МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

„Моделювання засобами С++”

Навчальний посібник.

.

Рекомендовано на засіданні
методичної ради НТУУ «КПІ»
протокол № 10 від 16 червня 2009р.

Київ
“АВЕРС”
200_

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

„Моделювання засобами С++”

Навчальний посібник.

Рекомендовано на засіданні
методичної ради НТУУ «КПІ»
протокол № 10 від 16 червня 2009р.

Київ
“АВЕРС”
200_

„Моделювання засобами С++” Навчальний посібник: Уклад.: О.В. Мачулянський, Д.Д. Татарчук.-К.: „АВЕРС” , 2008.-__ с.

Зміст посібника відповідає програмі базової вищої освіти з напрямку 6.050801 – “ мікроелектроніка та наноелектроніка”

Укладачі:
О.В. Мачулянський, к.т.н., доц.
Д.Д. Татарчук, к.т.н., доц.

Відповідальний
редактор О.В. Борисов, к.т.н., проф.

Рецензенти: В.Г. Вербицький, д.т.н., с.н.с.

О.М. Юрченко, д.т.н., проф.

Підп. до друку 00.00.0_. Формат $60 \times 84\ 1/16$. Папір друк. №3. Друк офс.

Ум. друк. арк. 0.0. Наклад 200пр.
Видавництво ПЦ “АВЕРС”
03056, Київ, вул. Політехнічна, 16, тел.:241 86 00

ЗМІСТ

ВСТУП

В зв’язку з швидким розвитком обчислювальної техніки широкого розповсюдження при проведенні наукових досліджень та інженерного проектування набув обчислювальний експеримент. Обчислювальний експеримент базується на побудові та аналізі за допомогою ЕОМ математичних моделей досліджуваного об’єкту.

Розглянемо схему обчислювального експерименту (рис. 1.1) [1].

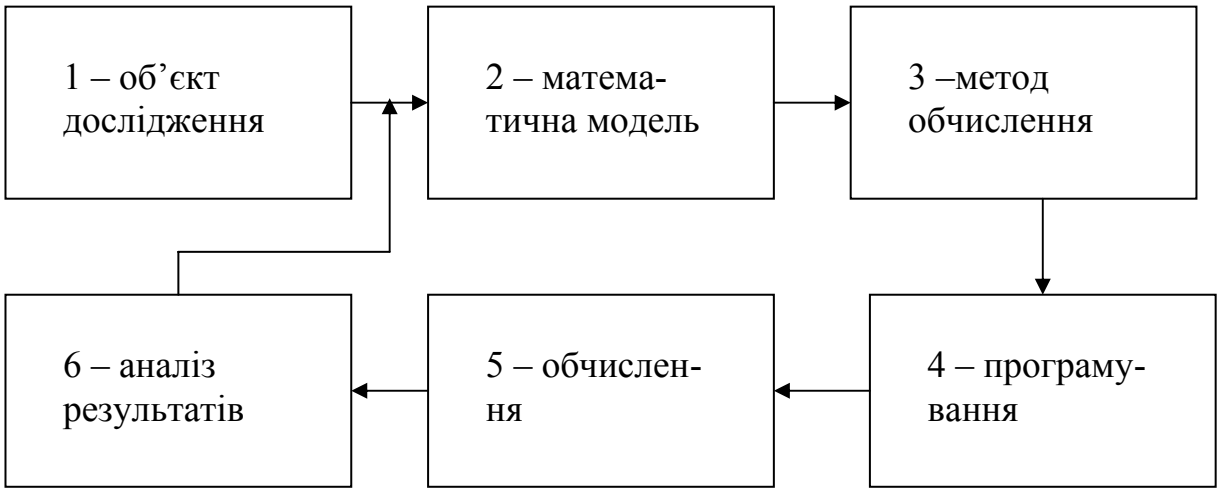

Рис. 1.1. Схема обчислювального експерименту

Нехай необхідно дослідити якийсь об’єкт, явище або процес (1). Тоді спочатку формулюються основні закони та взаємозв’язки, що описують даний об’єкт. На їх основі розробляється математична модель (2), що являє собою, як правило, запис цих законів у вигляді системи рівнянь (алгебраїчних, диференціальних, інтегральних і т.д.). Після того, як задачу сформульовано, її необхідно розв’язати. Тільки в досить простих випадках вдається отримати розв’язок у явному вигляді. В більшості випадків виникає необхідність використання того чи іншого наближеного методу (обчислювального методу або дискретної моделі). На основі отриманої дискретної моделі будується обчислювальний алгоритм, результатом реалізації якого є число або таблиця чисел.

Для реалізації обчислювального методу необхідно розробити програму для ЕОМ (4). Після розробки та відладки програми наступає етап проведення обчислень (5). Отримані результати детально аналізують (6) з точки зору їх відповідності досліджуваному явищу і, при необхідності вносяться зміни в математичну модель або обирається інший обчислювальний метод. Цей цикл повторюється доти, доки не буде отримано результати з необхідною точністю.

З огляду на вказане вище, одним з найважливіших питань при розробці програми є питання правильної організації даних, тому ***предметом посібника*** є методологія розробки програмного забезпечення, а саме аспекти вибору структури даних та алгоритмів їх обробки, які у значній мірі зумовлюють ефективність програми і всього обчислювального експерименту[2].

Метою посібника є оволодіння теоретичними основами обробки структурованих даних, методами формування абстрактних типів даних на основі концепції об’єктно орієнтованого програмування.

Основними задачами посібника є формування теоретичної бази використання сучасних методів обробки даних на ЕОМ, а також вироблення практичних навичок використання абстрактних типів даних для розв’язку інженерних задач.

Посібник може бути використаний при вивченні дисциплін «Програмування та алгоритмічні мови» та «Алгоритми і структури даних», які є базовими для вивчення дисциплін «Обчислювальна математика», «Моделювання в електроніці.», «Моделювання технології та інтегральних мікросхем».

Структурно посібник складається з двох частин. В першій частині вивчаються основи об’єктно-орієнтованого програмування з використанням мови програмування С++. В другій частині розглядаються типові структури даних та методи роботи з ними.

Чому за основу при викладанні курсу вибрано саме мову програмування С++ ? Відповідь на це запитання дуже проста. По-перше – мова програмування С++ на сьогодняшній день є однією з найпоширеніших і набільш універсальних мов програмування, яка може успішно використовуватись для реалізації різноманітних задач, а саме системних, обчислювальних і т.д. По-друге – мова програмування С++ реалізована для багатьох операційних систем, що дає змогу не прив’язуватись до конкретної операційної системи, а використовувати ті засоби, які є в наявності. По-третє – ця мова має розвинені засоби об’єктно-орієнтованого програмування, що дає майже необмежені можливості для розробки та підтримки нестандартних (визначених користувачем) типів даних при розв’язку реальних інженерних задач.

ОСНОВНІ ПОНЯТТЯ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

Моделювання процесів реального світу є досить складною задачею, оскільки модель повинна відповідати багатьом вимогам, а саме:

- відображати процеси, що моделюються з високою точністю;

- мати однозначну відповідність між параметрами системи і фізичними процесами;

- бути достатньо простою для реалізації.

Ці вимоги є досить суперечливими. Так, наприклад, моделі, що відображають явища з високою точністю можуть бути занадто складними і потребувати занадто великих машинних ресурсів. Тому при моделюванні реальних явищ користуються спрощеними моделями, які враховують лише ті особливості системи, що є найбільш загальними і значущими для даної системи, або іншими словами використовують певний рівень абстрагування. При цьому, в процесі моделювання, розроблюються структури даних, що характеризують дане явище чи систему. Ці структури даних формуються з урахуванням рівня абстрагування і доступних можливостей засобів розробки, а саме мови програмування, апаратної бази, операційної системи і т.д. Такі структури даних називаються ***абстрактними типами даних***.

Для програмної реалізації абстрактних типів даних в мові програмування С++ використовується об'єктно-орієнтоване програмування (ООП).

Основна ідея об'єктно-орієнтованого програмування полягає в об'єднанні даних і алгоритмів для їх обробки в єдине ціле[3], єдину інформаційну структуру, єдиний тип даних. Цей тип даних специфікує члени-дані, звані полями, і операції для роботи з цими даними, звані методами. Поля описують параметри та характеристики системи, а методи описують поведінку системи, а саме взаємозв'язок між параметрами системи та реакції системи на вплив зовнішніх по відношенню до неї чинників.

Наприклад нам необхідно написати програму, що моделює поведінку системи, яка складається з пружини, закріпленої одним кінцем до стелі, і тягарця закріпленого на вільному кінці пружини. Нехай нас цікавить положення тягарця в залежності від його маси , параметрів пружини та прикладеної зовнішньої сили. Для простоти будемо розглядати одновимірний випадок, коли зміщення тягарця може відбуватись лише вздовж осі пружини.

Тоді абстрактний тип даних, що описує дану систему повинен містити для збереження параметрів системи наступні поля – змінну для збереження маси тягарця, змінні для збереження жорсткості пружини та довжини пружини у вільному стані, змінну для збереження величини зовнішньої сили. Також цей тип даних повинен містити функцію, яка вираховує положення тягарця в залежності від величини вищевказаних змінних, а також додаткові (службові) функції для встановлення потрібних значень вищевказаних змінних і відображення результатів розрахунку.

Концепція ООП базується на наступних принципах:

- абстрагування;
- інкапсуляція;
- успадкування;
- поліморфізм;
- повторне використання коду.

Дамо означення цих принципів.

Абстрагування – це виділення найбільш загальних і значимих особливостей досліджуваної системи з метою спрощення результуючої моделі. Результатом абстрагування є розроблення абстрактного типу даних.

Інкапсуляція – це поєднання в одній абстрактній інформаційній структурі даних (полів) та алгоритмів їх обробки (методів).

Успадкування – це засіб побудови нових (***породжених***) абстрактних інформаційних структур на основі уже існуючих (***базових***). При цьому породжені структури утворюються шляхом додавання нових полів та методів або шляхом модифікації існуючих і можуть успадковувати деякі або всі

властивості базових структур. Це дозволяє повторно використовувати існуючий програмний код і, за рахунок цього, прискорити розробку програмного забезпечення. Породжені структури називають ***потомками,*** а базові – ***предками***.

Розглянемо приклад. Нехай існує інформаційна структура яка описує фігуру трикутник, а нам треба побудувати структуру, що описує чотирикутник. Тоді, використовуючи успадкування, нам достатньо до структури трикутника додати ще дві змінних, які описують положення четвертої вершини, а також замінити процедуру відображення фігури на екрані. При цьому змінні, які описують положення перших трьох вершин, будуть успадковані від структури трикутник, і нам не треба буде витрачати час на їх реалізацію.

Поліморфізм – властивість інформаційної структури, яка полягає у можливості зміни її поведінки у залежності від того з якими іншими структурами вона взаємодіє.

Типовим прикладом є реалізація арифметичних операторів у мовах програмування, коли їх застосування до цілих чисел виконується за правилами цілочисельної математики, а їх застосування до чисел з плаваючою комою за іншими правилами. При цьому вибір правил відбувається автоматично. Звісно, що така можливість повинна бути закладена до інформаційної структури на етапі її розробки.

Повторне використання коду означає, що нові абстрактні типи можна не розроблювати з нуля. Нові типи даних можна розроблювати на основі вже існуючих. Для цього спочатку знаходять в бібліотеці тип (або типи даних), який (які) найбільше відповідає (відповідають) потребам, і модифікують, використовуючи механізми успадкування та поліморфізму.

Принципи об'єктно-орієнтованого програмування реалізовані по-різному в різних мовах програмування. Особливості реалізації ООП в мові програмування C++ будуть розглянуті далі.

Контрольні запитання

1. З яких етапів складається обчислювальний експеримент?
2. Які вимоги висуваються до моделей?
3. Що таке абстрактний тип даних?
4. В чому полягає основна ідея ООП?
5. Які основні принципи ООП?
6. Що таке абстракція в контексті ООП?
7. Що таке інкапсуляції?
8. Що таке успадкування?
9. Що таке поліморфізм?
10. Що означає повторне використання коду в контексті ООП?
11. Які переваги ООП?

ОСОБЛИВОСТІ ОБ'ЄКТНОЇ МОДЕЛІ В С++

Ключовим поняттям в об'єктно-орієнтованому програмуванні на С++ є поняття класу.

Клас в С++ – це практична реалізація абстрактного типу даних засобами мови програмування С++[4]. Фактично клас – це визначений програмістом нестандартний тип даних, тому поняття полів і методів класу повністю співпадають з аналогічними поняттями абстрактного типу даних. Процес визначення класу складається з двох частин.

Перша частина – це ***опис класу***. В цій частині визначається структура класу, а саме кількість та типи полів(властивостей) класу, кількість методів, кількість та типи вхідних параметрів методів, а також типи результатів виконання методів. Крім того на цьому етапі визначається область бачимості полів та методів.

Опис класу починається з ключового слова ***class***, за яким слідує назва класу. Він складається з розділів, які виділяються модифікаторами public, private, protected, що є заголовками розділів і визначають їх початок та область бачимості полів та методів, розташованих у розділі.

У розділах із заголовками ***public*** розміщуються загальнодоступні поля та методи, які використовуються для інтерфейсу об'єктів даного класу з програмою. Доступ до цих полів і методів може бути здійснений прямим зверненням.

У розділах із заголовками ***private*** розміщуються закриті дані, доступ до яких може бути здійснений лише за допомогою методів самого класу або за допомогою друзів класу. Поняття друзів класу буде розглянуто пізніше.

Розділи із заголовками ***protected*** містять дані, доступ до яких може бути здійснений за допомогою методів самого класу, за допомогою методів породжених класів або за допомогою друзів класу.

Розділ починається заголовком та закінчується заголовком іншого розділу або межею опису класу. Опис класу обмежується заголовком класу та фігурними скобками. „{", „}". Кількість розділів не лімітується. Клас може

складатися як з одного розділу, так із кількох, порядок розташування розділів довільний в межах опису класу. В описі класу може існувати кілька розділів з однаковими заголовками. Перший розділ опису класу може бути без заголовку. В цьому випадку вважається, що заголовком розділу є модифікатор private.

Друга частина визначення класу – це ***опис методів класу***. В цій частині реалізуються засобами мови програмування алгоритми методів класу.

Обидві частини є обов'язковими при визначенні класу і жодна з них не може бути пропущена. Опис класу та опис методів класу повинні бути розміщені у одному файлі. Зазвичай їх розміщують у окремих файлах із розширенням „h" і потім включають за допомогою директиви #include до кожного з тих файлів проекту, в яких використовуються об'єкти даного класу.

Розглянемо приклад визначення класу.

Опис класу:

```
class MyClass
{
int i; //розділ із типом доступу private
public: //розділ із типом доступу public
int get_i();
void set_i(int);
};
```

Опис методів класу:

```
int MyClass::get_i() { return i; }
void MyClass::set_i(int x) { i=x; return; }
```

Якщо методи класу прості і складаються всього лише з кількох операторів, то їх опис можна розмістити прямо у описі класу.

```
class MyClass
{
int i; //розділ із типом доступу private
public: //розділ із типом доступу public
int get_i(){ return i; };
void set_i(int) { i=x; return; };
};
```

Для збільшення швидкодії програми при описі методів класу може використовуватись модифікатор inline (вбудований). Використання цього модифікатора зменшує кількість операцій процесора при визові таких функцій. Однак при цьому збільшується об'єм програми, тому такий модифікатор бажано використовувати тільки з невеликими методи, що складаються лише з кількох операторів. Як у нижченаведеному прикладі.

```
class MyClass
{
int i; //розділ із типом доступу private
public: //розділ із типом доступу public
inline int get_i(){ return i; };
inline void set_i(int) { i=x; return; };
};
```

Як було сказано вище клас – це абстрактний тип даних. Для того щоб використати розроблений клас у програмі необхідно об'явити змінну даного типу. Така змінна буде називатись об'єктом даного класу. Таким чином ***об'єкт класу*** – це конкретна змінна (екземпляр) класу (даного типу інформаційної структури).

Об'явити об'єкт даного класу можна таким же чином, як і змінну будь-якого іншого стандартного або заданого користувачем типу. Наприклад об'явити об'єкт показаного вище класу MyClass при умові, що даний клас описаний у файлі MyClass.h, який знаходиться у одному каталозі з проектом, можна наступним чином:

```
#include “MyClass.h”
…
MyClass MyObj;
```

Більш детально дане питання буде розглянуто пізніше.

Контрольні запитання

1. Що таке клас?
2. Що таке поле класу?
3. Що таке метод класу?
4. Як визначити клас у програмі?
5. Як описати структуру класу?
6. З яких розділів складається структура класу?
7. Що означає заголовок ***public*** ?
8. Що означає заголовок ***private*** ?
9. Що означає заголовок ***protected*** ?
10. Як описати методи класу?
11. Що означає директива inline при визначенні методів класу?
12. Що таке об’єкт?
13. Як об’явити об’єкт у програмі?

ІНІЦІАЛІЗАЦІЯ І ЗНИЩЕННЯ ОБ'ЄКТІВ. КОНСТРУКТОРИ ТА ДЕСТРУКТОРИ

Створюючи і використовуючи такі складні інформаційні структури як класи, програміст потребує наявності ефективних засобів керування об'єктами. Для класа потрібен механізм, який визначає поведінку об'єкта при ініціалізації та знищенні, контролює розподіл пам'яті в процесі ініціалізації та знищенні об'єктів, щоб програміст міг використовувати об'єкти класів аналогічно змінним стандартних типів.

В C++ передбачена реалізація такого механізму шляхом використання конструкторів та деструкторів.

Конструктор – це функція метод класу з таким самим ім'ям, як у класу. Цей метод призначений для створення та ініціалізацію об'єктів класу, виділення пам'яті під об'єкти та присвоєння початкових значень полям об'єкту.

Деструктор – це функція метод класу, яка відповідає за коректне вивільнення пам'яті при знищенні об'єкту. Деструктор завжди має ім'я таке ж, що і у класу, перед яким ставиться символ „~".

Конструктори і деструктори не можуть повертати значень і тому при їх описанні відсутній тип результату.

Конструктор може мати вхідні параметри, а може не мати. Конструктор, який не має вхідних параметрів називається ***конструктором за замовчуванням***.

Клас може не містити цих функцій у явному вигляді. Тоді розподіл пам'яті та ініціалізація об'єкту виконуються системою автоматично стандартним шляхом. Але оскільки стандартна процедура не враховує особливостей класу, то така процедура може бути неефективною і часто навіть може призводити до серйозних помилок в роботі програми, тому бажано при розробці класів розробляти конструктори та деструктори.

Клас може мати кілька конструкторів і лише один деструктор. Для прикладу добавимо конструктор в розглянутий вище клас MyClass.

```
class MyClass
{
int i; //розділ із типом доступу private
public: //розділ із типом доступу public
MyClass(){i=0;}//Конструктор. У описі відсутній тип результату.;
//Вхідні параметри відсутні. Це конструктор за замовчуванням;
inline int get_i(){ return i; };
inline void set_i(int) { i=x; return; };
};
```

В даному випадку конструктор дуже простий і не потребує параметрів при визові. Це конструктор по замовчуванню. Розглянемо більш складний приклад.

```
class MyClass
{
int i; //розділ із типом доступу private
public: //розділ із типом доступу public
MyClass(){i=0;}//Конструктор. У описі відсутній тип результату.;
//Вхідні параметри відсутні. Це конструктор за замовчуванням;
MyClass(int a){i=a;}//Це також конструктор, але з параметром.
inline int get_i(){ return i; };
inline void set_i(int) { i=x; return; };
};
```

В даному прикладі два конструктори. Один по замовчуванню, який ініціалізує поле і класу MyClass нулем, та другий, який ініціалізує це поле заданим значенням. Тепер об’явити об’єкт даного класу можна у два способи за замовчуванням та із заданим значенням ініціалізації:

```
#include “MyClass.h”
…
MyClass MyObj1; //ініціалізація за замовчуванням
//поле і ініціалізується значенням 0.
MyClass MyObj1(1); //ініціалізація заданим значенням
//поле і ініціалізується значенням 1.
```

ДИНАМІЧНИЙ РОЗПОДІЛ ПАМ'ЯТІ

Для більш ефективного використання пам'яті часто доводиться розроблювати класи, в яких розмір об'єкту залежить від даних, що в ньому зберігаються. В таких структурах часто об'єм потрібної пам'яті стає відомим лише безпосередньо перед ініціалізацією конкретного об'єкта даного класу. Це призводить до необхідності динамічного керування пам'яттю. Для цих цілей в С++ передбачено два оператори new та delete. Оператор new дозволяє виділити пам'ять об'єкту, оператор delete дозволяє звільнити пам'ять

Динамічний розподіл пам'яті вимагає використання покажчиків. Розглянемо приклади.

Динамічне виділення пам'яті під одновимірний масив

```
float* y;
y=new float [5];
```

Очищення пам'яті, виділеної динамічно під одновимірний масив

```
delete y;
```

Динамічне виділення пам'яті під двовимірний масив

```
float** y;
y=new float* [5];
for(i=0;i<5;i++) {y[i]= new float[5] };
```

Очищення пам'яті, виділеної динамічно під двовимірний масив

```
for(i=0;i<5;i++)  {delete y[i] };
delete y;
```

Зрозуміло, що оператор new використовується в конструкторі, а оператор delete в деструкторі. Розглянемо приклад.

Нехай нам потрібен клас одновимірний вектор, розмір якого визначається у момент ініціалізації. Такий клас повинен мати конструктор з параметром, що вказує на розмір вектора та конструктор за замовчуванням, щоб не виникали аварійні ситуації при ініціалізації без параметрів. Будемо

вважати, що за замовчуванням наш вектор буде тривимірним. Щоб не ускладнювати приклад в описі класу покажемо лише конструктори та деструктори, а інші необхідні методи не показуватимемо.

```
class MyVector
{
private:
        float *v;
        int size;
public:
  MyVector(){size=3; v= new float[size];}; //конструктор за замовчуванням
  MyVector(int a) {size= a; v= new float[size];}; //конструктор з параметром
  ~MyVector(){delete[] v;};//деструктор
...
};
```

Контрольні запитання

1. Що таке конструктор?
2. Навіщо потрібен конструктор?
3. Як формується ім’я конструктора?
4. Що таке деструктор?
5. Як формується ім’я деструктора?
6. Навіщо потрібен деструктор?
7. Що таке конструктор за замовчуванням?
8. Скільки конструкторів може мати клас?
9. Скільки деструкторів може мати клас?
10. Для чого використовується оператор new?
11. Для чого використовується оператор delete?

ДОСТУП ДО ПОЛІВ ТА МЕТОДІВ КЛАСУ. СТАТИЧНІ ЧЛЕНИ КЛАСУ

Доступ до відкритих полів та методів класу здійснюється прямим зверненням, з використанням операторів прямого і непрямого вибору, як в структурах і об'єднаннях.

Оператор прямого вибору – це точка „.”. Оператор непрямого вибору – це спеціальний символ, що складається з мінуса та знаку більше „->”.

Оператор прямого вибору використовується тоді, коли доступ до об’єкту класу у програмі виконується безпосередньо. Оператор непрямого (опосередкованого) вибору використовується, коли доступ до об’єкту класу у програмі виконується опосередковано через покажчик на об’єкт.

Доступ до закритих та захищених методів та полів здійснюється лише за допомогою відкритих методів класу або за допомогою друзів класу. Доступ за допомогою друзів класу буде розглядатись пізніше, а зараз розглянемо приклад без використання друзів класу.

```
class MyClass
{
int i; //закрите поле класу
public:
//відкриті методи класу
MyClass(){i=0};//конструктор
inline int get_i(){ return i; };
inline void set_i(int) { i=x; return; };
};
        ...

void main()
{
MyClass obj;//об’явлення об’єкту
MyClass * obj_ptr=&obj;// об’явлення покажчика на об’єкт

obj. set_i(5) ; //встановлюємо нове значення закритого поля і
//шляхом безпосереднього звернення до відкритого методу

obj_ptr-> set_i(6) ; // встановлюємо нове значення закритого поля і
//шляхом опосередкованого (через покажчик об’єкт)  звернення
```

```
//до відкритого методу
}
```

Як зазначалось раніше клас – це тип даних, інформаційна структура, а не об'єкт даних. Кожен об'єкт класу містить свою власну копію полів цього класу. Проте деякі типи найелегантніше реалізуються, якщо всі об'єкти цього типу можуть спільно використовувати (розділяти) деякі дані. При цьому бажано, щоб такі дані , що розділяються, були описані як частина даного класу. Наприклад для управління завданнями в операційній системі використовують список всіх завдань, або при створенні об'єктів класу необхідно, щоб ці об'єкти могли контролювати кількість створених об'єктів даного класу і т.д.

Для реалізації цих властивостей в C++ передбачено використання статичних полів класу. Статичні поля описуються за допомогою ключового слова static. Оскільки статичні члени класу використовуються всіма об’єктами класу, то бажано перед використанням такі поля завжди ініціалізувати при описі класу .

Наведемо приклад.

```
class MyClassStatic
{
 int i;
 static int count; //статичне поле класу
public:
 MyClassStatic(int a=0){i=a;};//конструктор
 int inc_count(){++count; return 0;}; //метод доступу до статичного поля
 int get_i(){return i;};//метод доступу до звичайного закритого поля
 int get_count(){return count;}; //метод доступу до статичного поля
};

int MyClassStatic::count=0; //ініціалізація статичного поля
```

РЕАЛІЗАЦІЯ МЕХАНІЗМІВ УСПАДКУВАННЯ ТА ПОЛІМОРФІЗМУ В С++. ДОСТУП ДО ЧЛЕНІВ БАЗОВИХ КЛАСІВ

Як уже зазначалося, успадкування і поліморфізм – відносяться до найважливіших механізмів ООП. З їх допомогою можна розробляти дуже складні класи, просуваючись від загального до приватного, а також "нарощувати" вже створені, одержуючи з них нові класи з необхідними властивостями. Основною відмінністю реалізації успадкування і поліморфізму в С++ є можливість успадкування від кількох класів одночасно – ***множинне успадкування***.

Приступаючи до проектування складного класу необхідно з'ясувати, які найбільш загальні властивості повинні бути притаманними даному класу, і чи немає вже готового класу, який би можна було "доробити".

Знайшовши такий клас, потрібно успадкувати необхідні властивості і додати нові(або змінити деякі з тих, що вже існують). При цьому потрібно пам'ятати, що в С++ можливе множинне успадкування.

Для ілюстрації вищесказаного наведемо приклад створення нового об'єкту на основі двох базових об'єктів з використанням успадкування їх членів і перевизначенням двох функцій членів, а саме конструктора і функції для виведення вмісту об'єкту на екран.

Настійно рекомендую звернути увагу на визначення конструктора результуючого класу і на звернення до конструкторів базових класів, а так само на використання оператора дозволу області бачимості ”::”.

```
#include <iostream.h>
#include <string.h>
class human //перший базовий клас
{
char* name;
char* surname;
public:
human(char* h_name="anybody", char*  h_surname="anybody");//конструктор
~human(){delete name; delete surname;return;};//деструктор
```

```
void get_info()//метод get_info першого базового класу
{cout <<"\n name - " << name << "\n surname - "<< surname <<"\n"; return ;};
};
//конструктор першого базового класу
human::human(char* h_name, char* h_surname)
{
name = new char[strlen(h_name)+1];
strcpy(name,h_name);
surname = new char[strlen(h_surname)+1];
strcpy(surname,h_surname);
return;
};
//другий базовий клас
class qualification
{
char* trade;
int   time;
public:
qualification(char* q_trade="NO", int q_time=0);//конструктор
~qualification(){delete trade;return;}//деструктор
void get_info()//метод get_info другого базового класу
{cout<<"\nmy trade is - "<<trade<<"\ni work in my trade - "<<time<<"-year(s)\n";
return;}
};
//конструктор другого базового класу
qualification::qualification(char* q_trade, int q_time)
{
trade=new char[strlen(q_trade)+1];
strcpy(trade,q_trade);
time=q_time;
return;
}
//клас породжений від двох базових
//базові класи вказуються через кому після символа двокрапки
class employee: human, qualification
{
employee(char* h_name, char* h_surname, char* q_trade, int q_time);
//метод get_info об’єднаного класу виконує лише коректний визов
//методів get_info базових класів
void get_info(){human::get_info();qualification::get_info();return;}
};
//конструктор породженого класу
//у даному випадку виконує лише коректний визов
//конструкторів базових класів, передаючи їм відповідні аргументи
```

```
employee::employee(char* h_name, char* h_surname, char* q_trade, int q_time):
        human(h_name,h_surname), qualification(q_trade,q_time)
{
//тіло конструктора породженого класу
//тут при необхідності можна виконувати додаткові дії
return;
}
```

Контрольні запитання

1. Яким чином здійснюється доступ до відкритих полів та методів класу?
2. Яким чином здійснюється доступ до відкритих полів та методів класу?
3. Яким чином здійснюється доступ до закритих полів та методів класу?
4. Яким чином здійснюється доступ до захищених полів та методів класу?
5. Що представляє собою оператор прямого вибору?
6. Що представляє собою оператор непрямого вибору?
7. Що таке статичне поле класу, які його властивості?
8. Що таке множинне успадкування?

ДРУЗІ КЛАСУ

Іноді виникає необхідність, щоб якась функція, що не є членом класу мала доступ до закритих та захищених полів об'єктів даного класу. У С++ є така можливість. Для цього необхідно і достатньо оголосити дану функцію ***другом*** цього класу. Зробити це можна за допомогою ключового слова "friend". Розглянемо це на прикладі.

```
class friend_prob
{
private:
int status;
public:
int get_status(){return status;}
void set_status(int x){ status=x; return;}
//функція prob_friend оголошується другом класу
void friend prob_friend(friend_prob &obj );
};

void prob_friend( friend_prob &obj)
{
//функція друг здійснює доступ до закритого члена
//класу шляхом прямого звертання.
obj.status++;
return;
}
```

У даному прикладі функція prob_friend (friend_prob &obj) описана, як друг класу friend_prob. Це дозволяє їй дістати доступ до закритого поля даного класу status. Зверніть увагу на те, що, оскільки в програмі може існувати одночасно декілька об'єктів даного класу, такій функції необхідно як параметр передавати покажчик на об'єкт, з яким вона повинна працювати в даний момент.

Аналогічним чином можна описати метод іншого класу або навіть цілий клас, як друг даного класу. Наприклад, можна визначити клас a, як друг класу b. Тоді методи об'єктів класа a отримають доступ до закритих та захищених полів об'єктів класу b.

Розглянемо приклад.

```
class MyClass1;//випереджуючий неповний опис класу MyClass1

class MyClass2
{
private://розділ закритих полів класу
int status;
public://розділ відкритих методів класу
int get_status(){return status;}
void set_status(int x){status=x;return;}
friend MyClass1;//об’являємо клас MyClass1 другом класу MyClass2
};

class MyClass1
{
public:
//метод set_status класу MyClass1 модифікує закрите
//поле status класу MyClass2 шляхом прямого звернення
void set_status(MyClass2& obj,int x){obj.status=x;return;}
};
```

У даному прикладі клас MyClass1 оголошується як друг класу MyClass. Це дозволяє методу set_status класу MyClass1 звертатися до закритого поля status класу MyClass2.

Зверніть увагу на те, що, для забезпечення можливості зробити це оголошення, довелося застосувати випереджуючий неповний опис класу MyClass1. Без такого опису компілятор видав би помилку. Крім того методу set_status класу MyClass1 для реалізації доступу до конкретного об'єкту необхідно знати адресу цього об'єкту, тому, як один з аргументів, йому передається покажчик на об'єкт класу MyClass2.

Наявність механізму установлення дружніх відносин між класами дозволяє моделювати досить складні відносини між класами, що значно полегшує створення програм для вирішення складних практичних завдань.

ШАБЛОНИ КЛАСІВ

Досить часто при використанні ООП виникає необхідність введення великої кількості класів, які виконують однакові дії і відрізняються лише типами даних, по відношенню до яких ці дії застосовуються. Для спрощення виконання цієї задачі в С++ передбачені шаблони класів[5]. ***Шаблони класів*** це елементи мови програмування, які дозволяють визначити структуру сімейства класів, за якою компілятор самостійно створює потрібні класи, грунтуючись на параметрах настройки, що задаються. Цей механізм аналогічний механізму шаблонів функцій.

Найбільш типовий приклад використання шаблонів класів – це створення контейнерних класів, наприклад, векторів для розміщення об'єктів довільних типів.

Приклад:

```
template <class T>//шаблон класу вектор
class Vector
{
private:
T *elements;
int size;
public:
Vector(int razm=0); //конструктор, його реалізація має особливості
//деструктор, його реалізація також може мати особливості
~Vector(){delete elements;}
//перевантажений оператор для класу
T& operator[](int i){return elements[i];}//перевантажений оператор-метод
//метод, його реалізація має особливості
void print_contents();
};

//конструктор, його реалізація має особливості
template <class T>
Vector<T>::Vector(int razm)
{
elements=new T[razm];
for(int i=0; i< razm; elements[i]=(T) 0 i++);
size=razm;
};
```

```
//метод, реалізація якого має особливості
template <class T>
void Vector<T>::print_contents()
{
cout << "elements num-"<<size<<"\n";
for(int i=0; i<size; i++)
cout <<"el["<<i<<"]="<<elements[i] <<"\n";
}
//головна функція
//зверніть увагу на визначення типу для кожного об’єкту
int main()
{
int razmer=10;
Vector <int> i(razmer);
Vector <float> x(razmer);
Vector <char> з(razmer);
…
return 0;
}
```

Зверніть увагу на те, що заголовок шаблону класу починається з ключового слова template і містить вказівку на те, що тип наперед невідомий і повинен вказуватись при об’вленні об’єкту <class T>. Замість літери T може бути використана інша літера. Головне, щоб у всіх методах і полях класу, де буде оброблюватись інформація даного типу стояла та ж сама літера. Це дасть змогу компілятору правильно сформувати об’єкт класу для заданого типу. При розробці шаблонів класів часто виникає проблема перевантаження операторів. Це пов’язано з тим, що з одного боку для розроблених програмістом класів, як правило, немає стандартних операторів, а з іншого боку дуже зручно, коли аналогічні операції для різних типів позначаються у програмі однаковими операторами.

ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ ДЛЯ КЛАСІВ

При розробці шаблонів класів часто виникає проблема перевантаження операторів. Це пов’язано з тим, що з одного боку для розроблених програмістом класів, як правило, немає стандартних операторів, а з іншого боку дуже зручно, коли аналогічні операції для різних типів позначаються у програмі однаковими операторами.

Навіть у попередньому прикладі нам довелося перевантажувати оператор індексу, щоб можна було звертатись до елемента вектора таким же чином, як до елемента масиву.

Перевантажувати оператори для класів можна або описуючи оператори як методи класу (дивись попередній приклад), або використовуючи дружню функцію. Розглянемо приклад використання дружньої функцію для перевантаження оператора складання. Перевантажимо цей оператор для складання двох об'єктів класу complex, що моделює комплексне число.

```
class complex
{
private:
float re,im;
public:
void set_value(float x,float y){re=x;im=y;return ;}
float get_re(){return re;}
float get_im(){return im;}
friend complex operator+(complex&,complex&);//перевантажений оператор „+”
};
complex operator+(complex& a complex& b)//реалізація алгоритму оператора
{
complex c;
c.re=a.re+b.re;
c.im=a.im+b.im;
return c;
}
```

Контрольні запитання

1. Для чого використовуються друзі класів?
2. Що може бути другом класу?
3. Як оголосити функцію чи клас другом класу?
4. Чому при оголошенні класу другом іншого класу використовують випереджуючий неповний опис?
5. Що таке шаблони класів?
6. У яких випадках використовуються шаблони класів?
7. Навіщо використовують перевантаження операторів для класів?
8. У які способи можна реалізовувати перевантаження операторів для класів?

СТАТИЧНІ ТА ДИНАМІЧНІ МАСИВИ

Масив – це набір однотипних елементів. Кожен елемент має свій номер (індекс). Всі елементи масиву упорядковані за своїм індексом. Масив може бути одновимірним чи багатовимірним. Елементи масиву можуть мати просту чи складну структуру в залежності від функціонального призначення даного масиву. Основною особливістю масивів є те, що на етапі виділення пам'яті для збереження елементів масиву відома точна кількість елементів масиву. Основними операціями при роботі з масивами є:

- виділення пам'яті для збереження елементів масиву;
- вивільнення пам'яті;
- сортування елементів масиву;
- пошук елемента масиву, що відповідає заданому критерію пошуку.

Виділення пам'яті під масив може бути ***статичним***, коли кількість елементів масиву відома на етапі розробки програми, або ***динамічним***, коли кількість елементів масиву стає відомою під час виконання програми. При статичному виділенні вивільнення пам'яті відбувається автоматично при закінченні роботи програми і програміст не має змоги виконати вивільнення пам'яті під час роботи програми. При динамічному виділенні вивільнення пам'яті також відбувається автоматично при закінченні роботи програми, але програміст може при необхідності вивільнити пам'ять в процесі виконання програми. Це робить програми більш гнучкими, тому в сучасному програмуванні все частіше використовують динамічне виділення пам'яті при роботі з масивами.

Методи сортування масивів.

При роботі з масивами найчастіше доводиться виконувати операції сортування та пошуку даних. Від ефективності реалізації цих операцій часто залежить ефективність всієї програми, тому розглянемо ці операції докладніше.

Сортування – це процес перегрупування даних у деякому заданому порядку. Основна мета сортування – полегшити пошук потрібної інформації у заданій послідовності даних[2,5].

Методи сортування поділяються на ***внутрішні*** (дані розміщуються в оперативній пам'яті) та ***зовнішні*** (дані розміщуються в файлах). Для внутрішнього сортування використовуються методи сортування за допомогою включення, за допомогою вибору, за допомогою обміну і т.д.

Розглянемо вказані внутрішні методи. Для визначеності будемо вважати, що необхідно виконати сортування у порядку зростання елементів масиву, і всі приклади, що будуть розглянуті відповідатимуть саме такому порядку сортування. Читачеві ж рекомендуємо для тренування модифікувати наведені програми для сортування у зворотному напрямку.

Сортування за допомогою включення

При сортуванні за допомогою ***включення*** елементи умовно ділять на вже відсортовану послідовність $a_1 \ldots a_{i-1}$ і вихідну послідовність $a_i \ldots a_n$. На кожному кроці починаючи з $i=2$ та збільшуючи кожен раз i на одиницю, із вихідної (початкової) послідовності видобувається i-тий елемент і перекладається в уже готову послідовність. При цьому він вставляється в потрібне місце готової послідовності. В процесі пошуку потрібного місця чергуються операція порівняння даного елемента з елементами послідовності та операція переміщення по послідовності, тобто вибраний елемент порівнюється з черговим елементом a_j , а тоді a_i або вставляється на місце елемента a_j (якщо виконано критерій сортування), тоді відповідно a_j зміщується на одну позицію вправо або a_i порівнюється з наступним елементом a_{j-1} (якщо критерій сортування не виконано), при цьому a_j знову ж таки зміщується на одну позицію вправо. Нижче наведено приклад сортування методом включення масиву цілих чисел . Для демонстрації на екран виводяться вихідний масив та результуючий масив.

Приклад:

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>

//Шаблон функції сортування елементів масиву методом включення
template <class T> void vstavka(T* massive,int size);

void main()
{
//Оголошення змінних, необхідних для реалізації алгоритму
int* mas;
int n,i;
//Очищення екрану
clrscr();
//Запит на введення розміру масиву
cout << "Input n" << endl;
cout << "n=";
cin >> n;
cout << "creating massive of random numbers:"<<endl;
//Динамічне виділення пам’яті під елементи масиву
mas=new int[n];
//Підключення генератору випадкових чисел
randomize();
//Формування масиву випадкових чисел
for (i=0;i<n;i++)
{
 //Формування елементу масиву випадковим чином
mas[i]=random(100)-50;
//Відображення на екрані сформованого елементу
 cout << mas[i]<<"   ";
}
//Переведення курсору на наступну строку екрану
cout << endl;
//Сортування елементів масиву методом включення
vstavka(mas, n);//Відображення на екрані відсортованого масиву
cout << "massiv after sorting:" << endl;
for(i=0;i<n;i++)
{
 cout << mas[i] << "   ";
}
//Очищення пам’яті, виділеної під масив
delete mas;
return;
```

```
}

//Реалізація шаблону функції сортування елементів масиву методом
//включення опис алгоритму див вище
template <class T> void vstavka(T* massive,int size)
{
 T tmp;
 int i,j;
for(i=1;i<size;i++)
{
 tmp=massive[i];
 j=i;
 while(tmp<massive[j-1])
 {
  massive[j]=massive[j-1];
  j--;
  if(j==0) break;
 }
 massive[j]=tmp;
}
return;
}
```

Сортування за допомогою прямого выбору

Сортування за допомогою ***прямого вибору*** базується на нижчеперелічених операціях:

1). Обирається елемент з найменшим значенням.

2). Цей елемент обмінюється місцями з першим елементом.

3). Потім п.1-2 повторюються з елементами від 2-го до n-го, потім від 3-го до n-го і т. д.

Наведемо приклад сортування методом прямого вибору масиву цілих чисел . Як і в попередньому прикладі для демонстрації на екран виводяться вихідний масив та результуючий масив.

Приклад:

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>

//Шаблон функції сортування масиву методом прямого вибору
template <class T> void vibir(T* massive,int size);

void main()
{
//Оголошення змінних, необхідних для реалізації алгоритму
int* mas;
int n,i;
//Очищення екрану
clrscr();
//Запит на введення розміру масиву
cout << "Input n" << endl;
cout << "n=";
cin >> n;
cout << "creating massive of random numbers:"<<endl;
//Динамічне виділення пам’яті під елементи масиву
mas=new int[n];
//Підключення генератору випадкових чисел
randomize();
//Формування масиву випадкових чисел
for (i=0;i<n;i++)
{
//Формування елементу масиву випадковим чином
 mas[i]=random(100)-50;
//Відображення на екрані сформованого елементу
 cout << mas[i]<<"   ";
}
//Переведення курсору на наступну строку екрану
cout << endl;
//Сортування елементів масиву методом прямого вибору
vibir(mas,n);
//Відображення на екрані відсортованого масиву
cout << "massiv after sorting:" << endl;
for(i=0;i<n;i++)
{
 cout << mas[i] << "   ";
}
//Очищення пам’яті, виділеної під масив
delete mas;
```

```
return;
}

//Реалізація шаблону функції сортування масиву методом прямого вибору
//опис алгоритму дивись вище
template <class T> void vibir(T* massive,int size)
{
T tmp;
int i,j,e;
tmp=massive[0];
for(i=0;i<size;i++)
{
 e=i;
 tmp=massive[i];
 for(j=i;j<size;j++)
 {
  if(massive[j]<tmp)
  {
   tmp=massive[j];
   e=j;
  }
 }
 if(e!=i)
 {
  massive[e]=massive[i];
  massive[i]=tmp;
 }
}
return;
}
```

Сортування за допомогою обміну

Сортування за допомогою обміну базується на процесі порівняння і при необхідності обміну місцями двох сусідніх елементів масиву. Ці операції повторюються доти, доки не буде упорядковано весь масив. Треба зазначити, що після першого проходу по всьому масиву максимальний елемент переміщається в крайнє праве положення, і на наступному етапі немає сенсу перевіряти весь масив. Тому на практиці при першому проході перевіряють елементи з номерами від 1 до n (останнього), на другому від 1 до (n-1) і т.д.

Наведемо приклад сортування методом обміну масиву цілих чисел . Як і в попередньому прикладі для демонстрації на екран виводяться вихідний масив, та результуючий масив.

Приклад:

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>

//Шаблон функції сортування елементів масиву методом обміну
template <class T> void obm_sort(T* massive,int size);

void main()
{
//Оголошення змінних необхідних для реалізації алгоритму
int* mas;
int n,i;
//Очищення екрану
clrscr();
//Запит на введення розміру масиву
cout << "Input n" << endl;
cout << "n=";
cin >> n;
cout << "creating massive of random numbers:"<<endl;
//Динамічне виділення пам’яті під елементи масиву
mas=new int[n];
//Підключення генератору випадкових чисел
randomize();
//Формування масиву випадкових чисел
for (i=0;i<n;i++)
{
//Формування елементу масиву випадковим чином
mas[i]=random(100)-50;
//Відображення на екрані сформованого елементу
 cout << mas[i]<<"   ";
}
//Переведення курсору на наступну строку екрану
cout << endl;
//Сортування елементів масиву методом обміну
obm_sort(mas,n);
//Відображення на екрані відсортованого масиву
cout << "massiv after sorting:" << endl;
for(i=0;i<n;i++)
{
```

```
 cout << mas[i] << "   ";
}
//Очищення пам’яті, виділеної під масив
delete mas;
return;
}
//Реалізація шаблону функції сортування елементів масиву методом обміну
template <class T> void obm_sort(T* massive,int size)
{//опис алгоритму дивись вище
int i,j,d;
T tmp;
d=size;
for(i=0;i<size-1;i++)
{
 for(j=0;j<d-1;j++)
 {
  if(massive[j]>massive[j+1])
  {
   tmp=massive[j];
   massive[j]=massive[j+1];
   massive[j+1]=tmp;
  }
 }
 d--;
}
return;
}
```

Продемонстровані методи можуть бути модифіковані для збільшення їх ефективності, але викладення цього матеріалу виходить за рамки даного курсу. Студентам же пропонується самостійно вивчити метод Шелла (модифікований метод сортування за допомогою включення) і метод швидкого сортування[5] та написати параметризовані функції сортування для цих методів, а також для всіх методів сортування розглянутих вище.

Методи пошуку у масивах

Пошук – це процес знаходження серед елементів даного типу елемента з заданими властивостями. Задане значення критерію пошуку називається ***ключем пошуку***. Це може бути умова рівності елемента заданій величині або

інша умова. При подальшому розгляді методів пошуку будемо вважати, що кількість елементів даного типу, в якій провадиться пошук – відома.

Прямий лінійний пошук

Найпростішим, але не самим оптимальним методом пошуку є ***прямий лінійний пошук***. Цей метод використовується тоді, коли немає ніякої додаткової інформації про групу елементів серед якої провадиться пошук.

Метод полягає в послідовному перегляді всіх елементів і перевірці їх на відповідність ключу пошуку. Умовою закінчення пошуку може бути або факт знаходження даного елемента, або той факт, що дану сукупність елементів перевірено повністю і не знайдено елементів, що відповідають критерію пошуку. Розглянемо приклад:

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
//Опис параметризованої функції
template <class T> int search(T* mas,T search_key,int size);

void main ()
{
//Опис змінних
int n,i,key;
int* massive;
float* massive1;
float key1;
//Очищення екрану
clrscr();
//Запит на введення розміру масивів
cout << "Input n=" ;
cin  >> n;
//Динамічне виділення пам’яті під масив цілих чисел
massive= new int[n];
// Динамічне виділення пам’яті під масив дійсних чисел
massive1= new float[n];
//Активація генератора випадкових чисел
randomize();
//Заповнення масивів випадковими числами
for(i=0;i<n;i++)
```

```
{
 massive[i]=random(50)-25;
 massive1[i]=massive[i]/2.0;
}
//Відображення на екрані масиву цілих чисел
cout << "Massive of integer numbers:"<<endl;
for(i=0;i<n;i++)
{
 cout.width(7);
 cout << massive[i];
}
cout <<endl;
//Запит на введення ключа для пошуку у масиві цілих чисел
cout << "Input integer number key=";
cin >> key;
//Пошук у масиві цілих чисел та виведення результату на екран
search(massive, key, n);
//Відображення на екрані масиву дійсних чисел
cout << "Massive of float numbers:"<<endl;
for(i=0;i<n;i++)
{
 cout << massive1[i] << "   ";
}
cout <<endl;
//Запит на введення ключа для пошуку у масиві дійсних чисел
cout << "Input float number key1=";
cin >> key1;
//Пошук у масиві дійсних чисел та виведення результату на екран
search(massive1, key1, n);
//Очищення пам’яті, виділеної під масиви
delete massive;
delete massive1;
return;
}

//Параметризована функція прямого пошуку
template <class T> int search(T* mas, T search_key, int size)
{
 int index=0;

 for (index=0;index<size;index++)
 {
  if(search_key==mas[index])
  {
   cout << "Found element with number n= " << index+1 << endl;
```

```
  return index;
 }

}

 cout << "Element not found"<<endl;
return -1;
}
```

Бінарний пошук

Прямий пошук вимагає великих затрат машинного часу. А чи можна якось прискорити пошук потрібного елемента? Очевидно, що без додаткової інформації про задану сукупність елементів, це неможливо. Проте пошук можна зробити значно ефективнішим, якщо відомо, що задана послідовність елементів є впорядкованою за критерієм пошуку. Прикладом такої впорядкованої послідовності може бути телефонний довідник, всі записи якого впорядковано відповідно до абетки.

Основною ідеєю пошуку у такій послідовності є вибір деякого випадкового елемента і порівняння його з критерієм пошуку. При цьому може виникнути три випадки:

- елемент відповідає критерію пошуку. Тоді шуканий елемент знайдено і пошук можна завершити;
- елемент має значення більше за величину ключа пошуку. Тоді треба продовжити пошук у тій частині сукупності де значення менші за значення обраного елемента;
- елемент має значення менше за величину ключа пошуку. Тоді треба продовжити пошук у тій частині сукупності де значення більші за значення обраного елемента.

При такій організації пошуку критерієм зупинки може бути або факт знаходження даного елемента, або той факт, що дану сукупність елементів перевірено повністю і не знайдено елементів, що відповідають критерію пошуку. Найпростішим способом реалізації такого алгоритму є ***метод***

бінарного пошуку (***метод поділу навпіл***). При такому методі розглядувану послідовність ділять пополам і порівнюють критерій пошуку з центральним елементом послідовності. Якщо критерій співпадає, то елемент знайдено, якщо значення елементу менше за заданий критерій то ділять пополам ту частину послідовності де значення елементів більше за значення обраного елемента, якщо ж воно більше, то ділять ту половину де значення елементів менше за значення обраного елемента. Ці дії виконують доти, доки не буде знайдено потрібний елемент, або поки у досліджуваній частині сукупності не залишиться лише один елемен.

Оскільки при цьому кожен раз кількість досліджуваних елементів зменшується вдвічі, то швидкість пошуку значно зростає порівняно з лінійним пошуком.

Розглянемо приклад:

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>

template <class T> int b_search(T* mas,T search_key,int size);
template <class T> void sort(T* mas, int size);

void main ()
{
//Опис змінних
int n,i,key;
int* massive;
float* massive1;
float key1;
//Очищення екрану
clrscr();
//Запит на введення розміру масивів
cout << "Input n=" ;
cin  >> n;
//Виділення пам’яті під масиви чисел
massive= new int[n];
massive1= new float[n];
//Активація генератора випадкових чисел
randomize();
```

```
//Формування масивів випадкових чисел
for(i=0;i<n;i++)
{
 massive[i]=random(50)-25;
 massive1[i]=massive[i]/2.0;
}
//сортування масиву цілих чисел
sort(massive,n);
//Відображення відсортованого масиву на екрані
cout << "Massive of integer numbers:"<<endl;
for(i=0;i<n;i++)
{
 cout.width(7);
 cout << massive[i];
}
//Запит на введення ключа для пошуку у масиві цілих чисел
cout <<endl;
cout << "Input integer number key=";
cin >> key;
//Пошук у масиві цілих чисел та відображення на екрані результату пошуку
b_search(massive, key, n);
//Сортування масиву дійсних чисел
sort(massive1,n);
//Відображення відсортованого масиву на екрані
cout << "Massive of float numbers:"<<endl;
for(i=0;i<n;i++)
{
 cout << massive1[i] << "   ";
}
cout <<endl;
//Запит на введення ключа для пошуку у масиві дійсних чисел
cout << "Input float number key1=";
cin >> key1;
//Бінарний пошук та відображення на екрані результату пошуку
b_search(massive1, key1, n);
//Вивільнення пам’яті виділеної під масиви
delete massive;
delete massive1;
return;
}

//Параметризована функція бінарного пошуку
template <class T> int b_search(T* mas, T search_key, int size)
{
 int low, high, mid;
```

```
 low=0;
 high=size-1;

 while(low<=high)
 {
  mid=(low+high)/2;

  if(search_key<mas[mid])
  {
   high=mid-1;
  }
  else
  {
   if(search_key>mas[mid])
   {
    low=mid+1;
   }
   else
   {
    cout << "Found element with number n= " << mid+1 << endl;
    return mid;
   }
  }
 }

  cout << "Element not found" << endl;
 return -1;
}
//Параметризована функція сортування методом перестановок
template <class T> void sort(T* mas, int size)
{

 int i,j,d,tmp;

 d=size;
for(i=0;i<size-1;i++)
{
 for(j=0;j<d-1;j++)
 {
  if(mas[j]>mas[j+1])
  {
   tmp=mas[j];
   mas[j]=mas[j+1];
   mas[j+1]=tmp;
  }
```

```
 }
 d--;
}
return;
}
```

Контрольні запитання

1. Що таке масив?
2. Які бувають масиви?
3. Чим статичний масив відрізняється від динамічного?
4. Які основні операції виконуються над масивами?
5. Що таке сортування?
6. Як класифікуються методи сортування?
7. Які методи називають внутрішніми?
8. Які методи називають зовнішніми?
9. Який алгоритм сортування методом обміну?
10. Який алгоритм сортування методом вибору?
11. Який алгоритм сортування методом включення?
12. Який алгоритм методу прямого пошуку?
13. Який алгоритм методу бінарного пошуку?

СПИСКИ. СОРТУВАННЯ СПИСКІВ, ПОШУК У СПИСКАХ

Список – це сукупність елементів, кількість яких може змінюватись в ході виконання програми. При цьому пам'ять для розміщення нового елементу виділяється і вивільняється динамічно по мірі необхідності.

Зрозуміло, що для реалізації такого типу даних необхідно використовувати покажчики.

Списки можуть реалізовуватись за допомогою одинарних або подвійних зв'язків[2,5]. В ***списку з одинарними*** зв'язками кожен елемент містить покажчик на наступний елемент списку. В списку з ***подвійними зв'язками*** кожен елемент містить покажчики на попередній і наступний елемент списку.

На сьогодняшній день найбільш часто використовуються списки з подвійними зв'язками. Це визвано тим, що список з подвійними зв'язками можна читати в обох напрямках, такий список легше відновлювати при пошкодженні і при використанні таких списків операції обробки інформації реалізуються простіше. Виходячи із сказаного ми будемо розглядати лише списки з подвійними зв'язками.

Основними операціями при роботі з списками є:

- додавання елементу до списку;
- вилучення елементу із списку;
- сортування елементів списку;
- пошук елемента списку, що відповідає заданому критерію пошуку.

Розглянемо ці операції на прикладі.

```
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
//Шаблон класу список
template <class T> class my_list
{
 private:
//Покажчик на наступний елемент списку
  my_list      *next;
```

```
//Покажчик на попередній елемент списку
  my_list      *prev;
//Поле даних елементу списку
  T            info;
//Поточна кількість елементів, занесених до списку
  int          list_size;
 public:
//Функція визначення адреси наступного елементу списку
  my_list<T> * get_next() {return next;}
//Функція визначення адреси попереднього елементу списку
  my_list<T> * get_prev() {return prev;}
//Функція запису адреси наступного елементу списку
  my_list<T> * set_next(my_list<T> *a) {return next=a;}
//Функція запису адреси попереднього елементу списку
  my_list<T> * set_prev(my_list<T> *a) {return prev=a;}
//Функція додавання елементу до списку
  void       add_elem(my_list<T> *first,T inf);
//Функція вилучення елементу із списку
  void       del_elem(my_list<T> *first);
//Функція сортування елементів списку за зростанням
  void       sort_elements(my_list<T> *first);
//Функція зчитування даних, збережених в елементі списку
  T          get_info() {return info;}
//Функція запису даних до елементу списку
  void       set_info(T c) {info=c;return;}
//Функція запису розміру списку
  void       set_size(int c) {list_size=c;return;}
//Функція зчитування розміру списку
  int        get_size() {return list_size;}
//Функція відображення інформаційних полів списку на екрані
  void       print_list(my_list<T> *first);
//Функція пошуку заданого значення у списку
  int        b_search(my_list<T> *first,T key);
}
//Реалізація функції видалення елементу із списку
template<class T>  void  my_list<T>::del_elem(my_list<T> *first)
                  {
                   my_list<T> *tmp;
                   tmp=first->next;
                   first->list_size--;
                   first->next=first->next->next;
                   delete tmp;
                   return;
                  }
```

```
//Реалізація функції відображення списку на екрані
template<class T>  void       my_list<T>::print_list(my_list<T> *first)
                {
                 int i;
                 my_list<T> *tmp;
                 tmp=first;
                 for(i=0;i< first->get_size();i++)
                  {
                   cout << tmp->get_info() << endl;
                   tmp=tmp->get_next();
                  }
                }
//Реалізація функції сортування елементів списку
template<class T>  void   my_list<T>::sort_elements(my_list<T> *first)
                {
                my_list<T> *tmp_ptr;
                T  tmp_elem_value;
                int i,j,d,size;
                tmp_ptr=first;
                size=first->list_size;
                d=size;
                for(i=0;i<size-1;i++)
                 {
                  for(j=0;j<d-1;j++)
                  {
                   tmp_elem_value=tmp_ptr->info;
                   if(tmp_ptr->info>tmp_ptr->next->info)
                    {
                     tmp_elem_value=tmp_ptr->info;
                     tmp_ptr->info=tmp_ptr->next->info;
                     tmp_ptr->next->info=tmp_elem_value;
                    }
                   tmp_ptr=tmp_ptr->next;
                  }
                  d--;
                  tmp_ptr=first;
                 }
                return;
                }
//Реалізація функції додавання елементу до списку
template <class T>  void my_list<T>::add_elem(my_list<T> *first,T inf)
                {
                my_list<T> *tmp;// тимчасовий покажчик для елемента
                tmp=new my_list<T>;//виділення пам'яті під елемент
                tmp->set_info(inf);//запис інформації до елементу
```

```
                tmp->next=first->next;//додавання до початку списку
                tmp->prev=first;
                first->prev=NULL;
                first->next=tmp;
                first->list_size++;
                return;
               }
//Реалізація функції пошуку значення у списку (бінарний пошук)
 template<class T> int my_list<T>::b_search(my_list<T> *first,T key)
               {
                my_list<T> *tmp,*tmp_beg;
                int beg,mid,end,i,num;
                if(first->info==key) return 1;//перевірка першого елементу
                tmp_beg=tmp=first;
                num=1;//налаштування початкових
                beg=1;//параметрів списку
                end=first->list_size;
                while(beg<end)//цикл пошуку
                {
                 mid=(end+beg)/2;//знаходження середини списку
                 for(i=beg;i<mid;i++)//
                 {
                  tmp=tmp->next;//перехід до центрального елементу
                 }
                 if(tmp->info==key)//закінчити пошук, якщо знайдено
                 {
                  num=mid;
                  return num;
                 }
                 if(tmp->info>key) //вибір потрібної для пошуку частини
                 {
                  end=mid;
                  tmp=tmp_beg;
                  num=beg;
                 }
                 else
                 {
                  tmp_beg=tmp;
                  num=mid;
                  beg=mid;
                 }
                 if (mid == (beg+end)/2) break;
                }
                tmp=first->next;
```

```
        for(i=2;i<first->list_size;i++)
                {
                 tmp=tmp->next;
                }
                if(tmp->info==key) return first->list_size;//повернути значення
                return -1;//якщо не знайдено повернути код помилки
               }

//Головна функція
void main(void)
{
 int i;
 my_list <int> *first,*last,*tmp;
//Очистка екрану
 clrscr();
//Створення списку
 first=new my_list<int>;
 last= new my_list<int>;
 tmp=  new my_list<int>;
 first->set_next(NULL);
 first->set_prev(NULL);
 first->set_info(0);
 first->set_size(1);
 last->set_next(NULL);
 last->set_prev(NULL);
 tmp=first;
//Додавання чотирьох елементів до списку
 for(i=1;i<5;i++)
 {
  tmp->add_elem(first,i);
 }
//Відображення списку
 first->print_list(first);
//Сортування списку
 first->sort_elements(first);
//Відображення списку
 first->print_list(first);
//Пошук елементу із значенням 3
 cout << endl << first->b_search(first,3)<<endl<<endl;
//Видалення всіх елементів списку, що знаходяться між першим і останнім
 for(i=0;i<3;i++)
 {
  first->del_elem(first);
 }
```

```
//Відображення списку на екрані
 first->print_list(first);
return;
}
```

З прикладу видно, що навіть елементарні операції із списками потребують досить високої кваліфікації, але вони дають змогу оптимально використовувати пам’ять.

В даному прикладі продемонстровано сортування списку методом обміну. Для простоти в прикладі було опущено ряд необхідних дій, таких як контроль виходу за межі списку, обробка помилок і т.і. Приклад добре прокоментовано, що спрощує розбір прикладу.

Для кращого засвоєння матеріалу рекомендується самостійно реалізувати сортування списку методом включення та методом прямого вибору.

Для пошуку елементів списку, що відповідають заданому критерію пошуку використовують ті ж методи, що і для масивів. В наведеному прикладі розглянуто бінарний пошук.

Контрольні запитання

1. Що таке список?
2. Які бувають списки?
3. Які основні операції, що виконуються над списками?
4. Які методи сортування використовуються для списків?
5. Які методи пошуку використовуються для списків?

СТЕКИ

Стек – одна з найбільш корисних та найчастіше використовуваних стандартних структур даних [2,5]. Він використовується для збереження змінних при визові процедур, для збереження змінних при виконанні переривань, для збереження даних про стан екрану при розробці інтерфейсу з користувачем на основі багаторівневого меню тощо. Доступ до елементів стеку здійснюється за принципом LIFO (last in, first out –„останнім зайшов, першим вийшов").

Концептуально цей тип даних дозволяє виконувати вставку та зчитування даних лише в одному елементі – вершині стеку. Елементи зчитуються у зворотному напрямі – першим зчитується елемент, що був записаний до стеку останнім. Довільний доступ до елементів стеку не допускається.

Стек може бути реалізований на основі масиву або на основі списку. Реалізація на основі масиву більш проста, але має ряд недоліків: обмежену кількість елементів, неефективне використання пам'яті та інші. Реалізація на основі списку більш складна, але позбавлена вказаних недоліків, тому в практичному програмуванні частіше реалізують стеки на основі списків. Саме тому ми будемо розглядати лише стеки на основі списків.

Стек повинен реалізовувати наступні операції:

- додавання елементу даних до вершини стеку (метод push);
- зчитування та вилучення елементу даних із вершини стеку (метод pop);
- визначення чи є в стеку дані.

Розглянемо ці операції на прикладі простого стеку на основі списку з подвійними зв'язками.

Приклад:

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <conio.h>
```

```
//Шаблон класу стек
template <class T> class user_stack
{
 private:
//Розмір стеку
        int stack_size;
//Елемент даних стеку
        T data;
//Покажчик на попередній елемент стеку
        user_stack *prev;
//Покажчик на наступний елемент стеку
        user_stack *next;
 public:
//Конструктор стеку
 user_stack();
//Функція, що перевіряє чи є в стеку елементи даних
 int  empty(){if (stack_size>0) return 0; else return -1;};
//Функція, що дозволяє прочитати елемент даних стеку
//Дана функція потрібна лише для відладки програми
 T    get_data(){return data;};
//Функція, що дозволяє отримати адресу попереднього елементу стеку
 user_stack<T> * get_prev(){return prev;};
//Функція, що дозволяє додати елемент до стеку (метод push)
 user_stack<T> * push(T var);
//Функція, що дозволяє видалати елемент стеку(метод pop)
 T   pop();
}
//Реалізація конструктора
template<class T> user_stack<T>::user_stack()
 {
  stack_size=0;
  data=NULL;
  prev=NULL;
  next=NULL;
 }

//Реалізація методу push
template <class T> user_stack<T> * user_stack<T>::push(T var)
 {
  this->next=new user_stack<T>;
  this->next->prev=this;
  this->next->data=var;
  this->next->stack_size=this->stack_size+1;
  return this->next;
 }
```

```
//Реалізація методу pop
template <class T>  user_stack<T>::pop()
 {
  T dat;
  dat=this->next->data;
  delete this->next;
  this->next=NULL;
  return dat;
 }
void main(void)
{
    user_stack<int> *my_stack;
    int my_data;
//Створення стеку
    my_stack=new user_stack<int>;
//Очистка екрану
    clrscr();
//Активація генератора випадкових чисел
    randomize();
//Присвоення змінній випадкового значення
    my_data=random(10)+5;
//Відображення значення змінної на екрані
    cout << "data="<<my_data << endl;
//Запис значення змінної в стек
    my_stack=my_stack->push(my_data);
//Відображення на екрані даних записаних у стек
    cout << "In stack stored - " << my_stack->get_data() << endl;
//Зміна значення змінної
    my_data+=20;
//Відображення нового значення змінної
    cout << "Changing data. Now  data ="<<my_data << endl;
//Відновлення старого значення змінної із стеку
    my_stack=my_stack->get_prev();
    my_data=my_stack->pop();
//Відображення відновленого значення змінної на екрані
    cout << "After popping data="<<my_data << endl;
return;
}
```

Контрольні запитання

1. Що таке стек?
2. Для чого використовуються стеки?
3. Які можливі варіанти програмної реалізації стеку?

ЧЕРГИ ПРОСТІ ТА ЦИКЛІЧНІ

Черга являє собою лінійний список, доступ до елементів якої здійснюється за принципом FIFO (first in, first out – „першим зайшов, першим вийшов"). Таким чином, першим із черги видаляється елемент, який був записаний до черги першим, потім – елемент, що був записаний до черги другим, і т. д. Для черги такий метод доступу і збереження даних являється єдиним. Довільний доступ до вказаного елементу не допускається. Черги можуть бути простими та циклічними.

Прості черги

Черги мають досить широке використання на практиці. Наприклад при моделюванні процесів реального часу, для диспетчеризації завдань операційної системи або для буферизації операцій вводу-виводу.

Черга повинна реалізовувати наступні операції:

- додавання елементу даних у кінець черги;
- зчитування та вилучення елементу даних з початку черги.

Розглянемо приклад, що демонструє методи роботи з чергами.

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <conio.h>

//Шаблон класу черга
template <class T> class queue
{
 private:
//Покажчик на динамічний масив, в якому розміщаються дані
        T *q;
//Номер позиції, з якої будуть зчитуватись дані
        int read_loc;
//Номер позиції, у яку будуть записуватись дані
        int write_loc;
//Розмір черги – максимальна кількість елементів,
//що може бути розміщено у черзі
```

```
        int length;
 public:
//Функція – конструктор, що створює чергу заданого розміру
        queue(int size);
//Функція – деструктор, що знищує чергу
        ~queue() {delete [] q;};
//Функція, що записує елемент даних до черги
        void write_data(T data);
//Функція, що зчитує елемент даних з черги
        T read_data();
}
//Реалізація функції – конструктора
template <class T> queue<T>::queue (int size)
{
//Виділення пам’яті під чергу
 q=new T[size];
//Якщо вільної пам’яті нема, то вивести на екран
//повідомлення про помилку
 if(!q)
 {
  cout << "Error! Can't create turn." << endl;
  exit(1);
 }
//Встановлення розміру черги
 length=size;
//Переведення індексів зчитування та запису в початок черги
 read_loc=write_loc=0;
}

//Реалізація функції, що записує елемент даних до черги
template <class T> void queue<T>::write_data(T data)
{
//Перевірка наявності вільного місця у черзі і при його відсутності
//виведення повідомлення про помилку
 if (write_loc==length)
 {
  cout << "Error! Turn is full." << endl;
  return;
 }
//Запис елементу даних у чергу при наявності вільного місця
 q[write_loc++]=data;
}
//Реалізація функції, що зчитує елемент даних до черги
template <class T> T queue<T>::read_data()
{
```

```
//Перевірка наявності незчитаних елементів у черзі та
//відображення повідомлення про помилку у разі їх відсутності
 if (write_loc==read_loc)
 {
  if(read_loc==length){ read_loc=write_loc=0;}
  cout << "Error! Turn is empty." << endl;
  return NULL;
 }
//Повернення зчитаних даних як результату функції
 return q[read_loc++];
}
//Головна функція
void main()
{
//Створюємо чергу цілих чисел, величиною 5 елементів
 queue<int> a(5);
 int turn_data;
//Очищуємо екран
 clrscr();
//Записуємо 5 елементів даних до черги
 a.write_data(1);
 a.write_data(2);
 a.write_data(3);
 a.write_data(4);
 a.write_data(5);
//Записуємо 6-ий елемент даних до заповненої черги.
//На екрані з’явиться повідомлення про помилку
 a.write_data(6);
//Зчитуємо елементи даних черги, поки не вичерпаємо їх.
//Після того, як черга стане пустою, при наступному зчитуванні на екрані
//з’явиться повідомлення про помилку
 while((turn_data=a.read_data())!=NULL)
 {
  cout << turn_data << endl;
 }
 return;
}
```

Як видно з прикладу – черга досить просто реалізується на основі динамічного масиву.

Така черга називається ***простою чергою***. Однак така черга має деякі недоліки, а саме: фіксований розмір, який не можна змінити в процесі роботи з чергою, можливість втрати даних при переповненні черги, оскільки не

можна продовжувати запис даних до заповненої черги, поки не будуть прочитані всі елементи. Лише після того, як буде прочитаний останній елемент черги, до неї можна знов записувати дані, починаючи з першого елементу. Цього можна уникнути, створюючи, в разі необхідності, нові черги в динамічному режимі. Але при такому підході не досить ефективно використовується пам'ять. Можна розробити безрозмірну чергу на основі списку. Однак це – досить складне завдання, розгляд якого виходить за рамки даного посібника. Деяким компромісним варіантом, що покращує використання пам'яті і, при цьому, досить просто реалізується є використання циклічних черг.

Циклічні черги

Циклічні черги мають таку ж саму структуру, як і прості черги з однією відмінністю. Ця відмінність полягає в тому, що після заповнення черги запис знов починається з першого елементу черги при умові, що цей елемент уже зчитано. Далі запис продовжується, як і в звичайній черзі при умові, що відповідні елементи черги уже прочитані. При правильному підборі розміру черги з урахуванням швидкості запису й зчитування даних практично можна уникнути проблем з переповненням черги.

Розглянемо приклад практичної реалізації циклічної черги.

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <conio.h>

//Шаблон класу циклічна черга
template <class T> class queue
{
 private:
//Покажчик на адресу, за якою починається черга
        T *q;
//Змінна, що вказує на елемент, який буде зчитано при черговому зчитуванні
        int read_loc;
```

```
//Змінна, що вказує куди буде записано елемент при черговому записі
        int write_loc;
//Розмір черги (максимальна кількість елементів, що можуть
//розміститись в черзі одночасно)
        int length;
//Кількість елементів, розміщених в черзі
        int elem_nums;
 public:
//Конструктор черги
        queue(int size);
//Деструктор черги
        ~queue() {delete [] q;};
//Функція, що записує елемент до черги
        void write_data(T data);
//Функція, що зчитує елемент із черги
        T read_data();
}
//Реалізація конструктора
template <class T> queue<T>::queue (int size)
{
//Виділення пам'яті під чергу
 q=new T[size];
//Повідомлення про помилку в разі невдачі
 if(!q)
 {
  cout << "Error! Can't create turn." << endl;
  exit(1);
 }
//Активація нулем кількості записаних до черги елементів
 elem_nums=0;
//Встановлення розміру черги
 length=size;
//Встановлення індексів зчитування та запису в початкову позицію
 read_loc=write_loc=0;
}

//Реалізація функції, що записує елемент даних до черги
template <class T> void queue<T>::write_data(T data)
{
//Перевірка наявності вільного місця у черзі і відображення повідомлення про
//помилку вразі відсутності вільного місця
 if (elem_nums==length)
 {
  cout << "Error! Turn is full." << endl;
  return;
```

```
 }
//Запис даних до черги при наявності вільного місця
//та збільшення індексу запису
 q[write_loc++]=data;
//Якщо досягнуто кінець черги, то перевести індекс запису у початок
if (write_loc>=length) {write_loc=0;}
//Збільшення числа зайнятих елементів на одиницю
 elem_nums++;
}

//Реалізація функції зчитування елементу даних із черги
template <class T> T queue<T>::read_data()
{
//Перевірка на наявність незчитаних елементів
//та відображення повідомлення про помилку при їх відсутності
 if (elem_nums==0)
 {
  if(read_loc==length){ read_loc=write_loc=0;}
  cout << "Error! Turn is empty." << endl;
  return NULL;
 }
//Зменшення числа незчитаних елементів на одиницю
 elem_nums--;
//Зчитування елементу даних у тимчасову змінну та збільшення
// індексу зчитування на одиницю
tmp=q[read_loc++];
//Якщо досягнуто кінець черги, то перевести
//індекс зчитування у початок черги
 if (read_loc>=length) { read_loc=0;}
//Повернути прочитане значення, як результат роботи функції
 return tmp;
}

//Головна функція програми
void main()
{
//Створення черги з п’яти елементів
 queue<int> a(5);
//Оголошення змінної для тимчасового зберігання елементу даних,
//зчитаних з черги
 int turn_data;
//Очистка екрану
 clrscr();
//Заповнення черги до кінця елементами даних
 a.write_data(1);
```

```
 a.write_data(2);
 a.write_data(3);
 a.write_data(4);
 a.write_data(5);
//Спроба записати дані в заповнену чергу.
//На екрані відобразиться повідомлення про помилку
 a.write_data(6);
//Зчитування елементу черги і вивільнення місця під один елемент даних
 turn_data=a.read_data();
//Відображення зчитаних даних на екрані
 cout << turn_data << endl;
//Запис нового елемнту даних до черги на звільнене місце
 a.write_data(6);
//Зчитування всіх елементів даних черги
//при досягненні кінця черги на екрані відобразиться
//повідомлення про помилку
 while((turn_data=a.read_data())!=NULL)
 {
  cout << turn_data << endl;
 }
//Повторне заповнення черги даними
 a.write_data(1);
 a.write_data(2);
 a.write_data(3);
 a.write_data(4);
 a.write_data(5);
//Зчитування першого елементу черги
 turn_data=a.read_data();
//Запис ще одного елемента даних на звільнене місце
 a.write_data(6);
//Зчитування всіх елементів даних черги
//при досягненні кінця черги на екрані відобразиться
//повідомлення про помилку
 while((turn_data=a.read_data())!=NULL)
 {
  cout << turn_data << endl;
 }
 return;
}
```

Контрольні запитання

1. Що таке черга?
2. Для чого використовують черги?
3. Які бувають черги?
4. В чому полягає відмінність між простими та циклічними чергами?

БІНАРНІ ДЕРЕВА

Остання структура даних, яку ми вивчимо в даному курсі – це дерево. Фактично дерево – це різновид динамічного списку. Існує багато типів дерев, але ми розглянемо лише бінарні дерева. Це визвано тим, що такий тип дерев є значно простішим і найбільш зручним у використанні, порівняно з іншими типами дерев[5].

Бінарне дерево – це структура даних, кожен елемент якої окрім самих даних містить покажчики на два наступних елементи структури. Один з цих наступних елементів умовно називається ***лівим***, а інший ***правим***.

Кожен елемент дерева називається ***вузлом*** або ***листом*** дерева. Перший вузол дерева (з якого дерево власне починається) називається ***коренем***.

Фрагмент дерева разом з вузлом, від якого він починається, називається ***піддеревом*** або ***віткою***.

Множина всіх вузлів, рівновіддалених від кореня, називається ***рівнем***.

Вузол, з якого не починається жодна вітка, називається ***кінцевим*** або ***термінальним*** вузлом.

Оскільки дерево бінарне, кожен вузол може породжувати два вузли наступного рівня. Породжені вузли є ***дочірніми*** по відношенню до вузла, що їх породив. Породжуючий вузол є ***батьківським*** по відношенню до своїх ***дочірніх*** вузлів. Батьківський вузол разом із своїми дочірніми складає ***ланку***.

Сумарна кількість рівнів дерева називається ***висотою дерева***.

Доступ до елементів дерева. Сортування бінарних дерев. Пошук у бінарних деревах

Основними операціями при роботі з деревами є:

- додавання елемента до дерева;
- пошук елемента дерева, що відповідає заданому критерію пошуку;
- сортування елементів дерева;
- вилучення елемента дерева.

Процес доступу до елементів дерева називається ***проходженням дерева***. Існує три способи проходження дерев: послідовний, низхідний та висхідний.

При послідовному методі доступу до елементів дерева порядок проходження дерева наступний: спочатку розглядається самий лівий відносно кореня елемент, потім його батьківський вузол, потім правий елемент даної ланки, потім переходять до елемента попереднього рівня, що є батьківським по відношенню до батьківського вузла даної ланки і т. д. Вверх до кореня, а потім від кореня вниз до самого правого елемента.

При низхідному проходженні дерева порядок обходу елементів зверху-вниз та зліва-направо. Тобто спочатку проходиться корінь, потім його лівий елемент, а потім правий і т.д.

При висхідному проходженні порядок проходження зліва-направо та знизу-вверх. Тобто спочатку проходиться лівий вузол найнижчої ланки, потім правий вузол цієї ланки, а потім їх батьківський вузол і т.д.

Розглянемо приклад. В прикладі будемо використовувати послідовний метод проходження дерева.

Для простоти вважатимемо, що дерево упорядковано (відсортовано) за зростанням відповідно до прямого порядку проходження. При такому упорядкуванні ліва вітка породжена вузлом містить значення менші за значення цього вузла, а права вітка, породжена даним вузлом, містить значення більші за значення цього вузла. Сам процес упорядкування виконуватимемо методом вставки під час формування дерева.

Операції видалення елемента не розглядатимемо, оскільки вона досить складна. Бажаючі можуть знайти потрібну інформацію у літературі вказаній у списку використаної літератури [2,5,6,7].

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
```

```
//клас бінарне дерево
template <class T> class tree
{
  T info; //поле, що містить інформацію
  tree * left; //покажчик на лівий елемент ланки
  tree * right; //покажчик на правий елемент ланки
public:
  tree * root;//покажчик на корінь дерева
  tree(){root=NULL;};//конструктор
//функція, що будує дерево
  void stree(tree<T> *r, tree<T> *previous, T info);
//функція, що відображає дерево на екрані
  void print_tree(tree *r, int l);
  //функція пошуку елемента дерева
  tree<T>  *search_tree(tree<T> *r, T key);
}

//реалізація функції, що будує дерево
template <class T> void tree<T>::stree(tree<T> *r, tree<T> *previous, T info)
{
  if (!r)
  {
    r=new tree<T>; //виділення пам'яті під елемент дерева
    if (!r)//якщо не вдалося виділити пам'ять під елемент
    {
      cout << "Out of memory." << endl;//повідомляємо про це

      exit(1);//і виходимо з процедури
    }
//ініціалізуємо елемент
    r->left=NULL;
    r->right=NULL;
    r->info=info;
    if(!root) //якщо треба робимо його коренем дерева
    {
      root=r;
    }
    else//в противному випадку робимо його звичайним вузлом
    {
      if(info < previous->info) //якщо елемент менший за попередній, то
      {
         previous->left=r;//переходимо до лівої гілки
      }
      else
      {
```

```
        previous->right=r;//інакше до правого
      }
    }
    return;
  }//коли дійшли до потрібного термінального листа, то розміщуємо елемент.
  if (info<r->info) Якщо елемент менший за попередній, то
  {
    stree(r->left,r,info); //розміщуємо його в лівому листі
  }
  else
  {
    stree(r->right,r,info); // інакше розміщуємо його в правому листі
  }
return;
};
//реалізація процедури відображення дерева
//відображуємо відповідно до порядку послідовного проходження дерева
template <class T> void tree<T>::print_tree(tree<T> *r, int l)
{
  int i;
  if(!r)
  {
     return; //якщо нема батьківського вузла, то це корінь і процедуру закінчено
  }
  print_tree(r->right,l+1); //друкуємо елементи правої гілки
  for(i=0;i<l;++i)
  {
    cout << " ";
  }
  cout << r->info << endl;
  print_tree(r->left,l+1); //друкуємо елементи лівої гілки
}
//реалізація процедури пошуку елемента
template <class T> tree<T> *tree<T>::search_tree(tree<T> *r, T key)
{
 if(!r)
 {
  return r; //якщо нема батьківського вузла, то це корінь і процедуру закінчено
 }
```

```
//перевіряємо елементи дерева поки не знайдемо шукане
//або не досягнемо кінця
 while(r->info!=key)
 {
  if(key<r->info)
  {
   r=r->left; //перевіряємо елементи лівої гілки
  }
  else
  {
   r=r->right; //перевіряємо елементи правої гілки
   if(r==NULL){break;}
  }
 }
 return r;
}
//основна програма
void main()
{
 char s[80], key;
 tree<char> chTree; //створюємо об'єкт класу бінарне дерево
 clrscr(); //очищуємо екран
 do //вводимо елементи дерева
 {
  cout << "Input character (. - for stopping):";
  cin >> s;
  if(*s!='.')
  {
    chTree.stree(chTree.root,NULL,*s);
  }
 } while(*s!='.'); //для закінчення вводу натискаємо крапку
 chTree.print_tree(chTree.root,0); //друкуємо дерево на екрані
 getch(); // затримуємо зображення, щоб побачити результат
 cout << "input character for searching-";
 cin >> key; //отримуємо значення для пошуку елемента
 chTree.print_tree(chTree.search_tree(chTree.root,key),0); //виконуємо пошук
 getch();// затримуємо зображення, щоб побачити результат
 return; //закінчуємо програму
}
```

Як бачимо бінарне дерево – досить складна структура даних. Фактично це ускладнений варіант списку. Деякі операції реалізуються значно складніше ніж у списку. Однак операції пошуку у відсортованому бінарному

дереві виконуються значно ефективніше ніж у списку, тому бінарні дерева використовуються на практиці досить часто.

Зверніть увагу на те, що майже всі методи класу бінарне дерево є рекурсивними, оскільки така реалізація при роботі з бінарним деревом – більш ефективна.

Контрольні запитання

1. Що таке бінарне дерево?
2. Що таке вузол (лист) дерева?
3. Що таке корінь дерева?
4. Що таке вітка (піддерево) дерева?
5. Що таке рівень?
6. Який вузол називається кінцевим(термінальним)?
7. Що таке висота дерева?
8. Що таке проходження дерева?
9. Які є методи проходження дерева?
10. Який порядок проходження дерева при послідовному методі?

КОНТРОЛЬНА РОБОТА

Варіант 1

1. Поняття класу основні принципи ООП .
2. Динамічні масиви.
3. Класифікація методів пошуку.

Варіант 2

1. Статичні члени класу.
2. Списки з одинарними зв’язками.
3. Класифікація методів сортування.

Варіант 3

1. Друзі класу.
2. Списки з подвійними зв’язками.
3. Метод прямого пошуку.

Варіант 4

1. Перенавантаження операторів для класів.
2. Стеки.
3. Метод бінарного пошуку.

Варіант 5

1. Доступ до членів класу по посиланню.
2. Прості черги.
3. Метод обміну.

Варіант 6

1. Динамічний розподіл пам’яті в С++.
2. Циклічні черги.
3. Метод вибору.

Варіант 7

1. Шаблони класів.
2. Бінарні дерева.
3. Метод включення.

Варіант 8

1. Перевантаження операторів для класів.
2. Сортування бінарних дерев.
3. Стеки.

Варіант 9

1. Статичні члени класів.
2. Пошук у бінарних деревах.
3. Прості черги.

Варіант 10

1. Поняття абстрактного типу даних.
2. Циклічні черги.
3. Класифікація методів пошуку.

Варіант 11

1. Ініціалізація об’єктів. Конструктори.
2. Списки з одинарними зв’язками.
3. Класифікація методів сортування.

Варіант 12

1. Знищення об’єктів. Деструктори.
2. Списки з подвійними зв’язками.
3. Метод поділу пополам.

Варіант 13

1. Динамічний розподіл пам’яті в С++.
2. Бінарні дерева.
3. Метод прямого пошуку.

Варіант 14

1. Шаблони класів.
2. Стеки.
3. Метод обміну.

Варіант 15

1. Доступ до членів базових класів.
2. Динамічні масиви.
3. Метод включення.

Варіант 16

1. Поняття абстрактного типу даних.
2. Прості масиви.
3. Метод вибору.

Варіант 17

1. Множинне успадкування
2. Особливості сортування списків.
3. Пошук у бінарних деревах

Варіант 18

1. Основні принципи ООП. Інкапсуляція.
2. Особливості пошуку у списках.
3. Класифікація методів сортування.

Варіант 19

1. Функції друзі класу.
2. Циклічні черги.
3. Сортування бінарних дерев.

Варіант 20

1. Етапи розробки моделі.
2. Прості черги.
3. Пошук у бінарних деревах.

Варіант 21

1. Доступ до базових членів класу.
2. Бінарні дерева.
3. Класифікація методів пошуку.

Варіант 22

1. Основні принципи ООП. Наслідування
2. Динамічні масиви.
3. Класифікація методів сортування.

Варіант 23

1. Шаблони класів.
2. Стеки.
3. Сортування бінарних дерев.

Варіант 24

1. Поняття класу і об’єкту.
2. Етапи розробки моделі.
3. Пошук у бінарних деревах.

Варіант 25

1. Основні принципи ООП. Поліморфізм.
2. Стеки.
3. Методи сортування масивів.

Варіант 26

1. Перевантаження операторів для класів.
2. Циклічні черги.
3. Методи пошуку у масивах.

Варіант 27

1. Поняття абстрактного типу даних.
2. Бінарні дерева.
3. Прості черги.

Варіант 28

1. Шаблони класів.
2. Поняття класу і об’єкту.
3. Особливості пошуку у бінарних деревах.

Варіант 29

1. Доступ до членів базових класів.
2. Поняття абстрактного типу даних.
3. Особливості сортування бінарних дерев.

Варіант 30

1. Статичні члени класу.
2. Доступ до полів та методів класу.
3. inline – методи.

ЛАБОРАТОРНІ РОБОТИ

Мета лабораторних робіт – закріплення вивченого матеріалу, вироблення навичок, необхідних для роботи з типовими структурами даних, а також умінь та навичок, необхідних при програмуванні інженерних задач мовою С++ з використанням методів об'єктно орієнтованого програмування.

Порядок виконання лабораторних робіт :

1) Отримати завдання згідно варіанту.
2) Ознайомитися з теоретичними відомостями, необхідними для виконання роботи та отримати у викладача дозвіл на виконання роботи.
3) Написати на мові С++ програму, що виконує дії, вказані у завданні до роботи.
4) Продемонструвати програму викладачеві і отримати дозвіл на оформлення звіту.
5) Оформити звіт.

Зміст звіту:

1) Номер роботи.
2) Варіант.
3) Завдання по роботі згідно варіанту.
4) Теоретичні відомості, необхідні для виконання роботи.
5) Лістинг програми.

Порядок захисту робіт.

Захист робіт відбувається після виконання всіх робіт і отримання у викладача допуску до захисту. Захист проводиться на основі опитування по

матеріалу даного посібника згідно переліку контрольних запитань, наведеного у посібнику.

Лабораторна робота №1.

Тема роботи : Робота з файлами в С++.

Мета роботи – вивчити особливості роботи з файлами в С++. Підготувати файл для виконання наступних робіт.

Завдання

Написати програму, що виконує наступні дії:

1) Обрати з таблиці 1 функцію згідно варіанту.

2) Обчислити значення функції $f(x)$ на інтервалі $x \in [0, \pi/4]$ з кроком $\pi/40$.

3) Обчислені значення зберегти у вигляді файлу, в якому кожній точці відповідає пара чисел x $f(x)$.Ім'я файлу сформувати наступним чином. Перші чотири символи – назва групи латинськими літерами, наступні два символи – варіант. Наприклад DP6102 – група ДП61, другий варіант.

4) Отриманий файл зберегти для виконання наступних робіт.

Таблиця 1.

Варіант	Функція	Варіант	Функція
1	$\sin(x)$	16	$\sin(x)+\cos(x)$
2	$\sin(\cos(x))$	17	$\sin(x)\cos(x)$
3	$\frac{\sin(x)}{x}+\cos(x)$	18	$\cos^2(x)$
4	$\sin(3x)+\cos(2x)$	19	$\cos(x^2)\sin^2(x)$
5	$\sin(\lvert x\rvert)$	20	$\cos^2(\lvert x\rvert)$
6	$x\sin(x)$	21	$\lvert x\rvert\sin(x)$
7	$\cos(x)$	22	$\frac{\sin(x)}{x}$
8	$\sin(x^2)$	23	$\cos(\frac{\sin(x)}{x})$
9	$\cos(x^2)$	24	$\sin(x)-\cos(x)$
10	$\frac{\sin^2(x)}{x}$	25	$\sin(x^2)\cos^2(x)$
11	$\sin^2(\lvert x\rvert)$	26	$\frac{\sin(x)}{\lvert x\rvert}$
12	$x\cos(x)$	27	$x\cos^2(x)$
13	$\sin^2(x)$	28	$\cos(\lvert x\rvert)$
14	$\sin(\frac{\sin(x)}{x})$	29	$\lvert\frac{\sin(x)}{x}\rvert$
15	$\sin(3x)-\cos(2x)$	30	$x\sin^2(x)$

Лабораторна робота №2.

Тема роботи : : Робота з масивами та списками. Методи сортування та пошуку.

Мета роботи – навчитись використовувати масиви та списки при розробці програм, вивчити методи сортування.

Завдання

Написати програму, що виконує наступні дії:

1) Зчитує дані із файлу отриманого в першій лабораторній роботі та зберігає їх у пам'яті у вигляді структури заданої у таблиці 2 відповідно до варіанту завдань.

2) Виводить дані на екран у вигляді двох стовпців . x $f(x)$, розділених трьома символами пробілу. Стовпці повинні мати заголовки X та Y відповідно.

3) Сортувати дані за зростанням або за спаданням за вибором користувача методом, заданим у таблиці 3 відповідно до варіанту.

4) Виводити на екран відсортовану послідовність.

5) Здійснювати бінарний пошук введеної з клавіатури величини та виводити на екран результат пошуку. Величини для пошуку (ключі пошуку) повинні зберігатись у програмі у вигляді черги, розмір якої також вводиться з клавіатури. Результати пошуку повинні відображатись на екрані у порядку введення ключів пошуку.

Таблиця 2.

Структура	Варіант									
Динамічний масив	1	4	7	10	13	16	19	22	25	28
Список з одинарними зв'язками	2	5	8	11	14	17	20	23	26	29
Список з подвійними зв'язками	3	6	9	12	15	18	21	24	27	30

Таблиця 3.

Метод	Варіант									
Метод вибору	1	4	7	10	13	16	19	22	25	28
Метод включення	2	5	8	11	14	17	20	23	26	29
Метод обміну	3	6	9	12	15	18	21	24	27	30

Лабораторна робота №3.

Тема роботи : Робота зі списками. Динамічна зміна розмірів списків.

Мета роботи – закріпити навички використання списків при розробці програм, вивчити прийоми динамічного розподілу пам’яті.

Завдання

Написати програму, що виконує наступні дії:

1) Зчитує дані із файлу отриманого в першій лабораторній роботі та зберігає їх у пам’яті у вигляді списку заданого у таблиці 4 відповідно до варіанту завдань.

2) Виводить дані на екран у вигляді двох стовпців . x $f(x)$, розділених символами табуляції. Стовпці повинні мати заголовки X та Y відповідно.

3) Обчислює значення функції у точках, що знаходяться посередині між сусідніми точками отриманими з файлу, як середнє значення двох сусідніх значень і добавляє ці точки до свого списку.

4) Виводить отриманий список на екран, як визначено у п.2.

5) Видаляє із списку 5 елементів, що містять дані введені з клавіатури та відображає отриманий список на екрані згідно п. 2.

Таблиця 4.

Структура	Варіант														
Список з подвійними зв’язками	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29
Список з одинарними зв’язками	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30

Лабораторна робота №4.

Тема роботи : Використання списків для збереження та відображення графічної інформації.

Мета роботи – навчитись використовувати списки для тимчасового збереження великих об’ємів інформації.

Завдання

Написати програму, що виконує наступні дії:

1) Зчитує дані із файлу отриманого в першій лабораторній роботі та зберігає їх у пам’яті у вигляді структури заданої у таблиці 5 відповідно до варіанту завдань.

2) Визначає максимальний та мінімальний елементи списку.

3) Розраховує масштабні коефіцієнти для відображення графіку функції на екрані. Масштабні коефіцієнти повинні бути розраховані таким чином, щоб графік функції займав весь екран, як по вертикалі так і по горизонталі.

4) Відображати графік функції на екрані. Крім графіку на екрані повинні відображатись осі координат, масштабна сітка та значення по осях X та Y.

Таблиця 5.

Структура	Варіант									
Список з подвій-ними зв’язками	1	4	7	10	13	16	19	22	25	28
Динамічний масив	2	5	8	11	14	17	20	23	26	29
Список з одинар-ними зв’язками	3	6	9	12	15	18	21	24	27	30

Лабораторна робота №5.

Тема роботи : Робота з бінарними деревами.

Мета роботи – вивчити особливості роботи з бінарними деревами.

Завдання

Написати програму, що виконує наступні дії:

1) Генерує за допомогою генератора випадкових чисел 10 символів латинського алфавіту.

2) З отриманих символів будує впорядковане бінарне дерево, відсортоване у заданому порядку відповідно до напряму проходження дерева.

3) Відображає отримане дерево на екрані.

4) Здійснює пошук серед елементів дерева значення введеного з клавіатури та виводить на екран повідомлення про номер знайденого елемента згідно порядку проходження дерева. Якщо такого елемента нема, то програма виводить повідомлення про його відсутність.

Порядок проходження дерева визначається відповідно до варіанта згідно таблиці 6. Порядок сортування елементів дерева за зростанням для парних варіантів та за зменшенням для непарних.

Таблиця 6.

Порядок проходження дерева	Варіант									
Висхідний	1	4	7	10	13	16	19	22	25	28
Низхідний	2	5	8	11	14	17	20	23	26	29
Прямий	3	6	9	12	15	18	21	24	27	30

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Прокопенко Ю.В., Татарчук Д.Д., Казміренко В.А. Обчислювальна математика [Текст]: Навч. посіб. / Ю.В. Прокопенко, Д.Д. Татарчук., В.А. Казміренко. – К.: ІВЦ „Видавництво ”Політехніка” ”, 2003. – 120 с. : іл. ; – Бібліогр.: с. 119. - 300 екз.
2. Вирт, Н. Алгоритмы и структуры данных [Текст] / Н. Вирт – 2-е изд., испр. – С-Пб.: Невский Диалект, 2001.-352 с.: ил.; – Библиогр.: с. 337. – Предм. указ.: с. 346 – 348. – Перевод с английского Д. Б. Подшивалова. – ISBN 5-7940-0065-1 (рус.).
3. Пол Ирэ Объектно-ориентированное программирование с использованием C++ [Текст] / Ирэ Пол – К.: НИПФ „ДиаСофт Лтд.”, 1995. – 480 с.: ил.; – Перевод с английского А.С. Климов. – ISBN 5-7707-7219-0 (рус.).
4. Карпов Б., Баранова Т. C++[Текст] : специальный справочник / Б. Карпов, Т. Баранова. – СПб.: Издательство «Питер», 2000. – 480 с.; – Предм. указ.: с. 465 – 479. – ISBN 5-272-00076-5 (рус.).
5. Шилдт, Г. Теория и практика C++ [Текст] / Г. Шилдт. – С-Пб.: “BHV – Санкт-Петербург”, 1996. – 416 с.: ил.; – Предм. указ.: с. 409 – 412. – Перевод с английского Ольга Кокарева. – 10000 экз. – ISBN 5-7791-0029-2 (рус.).
6. Фридман , А. C/C++. Архив программ [Текст] / А. Фридман, Л. Кландер, М. Михаєлис, Х. Шильдт; под общ. ред. В. Тимофеева – М.: ЗАО „Издательство БИНОМ”, 2001 г. – 640 с.: ил.; –Перевод с английского. – 4000 экз. – ISBN 5-7989-0205-6 (рус.).
7. Седжвик, Р. Фундаментальные алгоритмы на C++. Алгоритмы на графах [Текст] / Роберт Седжвик. – СПб: ООО „ДиаСофтЮП”, 2002. – 496 с. : ил.; – Предм. указ.: с. 479 – 484. – Перевод с английского. – 3000 экз. – ISBN 5-93772-054-7.