної операції або процедури, викликаної її об’єктом. Значення результату може ініціювати розгалуження потоку керування;

• “create” (створити) — повідомлення, що вимагає створення іншого об’єкта для виконання певних дій. Створений об’єкт може отримати фокус керування, а може і не отримати його;

• “destroy” (знищити) — повідомлення з явною вимогою знищити відповідний об’єкт. Посилається у тому випадку, коли необхідно припинити небажані дії з боку існуючого в системі об’єкта, або коли об’єкт більше не потрібен і повинен звільнити задіяні ним системні ресурси;

• “send” (послати) — позначає посилку об’єктом деякого сигналу, який асинхронно ініціюється одним об’єктом і приймається



(перехоплюється) іншим. Відмінність сигналу від повідомлення полягає у тому, що сигнал повинен бути явно описаний у тому класі, об'єкт якого ініціює його передачу.

Тимчасові обмеження на діаграмах послідовності

В окремих випадках виконання тих чи інших дій на діаграмі послідовності може вимагати явної специфікації тимчасових обмежень, що накладаються на сам інтервал виконання операцій або передачу повідомлень. У мові UML для запису тимчасових обмежень використовуються фігурні дужки.

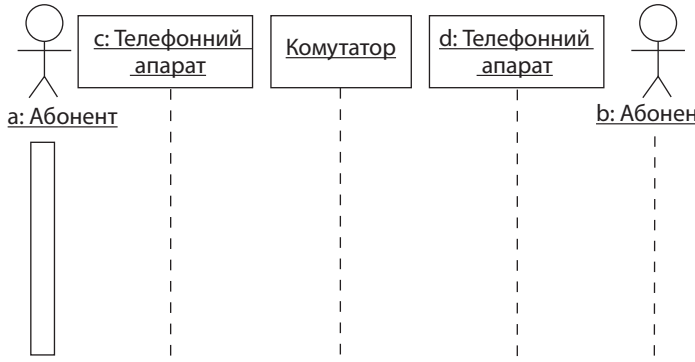
Приклад:

(час_прийому_повідомлення час_відправки_повідомлення<1 с);
(час_очікування_відповіді <5 с);
(час_передачі_пакета <10 с);
(об'єкт_1. час_подачі_сигналу_тривоги> 30 с).

Приклад побудови діаграми послідовності

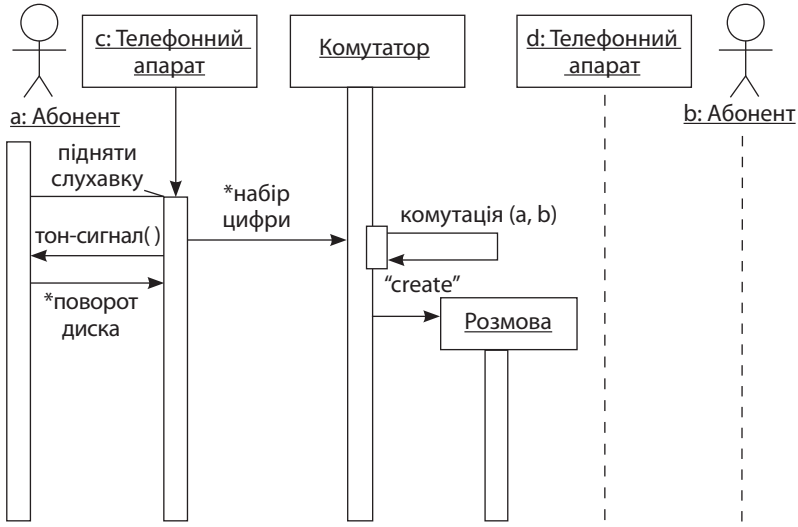
Як приклад розглянемо побудову діаграми послідовності для моделювання процесу телефонної розмови з використанням звичайної телефонної мережі. Об'єктами в цьому прикладі є: два абоненти a і b, два телефонних апарата c і d, комутатор та сама розмова як об'єкт моделювання. При цьому як комутатор, так і розмова є анонімними об'єктами.

Отже, маємо:



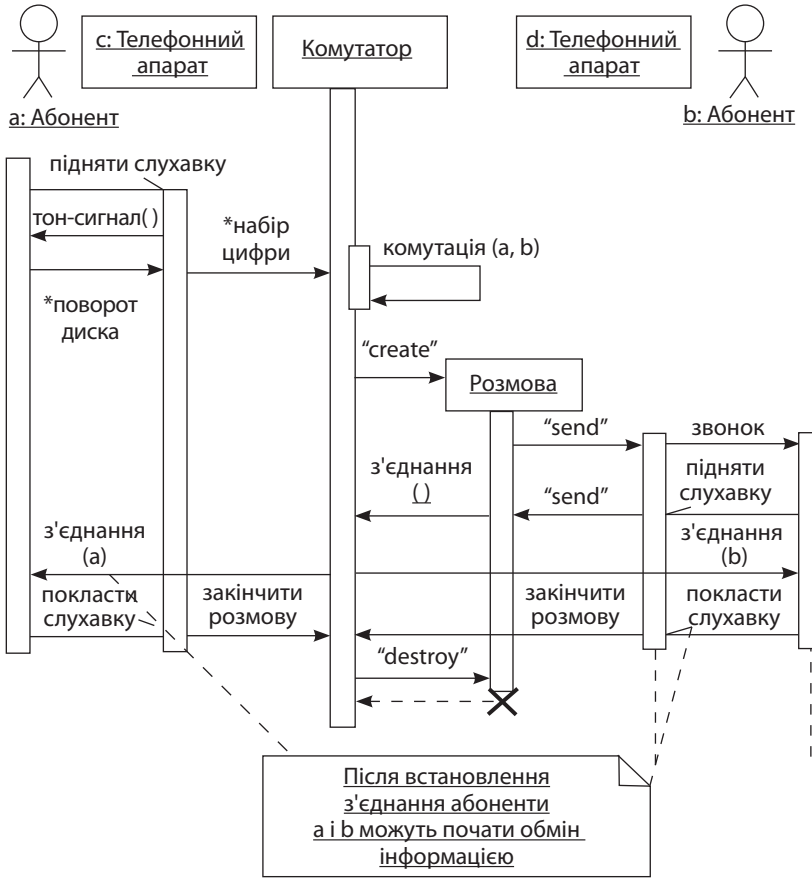
Процес взаємодії у цій системі починається з підняття трубки телефонного апарата першим абонентом. Тим самим він посилає повідомлення з телефонного апарата, що переводить цей апарат в активний стан і викликає дію — подачу тонового сигналу в телефонну трубку для першого абонента. Наступний крок також ініціюється першим абонентом — набір цифр телефонного номера. Це представлено у формі ітеративного повідомлення зі знаком "*" ліворуч від його імені.

Зауважимо, що підняття телефонної трубки і набір цифр номера є фізичними діями і тому зображуються у формі простих асинхронних повідомлень. Після набору цифр номера телефону апарат рекурсивно викликає процедуру посилки комутаційних імпульсів на комутатор. Останній ініціює створення нового об'єкта в системі, що моделюється — телефонної розмови. Отже, уточнена діаграма послідовності буде мати вигляд:



Після створення анонімний об'єкт "розмова" відразу отримує фокус активності та посилає повідомлення телефонного апарата *d* на виконання дії — дзвінок. При цьому другий абонент знімає трубку (асинхронне повідомлення), тим самим

встановлюється пряме з'єднання між абонентами *a* і *b*. Після того як абоненти покладуть трубки, розмова закінчується. Тим самим об'єкт "розмова" знищується. Остаточний варіант діаграми послідовності може містити деякі часові обмеження та коментарі.



Діаграма діяльності (activity diagram)

При моделюванні поведінки проектованої або аналізованої системи виникає необхідність не тільки уявити процес зміни її

станів, а й деталізувати особливості алгоритмічної і логічної реалізації виконуваних системою операцій.

Для моделювання процесу виконання операцій у мові UML використовуються так звані діаграми діяльності. Застосована у них графічна нотація багато в чому схожа на нотацію діаграми станів, оскільки на діаграмах діяльності також присутні позначення станів і переходів.

Стан дії

Стан дії (action state) є спеціальним випадком стану з деякими вхідними діями і принаймні одним переходом, що виходить зі стану. Цей перехід неявно припускає, що вхідна дія вже завершилась. Стан дії не може мати внутрішніх переходів, тому що вона є елементарною. Звичайне використання стану дії полягає у моделюванні одного кроку виконання алгоритму (процедури) або потоку управління.

Графічно стан дії зображується фігурою, яка нагадує прямокутник. Всередині цієї фігури записується вираз дії (action-expression), яка має бути унікальною в межах однієї діаграми діяльності.

проста дія
а

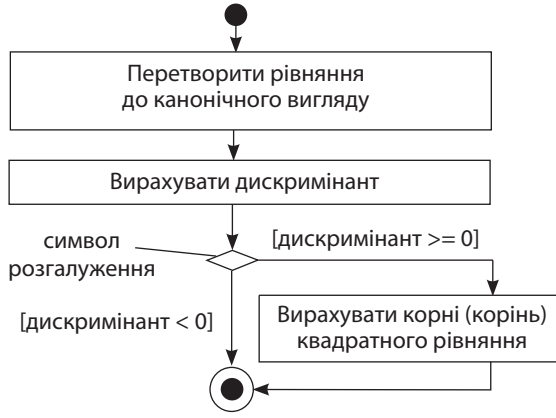
вираз
б

Перехід

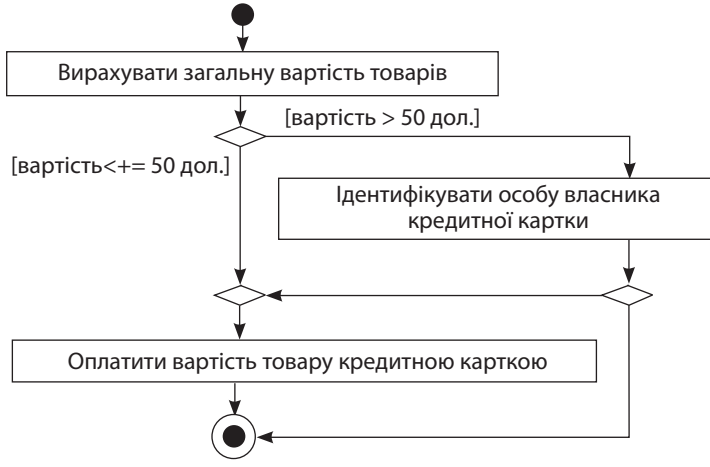
При побудові діаграми діяльності використовуються тільки нетрігерні переходи, тобто такі, які спрацьовують одразу після завершення діяльності або виконання відповідної дії. Цей перехід переводить діяльність у подальший стан відразу, як тільки закінчиться дія у попередньому стані. На діаграмі такий перехід зображується суцільною лінією зі стрілкою.

Як приклад розглянемо фрагмент відомого алгоритму знаходження коренів квадратного рівняння. У загальному випадку після приведення рівняння другого ступеня до канонічного ви-

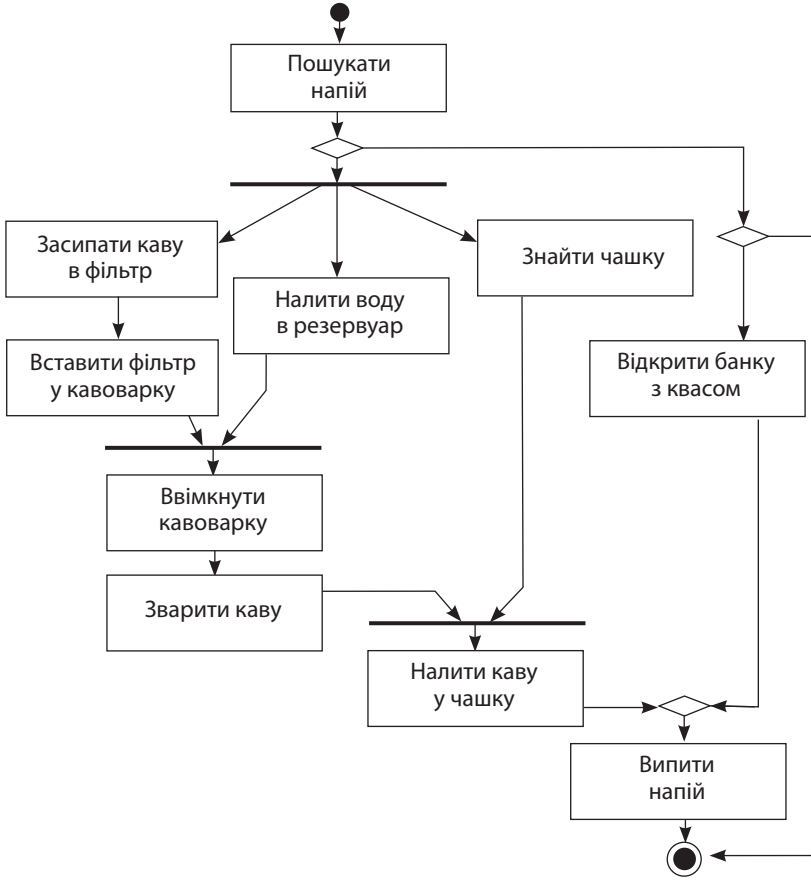
гляду: a • x • x + b • x + c = 0 необхідно обчислити його дискримінант. Діаграма діяльності має вигляд:



У наступному прикладі розраховується загальна вартість товарів, що купуються за кредитною карткою у супермаркеті. Якщо ця вартість перевищує 50 дол., то виконується аутентифікація особи власника картки. У разі позитивної перевірки (картка дійсна), або якщо вартість товарів не перевищує 50 дол., відбувається зняття суми з рахунка і оплата вартості товарів. При негативному результаті (картка недійсна) оплата не відбувається і товар залишається у продавця.



Для ілюстрації особливостей паралельних процесів виконання дій розглянемо приклад з приготуванням напою.



Діаграми діяльності можуть бути використані не тільки для специфікації алгоритмів обчислень або потоків управління у програмних системах.

СПИСОК ЛІТЕРАТУРИ

1. *Липпман С., Лажойе Ж.* Язык программирования C++. Вводный курс. — М.: ДМК, 2001. —1104 с.

2. *Страуступ Б.* Язык программирования C++. Специальное издание. — М.: БИНОМ, 2001. — 420 с.

3. *Саттер Г.* Решение сложных задач на C++. — М.: Изд. дом "Вильямс", 2002. — 395 с.

4. *Мацяшек Л.* Анализ требований и проектирование систем. — М.: Изд. дом "Вильямс", 2002. — 428 с.

5. *Влиссидес Дж.* Применение шаблонов проектирования. Дополнительные штрихи. — М.: Изд. дом "Вильямс", 2003 — 144 с.

6. *Павловская Т. А.* C/C++ Программирование на языке высокого уровня. — СПб.: Питер, 2007. — 461 с.

7. *Лаптев В. В., Морозов А. В., Бокова А. В.* C++ объектно-ориентированное программирование. Задачи и упражнения. — СПб.: Питер, 2007. — 287 с.

ДОДАТОК

ЗАДАЧІ

Процедурне програмування

1. Дано 36-значне число, що містить не більше 100 цифр (цифри 10, 11, ..., 35 кодуються заголовними латинськими літерами *A*, *B*, ..., *Z*). Переставити цифри числа таким чином, щоб воно стало "щасливим". "Щасливим" будемо називати число з *N* цифр, у якого сума перших [*N*/2] цифр дорівнює сумі останніх [*N*/2] цифр. Якщо така перестановка неможлива, вивести повідомлення "impossible".

2. Написати програму, що виконує такі функції:

- введення послідовності;
- перегляд даних;
- пошук заданого елемента послідовності;
- сортування методом включення.

3. Обчислити визначений інтеграл методами Сімпсона й трапецій із заданою точністю.

Інтеграл від ***a*** до ***b (sin(x) · x*** в ступені 1/7).

4. Обчислити методом Монте-Карло 2-кратний інтеграл ***f(x,y) = x·x + y·y.***

Область інтегрування: ***0.5 <= x <= 1, 0 <= y <= 2 x-1.***

5. Створити однозв'язний лінійний список (у кожного вузла 1 інформаційне поле типу Integer) з можливістю додавати й видаляти вузли.

Після завершення редагування списку запросити число *N* і розбити список на два інших, не змінюючи розташування елементів у пам'яті, у такий спосіб: якщо значення інформаційного поля вузла > *N*, включити його у 1 список, якщо ні, то включити його у 2 список.

Після завершення роботи всі списки знищити.

6. Програма підраховує число вершин на *n*-му рівні непорожнього дерева *T* (корінь вважається вершиною нульового рівня).

7. Реалізувати роботу динамічної структури ЧЕРГА:

- додавання елемента;

• видалення;

• сортування. У роботі використовувати два методи сортування: швидкий і вставками. Порівняти ефективність;

• пошук. Вивід на екран.

8. Інтерполяція:

• знайти багаточлен найменшого ступеня, що приймає у даних точках ***x*** задані значення ***y:***

x -1 2 9 10;

y 1 2 4 8;

• функція ***f(x)*** задана таблицею. Використовуючи інтерполяційний багаточлен Ньютона, визначити пропущене значення у вільній комірці цієї таблиці:

x -0,1 0,2 0,5 0,7 0,9 1,2

f(x) 0,02 -1,73 -1,62 -1,48 ? -0,97;

• функція ***f(x)*** задана таблицею своїх значень. Побудувати кубічний сплайн і оцінити значення функції у зазначеній точці ***x0***.

x -0.5 1.5 2.0 3.5 6.0 9.0

f(x) 2.5 6 -3.5 -7 -10 2

x0=2.5.

9. Зробити інтерполяцію для заданого довільного набору вузлів і значень функції в них за допомогою полінома Лагранжа.

Визначити значення функції у будь-якій точці на вихідному інтервалі.

Вихідні дані (вузли й значення функції в них) зчитуються з файла.

10. Знайти власні значення матриці методом Крилова і Левер'є.

11. Знайти Ейлерів цикл у графі, що не містить вершин непарного ступеня і заданому списками інцидентності:

• граф задається списком інциндентності у текстовому файлі;

• кожний рядок містить пару: ребро та вершина;

• у файлі не повинно бути порожніх рядків;

• при пошуку циклу використовується пошук у глибину.

12. Текст програми на Паскалі зберігається у файлі на диску.

Скласти програму обробки тексту програми:

• перші букви службових слів зробити заголовними;

• текст коментарю замінити на номер коментарю один по одному.

Переписати текст програми в новий файл із мінімальною кількістю пробілів, зберігши пробіли тільки там, де вони необхідні.

13. Видалити з матриці [*A*] рядок і стовпчик, що містять найбільший елемент матриці. Матриця [*A*] є розрядженою і зберігається у вигляді мультисписків.

14. Написати програму, що видає для шахового поля заданого розміру всі можливі варіанти обходу його конем (тобто буквою "Г"). Наприклад:

1 14 9 20 3
16 21 2 25 10
13 8 15 4 19
22 17 6 11 24
7 12 23 18 5

15. Написати програму, що здійснює пірамідальне сортування масиву з використанням рекурсії.

16. Використовуючи метод пошуку завширшки, знайти найкоротший шлях від початкової до будь-якої довільної вершини зв'язного неорієнтованого графа, заданого списками інцидентності (ваги всіх ребер прийміть рівними одиниці).

17. Написати програму, що здійснює пошук у кореневому дереві піддерева, ізоморфних заданому дереву.

Використовувати алгоритм повного обходу дерева з бектрекінгом (backtracking).

18. Дано файл, компонентами якого є дійсні числа.

Знайти максимальне й мінімальне значення серед компонентів файла.

19. Використовуючи алгоритм пошуку в глибину, написати програму, що знаходить множину фундаментальних циклів зв'язного неорієнтованого графа, заданого таблицею інцидентності.

20. Використовуючи метод пошуку в ширину, знайти стягуюче дерево для довільного зв'язного неорієнтованого графа, заданого списками інцидентності.

21. Кожне число являє собою масив байт.

Перший байт масиву містить код знака числа: 0 — “+”, 225 — “-”, а інші байти — значущі двійкові цифри числа. У програмі розробити процедури алгебраїчних операцій із двома цілими числами, кожне з яких представлене масивом байтів.

Операції:

- плюс;
- мінус;
- множення;
- ділення на число.

При виконанні операцій враховується можливість переповнення.

22. У файлі *f1.txt* перебувають рядки тексту. Відкрити його для читання й построково читати з нього інформацію. У кожному рядку видалити всі цифри за допомогою зміщення. Потім отриманий рядок записати в інший файл *f2.txt*.

23. Спочатку вводяться рядки (кількість задана в константі ***k***). Потім береться рядок і ділиться на підрядки, роздільниками служать латинські літери. Після з отриманих підрядків шукаються ті підрядки, в яких є цифри й квадратні дужки. Якщо такі підрядки є, то в початковому рядку цього підрядка шукається перша цифра, після неї вставляються три зірочки ***, відбувається вихід із програми. Якщо такого підрядка немає, то переходимо до обробки наступного рядка.

24. Написати програму, що в діалоговому режимі створює таблицю, яка в свою чергу (теж у діалоговому режимі) вміє:

- створювати, додавати, видаляти стовпчики, рядки й комірки;
- читатися й записуватися у файл;
- редагувати стовпчики.

У стовпчиках можуть бути типи: integer, short, char, string, double, bool.

Типи можна змінювати й перетворювати за можливістю, тобто ті, які можливо. Наприклад, short в integer.

25. Написати програму, що форматує заданий у файлі текст у такий спосіб:

• текст вирівнюється по 50 символів у рядку;
• розставляються знаки переносу, де це необхідно;
• кожні 60 рядків виставляється номер сторінки.

26. Розробити програму формування черги, що містить цілі числа і впорядковує за зростанням елементів у цій черзі.

У процесі впорядкування елементи черги переміщуватися не повинні.

27. Розробити схему алгоритму й програму для шифрування даних у файлі методом гаммирування. Вихідний текст і ключ, на якому виробляється шифрування, перебувають у файлах. Результатом роботи програми є файл із зашифрованим текстом.

28. На основі даного тексту скласти частотний словник.

В окремий файл вивести список слів, кількість разів у тексті, в якому вони вживаються, і процентний вміст.

Впорядкувати за частотою вживаності.

29. Реалізувати алгоритм сортування вичерпуванням і оцінити його часову складність

30. Написати програму, що обчислює як ціле число значення виразу (без змінних), записане у постфіксній формі в текстовий файл ***postfix***.

Постфіксною формою запису вираження ***a^b*** називається запис, в якому знак операції розміщений за операндами: ***ab^***.

Наприклад:

a+ b-c це ab+ c-c-

a+b*c це bc*a+.

Об'єктно-орієнтоване програмування

1. Реалізувати клас шаблон "матриця", перевизначити операції додавання, віднімання, множення, транспонування. Реалізувати всередині класу також процедури виводу матриці на друк. Передбачити всі можливі виключення.

2. Реалізувати послідовність покажчиків на об'єкт. Послідовність повинна бути реалізована як окремий клас, передбачити всі характерні операції. (Послідовність може бути реалізована за допомогою списку або масиву.)

3. Описати клас “Кут” для вимірювання кутів.

Дані класу:

число градусів і хвилин.

Конструктори класу:

конструктор за замовчуванням, конструктор перетворення із цілого в градуси, конструктор довільного кута.

Визначити методи:

введення кута, друк на екран, визначення значення кута в радіанній мірі, один із методів повинен виконувати приведення, якщо дані задані некоректно.

Записати ціле і довільний кут як статичні об’єкти створеного типу, знайти значення кутів у радіанах.

Записати динамічний об’єкт за замовчуванням, ввести дані й надрукувати на екрані.

Записати масив із 2–3 кутів. Знайти спосіб задати значення при створенні об’єктів.

Розробити метод графічного відображення об’єкта на екрані.

Перевантажити операцію додавання кутів з використанням операції — методу класу, а операцію віднімання з використанням дружньої функції.

Перевантажити унарну операцію ++ для інкремента кута як метод класу; -- — як дружню функцію.

Перевантажити операцію присвоювання для кутів.

Перевантажити операцію порівняння = = для кутів.

Описати клас “Сектор”, похідний від класу “Кут”, як частину кола довільного радіуса між двома кутами. Визначити конструктор сектора.

Виведіть на екран площу сектора.

Для виводу повної інформації про об’єкт використовувати методи базового класу.

4. Написати максимально розширюваний проект множини **Set** (множина може містити різні типи, може бути реалізована на базі списку, масиву й т. д.).

5. Розробити безпечний код для класу **Stack**.

6. Дано наступний код:

```
    class A
{
public:
virtual ~A();
string Name(); private:
virtual  string DoName(); };
class  Bl  :  virtual  public A
string DoName(); };
class B2   :  virtual  public A
string DoName(); };
A::~A()  {}
string A::Name()        {return DoName();}
string A::DoName()    {return  "A";}
string Bl::DoName()   {return  "Bl";}
string B2::DoName()  {return  "B2";}
class D  :  public Bl,  public B2
string DoName()  {return  "D";}.
```

Написати еквівалентний код без використання множинного спадкування.

7. Написати програму, що демонструє роботу з об'єктами двох типів: **SymbString** символьний рядок (довільний рядок символів) і **OctString** восьмеричний рядок (зображення восьмеричного числа), для чого створити систему відповідних класів. Кожний об'єкт повинен мати ідентифікатор (у вигляді довільного рядка символів) і одне поле (або кілька) для зберігання стану об'єкта (один клас є нащадком іншого). Клієнту (функції main) повинні бути доступні такі основні операції (методи): створити об'єкт, видалити об'єкт, показати значення об'єкта та інші додаткові операції (залежать від варіанта). Операції по створенню й видаленню об'єктів інкапсулювати в класі **Factory**. Передбачити меню, що дає змогу продемонструвати задані операції.

За необхідністю в розроблювальні класи додаються додаткові методи (наприклад, конструктор копіювання, операція присвоювання й т. п.) для забезпечення належного функціонування цих класів. Проект повинен бути максимально розширюваним.

8. Потрібно створити шаблон деякого цільового класу **Vect**, можливо, реалізований із застосуванням деякого серверного класу **List**. Це означає, що об'єкт класу **List** використовується як елемент класу **Vect**. В якості серверного класу може бути використаний клас, створений програмістом. Можливі виключення повинні бути оброблені в програмі.

9. Написати гру "морський бій".

10. Розробити параметризований контейнерний клас **List**. Специфікація розміщується у файлах *Lists.h, List.h, ListIterator.h*.

11. Створити калькулятор, що обчислює введений арифметичний вираз на основі патерну **Composite**. У калькуляторі повинна бути процедура синтаксичного розбору вихідного виразу, результатом якої є структура даних типу дерева.

12. Проаналізувати специфікації й реалізувати варіанти обмеженого стека на базі масиву. Визначити клас циклічний масив і побудувати на його основі клас черга (файли *Stack, ProtectedArray, PeekBackstack, BoundedStackDerived FromArray, Bo-undedStackAgregatingArray, Array*).

13. Створити абстрактний базовий клас **Array** з віртуальними методами додавання і поелементної обробки масиву **foreach()**. Розробити похідні класи **AndArray** і **OrArray** (вибір). У першому класі операція додавання реалізується як перетинання множин, а поелементна обробка є обчислення квадратного кореня. У другому класі операція додавання реалізується як об'єднання, а поелементна обробка — обчислення логарифма.

14. Створити клас **ListPayment** (зарплата). У класі міститься список співробітників, для яких розраховується заробітна плата. Співробітник представлений класом **Person** з полями: табельний номер, прізвище, ім'я, по батькові, оклад, рік вступу на роботу, відсоток надбавки, прибутковий податок, кількість відпрацьованих днів у місяці, кількість робочих днів у місяці, нарахування, утримання. Реалізувати методи обчислення класу **Person**: нарахованої суми, утриманої суми, суми, видаваної на руки, і стажу. Стаж обчислюється як повна кількість років, що

пройшли від року влаштування на роботу, до поточного року. Нарахування — це сума, що нарахована за відпрацьовані дні й надбавки — частки від суми, нараховані за відпрацьовані дні. Утримання — це відрахування у пенсійний фонд (1 % від нарахованої суми) і прибутковий податок. Прибутковий податок становить 13 % від нарахованої суми без відрахувань у пенсійний фонд. Реалізувати методи додавання співробітника в список і видалення з нього; методи об'єднання списків; методи пошуку по полях класу **Person**. Реалізувати методи обчислення повних сум за списком: нараховано, утримано, на руки, прибутковий податок, пенсійний фонд. Реалізувати операцію генерації об'єкта **Group** (група), що містить список співробітників з однаковим стажем роботи, з об'єкта типу **ListPayment**.

15. Написати програму з використанням шаблону **Abstract-Factory**.
16. Написати програму з використанням шаблону **Builder**.
17. Написати програму з використанням шаблону **Decorator**.
18. Написати програму з використанням шаблону **Facade**.
19. Написати програму з використанням шаблона **Proxy**.
20. Написати програму з використанням шаблону **Strategy**.

ЗМІСТ

У конспекті лекцій розглядаються всі основні можливості мови С++ і їх застосування при розробці об'єктно-орієнтованих програм. Дається короткий опис бібліотек мови С++, необхідних для створення типових програм. Систематизовано викладаються основні поняття й описуються можливості мови С++.

Для студентів вищих навчальних закладів, які вивчають програмування на мові С++.

Навчальне видання

ГАЛКІН ОЛЕКСАНДР ВОЛОДИМИРОВИЧ
ВЕРЕС МАКСИМ МАКСИМОВИЧ

МОВА ПРОГРАМУВАННЯ С++

Конспект лекцій

Редактор *Н. К. Чумаченко*
Коректор *Т. К. Валицька*
Комп'ютерне верстання *Ю. Ю. Попова*
Оформлення обкладинки *О. О. Стеценко*

Підп. до друку 17.04.15. Формат 60x84/16. Папір офсетний. Друк офсетний.
Ум. друк. арк. 15,34. Обл.-вид. арк. 6,34. Наклад 1000 пр.

Міжрегіональна Академія управління персоналом (МАУП)
03039 Київ-39, вул. Фрометівська, 2, МАУП

ДП "Видавничий дім "Персонал"
03039 Київ-39, просп. Червонозоряний, 119, літ. ХХ

Свідоцтво про внесення до Державного реєстру суб'єктів видавничої справи ДК № 3262 від 26.08.2008 р.

Надруковано в друкарні ДП «Видавничий дім «Персонал»