ЛАБОРАТОРНА РОБОТА №2

Тема: Робота з лінійними списками. Конструктор і деструктор класу

Мета: Навчитись використовувати конструктори і деструктори класів, створювати класи для опису лінійних списків

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

1. Конструктори та деструктори

Конструктор – це метод (функція-член класу), який викликається при створенні нового об'єкту класу. Конструктор являє собою метод класу, що полегшує програмам ініціалізацію елементів даних класу. Він викликається автоматично кожен раз, коли створюється об'єкт класу. Конструктор є обов'язковим, якщо використовуються віртуальні правила (поліморфізм). Кожен окремий екземпляр об'єкта повинен ініціалізуватись з використанням конструктора.

Конструктор має таке ж ім'я, як і клас. Конструктор не повертає результату (void писати не потрібно). Описується конструктор так:

```
class class_name   // якийсь клас з назвою class_name
{      …      …  // закриті елементи даних класу
public:       …            // відкриті елементи даних та методи класу
       class_name(список параметрів); //конструктор (інтерфейсна частина)
// має таке ж ім'я, як і назва класу
}
class_name::class_name(); // опис реалізації конструктора
{   ….   }
```

Якщо клас є похідним від базового класу, або містить в собі об'єкти, які мають власні конструктори, то конструктор базового класу та конструктори об'єктів-членів класу викликаються автоматично перед викликом конструктора похідного класу. В тілі конструктора не можна використовувати оператор return. Конструктор не може бути віртуальною функцією.

Деструктор – навпаки викликається при знищенні об'єкту, і тому в ньому ми повинні звільнити пам'ять зарезервовану в конструкторі. Він викликається автоматично кожен раз, коли ви знищуєте об'єкт класу. Для глобальних статичних змінних це відбувається при завершенні роботи програми, в порядку, зворотньому до їх оголошення, для автоматичних — при виході з блоку, для динамічних — при виклику оператора **delete**.

Деструктор має таке ж ім'я, як і клас, за винятком того, що перед ним додається символ "~" (тильда). Деструктор не повертає результату (void писати не потрібно) і не отримує ніяких параметрів. Описується деструктор так:

```
class class_name   // якийсь клас з назвою class_name
{      …      …
public:       …
       class_name(); //конструктор (має ім'я, як назва класу)
       ~class_name(); // деструктор (назві передує "тильда")
}
class_name::~class_name(); // опис реалізації деструктора
{   ….   }
```

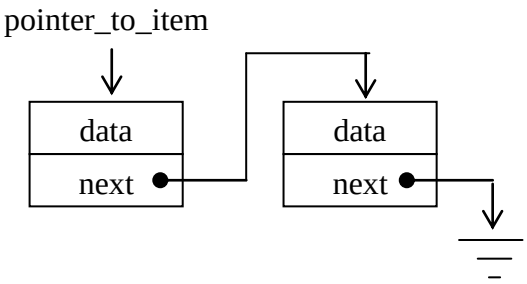
Деструктор для базового класу виконується після деструктора похідного від нього класу. Деструктори для об'єктів-членів виконуються після деструктора для об'єкта, членами якого вони є. Деструктор може бути віртуальною функцією.

Динамічні об'єкти конструюються та знищуються з використанням операторів **new** та **delete**. Пам'ять для збереження динамічних об'єктів в C++ виділяється за допомогою оператора **new** . Коли за допомогою оператора **new** створюється об'єкт класу, то неявно викликається конструктор цього класу, який ініціалізує елементи класу та виділяє під них пам'ять. Оператор **delete** для об'єкта неявно викликає його деструктор для звільнення пам'яті.

2. Лінійні списки

Для представлення взаємопов'язаних даних крім структур та масивів використовується поняття лінійного списку. Кожен елемент такого списку містить корисні дані та покажчик (адресу) на наступний елемент списку (або порожній покажчик **null** для останнього елемента списку). Оскільки довжина списку наперед невідома, то найкраще його елементи розміщувати у динамічній пам'яті, створюючи та знищуючи їх операторами **new** та **delete**. Враховуючи неоднорідний характер даних для кожного елемента списку (покажчик та елемент іншого типу) можна представляти його з допомогою покажчика на структуру:

```
struct ITEM
{
       ITEM *next;
       SOMETYPE data;
} var_item, *pointer_to_item;
```



тут ITEM — ідентифікатор елемента списку, `var_item`— змінна типу список (елемент списку), `pointer_to_item` — покажчик на список, `next` — поле, що вказує на наступний елемент списку, `data` — корисні дані типу `SOMETYPE` (наприклад, ціле число або покажчик на якийсь складніший тип даних).

Основні операції, що виконуються над списками — це:

1) пошук у списку елемента із заданою властивістю;
2) визначення *i*-го елемента у списку;
3) внесення додаткового елемента до або після вказаного вузла;
4) вилучення певного елементу зі списку;
5) упорядкування лінійного списку за певною ознакою;
6) вивід списку на екран;
7) повне знищення списку.

Всі ці операції можуть бути зреалізовані за допомогою зовнішніх функцій, проте найзручніше їх зробити методами класу "список". Приклад такої реалізації наведений нижче:

```
class LIST
{
  LIST *next;
  int data;
public:
  LIST(int newdata, LIST *oldlist=0) {data=newdata; next=oldlist;}
  ~LIST();
  virtual void print();
};
```

Клас `LIST` являє собою список (фактично перший елемент — "голову" списку — який може містити покажчики на наступні елементи списку). Якщо створити статичну змінну типу `LIST` то отримаємо список, що складається з одного елементу, покажчик на `LIST`, що рівний **null** (не ініціалізований) являє собою порожній список (список, в якому немає ні одного елемента).

Кожен елемент списку містить поле `data` для зберігання цілого числа та покажчик `next` на наступний об'єкт типу `LIST`. Ці дані є закритими (private), тому доступ до них можуть мати тільки методи класу `LIST`.

Конструктор `LIST(…)` ініціалізує поля `data` та `next` значеннями `newdata` та `oldlist`, заданими в його параметрах. Оскільки він дуже простий, то записаний у вигляді inline-функції, тобто в середині опису класу. Якщо об'єкт створюється в динамічній пам'яті з допомогою операторами **new**, то цей конструктор викликається автоматично і крім ініціалізації забезпечує ще і виділення потрібної кількості пам'яті для зберігання `data` та `next`. При виклику конструктора, другий параметр (покажчик на наступний елемент списку) можна опускати, тоді йому присвоїться значення за замовчуванням **null** (див. в описі конструктора: … `LIST *oldlist=0`), тобто створиться новий список, що складається тільки з одного елемента.

Враховуючи рекурсивну природу списку, операції над ним найзручніше реалізовувати з використанням рекурсивних функцій. Так деструктор `~LIST()` знищує об'єкт типу `LIST`, звільняє зайняту ним пам'ять, а також повинен знищити всі наступні елементи списку, на який вказує покажчик `next`, якщо він не **null** (тобто поки не досягнуто "хвоста" списку):

```
LIST::~LIST()
{
   if (next)
      delete next;
}
```

Тут деструктор буде викликатись рекурсивно для кожного елемента списку. Використання умови `if (next)` є обов'язковим, щоб рекурсія завершилась на останньому елементі списку ("хвості"), а не стала нескінченною.

Аналогічно зреалізований рекурсивний метод `virtual void print()`, що виводить на екран послідовно всі значення поля списку `LIST`, розділені символом табуляції.

```
void LIST::print()
{
   cout<<data<<'\t';
   if (next)
      next->print();
}
```

Вивід здійснюється починаючи від "голови" списку до його "хвоста". Ця функція зроблена віртуальною, щоб можна було створювати породжені класи (списки з іншими типами даних), які б виводили свої дані.

Функція `main()` для робити зі списком `LIST` може мати такий вигляд:

```
void main(){
   LIST *mylist = new LIST(1);     //створити список з елементом "1"
   mylist = new LIST(2,mylist);    //додати елемент "2"
   mylist = new LIST(3,mylist);    //додати елемент "3"
   mylist->print();  //вивід списку на екран
   delete mylist;    //знищення списку
}
```

Тут створюється список `mylist`, що складається з одного елемента зі значенням поля `data=1`, потім в його "голову" додається ще два елементи (місять числа 2 та 3), далі весь список виводиться на екран і знищується.

Результати виконання програми (числа виводять від "голови" до "хвоста"):

```
3       2       1
```

ПОРЯДОК ВИКОНАННЯ РОБОТИ

1 Реалізувати один простий клас згідно варіанту індивідуального завдання, що містить закриті данні, а саме два типа даних: числове та рядкове, реалізоване через покажчик на char. Потрібно створити декілька екземплярів класу статично і динамічно, а також масив та продемонструвати дію всіх конструкторів і методів. Реалізувати методи:

- Scan - ввід даних з клавіатури у поля класу;
- Print - констатний метод виводу даних на екран;
- Конструктор по замовчуванню;
- Конструктор ініціалізації Клас(char*, int);
- Конструктор копіювання Клас(const Клас&);
- Деструктор
- Методи доступу та закритих данних Get та Set.

Індивідуальні завдання

Варіант 1.

```
class Toys
{
  char *Owner;
  int Old+;
public:
  Toys();
  Toys( char * , int  );
  Toys(  const Toys& );
    void SetOwner( char * );
  char * GetOwner( );
  void Setchar *( int  );
  int  Getchar *( );
  void Print ();
  void Input ();
  ~Toys();
};
```

Варіант 2.

```
class Animal
{
  char *Species;
  int Old;
public:
  Animal();
  Animal( char * , int  );
  Animal(  const Animal& );
    void SetSpecies( char * );
  char * GetSpecies( );
  void SetOld( int  );
  int  GetOld( );
  void Print ();
  void Input ();
  ~Animal();
};
```

Варіант 3.

```
class House
{
  char *Type;
  int Number;
public:
  House();
  House( char * , int  );
```

Варіант 4.

```
class Plant
{
  char *Species;
  int Height;
public:
  Plant();
  Plant( char * , int  );
```

```
  House(  const House& );
    void SetType( char * );
  char * GetType( );
  void SetNumber( int  );
  int  GetNumber( );
  void Print ();
  void Input ();
  ~House();
};
```

```
  Plant(  const Plant& );
  void SetSpecies( char * );
  char * GetSpecies( );
  void SetHeight( int  );
  int  GetHeight( );
  void Print ();
  void Input ();
  ~Plant();
};
```

Варіант 5.

```
class City
{
  char *Name;
  int Arial;
public:
  City();
  City( char * , int  );
  City(  const City& );
  void SetName( char * );
  char * GetName( );
  void SetArial( int  );
  int  GetArial( );
  void Print ();
  void Input ();
  ~City();
};
```

Варіант 6.

```
class Country
{
  char *Title;
  int People;
public:
  Country();
  Country( char * , int  );
  Country(  const Country& );
  void SetTitle( char * );
  char * GetTitle( );
  void SetPeople( int  );
  int  GetPeople( );
  void Print ();
  void Input ();
  ~Country();
};
```

Варіант 7.

```
class Airplane
{
  char *Model;
  int Power;
public:
  Airplane();
  Airplane( char * , int  );
  Airplane(  const Airplane& );
  void SetModel( char * );
  char * GetModel( );
```

Варіант 8.

```
class Food
{
  char *Type;
  int Calories;
public:
  Food();
  Food( char * , int  );
  Food(  const Food& );
  void SetType( char * );
  char * GetType( );
```

```
  void SetPower( int  );
  int  GetPower( );
  void Print ();
  void Input ();
  ~Airplane();
};
```

```
  void SetCalories( int  );
  int  GetCalories( );
  void Print ();
  void Input ();
  ~Food();
};
```

Варіант 9.

```
class Vegetables
{
  char *Color;
  int Price;
public:
  Vegetables();
  Vegetables( char * , int  );
  Vegetables(  const Vegetables& );
  void SetColor( char * );
  char * GetColor( );
  void SetPrice( int  );
  int  GetPrice( );
  void Print ();
  void Input ();
  ~Vegetables();
};
```

Варіант 10.

```
class Furniture
{
  char *Room;
  int Weight;
public:
  Furniture();
  Furniture( char * , int  );
  Furniture(  const Furniture& );
  void SetRoom( char * );
  char * GetRoom( );
  void SetWeight( int  );
  int  GetWeight( );
  void Print ();
  void Input ();
  ~Furniture();
};
```

Варіант 11.

```
class Flat
{
  char *Size;
  int Size;
public:
  Flat();
  Flat( char * , int  );
  Flat(  const Flat& );
  void SetSize( char * );
  char * GetSize( );
  void SetSize( int  );
  int  GetSize( );
  void Print ();
```

Варіант 12.

```
class Computer
{
  char *Owner;
  int Processor;
public:
  Computer();
  Computer( char * , int  );
  Computer(  const Computer& );
  void SetOwner( char * );
  char * GetOwner( );
  void SetProcessor( int  );
  int  GetProcessor( );
  void Print ();
```

```
  void Input ();
  ~Flat();
};
```

```
  void Input ();
  ~Computer();
};
```

Варіант 13.

```
class Worker
{
  char *Surname;
  int Year;
public:
  Worker();
  Worker( char * , int  );
  Worker(  const Worker& );
  void SetSurname( char * );
  char * GetSurname( );
  void SetYear( int  );
  int  GetYear( );
  void Print ();
  void Input ();
  ~Worker();
};
```

Варіант 14.

```
class Firm
{
  char *Type;
  int Size;
public:
  Firm();
  Firm( char * , int  );
  Firm(  const Firm& );
  void SetType( char * );
  char * GetType( );
  void SetSize( int  );
  int  GetSize( );
  void Print ();
  void Input ();
  ~Firm();
};
```

Варіант 15

```
class Clothing
{
  char *Color;
  int Age;
public:
  Clothing();
  Clothing( char * , int  );
  Clothing(  const Clothing& );
  void SetColor( char * );
  char * GetColor( );
  void SetAge( int  );
  int  GetAge( );
  void Print ();
  void Input ();
  ~Clothing();
};
```

Варіант 16

```
class Car
{
  char *Model;
  int Size;
public:
  Car();
  Car( char * , int  );
  Car(  const Car& );
  void SetModel( char * );
  char * GetModel( );
  void SetSize( int  );
  int  GetSize( );
  void Print ();
  void Input ();
  ~Car();
};
```

2 Виконати завдання згідно варіанту індивідуального завдання:

1. Створити клас для реалізації однозв’язного списку з динамічним виділенням пам’яті типу “черга” та методи додавання числа в чергу та вилучення з неї.
2. Створити клас для реалізації однозв’язного списку з динамічним виділенням пам’яті типу “стек” та методи додавання числа в стек та вилучення з нього.
3. Створити клас для реалізації однозв’язного списку з динамічним виділенням пам’яті типу “дек” та методи додавання числа в дек та вилучення з нього. Додавати з хвоста і вилучати з хвоста.
4. Створити клас для реалізації однозв’язного списку з динамічним виділенням пам’яті типу “дек” та методи додавання числа в дек та вилучення з нього. Додавати з хвоста і вилучати з голови.
5. Організувати список, заповнити його випадковими числами і зробити функції впорядкування списку та пошуку. Функція пошуку повинна виводити число на екран і вертати адресу даного елемента і його порядковий номер в списку.
6. Організувати два списки, заповнити їх впорядкованими за зростанням числами і зробити функції злиття цих списків.
7. Організувати чергу, заповнити її випадковими числами і зробити функції додавання в чергу та знаходження середнього арифметичного чисел записаних у чергу та їх кількості. Величина черги наперед невідома.
8. Створити чергу з числами та функції, які б обчислювали добуток усіх від’ємних членів та середнє арифметичне додатних членів черги.
9. Створити список з функціями його упорядкування та додавання числа в список без порушення порядку.
10. Створити двозв’язний список з пошуком в 2 напрямках.
11. Організувати список в якого елементами є масив з трьох рядкових змінних, заповнити його послідовно введеними рядками з прізвищами ім’ям та по батькові. Створити методи, які виводять кожен елемент списку у новому рядку послідовно у повній (ім’я, по батькові, прізвище) та скороченій формі (прізвище та ініціали).
12. Організувати список, кожен елемент якого складається з рядкової змінної та цілого числа, заповнити його послідовно введеними словами, не допускаючи повторів. Число у кожному елементі списку повинно відповідати кількості повторів для слова у цьому елементі. Створити метод, що виводить список слів та кількість повторів.
13. Створити двозв’язний список та заповнити його n-ою кількістю простих чисел.
14. Створити список типу “кільце” з корисною інформацією: рядки, що містять стадії кругообігу води в природі. Створити методи, що виводять поточний елемент кільця та переходять до наступного.
15. Створити двозв’язний список з простими числами. Числа заповнити за таким алгоритмом: спочатку заповнити список числами від 1 і до якогось n. Потім починаючи з двійки перевіряти і вилучати всі числа які діляться на неї, потім брати наступний елемент списку і робити аналогічну операцію, поки не буде чисел, що діляться на інші.

ОФОРМЛЕННЯ ЗВІТУ:

1. Титульний лист.
2. Мета роботи.
3. Короткі теоретичні відомості.
4. Текст завдання згідно варіанту.
5. Код програми.

6. Screen-shot вікна виконання програми.
7. Висновки.

Контрольні питання

1. Що таке конструктор?
2. В яких випадках використовуються конструктори?
3. Яким чином здійснюється ініціалізація об’єктів класу?
4. Чи можна замість ініціалізації динамічного об’єкту використовувати оператор присвоювання йому іншого об’єкта того ж класу?
5. Як описується конструктор?
6. Який тип результату може повертати конструктор?
7. Чи може бути конструктор віртуальною функцією?
8. Вкажіть послідовність виклику конструкторів базового, похідного класу та конструкторів об’єктів-членів класу.
9. Як можна описати реалізацію (тіло) конструктора? Наведіть приклад.
10. Що таке деструктор?
11. В яких випадках використовуються деструктори?
12. Яким чином звільнюється пам’ять, яку займають динамічні об’єкти?
13. Коли викликаються деструктори для статичних, автоматичних та динамічних об’єктів?
14. Як описується деструктор?
15. Яке ім’я може мати деструктор?
16. Який тип результату може повертати деструктор?
17. Скільки параметрів може мати деструктор?
18. Чи може бути деструктор віртуальною функцією?
19. В якому порядку викликаються деструктори базового, похідного класу та об’єктів-членів класу.
20. Як можна описати реалізацію (тіло) деструктора? Наведіть приклад.
21. Які оператори використовуються для створення та знищення динамічних об’єктів?
22. Що відбувається при виклику оператора new для об’єкта?
23. Що відбувається при виклику оператора delete для об’єкта?
24. Що таке лінійний список?
25. Які типи даних використовуються для реалізації динамічного лінійного списку?
26. Які основні операції виконуються над списками?
27. Що містить у собі список, реалізований у вигляді класу?
28. Які методи повинен містити клас, що реалізує динамічний список?
29. Які дії повинен виконувати конструктор класу, що реалізує динамічний список?
30. Які дії повинен виконувати деструктор класу, що реалізує динамічний список?
31. Який вид функцій найзручніше використовувати для реалізації методів класу “Динамічний список”?
32. Наведіть приклад методу, що виводить на екран значення елементів динамічного списку.