**Лекція 21. Абстрактні класи. Базові поняття та основні принципи**

**Опис протоколу класу.**

**class *Ім’я\_Класу***

**{ private:**

***тип*  ідентифікатор;**

**. . . . . . . . . . . . . . . . . . .**

***тип*  ідентифікатор;**

***тип\_результату Ім’я\_Функції*11 (сигнатура);**

**. . . . . . . . . . . . . . . . . . .**

***Тип*\_*результату Ім’я\_Функції*1N1(сигнатура);**

**protected:**

***тип*  ідентифікатор;**

**. . . . . . . . . . . . . . . . . . .**

***тип*  ідентифікатор;**

***тип\_результату Ім’я\_Функції2*1(сигнатура);**

**. . . . . . . . . . . . . . . . . . .**

***Тип\_результату Ім’я\_Функції*2N2(сигнатура);**

**public:**

***тип* ідентифікатор;**

**. . . . . . . . . . . . . . . . . . .**

***тип* ідентифікатор;**

***тип\_результату Ім’я\_Функції*31(сигнатура);**

**. . . . . . . . . . . . . . . . . . .**

***Тип\_результату Ім’я\_Функції*3N3(сигнатура);**

***Ім’я\_Класу*(сигнатура А);**

***Ім’я\_Класу*(сигнатура Б)**

**{ тіло функції }**

**~*Ім’я\_Класу*(void);**

**};**

***Тип\_результату* *Ім’я\_Класу*:: *Ім’я\_Функції*11(сигнатура)**

**{ тіло функції}**

***Тип\_результату* *Ім’я\_Класу*:: *Ім’я\_Функції*12(сигнатура)**

**{тіло функції}**

**. . . . . . . . . . . . . . . . . . . . .**

***Тип\_результату* *Ім’я\_Класу*:: *Ім’я\_Функції*3N3(сигнатура)**

**{тіло функції}**

***Ім’я\_Класу::Ім’я\_Класу*(сигнатура А)**

**{тіло функції}**

***Ім’я\_Класу::Ім’я\_Класу* (сигнатура Б)**

**{ тіло функції }**

***Ім’я\_Класу::Ім’я\_Класу :*~ *Ім’я\_Класу*(void)**

**{ тіло функції }**

Всі програмні компоненти, що містяться між відкриваючою та закриваючою фігурними дужками опису класу (“{“ - ”};”) утворюють *формальний опис класу*. Під *протоколом класу* будемо розуміти область, що складається з області формального опису класу та тіл функцій-членів, визначених за межами формального опису класу при допомозі операції розширення області видимості.

Приклад.1. Цей приклад будемо використовувати далі за текстом.

**class Ttime {**

**private:**

**long dt;**

**char\*dts;**

**void DeleteDts(void);**

**publiс:**

**TTime();**

**TTime(int m,int d=-1,int y=-1,int hr=-1,int min=-1);**

**~TTime(); //деструктор**

**void SetTime (int m=-1,int d=-1,int y=-1,int hr=-1,int min=-1);**

**const char\*GetsTime(void);**

**void ChangeTime(long nminutes)**

**{dt+=(nminutes\*60);**

**DeleteDts();}**

**char \* GetsTime(void)**

**{ if(dts) return dts;**

**dts = strdup(ctime(&dt));**

**return dts;**

**}**

**};**

**TTime ::TTime()**

**{dts=NULL;**

**SetTime(-1,-1,-1,-1,-1);**

**}**

**TTime ::Ttime(int m,int d,int y,int m,int min)**

**{dts=NULL.**

**SetTime(m,d,y,hr,min);}**

**TTime ::~TTime()**

**{DeleteDts;}**

**Масиви об’єктів .**

Масив об’єктів синтаксично задається як звичайний масив:

Ttime tentimes[10];

При цьому клас, екземпляри якого утворюють масив, повинен обов’язково мати конструктор по замовчуванню. Розглянемо приклад роботи з масивами об’єктів:

main(void)

{ Ttime tentimes[10];

for(int i=0;i<10;i++) tentimes[i].display();

return 0;

}

Очевидно, що тут буде надруковано 10 однакових значень.

Можна ініціалізувати масив , наприклад, так:

main(void)

{

Ttime tarray[3]={ Ttime(),Ttime(8),Ttime(8,1)};

for(int i=0;i<3;i++) tarray[i].display();

return 0;

}

Масиви об’єктів можуть розміщуватись в кучі та адресуватись за допомогою вказівників:

main()

{Ttime\* tarrayp;

tarrayp=new Ttime[6];

for(int i=0;i<6;i++)

tarrayp[i].display();

delete[ ] tarrayp;

return 0;

}

Відмітимо, що в цьому прикладі 6 разів викликається конструктор по замовчуванню при виділенні пам’яті оператором new та 6 разів викликається деструктор при звертанні до delete, яка вжита в специфічній формі - з квадратними дужками (C++ ігнорує будь-які значення всередині цих дужок)

**Чисті віртуальні функції. Абстрактні класи.**

Чиста або суто вiртуальна функцiя-член - це прототип функцiї, для якого не потрiбне визначення в протоколi класу. Cинтаксично чиста віртуальна функція задається так:

**virtual<тип><iм'я>(сигнатура)=0;**

Клас, який мiстить чисті вiртуальнi функцiї називається абстрактним.

Приклад:

**class AbstractClass {**

**public:**

**virtual void f1(void);**

**virtual void f2(void)=0;**

**//...};**

Специфiка абстрактного класу полягає в тому, що вiн не може мати екземплярiв. При оголошенні AbstractClass my; буде помилка на етапі компіляції. Для того, щоб використовувати абстрактні класи, необхідно визначити похідний від нього клас, в якому чиста віртуальна функція оголошується як звичайна віртуальна функція. Причому в цьому випадку вона повинна мати тіло. Розглянемо, наприклад, такий клас:

**class MyClass:public AbstractClass{**

**public:**

**virtual void f2(void);**

**// }**

**void MyClass::f2(void)**

**{\\тіло}**

Звернемо увагу на те, що у класi MyClass f2() є вже звичайною вiртуальною функцією (а не чистою), отже повинне бути її визначення(в даному випадку-за межами формального опису класу). Тодi, очевидно, можемо описати екземпляр класу MyClass.

Чиста вiртуальна функцiя може вiльно використовуватись в протоколi абстрактного класу.

Похідний клас, який не визначає всi чистi вiртуальнi функцiї базового класу, також являється абстрактним.

Уявимо собі, що базовий клас настільки абстрактний, що визначити заздалегідь, як повинна виглядати його віртуальна функція-член, неможливо. У таких ситуаціях її конкретне втілення конкретизується лише в похідних класах на основі додаткової інформації. Крім того, іноді необхідно забезпечити заміщення віртуальних функцій в усіх без винятку похідних класах. Як ми бачили вище, механізму звичайних віртуальних функцій для цього недостатньо — компілятор ніяк не реагує на відсутність заміщеної версії функції print() у класі TDerived2.

Для вирішення цієї проблеми в мові С++ реалізований механізм суто віртуальних функцій. Як визначалось вище: ***Суто або чисто віртуальною функцією називається віртуальна функція-член базового класу, що не має визначення.***

Пригадаємо приклад з попередньої, причому закоментуємо заміщення функції print() в об'єкті objDerived2.

#include <stdio.h>

class TInner

{

public:

int i;

TInner(int n):i(n){}

~TInner(){printf("Dtor TInner\n");}

};

class TBase

{

public:

double a;

TInner b;

TBase():b(10),a(5.0){printf("Ctor TBase\n");}

~TBase(){printf("Dtor TBase\n");}

virtual void print() {printf("TBase::TInner::i = %d a = %lf\n",b.i, a);}

};

class TDerived: public TBase

{

public:

char c;

TDerived():c('Z'){printf("Ctor TDerived\n");}

~TDerived(){printf("Dtor TDerived\n");}

void print() {printf("TDerived::TInner::i = %d c = %c\n",b.i,c);}

};

class TDerived2: public TDerived

{

public:

float f;

TDerived2():f(10.0){printf("Ctor TDerived2\n");}

~TDerived2(){printf("Dtor TDerived2\n");}

void print() {printf("TDerived2::TInner::i = %d f = %lf\n",b.i,f);}

};

int main()

{

TBase objBase;

TDerived objDerived;

TDerived2 objDerived2;

TBase\* pObj = &objBase;

pObj->print();

pObj=&objDerived;

pObj->print();

pObj=&objDerived2;

pObj->print();

return 0;

}

Результати роботи програми тепер виглядають так.

1. Ctor TBase

2. Ctor TBase

3. Ctor TDerived

4. Ctor TBase

5. Ctor TDerived

6. Ctor TDerived2

7. TBase::TInner::i = 10 a = 5.000000

8. TDerived::TInner::i = 10 c = Z

9. TDerived::TInner::i = 10 c = Z

10. Dtor TDerived2

11. Dtor TDerived

12. Dtor TBase

13. Dtor TInner

14. Dtor TDerived

15. Dtor TBase

16. Dtor TInner

17. Dtor TBase

18. Dtor TInner

Рядок 9 свідчить про те, що при виклику функції print() через вказівник на об'єкт objDerived2 виробляється виклик попередньої заміщеної версії. Отже, віртуальні функції утворять ієрархію*.*

Допустимо, нам невідома конкретна реалізація класу TInner. Закриємо очі на те, що зараз він містить поле типу int. Можливо, замість цілого числа там буде число типу double чи об'єкт іншого класу. У такому випадку клас TBase не повинний залежати від конкретної реалізації класу TInner. Тоді незрозуміло, як виводити на екран поля класу TBase. У цьому випадку необхідно оголосити функцію print() суто віртуальною, поклавши відповідальність за вивід на екран полів базового класу на функції, що заміщають

Приклад:

**#include <iostream.h>**

**#include <string>**

**#include <stdio.h>**

**class TInner**

**{**

**public:**

**int i;**

**TInner(int n):i(n){}**

**~TInner(){printf("Dtor TInner\n");}**

**};**

**class TBase**

**{**

**public:**

**double a;**

**TInner b;**

**TBase():b(10),a(5.0){printf("Ctor TBase\n");}**

**~TBase(){printf("Dtor TBase\n");}**

**virtual void print() = 0;**

**};**

**class TDerived: public TBase**

**{**

**public:**

**char c;**

**TDerived():c('Z'){printf("Ctor TDerived\n");}**

**~TDerived(){printf("Dtor TDerived\n");}**

**void print() {printf("TDerived::TInner::i = %d c = %c\n",b.i,c);}**

**};**

**class TDerived2: public TDerived**

**{**

**public:**

**float f;**

**TDerived2():f(10.0){printf("Ctor TDerived2\n");}**

**~TDerived2(){printf("Dtor TDerived2\n");}**

**};**

**int main()**

**{**

**//TBase objBase; Помилка! Клас TBase — абстрактний!**

**TDerived objDerived;**

**TDerived2 objDerived2;**

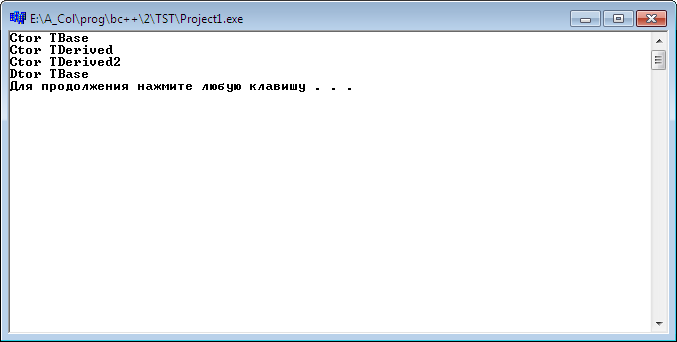
**objDerived.print();**

**objDerived2.print();**

**return 0;**

**}**

Результат роботи програми



Варто мати на увазі, що похідні класи повинні заміщати суто віртуальну функцію. Це не суперечить наведеній вище програмі, оскільки в сформульованому вище правилі маються на увазі безпосередні спадкоємці базового класу. Оскільки клас TDerived2 є похідним від класу TDerived, на нього це правило не поширюється. Однак якби клас TDerived2 був прямим спадкоємцем класу TBase, виникла б помилка компіляції, тобто при визначенні:

class TDerived2: public **TBase**

{

public:

float f;

TDerived2():f(10.0){printf("Ctor TDerived2\n");}

~TDerived2(){printf("Dtor TDerived2\n");}

// Відсутнє перевизначення суто віртуальних функцій**.**

};

**Механізм віртуальних функцій**

Об'єкт класу розміщається в суцільній області пам'яті, адреса якої зберігається в неявному вказівнику this. При виклику звичайної функції-члена цей вказівник передається їй як додатковий аргумент. Створюючи об'єкт похідного класу, компілятор поєднує його поля і поля базового класу в одне ціле. Якщо базовий клас містить віртуальні функції, їх адреси заносяться в таблицю віртуальних функцій. Усі класи, що утворюють поліморфічний кластер, містять вказівник на цю таблицю, у якій зберігаються вказівники на усі віртуальні функції-члени класів.

Розглянемо наступну програму. Вказівник на таблицю віртуальних функцій

**#include <stdio.h>**

**class TBase**

**{**

**public:**

**char a;**

**TBase():a('X'){printf("Ctor TBase\n");}**

**~TBase(){printf("Dtor TBase\n");}**

**//virtual void print() {printf("TBase::a = %c\n",a);}**

**};**

**class TDerived: public TBase**

**{**

**public:**

**char b;**

**TDerived():b('Y'){printf("Ctor TDerived\n");}**

**~TDerived(){printf("Dtor TDerived\n");}**

**void print() {printf("TDerived::b = %c\n",b);}**

**};**

**class TDerived2: public TDerived**

**{**

**public:**

**char c;**

**TDerived2():c('Z'){printf("Ctor TDerived2\n");}**

**~TDerived2(){printf("Dtor TDerived2\n");}**

**void print() {printf("TDerived2::c %c \n",c);}**

**};**

**int main()**

**{**

**printf("Sizeof(char) = %d\n",sizeof(char));**

**printf("Sizeof(TBase) = %d\n",sizeof(TBase));**

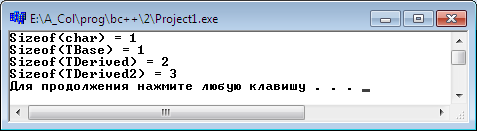
**printf("Sizeof(TDerived) = %d\n",sizeof(TDerived));**

**printf("Sizeof(TDerived2) = %d\n",sizeof(TDerived2));**

**return 0;**

**}**

Результат виконання програми.

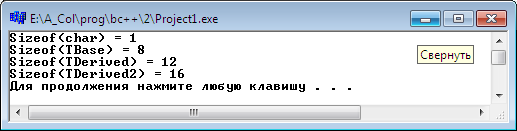


Отже, розмір типу char дорівнює одному байту. Клас TBase містить одне поле типу char, отже, його розмір також дорівнює одному байту. Клас TDerived успадковує поле типу char, додаючи його до власного члена b. Отже, розмір збільшується вдвічі. Аналогічна ситуація виникає в класі TDerived2, що містить три поля типу char — одне своє і два успадкованих.

Тепер, якщо зняти коментар з рядка:

// virtual void print() {printf("TBase::a = %c\n",a);}

Результати роботи програми стануть зовсім іншими.



Розмір вказівника на таблицю віртуальних функцій дорівнює 4 байт, отже, справжній розмір класу TBase повинний бути дорівнює 5 байт. Однак за рахунок вирівнювання машинного слова він збільшується до 9 байт. Отже, розмір класу TDerived повинний бути рівним 9+4=12 байт. Як бачимо, це відповідає дійсності. Аналогічно, sizeof(TDerived2)= sizeof(TDerived)+sizeof(вказівник)= 12+6=16 байт. Таким чином, ця програма підтверджує загальновідомий факт, що кожен об'єкт поліморфічного кластера містить вказівник на таблицю віртуальних функцій. Виключення перевизначених версій з похідних класів нічого не змінює — віртуальні функції утворять ієрархію й успадковуються всіма похідними класами.

Чиста віртуальна функція корисна, коли маємо функцію, яку хочемо помістити до базового класу, а реалізацію залишити похідним класам. Чиста віртуальна функція абстрактного базового класу змушує похідні класи перевизначити цю функцію, інакше об’єкти цих класів створювати буде неможливо.

При визначенні чистої віртуальної функції, її тіло (визначення) повинно бути записано окремо (не вбудоване), якщо ви хочете, щоб похідні класи мали можливість перевизначати віртуальну функцію або залишити її реалізацію по замочуванню (яку надає базовий клас). Якщо похідний клас задовольняє замовчувана реалізація, він може просто викликати її напряму. Наприклад:

#include <iostream>

#include <string>

#include <Windows.h>

class Animal // це абстрактний батьківський клас

{

protected:

    std::string m\_name;

public:

    Animal(std::string name): m\_name(name) {}

    std::string getName() { return m\_name; }

    virtual const char\* speak() = 0;

// Увага, speak() є чиста віртуальна функція

};

const char\* Animal::speak()

{

    return "buzz"; // реалізація по замовчанню

}

class Dragonfly: public Animal

{

public:

    Dragonfly(std::string name): Animal(name) {    }

    virtual const char\* speak()

// цей клас вже не є абстрактним, оскільки

// перевизначили функцію speak()

    {

        return Animal::speak();

// використовується реалізація по замовчуванню класу Animal

    }

};

int main()

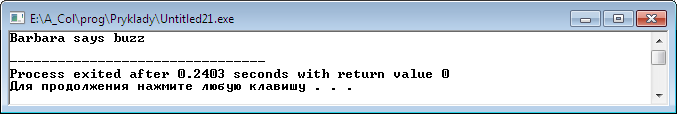
{system("color F0");

    Dragonfly dfly("Barbara");

    std::cout << dfly.getName() << " says " << dfly.speak() << '\n';

}

Результат



**Інтерфейс**

**Інтерфейс** — це клас, який не має змінних-членів і всі методи якого є чистими віртуальними функціями! інтерфейси ще називають «**класами-інтерфейсами**» або «**інтерфейсними класами**». Інтерфейсні класи зазвичай називають з початкової I. Приклад.

class IErrorLog

{

public:

    virtual bool openLog(const char \*filename) = 0;

    virtual bool closeLog() = 0;

    virtual bool writeError(const char \*errorMessage) = 0;

    virtual ~IErrorLog() {};

/\* створюємо віртуальний деструктор на випадок вилучення вказівника на IErrorLog, щоб викликався відповідний деструктор дочірнього класу \*/

};

Будь-який клас, що спадкує IErrorLog, повинен надати свою реалізацію всіх трьох методів класу IErrorLog. Можна створити похідний клас з іменем FileErrorLog, де openLog() відкриває файл на диску, closeLog() закриває файл, а writeError() записує повідомлення до файлу. Можна створити ще один похідний клас з іменем ScreenErrorLog, де openLog() та closeLog() нічого не роблють, а writeError() виводить повідомлення.

Якщо потрібно написати програму, яка використовує журнал помилок, то написання класів FileErrorLog або ScreenErrorLog напряму буде не ефективним. Наприклад, наступна функція змушує всі об’єкти, що викликають mySqrt(), використовувати FileErrorLog, що може бути недоцільним:

#include <cmath> // для sqrt()

double mySqrt(double value, *FileErrorLog &log*)

{

    if (value < 0.0)

    {

        log.writeError("Tried to take square root of value less than 0");

        return 0.0;

    }

    else

        return sqrt(value);

}

Кращим варіантом буде реалізація через IErrorLog:

#include <cmath> // для sqrt()

double mySqrt(double value, *IErrorLog &log*)

{

    if (value < 0.0)

    {

        log.writeError("Tried to take square root of value less than 0");

        return 0.0;

    }

    else

        return sqrt(value);

}

Тепер користувач через передачу об’єктів може визначити самостійно, який клас потрібно викликати. Якщо він хоче записати помилку до файлу, то він передасть в функцію mySqrt об’єкт класу FileErrorLog. Якщо хоче, вивести помилку на екран - передасть об’єкт класу ScreenErrorLog. А якщо захоче відправити комусь Email-ом повідомлення про помилку (що не передбачене), то він може створити новий похідний клас EmailErrorLog, який буде спадковувати IErrorLog, та передавати об’єкт цього класу. Таким чином, реалізація через IErrorLog робить нашу функцію більш гнучкою та незалежною.

Потрібно пам’ятати про підключення [**віртуальних деструкторів**](https://ravesli.com/urok-165-virtualnye-destruktory-i-prisvaivanie/) до інтерфейсних класів, щоб при видаленні покажчикана інтерфейс викликався деструктор відповідного (похідного) класу.

#include <iostream>

#include <string>

#include <Windows.h>

class Animal // це абстрактний батьківський клас

{

public:

    Animal(){std::cout << "Animal Ctor \n" ;}

    virtual ~Animal(){std::cout << "Animal Dtor \n" ; }

    virtual const char\* speak() = 0;

// Увага, speak() є чиста віртуальна функція

};

const char\* Animal::speak()

{

    return "buzz"; // реалізація по замовчанню

}

class Dragonfly: public Animal

{

public:

    Dragonfly() {std::cout << "Dragonfly Ctor \n" ;};

    virtual const char\* speak()

// цей клас вже не є абстрактним, оскільки

// перевизначили функцію speak()

    {

        return Animal::speak();

// використовується реалізація по замовчуванню класу Animal

    }

};

int main()

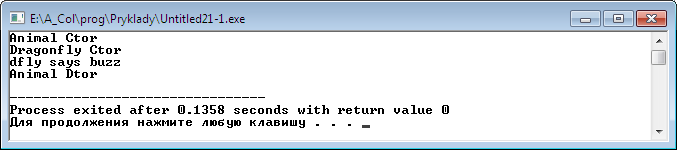
{system("color F0");

    Dragonfly dfly;

    std::cout << "dfly says " << dfly.speak() << '\n';

}

Результат



Класи-інтерфейси є відомим та популярним засобом проектування гнучких архітектурних рішень програмних засобів шляхом роз’єднання інтерфейсів та їхньої реалізації. Можна відзначити цінність підходу використання класів-інтерфейсів при командній розробці складних проектів з множини слабо зв’язаних класів та модулів.

Існує традиція назви класів-інтерфейсів розпочинати з букви I.

Приклад класу-інтерфейсу:

class IOpenable

{

public:

virtual int Open(void) = 0;

virtual void Close(void) = 0;

};

Таким чином, клас-інтерфейс записується тільки у вигляді оголошення.

Згідно канонів ООП класи-інтерфейси мають такі загальні риси:

- Клас-інтерфейс, або просто інтерфейс – це абстракція, яка описує деякий набір функцій

- Може мати тільки public функції-члени (методи). Для жодної з функцій класу-інтерфейсу не може бути у цьому класі визначена реалізація за замовчуванням. Не може містити дані-члени (поля), конструктор та деструктор

- Інтерфейс може розширювати інший інтерфейс

- Клас який успадковує (реалізує) клас-інтерфейс для створення об’єктів, повинен реалізовувати усі функції інтерфейсу

- Класи можуть успадковувати (реалізовувати) декілька інтерфейсів

Існує думка, що в мові С++ класи-інтерфейси (або просто інтерфейси) в «чистому» канонічному вигляді реалізувати не можна. Впродовж численних ітерацій розширення синтаксису мови С++ у багатьох стандартах так і не була втілена «чиста» конструкція інтерфейсу. Вважається, що поняття абстрактного класу разом із можливостями множинного успадкування в С++ повністю заперечує необхідність існування інтерфейсу як окремого синтаксичного елементу мови. Проте у деяких реалізаціях мови С++ можна зустріти спроби ввести слово «інтерфейс». Так, зокрема, у Microsoft Visual C++ є навіть два ключових слова: interface та \_\_interface. Перше з них призначене для .Net Framework, а друге – для native C++.

*Для самостійного вивчення*: Поглибити матеріал лекції за наданою літературою. Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.

*Контрольні запитання для самоперевірки*.

1. Чому при знищенні масиву об’єктів необхідно використовувати специфічну форму оператора delete ?
2. Як впливають віртуальні функції на розмір класу?
3. Навіщо потрібні віртуальні функції?
4. Чим об'єкт класу, що має віртуальні функції, відрізняється від об'єкта класу без віртуальних функцій?
5. Що станеться, якщо віртуальна функція не заміщується у якомусь похідному класі?

*Контрольні запитання для надання письмових відповідей*.

1. Поясніть різницю між успадкуванням інтерфейсу та успадкуванням реалізації
2. Що таке таблиця віртуальних функцій? Опишіть її заповнення в процесі виконання програми.

*Література*

1. Джейс Либерти Освой самостоятельно С++ за 21 день: 3-е изд. пер. с англ.: Уч. пос. – М.: Издательский дом „Вильямс”, 2001. – 816 с.: ил..
2. Павловская Т.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2003. – 461 с. URL:  <http://www.ph4s.ru/bookprogramir_1.html>
3. Липпман С. Б., Лажойе Ж. Язык программирования С++: Вводный курс. — М.: ДМК, 2001. URL: <http://www.insycom.ru/html/metodmat/inf/Lipman.pdf>
4. Дейтел Х., Дейтел П. Основы программирования на С++. – М.: Бином, 1999. – 1024 с. URL:  <http://ijevanlib.ysu.am/wp-content/uploads/2018/03/deytel.pdf>
5. Бадд Т. Объектно-ориентированное программирование в действии. [2-е изд.] – СПб.: Изд-во "Питер". 1997.  URL: <http://khizha.dp.ua/library/Timothy_Budd_-_Introduction_to_OOP_(ru).pdf>.
6. *Герб Саттер,Андрей Александреску* "Стандарты программирования на С++ ", [*Вильямс, 2005*](http://www.williamspublishing.com/)*; 304 с.*
7. *Скотт Мейерс.* Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов. *"ДМК", 2000; 240 с.*
8. *Скотт Мейерс.* Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов.*"ДМК",2000;304 с.*