МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВАСИЛЯ СТУСА

Ю.С. Антонов, Ю.В. Шамарін

Об’єктно-орієнтоване програмування

Методичні рекомендації до виконання індивідуальних завдань

Вінниця 2018

УДК 004.415 (072)
Авт. знак А724

Затверджено на засіданні вченої ради факультету математики та інформаційних технологій (протокол № 5 від «21» грудня 2017 р.)

Рецензенти:

Мічківський С.М. доцент, кандидат економічних наук, доцент кафедри прикладної математики і теорії систем управління

Парамонов А.І доцент, кандидат технічних наук, доцент кафедри комп'ютерних технологій

Ю.С. Антонов, Ю.В. Шамарін
Об'єктно-орієнтоване програмування. Методичні рекомендації до виконання індивідуальних завдань. – Вінниця: ДонНУ імені Василя Стуса, 2018. – 50 с.

Методичні вказівки містять загальну інформацію та терміни об'єктно-орієнтованого програмування, приклади розробки програм та варіанти індивідуальних завдань, критерії оцінювання та перелік запитань для підготовки до іспиту, відповідно до робочої програми дисципліни «Об'єктно-орієнтоване програмування».

Для студентів факультету математики та інформаційних спеціальностей «Прикладна математики» та «Комп'ютерні науки та інформаційні технології» усіх форм навчання.

Зміст

Вступ

Об’єктно-орієнтоване програмування (ООП) існує з 60-х років минулого сторіччя. Першою мовою програмування, в якій було запропоновано використовувати класи, об’єкти, віртуальні методи є Simula 67.

Існує точка зору, що ООП – це надмірність і воно нікому непотрібно, а з іншого боку існує точка зору, що ООП – це єдиний або самий правильний спосіб програмування.

ООП необхідно сприймати як інструмент, що дозволяє максимально ефективно розв’язувати певне коло задач. Як і будь-який інший інструмент, ООП потребує правильного використання – Ви же не будете намагатися забивати цвяхи викруткою? Можливо, раз або два в екстремальній ситуації, але не щоденно.

Підтримка ООП наразі реалізована у багатьох мовах програмування, ось деякі з них:

- C++,
- C#,
- Java,
- PHP,
- Python,
- Perl,
- Ruby,
- JavaScript,
- VBA,
- Fortran,
- Go.

Мови C++, PHP, Python, Perl, Ruby, JavaScript, Fortran, Go – дозволяють створювати програми як з використанням технології ООП, так і без неї. Мови C# та Java працюють виключно з ООП.

Дуже часто для того, щоб зрозуміти, чому зараз так, треба подивитись у минуле і зрозуміти, як це було раніше (рис. 1). Перші програми, що створювались на комп’ютерах, не мали навіть процедур або функцій один великий шматок коду, а для реалізації циклів використовувався оператор goto (або подібний). Потім з’явились цикли, і писати програми стало легше, а використання оператора goto поступово ставало поганим тоном. Потім розмір програм поступово збільшувався, і ніхто не хотів писати код повторно, і поступово з’явились функції[1]. З появою функцій змінилась логіка написання програм та їхня якість. Написавши один раз функцію, її потім можна було використовувати у різних програмах. У разі помилки треба було виправити лише цю функцію а не стрибати по коду як «мавпочка» з думкою «де я ще міг

[1] Процедури або підпрограми залежно від мови програмування

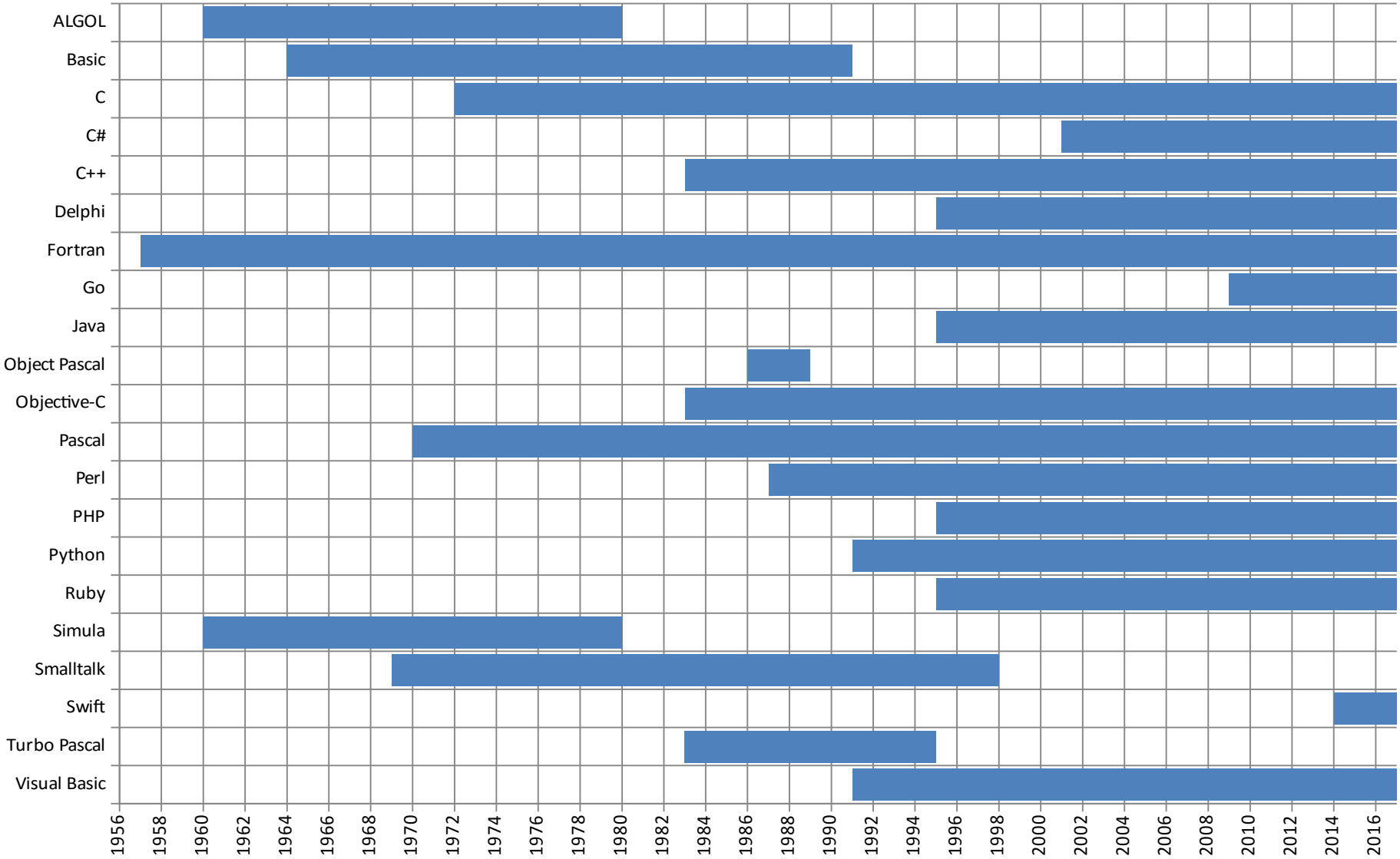

Рис. 1. – Періоди існування деяких мов програмування

таке написати?». З часом виявилось, що у дуже великих або складних програмах можливостей процедурних мов програмування вже не вистачає. До того ж стала важливою швидкість написання програм: якщо одну й ту саму програму можна написати двома мовами, і ці програми будуть працювати однаково, то програмісти оберуть ту, яка дозволить це зробити швидше. І тоді з'явились мови програмування, що підтримували ООП. Загалом еволюція мов програмування є дуже цікавою, при бажанні можна побачити, як одна мова впливала на інші, як при створенні нової мови намагалися взяти найкраще від вже існуючих, та позбавитись недоліків (рис. 2).

Наприклад, мови Assembler та C використовуються при створенні операційних систем та драйверів, що є цілком доцільним, оскільки вони дозволяють програмам працювати максимально швидко та ефективно. За допомогою C++ та класів можна створювати антивіруси, фаєрволи, браузери, системи управління базами даних.

ЗАУВАЖЕННЯ. У цій роботі під періодом існуванням слід розуміти проміжок часу, початок якого співпадає з роком створення мови програмування, а кінець – з роком, коли ця мова досить активно використовувалась востаннє. Слід зазначити, що кінець цього періоду може трактуватись спостерігачами по-різному [3].

ЗАУВАЖЕННЯ. Рисунок 2 ілюструє зв'язок лише частки мов програмування, що зумовлено великою кількістю мов програмування та обмеженими можливостями посібника. Тому не слід вважати, що C або LISP не зазнали впливу інших мов.

Мета цієї роботи – підійти максимально об'єктивно до процесу ООП, дізнатись його переваги та недоліки, навчитися правильно використовувати цю можливість.

Робота авторів була розподілена так: Антонов Ю.С. написав розділи «Створення класів та об'єктів», «Об'єктно-орієнтороване програмування на C++», «Довідкова інформація», «Завдання для індивідуальних робіт та приклади їх розв'язання» та параграф «Запитання для підготовки до іспиту»; Шамарін Ю. В. написав вступ та параграф «Критерії оцінювання та розподіл балів».

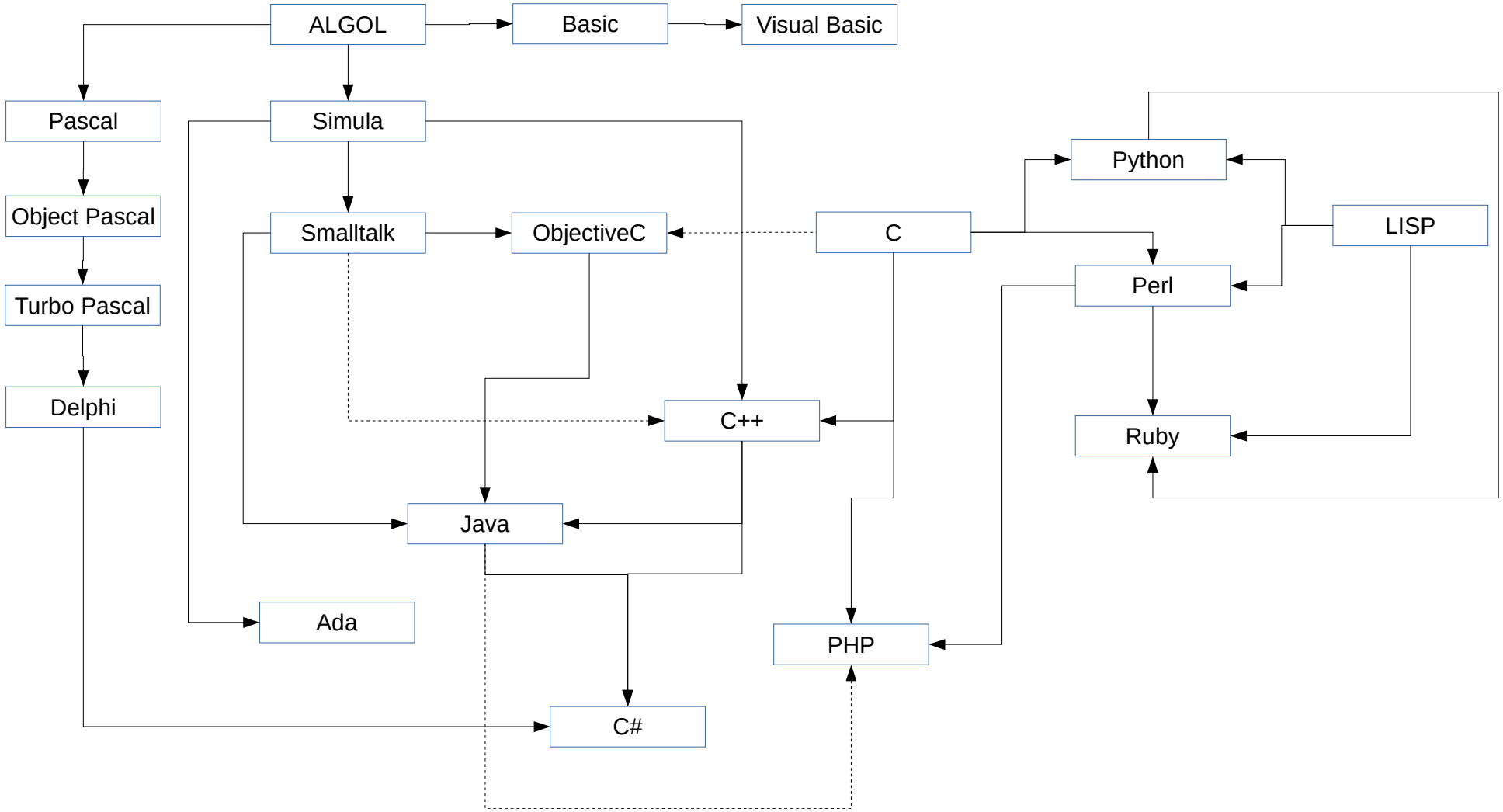

Рис. 2. – Вплив певних мов програмування одна на одну

1 «Створення класів та об’єктів»

Визначення класу та об’єкта

Обєктно-орієнтоване програмування **–** це методологія програмування, що базується на представленні програми як сукупності об’єктів, що взаємодіють між собою. Кожен із цих об’єктів є екземпляром певного класу, а класи є членами певної ієрархії наслідування.

Інкапсуляція, наслідування та поліморфізм є трьома стовпами об’єктно-орієнтованого програмування.

На сьогодні існує декілька визначень класу та об’єкта, які не суперечать одне одному, але трохи відрізняються. Ми з вами будемо використовувати такі визначення:

об’єкт **–** сутність, якою можна оперувати, має стан, поведінку та індивідуальність. Структура та поведінка схожих об’єктів визначається у спільному для них класі. Методи визначають як об’єкт взаємодіє з навколишнім середовищем [3]*;*

клас – множина об’єктів, що мають спільну (однакову) структуру, поведінку та семантику [3];

модульність – властивість системи, що розкладена на цілісні та слабо пов’язані між собою модулі [3].

абстракція – виділяє важливі характеристики об’єкту, що відрізняють його від усіх інших видів об’єктів, і чітко визначає концептуальні межі об’єкта з точки зору спостерігача [3].

Можна вважати, що клас – це схема будинку (автомобіля), а об’єкти – це конкретні будинки (автомобілі), що створюються за цією схемою.

У мові програмування C++ для опису класу використовується ключове слово ***class.*** Неповний опис класу здійснюється так:

```
class Ім’яКласу;
```

а повний:

```
class Ім’яКласу
{
  //тіло класу
};
```

Інкапсуляція та керування доступом

Інкапсуляція – це одна з фундаментальних концепцій об'єктно-орієнтированого програмування,

Інкапсуляція **–** це концепція об'єктно-орієнтированого програмування, що означає поєднання властивостей (атрибутів або даних) та методів в одне ціле (рис. 3). Деякі автори ототожнюють інкапсуляцію з приховуванням (англ. information hiding) [3], у той час як інші ні (відокремлюють ці визначення) [12].

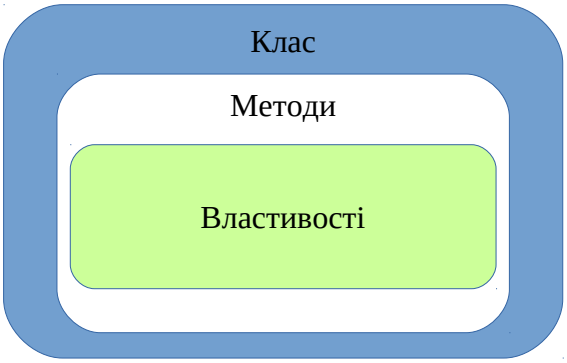

Рис. 3. – Інкапсуляція властивостей та методів у класі

Наприклад, клас, що відповідає точці на декартовій площині, можна записати так:

```
class CPoint
{
public:             //Специфікатор доступу
    double x;       //властивість (данні-член)
    double y;       //властивість (данні-член)
    double Distance(CPoint &p);//метод (функція-член)
};
```

У наведеному вище прикладі з'являється нове ключове слово – **public** – яке є специфікатором доступу. Всього таких специфікаторів три, а саме: **public**, **private**, **protected;** вони дозволяють керувати доступом до членів класу (рис. 4). Розглянемо більш детально призначення кожного з них:

public – відкритий блок класу, будь-який елемент, що знаходиться в цьому блоці, буде доступний у цьому класі, його нащадках, та зовнішнім операторам;

private – приватний (зачинений) блок класу, будь-який елемент, що знаходиться в цьому блоці, буде доступний лише в цьому класі. Використовується за замовчуванням;

protected – захищений блок класу, будь-який елемент, що знаходиться в цьому блоці, буде доступний в цьому класі та його нащадках.

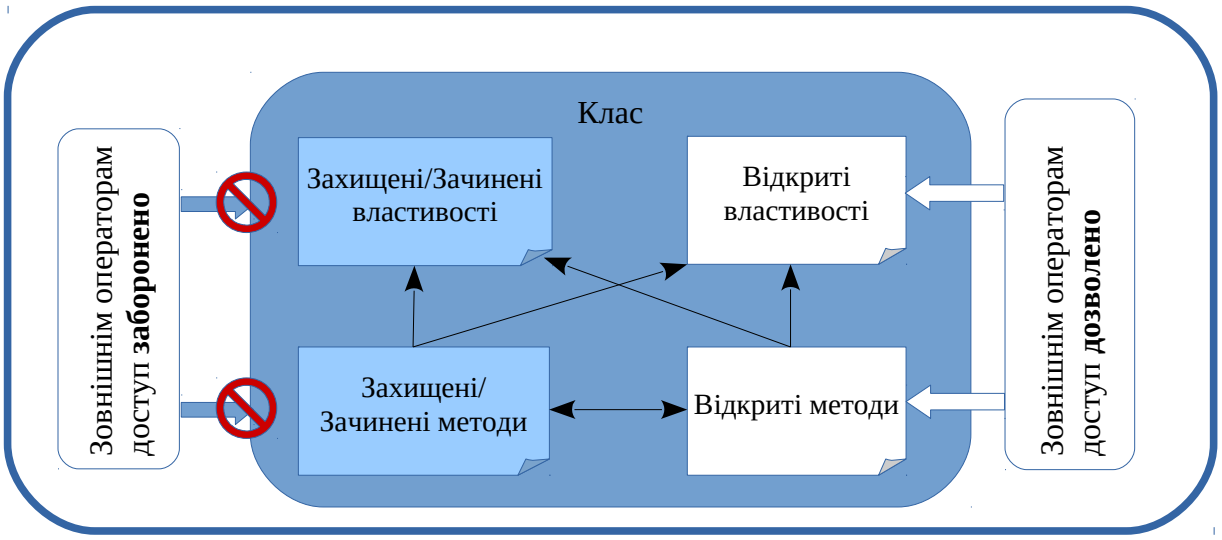

Рис. 4. – Керування доступом у класі

У класі можуть бути присутні численні відкриті, приватні або захищені секції, які можуть розташовуватись у довільному порядку, наприклад:

```
class CPerson
{
    string fname;          //приватний атрибут
public:
    void SetName(string);//відкритий метод
    string GetName();     //відкритий метод
private:
    string name;           //приватний атрибут
    string sname;          //приватний атрибут
public:
    string ToString();     //відкритий метод
    //.......
};
```

Зазвичай більшість класів краще оголошувати у заголовкових файлах, які потім можна додавати до різноманітних програмних модулів з метою спільного використання одних і тих самих класів.

ЗАУВАЖЕННЯ. Зазвичай для кожного поля певного об’єкта класу виділяється своя ділянка пам’яті. Це правило не діє, якщо оголошення поля супроводжується ключовим словом static. Пам’ять для статичних полів класу виділяється один раз, не залежно від того, скільки об’єктів існує у програмі.

ЗАУВАЖЕННЯ. Кожен об’єкт класу має вказівник *this,* який фактично посилається на поточний об’єкт.

Методи класу та робота з ними

Якщо ми у якійсь програмі плануємо працювати з часом, то досить зручно зробити це за допомогою класів. Такий клас має містити три атрибути – години, хвилини та секунди, а також методи, що дозволяють встановлювати, отримувати та змінювати час, а також конвертувати його у текстовий рядок. Один з варіантів оголошення та реалізації такого класу наведено відповідно у лістингах 1.1 та 1.2. Код головного модуля наведено у лістингу 1.3.

Лістинг 1.1. CTime.h (оголошення класу Ctime)

```
1: #ifndef CTIME_H
2: #define CTIME_H
3: #include <string>
4: #include <iostream>
5: #define UINT32 unsigned long int
6:
7: using namespace std;
8:
9: class CTime
10: {
11: private:
12:    UINT32 hours;
13:    UINT32 minutes;
14:    UINT32 seconds;
15: public:
16:    string ToString();
17:    void SetTime(UINT32 p_hours,UINT32 p_minutes,UINT32
          p_seconds);
18:    void SetTime(UINT32 p_hours,UINT32 p_minutes);
19:    void SetTime(UINT32 p_hours);
20:    void SetHours(UINT32 p_hours){hours=p_hours;};
21:    void SetMinutes(UINT32 p_minutes)
          {minutes=p_minutes;};
22:    void SetSeconds(UINT32 p_seconds)
          {seconds=p_seconds;};
23:    int GetHours(){return hours;};
24:    int GetMinutes(){return minutes;};
25:    int GetSeconds(){return seconds;};
26:    void AddSeconds(UINT32);
27:    void AddSeconds(float)=delete;
28:    void AddSeconds(double)=delete;
29: protected:
30: };
31: #endif // CTIME_H
```

Рядки 1, 2, 31 захищають включення змісту заголовкового файлу в один і той самий програмний модуль декілька разів. Оголошення класу CTime містить три приватних властивості, а саме: hours, minutes, seconds. У відкритій секції оголошені різноманітні методи класу, а саме:

- ToString() – метод що повертає час у вигляді рядку;
- SetTime(...) – метод, що перевантажений та дозволяє задавати значення приватним властивостям трьома різними способами;
- SetHours(...), SetMinutes(...), SetSeconds(...) – inline-методи, що дозволяють встановлення значень відповідних приватних полів;
- GetHours(), SetMinutes(), SetSeconds() – inline-методи, що дозволяють отримувати значення відповідних приватних полів;
- AddSeconds(...) – метод для збільшення часу на задану кількість секунд.

ЗАУВАЖЕННЯ. У C++ метод класу буде inline-методом, якщо він реалізується під час оголошення класу.

Якщо уважно подивитися на 26 – 28 рядки лістингу 1.1, то можна побачити, що ми наче перевантажуємо метод *AddSeconds*, але у рядках 27 – 28 після назви методу стоїть ключове слово ***delete.*** У стандарті ***C++11*** це означає, що ми забороняємо (видаляємо) використання вказаних методів.

Лістинг 1.2. CTime.cpp (реалізація класу CTime)

```
1: #include "CTime.h"
2: #include <sstream>
3:
4: string CTime::ToString()
5: {
6:     stringstream ss;
7:     if(hours<10)
8:         ss<<'0';
9:     ss << hours<<':';
10:     if(minutes<10)
11:         ss<<'0';
12:     ss << minutes<<':';
13:     if(seconds<10)
14:         ss<<'0';
15:     ss << seconds;
16:     return ss.str();
17: }
18: void CTime::SetTime(UINT32 p_hours,UINT32
          p_minutes,UINT32 p_seconds)
19: {
20:     hours=p_hours;
21:     minutes=p_minutes;
```

```
22:     seconds=p_seconds;
23: }
24: void CTime::SetTime(UINT32 p_hours,UINT32 p_minutes)
25: {
26:     hours=p_hours;
27:     minutes=p_minutes;
28:     seconds=0;
29: }
30: void CTime::SetTime(UINT32 p_hours)
31: {
32:     hours=p_hours;
33:     minutes=0;
34:     seconds=0;
35: }
36: void CTime::AddSeconds(UINT32 p_seconds)
37: {
38:   if((p_seconds+seconds)>59)
39:     {
40:       seconds=(seconds+p_seconds)%60;
41:       p_seconds/=60;
42:       if((p_seconds+minutes)>59)
43:       {
44:         minutes=(p_seconds+minutes)%60;
45:         p_seconds/=60;
46:         hours+=p_seconds;
47:       }
48:       else
49:         minutes=p_seconds+minutes;
50:     }
51:     else
52:       seconds=(seconds+p_seconds);
53: }
```

Лістинг 1.3. main.cpp (реалізація головної програми)

```
1: #include <iostream>
2: #include "CTime.h"
3:
4: using namespace std;
5:
6: int main()
7: {
8:     CTime time,*t2=new CTime;
9:     time.SetTime(12,8,4);
10:     cout << time.ToString()<< endl;
```

```
11:      time.AddSeconds(6721UL);
12:      cout<<time.ToString()<<endl;
13:      t2->SetTime(18,0,0);
14:      t2->AddSeconds(333UL);
15:      cout<<t2->ToString();
16:      return 0;
17: }
```

З лістингу 1.3 видно, як використовуються методи класу залежно від контексту (змінна або вказівник). Також у цій програмі буде неможливо викликати метод *AddSeconds* з параметрами 2.67, 3.57D тощо. Фактично, використовуючи інструкцію *=delete,* ми заборонили неявне перетворення типів до *unsigned long int,* тому якщо аргумент функції має інший тип, необхідно задавати перетворення типів явно, наприклад:

```
double ftime=23434.234;
t2->AddSeconds((UINT32)2.67);
t2->AddSeconds((UINT32)3.57D);
t2->AddSeconds((UINT32)ftime);
```

Метод SetTime можна було не перевантажувати тричі, а зробити методом з двома параметрами за замовчуванням. Для цього у лістингах 1.1 та 1.2 необхідно замінити рядки 17 та 18 відповідно на наступний рядок коду:

```
void SetTime(UINT32 p_hours,UINT32 p_minutes=0,UINT32
       p_seconds=0);
```

ЗАУВАЖЕННЯ. Насправді методи класу після компіляції будуть дещо відрізнятись від того, що написано в коді програми. Будь-який метод класу насправді в якості першого аргументу отримує посилання на об’єкт, для якого він викликається.

Конструктори та деструктори

В об’єктно-орієнтованому програмуванні конструктор класу це спеціальний метод класу, що викликається при створенні об’єкта.

Види конструкторів:

- ***конструктор за замовчуванням*** – конструктор, що не має обов’язкових аргументів. Використовується при створенні масивів об’єктів (виклик здійснюється для кожного елементу масиву). Якщо конструктор за замовченням не задано явно його код генерується компілятором автоматично (не впливає на початковий код програми);
- ***конструктор з параметрами;***
- ***конструктор копіювання*** – конструктор що приймає у якості аргументу посилання на об’єкт того ж класу. Використовується для передачі об’єктів у функції за значенням, повернення значень із функції та ініціалізації об’єктів під час оголошення. Конструктор копіювання зазвичай потрібен, коли об’єкт має вказівники на об’єкти, що виділені у купі. Якщо програміст не створює конструктор копіювання, то компілятор створює неявний конструктор копіювання, який копіює вказівники як є, без копіювання даних. Тобто два об’єкти будуть використовувати одну й ту саму ділянку пам’яті, це означає, що будь-яка спроба змінити копію зіпсує оригінал;
- ***конструктор переміщення*** – у C++11 конструктор переміщення приймає на вході значення не константного посилання на об’єкт класу, і використовується для передачі володіння ресурсами цього об’єкта. Конструктори переміщення були створені для вирішення проблеми ефективності, пов’язаної зі створенням тимчасових об’єктів.

ЗАУВАЖЕННЯ. Ім’я конструктора має збігатися з ім’ям класу. Дозволяється використовувати декілька конструкторів з однаковим ім’ям, але різними параметрами (перевантаження конструкторів).

Деструктор – спеціальний метод класу, що використовується для деінціалізації об’єкта.

Так оголошення класу, призначеного для роботи з векторами, може мати такий вигляд:

```
class CVector
{
protected:
   double *Data;
   UInt64 VectorSize;
public:
```

```
    //конструктор за замовченням
    Cvector(){Data=nullptr;VectorSize=0;};
    //конструктор з параметором
    CVector(UInt64 NewSize){Data=new double[NewSize];
            VectorSize=NewSize;};
    //конструктор копії
    CVector(CVector &v);
    //конструктор переносу
    CVector(CVector &&v);
    //деструктор
    ~CVector(){delete []Data;};
};
```

Перевантаження операторів

Мова C++ дозволяє перевантажувати будь-які оператори, що наведені у табл., що може бути корисним при роботі з класами.

Таблиця.

Перелік операторів які можна перевантажити

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

Приклади перевантаження операторів можна побачити у лістингах 1.4 та 1.5.

Друзі

Дружня функція – це функція, яка не є членом класу, але має доступ до членів класу, оголошеним у розділах private або protected. Дружні функції – це звичайні зовнішні функції з особливими правами доступу. Кому надати доступ, визначає клас, функція не може самостійно нав'язатись у друзі класу. Для оголошення дружньої функції використовується ключове слово ***friend.***

Лістинг 1.4. CVector.h (оголошення класу Ctime)

```
1: #ifndef CVECTOR_H
2: #define CVECTOR_H
3: #include <iostream>
4: using namespace std;
5: typedef unsigned long long int UInt64;
6:
7: class CVector
8: {
9: protected:
10:     double *Data;
11:     UInt64 VectorSize;
12: public:
13:     CVector(){ Data=nullptr; VectorSize=0; cout<<"\
            nConstruct Def ptr="<<Data;};
14:     CVector(UInt64 NewSize){ Data=new double[NewSize];
            VectorSize=NewSize; cout<<"\nConstruct Standart
            ptr="<<Data;};
15:     CVector(CVector &v);
16:     CVector(CVector &&v);
17:     void operator=(const CVector &copy);
18:     CVector operator+(CVector &);
19:     friend ostream& operator<<(ostream& ,CVector&);
20:     friend istream& operator>>(istream& ,CVector&);
21:     ~CVector(){cout<<"\nDestroy "<<Data;
            delete []Data;};
22: };
23: #endif // CVECTOR_H
```

Лістинг 1.5. CVector.cpp (реалізація класу CVector)

```
1: #include "CVector.h"
2:
3: CVector::CVector(CVector &v)
4: {
```

```
5:    Data=nullptr;
6:    *this=v;
7:    cout<<"\nConstruct Def ptr="<<Data;
8: }
9:  CVector::CVector(CVector &&v)
10: {
11:    Data=v.Data;
12:    v.Data=nullptr;
13:    VectorSize=v.VectorSize;
14:    v.VectorSize=0;
15:    cout<<"\nConstruct Move ptr="<<Data;
16: }
17:
18: void CVector::operator=(const CVector &copy)
19: {
20:     cout<<"\nOperator=";
21:   if(this==&copy)
22:     return;
23:   delete []Data;
24:   if(nullptr==copy.Data)
25:     {
26:         Data=nullptr;
27:         VectorSize=0;
28:         return;
29:     }
30:   Data=new double [copy.VectorSize];
31:   cout<<"ptr="<<Data;
32:   VectorSize=copy.VectorSize;
33:   for(UInt64 i=0;i<VectorSize;i++)
34:         Data[i]=copy.Data[i];
35: }
36: CVector CVector::operator+(CVector &v2)
37: {
38:    CVector v3(VectorSize);
39:    for(UInt64 i=0;i<VectorSize;i++)
40:       v3.Data[i]=Data[i]+v2.Data[i];
41:    return v3;
42: }
43:
44: ostream& operator<<(ostream& stream,CVector& V)
45: {
46:    stream<<"\nVector["<<V.VectorSize<<"] at
          "<<V.Data<<"=";
47:    for(UInt64 i=0;i<V.VectorSize;i++)
48:       stream<<V.Data[i]<<' ';
```

```
49:    return stream;
50: }
51: istream& operator>>(istream& stream,CVector& V)
52: {
53:     for(UInt64 i=0;i<V.VectorSize;i++)
54:       stream>>V.Data[i];
55:     return stream;
56: }
```

Лістинг 1.6. main.cpp (реалізація програми для роботи з векторами)

```
1: #include <iostream>
2: #include "CVector.h"
3:
4: using namespace std;
5:
6: int main()
7: {
8:     CVector v1(5),v2(5),v3;
9:     cout<<endl;
10:     cin>>v1>>v2;
11:     v3=v1+v2+v1;
12:     CVector v4(v1+v2+v2);
13:     cout << "\n Vector is::" << v1<<v2<<v3<<v4<<endl;
14:     return 0;
15: }
```

Масиви об’єктів

Досить часто виникає необхідність створювати декілька об’єктів одного класу, поєднавши їх у масив. Фактично це робиться так само, як і для базових типів або структур. Так, для класу векторів, розглянутого раніше, код буде таким:

```
CVector myVectors[15];
int i;
for(i=0;i<15;++i)
     cin>>myVectors[i];
for(i=0;i<14;++i)
     myVectors[i]=myVectors[i]+myVectors[i+1];
for(i=0;i<15;++i)
     cin>>myVectors[i];
CVector myArray[5]={
     CVector(4),
     CVector(6),
     CVector(8),
     CVector(5),
     CVector(3)
     };
```

Виключні ситуації

Більшість сучасних мов програмування дозволяють здійснювати обробку помилок не тільки за допомогою повернення функціями у той чи інший спосіб інформації про помилки що виникли під час роботи, а й використовуючи механізм виключних ситуацій (***try***, ***catch***, ***throw***)**.**

Так, у лістингу 1.7 описується клас CError, за допомогою якого можна посилати повідомлення про помилку, перехоплювати та обробляти його в потрібному місці.

Лістинг 1.7. CError.h (оголошення класу CError)

```
16: /*
17: * File:   CErrors.h
18: * Author: Антонов Ю.С.
19: * Created on 25 Сентябрь 2012 г., 19:43
20: * Клас CError призначений для зберігання інформації про
          помилку,
21: * її код, метод та об’єкт який викликликав цю помилку.
22: * Призначений для використання з try catch throw
23: */
24: #ifndef CERRORS_H
25: #define CERRORS_H
26:
```

```
27: #include<string>
28: #include<iostream>
29: using namespace std;
30:
31: class CError
32: {
33: protected:
34:        int ErrorCode;
35:        string ErrorDescription;
36:        string ErrorInitObject;
37:        string ErrorInitMethod;
38: public:
39:     CError()
40:     {
41:         ErrorCode=0;
42:     };
43:
44:     CError(string Description, int Code=0,string
          Method="Undefined", string ErrObject="Undefined")
45:          {
46:             ErrorDescription=Description;
47:             ErrorCode=Code;
48:             ErrorInitMethod=Method;
49:             ErrorInitObject=ErrObject;
50:          };
51:     string SpellError()
52:          {
53:             return "\nError ocured!\n"+ErrorInitMethod+"
          generate error in"+ErrorInitObject+" object"+"\
          nErrorDescription: "+ErrorDescription;
54:          };
55: };
56: #endif
```

Тепер спробуємо використати цей клас на практиці. У лістингу 1.8 здійснюється спроба знайти цілу частину та залишок від ділення двох змінних. У рядку 8 цієї програми відкривається блок *try,* а у рядку 10 відбувається перевірка, і якщо знаменник дорівнює 0, то за допомогою оператора *throw* посилається об’єкт, що буде містити інформацію про помилку.

Лістинг 1.8. main.cpp (програма ілюструє механізм виключних ситуацій)

```
1: #include "CErrors.h"
2:
3: int main(int argc, char* argv[])
4: {
```

```
5:          int a,b,j;
6:      cin>>a>>b;
7:          cout<<"\na="<<a<<"\nb="<<b;
8:          try
9:          {
10:         if(0==b)
11:            throw CError("You can not devide by
           zero!!!",1,"main");
12:         cout<<"\nresult: \na/b="<<a/b<<"\na%b="<<a%b;
13:         }
14:         catch(CError &e)
15:         {
16:            cout<<e.SpellError();
17:         }
18:         catch(...)
19:         {
20:             cout<<"\nUnresolved Exception";
21:         }
22: }
```

У рядку 14 оператор *catch* перехоплює об’єкт класу *CError*, після чого виводить сповіщення про зміст помилки. У 18 рядку оператор *catch* перехоплює помилки усіх інших типів.

.

2 «Об’єктно-орієнторованe програмування на С++»

Інкапсуляція та приховування внутрішньої реалізації

Одним із дуже важливих питань в ООП є проблема приховування внутрішньої реалізації, а також розуміння різниці між public, private, protected та правильного їх застосування. Для цього розглянемо наступний приклад. Нехай компанія «XYZ» займається розробкою програмного забезпечення. У поточному проекті виникла необхідність роботи з часом у форматі години, хвилини та секунди. Фактично одразу програмістами компанії було створено клас, наведений у лістингах 1.1 – 1.2. Через пів року після створення цього класу, один із керівників підприємства проводив перевірку коду і прийшов до висновку, що він не є оптимальним, і його слід переробити. Для реалізації поставленої мети він запропонував використовувати бітові поля. Після переробки код став виглядати так, як вказано у лістингу 2.1.

Лістинг 2.1. CTime.h (оголошення класу CTime – варіант 2)

```
1: #ifndef CTIME_H
2: #define CTIME_H
3: #include <string>
4: using namespace std;
5: typedef unsigned long int UINT32;
6: class CTime
7: {
8:     private:
9:       unsigned int hours:5;
10:       unsigned int minutes:6;
11:       unsigned int seconds:6;
12:     public:
13:       string ToString();
14:       void SetTime(UINT32 p_hours,UINT32
           p_minutes,UINT32 p_seconds);
15:       void SetTime(UINT32 p_hours,UINT32 p_minutes);
16:       void SetTime(UINT32 p_hours);
17:       void SetHours(UINT32 p_hours){hours=p_hours;};
18:       void SetMinutes(UINT32 p_minutes)
           {minutes=p_minutes;};
19:       void SetSeconds(UINT32 p_seconds)
           {seconds=p_seconds;};
20:       int GetHours(){return hours;};
21:       int GetMinutes(){return minutes;};
22:       int GetSeconds(){return seconds;};
23:       void AddSeconds(UINT32);
24:       void AddSeconds(float)=delete;
```

```
25:       void AddSeconds(double)=delete;
26: };
```

Через шість місяців, компанія найняла молодого, але талановитого студента, який дослідивши програмний код, прийшов до висновку, що обидві попередні реалізації не є ефективними. Він запропонував зберігати час у вигляді кількості секунд, що пройшла з опівночі у беззнаковому цілому числі. І, щоб не бути балакуном, підкріпив свої слова кодом, який наведено у лістингах 2.2 та 2.3.

Лістинг 2.2. CTime.h (оголошення класу CTime – варіант 3)

```
1: #ifndef CTIME_H
2: #define CTIME_H
3: #include <string>
4: using namespace std;
5: typedef unsigned long int UINT32;
6: class CTime
7: {
8:     private:
9:        unsigned int seconds;
10:     public:
11:       string ToString();
12:       void SetTime(UINT32 p_hours,UINT32
           p_minutes,UINT32 p_seconds);
13:       void SetTime(UINT32 p_hours,UINT32 p_minutes);
14:       void SetTime(UINT32 p_hours);
15:       int GetHours(){return seconds/3600;};
16:       int GetMinutes(){return seconds/60%60;};
17:       int GetSeconds(){return seconds%60;};
18:       void AddSeconds(UINT32);
19:       void AddSeconds(float)=delete;
20:       void AddSeconds(double)=delete;
21: };
22: #endif // CTIME_H
```

Лістинг 2.3. CTime.cpp (реалізація класу CTime – варіант 3)

```
1: #include "CTime.h"
2: #include "CTime.h"
3: #include <sstream>
4:
5: string CTime::ToString()
6: {
7:     stringstream str;
8:     int ss=seconds%60;
```

```
9:     int mm=(seconds/60)%60;
10:     int hh=seconds/3600;
11:     if(hh<10)
12:         str<<'0';
13:     str << hh<<':';
14:     if(mm<10)
15:         str<<'0';
16:     str << mm<<':';
17:     if(ss<10)
18:         str<<'0';
19:     str << ss;
20:     return str.str();
21: }
22: void CTime::SetTime(UINT32 p_hours,UINT32
           p_minutes,UINT32 p_seconds)
23: {
24:     seconds=3600*p_hours+60*p_minutes+p_seconds;
25: }
26: void CTime::SetTime(UINT32 p_hours,UINT32 p_minutes)
27: {
28:     seconds=3600*p_hours+60*p_minutes;
29: }
30: void CTime::SetTime(UINT32 p_hours)
31: {
32:     seconds=3600*p_hours;
33: }
34: void CTime::AddSeconds(UINT32 p_seconds)
35: {
36:       seconds+=p_seconds;
37: }
```

Тепер спробуємо проаналізувати ситуацію, що склалась.

Уявімо собі, що усі атрибути класу були у секції public. У цьому випадку програмістам було б необхідно двічі вносити зміни у методи цього класу, перевірити методи усіх нащадків цього класу та весь інший код, що міг використовувати ці властивості. Звісно, така ситуація потребує досить багато часу та уваги. Під час такої рутинної роботи можна припуститись помилок, і зробити код, що працює таким, що вже не працює.

У випадку, коли усі атрибути класу розташовані у секції protected, обсяги роботи трошки зменшуються – програмістам необхідно внести зміни у методи цього класу, перевірити методи усіх нащадків даного класу. Звісно, така ситуація вже краща за попередню, однак не є ідеальною. І ризик зробити код непрацездатним залишається.

Якщо усі властивості класу розташовані у секції private, то програмістам було б необхідно змінити лише методи цього класу, і ризик отримати непрацездатну програму буде мінімальним.

Дуже багато авторів рекомендують наступний підхід при реалізації класів:

- У секції public розташовується виключно інтерфейс класу – методи, що дозволяють зовнішнім об'єктам взаємодіяти з об'єктами цього класу. Інколи дозволяється розташовувати тут атрибути, форма представлення яких є незмінною, але ми б не радили робити це, оскільки те, що є незмінним сьогодні, може змінитися згодом.
- У секції protected розташовуються атрибути та методи, що є спільними для цього класу та його нащадків.
- У секції private розташовуються методи та атрибути, що притаманні лише цьому класу.

Ендрю Троелсєн визначає інкапсуляцію як можливість мови програмування приховувати деталі внутрішньої реалізації об'єктів від користувача об'єкта [12].

Наслідування

Наслідування – принцип ООП, що стосується здатності мови програмування будувати нові визначення класів на підставі вже існуючих визначень [12].

Наслідування – відносини між класами, при яких клас використовує структуру або поведінку, визначену в іншому класі (одиночне наслідування) або в декількох інших класах (множинне наслідування). Наслідування визначає ієрархію класів «загальне/приватне», в якій підклас наслідує властивості одного або кількох більш загальних суперкласів. Підклас зазвичай конкретизує суперклас, доповнюючи його структуру та поведінку [3].

Розглянемо, як на практиці може використовуватися, наслідування. Нехай нам необхідно створити програму, що буде працювати з домашніми улюбленцями, а саме кішками, собаками. Якщо дослідити предметну галузь, то ми побачимо, що кожен домашній улюбленець може мати ім'я (прізвисько) та подавати якісь звукові сигнали. Тому ми створюємо базовий клас CPet, що буде нашим базовим класом (лістинг 2.4).

Лістинг 2.4. CPet.h (оголошення класу CPet)

```
1: #ifndef CPET_H
2: #define CPET_H
3: #include <string>
4: #include <iostream>
5: using namespace std;
```

```
6: class CPet
7: {
8:     public:
9:         CPet();
10:         void   SetNickname(string nick)
           {nickname=nick;};
11:         string GetNickname(){return nickname;};
12:         string Say(){return "abstract pet cannot
           talk!!!";};
13:         virtual ~CPet();
14:     protected:
15:         string nickname;
16:         string color;
17: };
18: #endif // CPET_H
```

Цей клас містить ім'я улюбленця (15), його колір (16), та методи для встановлення та отримання прізвиська. Класи, що будуть відповідати кішкам та собакам, наведено у лістингах 2.5 та 2.6 відповідно.

Лістинг 2.5. CCat.h (оголошення класу CCat)

```
1: #ifndef CCAT_H
2: #define CCAT_H
3: #include "CPet.h"
4:
5: class CCat:public CPet
6: {
7:     public:
8:         CCat();
9:         virtual ~CCat();
10:         string Say(){return "Mr-rr Mr-rr";};
11:     protected:
12:     private:
13: };
14: #endif // CCAT_H
```

Лістинг 2.6. CDog.h (оголошення класу CDog)

```
1: #ifndef CDOG_H
2: #define CDOG_H
3: #include "CPet.h"
4:
5: class CDog:public CPet
6: {
7:     public:
8:         CDog();
```

```
9:          virtual ~CDog();
10:          string Say(){return "Gav Gav";};
11:     protected:
12:     private:
13: };
14: #endif // CDOG_H
```

Приклад програми, що буде працювати з домашніми улюбленцями можна побачити у лістингу 2.7.

Лістинг 2.7. mypets.cpp (програма для роботи з улюбленцями)

```
1: #include <iostream>
2: #include "CCat.h"
3: #include "CDog.h"
4:
5: using namespace std;
6:
7: int main()
8: {
9:      CCat *cat=new CCat;
10:     CDog *dog=new CDog;
11:     cout<<"Begin talk with cat"<<endl;
12:     cat->SetNickname("Mashka");
13:     cout<<cat->GetNickname()<<endl;
14:     cout<<cat->Say()<<endl;
15:     cout<<"\nBegin talk with dog"<<endl;
16:     dog->SetNickname("Bobik");
17:     cout<<dog->GetNickname()<<endl;
18:     cout<<dog->Say()<<endl;
19:     cout<<"\nBegin talk with pets"<<endl;
20:     CPet *mypet[2]={cat,dog};
21:     for(int i=0;i<2;i++)
22:        {
23:          cout<<mypet[i]->GetNickname()<<endl;
24:          cout<<mypet[i]->Say()<<endl;
25:        }
26:     return 0;
27: }
```

Результат роботи нашої програми можна побачити на рис. 5. У рядках 9 та 10 програми оголошуються та ініціалізуються вказівники для роботи з кішкою та собакою. Ці вказівники належать відповідно класам нащадкам CCat та CDog. Рядки 11 – 14 та 16 – 19 виводять інформацію про наших улюбленців, так, як і очікувалось. Далі у рядку 22 оголошується масив вказівників базового класу CPet, який ініціалізується значеннями вказівників cat та

dog за допомогою спискової ініціалізації.

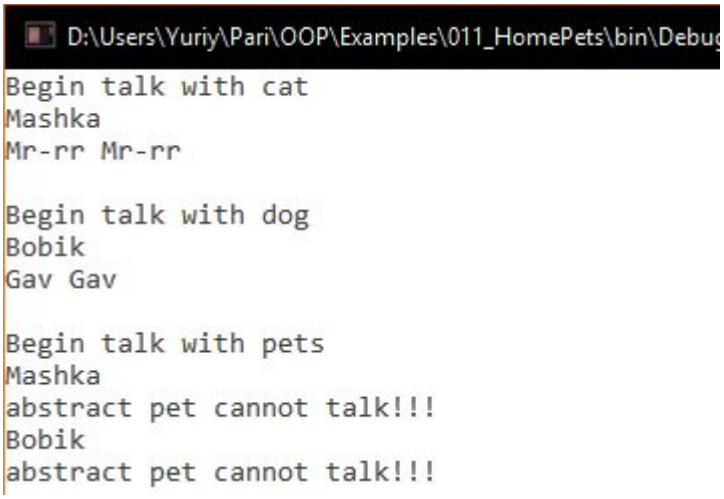

Рис. 5. – Вивід інформації про домашніх тварин

Як видно з рис. 5, отримання прізвиська улюбленця відбувається без перешкод, але отримати звуки, які він подає, нам не вдається. Замість очікуваних фраз ми бачимо фразу *«abstract pet cannot talk»*. Тобто замість очікуваного виклику методів *Say()* з класів нащадків компілятор здійснив виклик методу *Say()* з базового класу. Насправді така поведінка компілятора є цілком нормальною, оскільки масив вказівників відноситься до базового класу, то і виклик метода відбувається з цього класу.

Поліморфізм та віртуальні функції

У попередньому розділі ми створили програму, яка виводила інформацію про улюбленців, але не зовсім так, як дехто міг очікувати. Для того, щоб змусити нашу програму правильно виводити у циклі інформацію про улюбленців необхідно використовувати віртуальні функції та поліморфізм, тоді класи *CPet*, *CCat*, *CDog* матимуть вигляд, зображений у лістингах 2.8, 2.9 та 2.10 відповідно (зміняться рядки 12, 10 та 10 відповідно).

Лістинг 2.8. CPet.h (оголошення класу CPet)

```
1: #ifndef CPET_H
2: #define CPET_H
3: #include <string>
4: #include <iostream>
5: using namespace std;
6: class CPet
7: {
8:    public:
9:      CPet();
10:     void   SetNickname(string nick){nickname=nick;};
```

```
11:     string GetNickname(){return nickname;};
12:     virtual string Say(){return "abstract pet cannot
           talk!!!";};
13:     virtual ~CPet();
14:   protected:
15:     string nickname;
16:     string color;
17: };
18: #endif // CPET_H
```

Лістинг 2.9. CCat.h (оголошення класу CCat)

```
1: #ifndef CCAT_H
2: #define CCAT_H
3: #include "CPet.h"
4:
5: class CCat:public CPet
6: {
7:     public:
8:         CCat();
9:         virtual ~CCat();
10:        virtual string Say(){return "Mr-rr Mr-rr";};
11:     protected:
12:     private:
13: };
14: #endif // CCAT_H
```

Лістинг 2.10. CDog.h (оголошення класу CDog)

```
1: #ifndef CDOG_H
2: #define CDOG_H
3: #include "CPet.h"
4:
5: class CDog:public CPet
6: {
7:     public:
8:         CDog();
9:         virtual ~CDog();
10:        virtual string Say(){return "Gav Gav";};
11:     protected:
12:     private:
13: };
14: #endif // CDOG_H
```

Лістинг 2.11. mypet2.cpp (програма для роботи з улюбленцями)

```
1: #include <iostream>
2: #include "CCat.h"
3: #include "CDog.h"
4:
5: using namespace std;
6:
7: int main()
8: {
9:     CCat *cat=new CCat;
10:     CDog *dog=new CDog;
11:     cout<<"Begin talk with cat"<<endl;
12:     cat->SetNickname("Mashka");
13:     cout<<cat->GetNickname()<<endl;
14:     cout<<cat->Say()<<endl;
15:
16:     cout<<"\nBegin talk with dog"<<endl;
17:     dog->SetNickname("Bobik");
18:     cout<<dog->GetNickname()<<endl;
19:     cout<<dog->Say()<<endl;
20:
21:     cout<<"\nBegin talk with pets"<<endl;
22:     CPet *mypet[2]={cat,dog};
23:     for(int i=0;i<2;i++)
24:        {
25:          cout<<mypet[i]->GetNickname()<<endl;
26:          cout<<mypet[i]->Say()<<endl;
27:        }
28:      CPet myNewPet;
29:      cout<<"\n\n\n\ntest  pet say:
           "<<myNewPet.Say()<<"\n\n\n\n";
30:     return 0;
31: }
```

Як видно з рис. 6, результат роботи нашої програми змінився порівняно з результатами, зображеними на рис. 5. Тепер повідомлення про звуки, що притаманні нашим тваринам, виводяться правильно. Завдяки використанню віртуальних функцій код у рядках 23 – 27 лістингу 2.11 змінює свою поведінку, тобто є поліморфним. Більше того, цей код буде працювати навіть з тими класами, які не були створені на момент написання головної програми (єдиною умовою є те що ці класи мають також наслідуватись від базового класу *CPet* та містити відповідні віртуальні функції). Однак, з нашою програмою ще не все гаразд. У рядку 28 оголошується змінна класу *CPet.* Замість очікуваних фраз ми бачимо фразу *«abstract pet cannot talk»*. Тобто

замість очікуваного виклику методів *Say()* з класів нащадків компілятор здійснив виклик методу *Say()* з базового класу. Насправді така поведінка компілятора є цілком нормальною, оскільки масив вказівників відноситься до базового класу, то і виклик методу відбувається з цього класу.

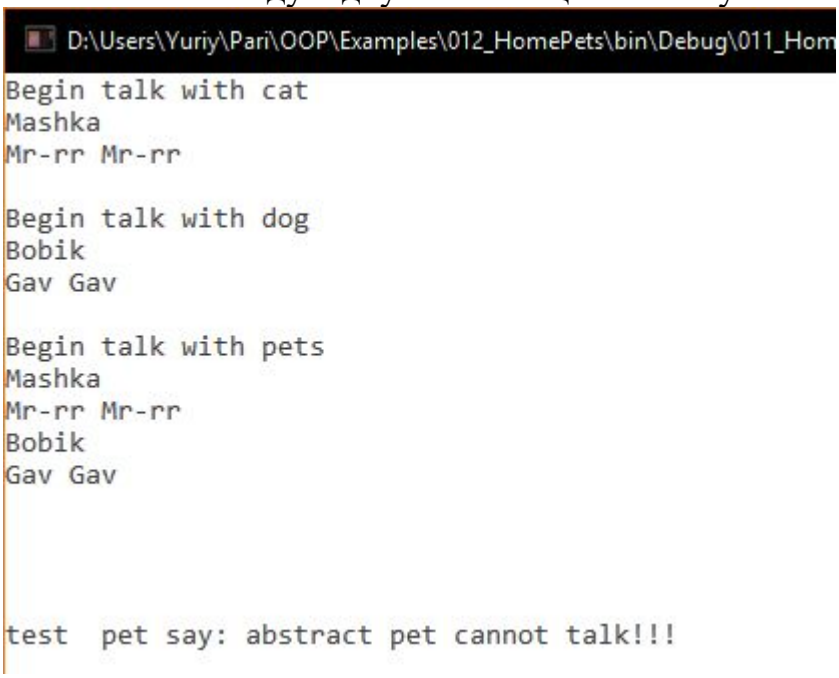

Рис. 6. – Вивід інформації про домашніх тварин з поліморфізмом

Абстрактні класи

У попередньому розділі ми навчили наших улюбленців правильно виводити їхні прізвиська та звуки, які вони можуть видавати. Однак код, який ми отримали дозволяє створювати об'єкти, що є членами базового класу *CPet,* але велика проблема в тому, що насправді така абстрактна тварина не може існувати!

Лістинг 2.12. CPet3.h (оголошення абстрактного класу CPet)

```
1: #ifndef CPET_H
2: #define CPET_H
3: #include <string>
4: #include <iostream>
5: using namespace std;
6: class CPet
7: {
8:     public:
9:         CPet();
10:         void   SetNickname(string nick)
           {nickname=nick;};
11:         string GetNickname(){return nickname;};
12:         virtual string Say()=0;
13:         virtual ~CPet();
14:     protected:
15:         string nickname;
16:         string color;
17: };
18:
19: #endif // CPET_H
```

На щастя, існує інструмент, який дозволяє виправити це непорозуміння – абстрактні класи. У мові C++ клас вважається абстрактним, якщо в ньому є хоча б одна чиста віртуальна функція.

Віртуальна фунція є чистою, якщо після її імені стоїть =0 (у цьому випадку тіло функції відсутнє).

ЗАУВАЖЕННЯ. У класах нащадках чисті віртуальні функції мають бути перевизначені. Якщо у класі нащадку хоча б одна чиста віртуальна функція залишиться невизначеною, то такий клас також буде абстрактним.

В інших мовах програмування, наприклад *C#,* для створення абстрактного класу необхідно використовувати ключове слово *abstract*. Також у *C#* ключове слово abstract може використовуватися для створення абстрактної функції – аналогу чистої віртуальної функції у *C++*.

У рядку 12 лістингу 2.12 оголошено чисту віртуальну функцію *Say(),* завдяки якій базовий клас став абстрактним, завдяки чому відтепер створення

екземплярів базового класу є неможливим. Як видно з рис. 7, будь-яка спроба створення об'єкта призведе до помилки під час компіляції – *Cannot declare variable 'myNewPet' to be of abstract type 'CPet'*, та пояснення чому саме це неможливо зробити *because the following virtual functions are pure within 'CPet'*.

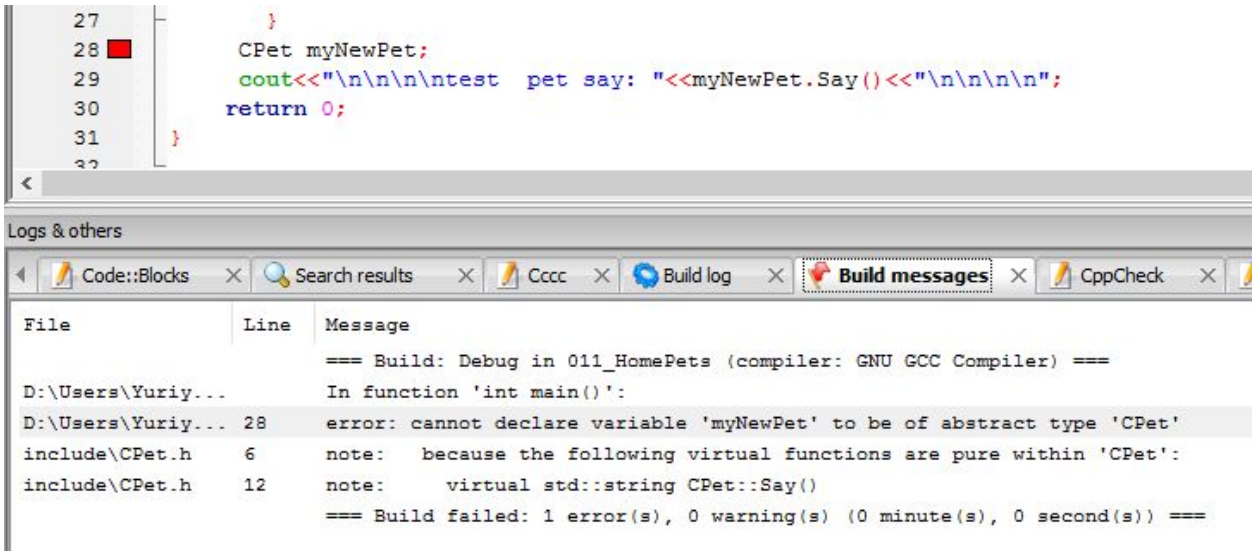

Рис. 7. – Заборона створення екземплярів абстрактних класів

Множинне наслідування

Мова C++, на відміну від Java та C#, підтримує можливість множинного наслідування. Це означає, що новий клас може наслідувати методи та властивості від двох або більше базових класів. У мовах програмування, які не підтримують множинне наслідування, для досягнення подібного ефекту дуже часто використовується механізм інтерфейсів або оголошення полів, що є екземплярами відповідних класів. На рис. 8 зображено приклад ієрархії класів геометричних фігур: коло, паралелограм, прямокутник, ромб, квадрат.

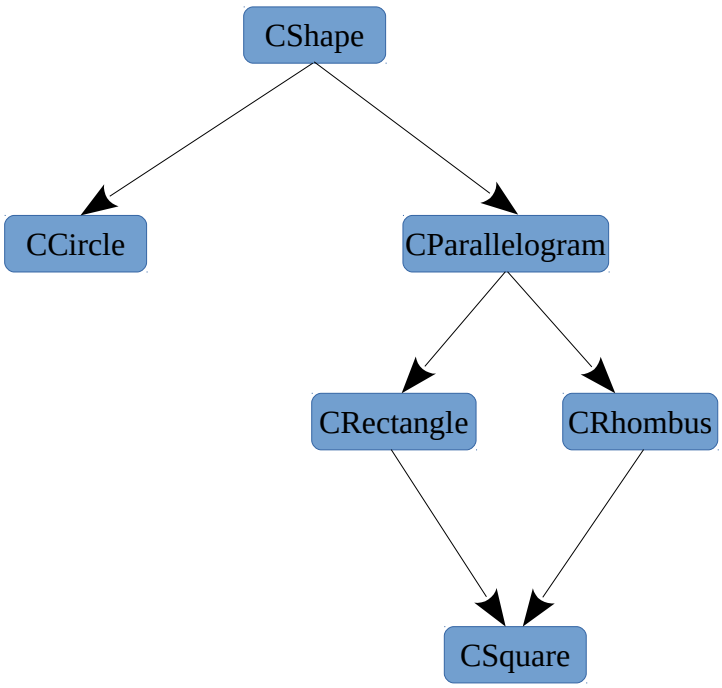

Рис. 8. – Приклад ієрархії класів геометричних фігур

Якщо ми спробуємо записати цю ієрархію мовою C++, то отримаємо такий код

```
class CShape {
    public:
         CShape();
……
};

class CParallelogram: public CShape {
    public:
         CParallelogram();
……
};

class CRhombus: public CParallelogram {
```

```
    public:
         CRhombus();
……
};

class CRectangle: public CParallelogram{
    public:
         CRectangle();
……
};

class CSquare: public CRectangle, public CRhombus{
    public:
         CCSquare();
……
};
```

Оголошення останнього класу CSquare містить одну проблему: він походить від двох класів, що мають спільний батьківський клас. Якби не це, то наш код працював би без проблем. Щоб уникнути проблеми під час наслідування, треба використовувати ключове слово ***virtual***, тоді код стане робочим та матиме такий вигляд:

```
class CSquare
    :virtual public CRectangle,
     virtual public CRhombus{
    public:
         CCSquare();
……
};
```

Шаблони

Шаблони функцій та класів – це потужний інструмент у руках програміста, який дозволяє підвищити повторне використання коду. Завдяки шаблонам робота програмістів дуже спрощується. Якщо, наприклад, нам необхідно створити функцію, що буде виводи масив, то нам знадобиться створювати функції для кожного типу даних, а завдяки шаблонам ми можемо написати цей код лише один раз. Так само і з класами – можна створити один раз шаблон класу (без прив'язки до типу даних) замість того, щоб створювати власноруч окремі класи для кожного типу.

Загальний синтаксис оголошення класу-шаблону у старій нотації такий

```
template<class T>
class ім'яКласу
{
 //тіло класу
};
```

Нова нотація дозволяє використання такого синтаксису

```
template<typename T>
class ім'яКласу
{
 //тіло класу
};
```

Усім відомо, що при роботі зі звичайними масивами у C/C++ не відбувається контролю границь масивів. Таку проблему можна виправити, створивши клас масивів, який би виправляв усі ці недоліки, але при звичайному підході проблема у тому, що нам треба буде описувати клас для кожного типу даних, що не є доцільним. Тому у цьому випадку використання шаблонів буде оптимальним варіантом.

Лістинг 2.13. TArray.h (оголошення класу шаблону TArray)

```
1: #ifndef TARRAY_H_INCLUDED
2: #define TARRAY_H_INCLUDED
3: #include <iostream>
4: using namespace std;
5:
6: template<class T>
7: class TArray
8: {
9: protected:
10:     T *data;
11:     unsigned int N;
12: public:
13:     TArray(unsigned int n=1){data=new T[n];N=n;};
```

```
14:     TArray(unsigned int n,const T &v);
15:     virtual ~TArray(){delete[] data;};
16:     void Show();
17:     virtual T& operator[](unsigned int i);
18:     virtual T operator[](unsigned int i) const;
19: };
20: template<class T>
21: TArray<T>::TArray(unsigned int n,const T &v)
22: {
23:     data=new T[n];
24:     N=n;
25:     for(unsigned int i=0;i<N;i++)
26:         data[i]=v;
27: }
28: template<class T>
29: T& TArray<T>::operator[](unsigned int i)
30: {
31:     if(i>=N)
32:     {
33:        cout<<"\n Index out of range!";
34:        throw 1;
35:     }
36:     return data[i];
37: }
38: template<class T>
39: void TArray<T>::Show()
40: {
41:     cout<<"\n";
42:     for(unsigned int i=0;i<N;i++)
43:        cout<<data[i]<<" ";
44: }
45: template<class T>
46: T TArray<T>::operator[](unsigned int i) const
47: {
48:     if(i>=N)
49:     {
50:        cout<<"\n Index out of range!";
51:        throw 1;
52:     }
53:     return data[i];
54: }
55: #endif // TARRAY_H_INCLUDED
```

Лістниг 2.13 містить реалізацію такого шаблонного класу, а у лістингу 2.14 наведено спосіб використання цього шаблонного класу

Лістинг 2.14. main.cpp (використання класу шаблону TArray)

```
1: #include <iostream>
2: #include "TArray.h"
3:
4: using namespace std;
5:
6: int main()
7: {
8:     TArray<int> arrInt(10,5),arrInt2(10,15),arrInt3(10);
9:     TArray<double> arrDbl(50);
10:     for(int i=0;i<10;i++)
11:         arrInt3[i]=arrInt[i]+arrInt2[i];
12:     arrInt.Show();
13:     arrInt2.Show();
14:     arrInt3.Show();
15:     arrDbl.Show();
16:     return 0;
17: }
```

Довідкова інформація

Ключові слова ANSI C++ 11

alignas	alignof	asm	auto	bool
break	case	catch	char	char16_t
char32_t	class	const	const_cast	constexpr
continue	decltype	default	delete	do
double	dynamic_cast	else	enum	explicit
export	extern	false	float	for
friend	goto	if	inline	int
long	mutable	namespace	new	noexcept
nullptr	operator	private	protected	public
registred	reinterpret_cast	return	short	signed
sizeof	static	static_assert	static_cast	struct
switch	template	this	thread_local	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	wchar_t	while		

Альтернативні лексеми ANSI C++ 11

Лексема	Призначення	Лексема	Призначення	Лексема	Призначення
`and`	`&&`	`compl`	`~`	`or_eq`	`|=`
`and_eq`	`&=`	`not`	`!`	`xor`	`^`
`bitand`	`&`	`not_e`	`!=`	`xor_eq`	`^=`
`bitor`	`|`	`or`	`||`		

Перетворення типів C++

На сьогодні у мові програмування C++ існує декілька способів перетворення типів:

- ***C-style cast*** має синтаксис ***TYPE (TYPE*) object.*** Не рекомендовано використовувати;
- ***static_cast*** має синтаксис ***TYPE static_cast<TYPE> (object).*** Перетворює вираз одного статичного типу в об'єкт або значення іншого статичного типу. Підтримується перетворення чисельних типів, вказівників та посилань за ієрархією наслідування як угору, так і вниз. Перевірка відбувається на етапі компіляції, тому, у разі

помилки, повідомлення буде отримано під час побудови програми або бібліотеки.

- ***dynamic_cast*** має синтаксис
  TYPE& dynamic_cast<TYPE&> (object)
  або
  TYPE* dynamic_cast<TYPE*> (object),
  який використовується для динамічного приведення типу під час виконання програми. У випадку неправильного приведення типу для посилань генерується виключна ситуація **std::bad_cast,** а для вказівників повертається 0. Використовує систему **RTTI** (**Run-Time Type Information**). Безпечне приведення типу за ієрархією наслідування, у тому числі для віртуального наслідування.
- ***const_cast*** має синтаксис
  TYPE const_cast<TYPE> (object).
  Мабуть найпростіше перетворення типу. Знімає (встановлює) const та volatile кваліфікатори, тобто константність та відмову від оптимізації змінної компілятором. Це перетворення відбувається на етапі компіляції, і у разі неможливості такого перетворення компіляцію буде зупинено та повідомлено про помилку.
- ***reinterpret_cast*** має синтаксис
  TYPE reinterpret_cast<TYPE> (object).
  Приведення типу без перевірки reinterpret_cast – це безпосередня вказівка компілятору. Використовується тільки у випадку повної впевненості програміста у власних діях. Використовується для приведення вказівника до вказівника, вказівника до цілого, цілого до вказівника.

Завдання для індивідуальних робіт та приклади їхнього розв'язання

Індивідуальна робота № 1. Класи та об'єкти

ЗАВДАННЯ № 1.1. Створити клас векторів, що підтримує такі операції:

- додавання двох векторів;
- скалярний добуток векторів;
- множення вектора на константу та константи на вектор;
- вивід значень вектора на екран;
- отримання значення певного елемента;
- встановлення значення певного елемента.

Програма має обробляти помилки за допомогою виключних ситуацій та класу CError.

ЗАВДАННЯ № 1.2. Відповідно до Вашого варіанту створити проект, в якому за допомогою ООП реалізувати вказаний функціонал. Кожен клас має містити: властивості, конструктори, деструктори та методи для роботи з об'єктами.

Варіант	Опис
1	Для роботи з кутами у форматі: *градуси, хвилини, секунди* створити клас, який дозволяє: a) вводити значення кута з клавіатури (файла); b) виводити значення кута на екран (файл); c) знаходити суму/різницю між двома кутами; d) множення кута на дійсне число; e) переведення кута у радіани та навпаки. Створити приклад для демонстрації усіх функціональних можливостей
2	Для роботи з часом у форматі: *години, хвилини, секунди* створити клас, який дозволяє: a) вводити значення часу з клавіатури (файла); b) виводити значення часу на екран (файл); c) знаходити суму/різницю двох відповідних міток часу (структур);

	d) переведення часу у секунди та навпаки. Створити приклад для демонстрації усіх функціональних можливостей
3	Для роботи з датами у форматі: *рік, місяць, день* створити клас, який дозволяє: a) вводити значення дати з клавіатури (файла); b) виводити значення дати на екран (файл); c) знаходити суму/різницю двох відповідних дат; d) переведення дати у дні та навпаки. Створити приклад для демонстрації усіх функціональних можливостей
4	Для роботи з грошима у форматі: *гривні, копійки* створити клас, який дозволяє: a) вводити грошову суму з клавіатури (файла); b) виводити грошову суму на екран (файл); c) знаходити суму/різницю двох грошових сум; d) переведення грошової суми у копійки та навпаки. Створити приклад для демонстрації усіх функціональних можливостей
5	Для роботи з раціональними числами (M/N, M – ціле число, N – натуральне число) створити клас, який дозволяє: a) вводити раціональне число з клавіатури (файла); b) виводити раціональне число на екран (файл); c) знаходити суму / різницю / добуток / двох раціональних чисел; d) приведення дробу до нескоротного виду; e) переведення раціонального числа у double. Створити приклад для демонстрації усіх функціональних можливостей
6	Для роботи з трикутниками на площині створити клас, який дозволяє: a) вводити координати трикутника з клавіатури (файла); b) виводити координати трикутника на екран (файл); c) знаходити площу/периметр трикутника; d) знаходиити радіуси вписаної та описаної окружностей. Створити приклад для демонстрації усіх функціональних можливостей
7	Для роботи з відрізками на площині створити клас, який дозволяє: a) вводити координати відрізка з клавіатури (файла);

	b) виводити координати відрізка на екран (файл); c) знаходити довжину відрізка; d) знаходити координати середини відрізка; e) автоматичне масштабування відрізка (функція повертає масштабовану копію відрізка, перша координата залишається без змін). Створити приклад для демонстрації усіх функціональних можливостей
8	Для роботи з колами на площині створити клас, який дозволяє: a) вводити параметри кола з клавіатури (файла); b) виводити параметри кола на екран (файл); c) знаходити площу кола; d) знаходити довжину околу; e) автоматичне масштабування кола (функція повертає масштабовану копію кола зі зміненим радіусом, координати центру залишаються без змін). Створити приклад для демонстрації усіх функціональних можливостей
9	Для роботи з прямокутниками на площині створити клас, який дозволяє: a) вводити координати прямокутника з клавіатури (файла); b) виводити координати прямокутника на екран (файл); c) знаходити площу/периметр прямокутника; d) знаходити радіуси вписаної та описаної окружностей. Створити приклад для демонстрації усіх функціональних можливостей
10	Для роботи з точками на площині створити клас, який дозволяє: a) вводити координати точки з клавіатури (файла); b) виводити координати точки на екран (файл); c) знаходити відстані між двома точками; d) знаходити найменшу/найбільшу відстань від заданої точки, до множини точок (заданих масивом). Створити приклад для демонстрації усіх функціональних можливостей

ЗАВДАННЯ №1.3. Для завдання 1.1 реалізувати клас шаблонів.

Індивідуальна робота № 2. Об’єктно-орієнтоване програмування

ЗАВДАННЯ № 2.1. Необхідно створити програму, що зчитує з файла параметри геометричних фігур та створює у пам’яті масив об’єктів, які відповідають цим фігурам. Після чого виводять назву та параметри кожної з фігур, а також її периметр та площу.

Програма має підтримувати такі геометричні фігури:

- трикутник;
- рівносторонній трикутник;
- прямокутник;
- квадрат;
- ромб;
- коло;
- еліпс;

Ім’я текстового файлу має передаватися в якості параметра командного рядка. Текстовий файл мусить мати такий формат:

- перший рядок – кількість фігур;
- далі – ідентифікатор та параметри фігури в окремому рядку

наприклад:

3
triangle 3.0 3.0 0.0 0.0 0.0 3.0
square 0.0 0.0 0.0 4.0 4.0 4.0 4.0 0.0
circle 3.0 3.0 6.0

Організація контролю знань

Запитання для підготовки до іспиту

1. Класи та об’єкти.
2. Створення класів та об’єктів.
3. Методи, дані та властивості.
4. inline та не inline методи.
5. Специфікатори public, private, protected.
6. Конструктор.
7. Конструктор копії.
8. Конструктор переміщення.
9. Деструктор.
10. Використання методів.
11. Вказівник this.
12. Константні методи класу.
13. Перевантаження методів.
14. Копіювання та переміщення об’єктів.
15. Перевантаження операторів присвоєння, що копіюють та переміщують.
16. Перевантаження операторів.
17. Дружні функції.
18. Дружні класи.
19. Дружні операції.
20. Масиви об’єктів.
21. Абстракція.
22. Інкапсуляція.
23. Наслідування.
24. Відкрите, закрите та захищене наслідування.
25. Виклик конструкторів та деструкторів при наслідуванні.
26. Перевантаження методів у класі-насліднику.
27. Поліморфізм.
28. Віртуальні функції.
29. Раннє зв’язування.
30. Пізнє зв’язування.
31. Абстрактні класи.
32. Абстрактні функції.
33. Перекриття методів.
34. Приведення типів.
35. Вкладені класи.
36. Множинне наслідування.

37. Віртуальне наслідування.
38. Переваги використання наслідування.
39. Недоліки використання наслідування.
40. Способи обробки помилок.
41. Генерація та перехоплення виключень.
42. Розробка коду, що є безпечним до виникнення виключень.
43. Шаблони функцій.
44. Шаблонні оператори.
45. Шаблони класів.
46. Параметри шаблонів, що не є типами.
47. Узагальнене програмування.
48. Спеціалізація шаблонів.
49. Шаблонні методи класу.
50. Шаблони та наслідування.
51. Переваги та недоліки використання наслідування.
52. STL – стандартна бібліотека шаблонів мови C++ (вектор, двозв'язний список, множина).
53. Ключові слова ANSI C++11.
54. Перечислення з областю видимості.
55. Спискова ініціалізація.
56. Оголошення: auto та decltype.
57. Цикл for оснований на діапазоні.
58. Хвостовий специфікатор типа повертаємого значення.
59. Нульовий вказівник nullptr.
60. Посилання lvalue та rvalue.
61. Посилання на тимчасові об'єкти.
62. Простір імен.

Критерії оцінювання та розподіл балів

Поточний контроль знань складається з двох індивідуальних робіт. Правильно виконані та захищені індивідуальні роботи оцінюються в 33 та 17 балів відповідно. Загальна кількість балів за поточний контроль – 50.

Завдання на підсумковий модульний контроль складається як з теоретичних, так і практичних запитань. Максимальна кількість балів за модульний контроль становить 20 балів.

Екзаменаційний білет складається з двох теоретичних питань та двох практичних завдань. Максимальна сума балів за екзаменаційну роботу становить 30.

Розподіл балів за поточний контроль, підсумковий контроль, індивідуальні завдання, іспит наведено в таблиці.

Вид роботи	Кількість балів
Індивідуальна робота № 1. Завдання 1.1	10
Індивідуальна робота № 1. Завдання 1.2	15
Індивідуальна робота № 1. Завдання 1.3	8
Індивідуальна робота № 2. Завдання 2.1	17
Модульний контроль	20
Екзамен	30
Разом	**100**

Шкала оцынювання знань:

Сума балів за всі види навчальної діяльності	Оцінка ECTS	Оцінка за національною шкалою
90 – 100	**A**	відмінно
82 – 89	**B**	добре
75 – 81	**C**	
67 – 74	**D**	задовільно
60 – 66	**E**	
35 – 59	**F**	незадовільно з можливістю повторного складання
0 – 34	**FX**	незадовільно з обов’язковим повторним вивченням дисципліни

Список рекомендованих джерел

1. Прата Стивен. Язык программирования C++. Лекции и упражнения / Стивен Прата. – 6-е изд.; пер. с англ. – М.: ООО «И. Д. Вильямс», 2012. – 1248 с.
2. Кравець П. О. Обєктно-орієнтоване програмування: навч. посібник / П. О. Кравець. – Львів: Видавництво Львівської політехніки, 2012. – 624 с.
3. Объектно-ориентированый анализ и проектирование с примерами приложений / Г. Буч, Р. Максимчук, Є. Майкл и др. – 3-е изд. – М.: ООО «И. Д. Вильямс», 2010. – 720 с.
4. Вайсфельд М. Объектно-ориентированное мышление / М. Вайсфельд. – СПб.: Питер, 2014. – 304 с.
5. Ray Lischner. Exploring C++11 / Lischner Ray. – Apress, 2013. – 617 p.
6. Павловская Т. А. C/C++. Структурное программирование: Практикум. / Т. А. Павловская, Ю. А. Щупак. – Спб.: Питер, 2003. – 240 с.
7. Васильев А. Н. Программирование на C++ в примерах и задачах / А. Н. Васильев. – Москва: Издательство «Э», 2017. – 368 с.
8. Кононова Е. А. Алгоритмы и программы. Язык C++: Учебное пособие / Е. А. Кононова, Г. А. Поллак. – 2-е изд. – СП.: Издательство «Лань», 2017. – 384 с.
9. Grimes Richard. Beginning C++Programming. / Richard Grimes. – Birmingham: «Packt Publishing», 2017. – 576 p.
10. Задания для занятий по программированию на языке C++ / Сост.: С. А. Калоеров, Е. В. Авдюшина, А. И. Ануфриева и др.– Донецк: Юго-Восток, 2010. – 96 с.
11. Сван Том. Освоение Borland C++ 4.5. Практический курс. / Том Сван. – К.: «Диалектика», 1996. – 544 с.
12. C plus plus. [Електронний ресурс] – Режим доступу http://www.cplusplus.com/