Лекція 9 : ООП - наслідування

Наслідування

Однією з найважливіших концепцій об'єктно-орієнтованого програмування є наслідування (успадкування). **Наслідування *дозволяє визначити клас у термінах іншого класу***, що полегшує створення та підтримку програми. Це також дає можливість повторного використання функціональності коду та пришвидшення часу реалізації класу та модифікації класу.

При створенні класу, замість написання абсолютно нових членів даних і методів, програміст може призначити, що новий клас повинен успадковувати члени і методи (можливо не всі) існуючого класу. Цей існуючий клас називається ***базовим*** класом (***батьківським*** класом), а новий клас називається ***похідним*** класом (класом -***нащадком***).

Ідея успадкування реалізує взаємозв'язок між об’єктами, який можна назвати «належить до» або «IS-A». Наприклад, ссавець «IS-A» тварина, собака «IS-A» ссавець, отже, собака «IS-A» тварина також і так далі.

Базові та похідні класи

Клас може бути отриманий з більш ніж одного класу, що означає, що він може успадковувати дані та функції з декількох базових класів. Щоб визначити похідний клас, ми використовуємо список батьків класу для визначення базового класу. Список батьківських класів вказує один або більше базових класів і має вигляд:

КЛАС_НАСЛІДНИК: <специфікатор-доступу> БАЗОВИЙ_КЛАС

Специфікатор доступу є одним із наступних:

- загальнодоступний (public),
- захищений(protected);
- приватний(private),

а БАЗОВИЙ_КЛАС- це назва раніше визначеного класу. Якщо специфікатор доступу не використовується, то за замовчуванням він є приватним(private).

Розглянемо базовий клас **Shape** і його похідний клас **Rectangle** наступним чином

```
#include <iostream>
using namespace std;
// Базовий клас
class Shape {
  public: // публічні методи
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
```

```
    }

  protected:
    int width;
    int height;
};

// Клас-нащадок
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);
    }
};

int main(void) {
  Rectangle Rect;
  Rect.setWidth(5);
  Rect.setHeight(7);
  // Виведемо площу прямокутнику
  cout << "Total area: " << Rect.getArea() << endl;
  return 0;
}
```

Результат

```
Total area: 35
```

Контроль доступу та успадкування

Похідний клас може отримати доступ до всіх не приватних членів свого базового класу. Таким чином, члени базового класу, які не повинні бути доступними для функцій-членів похідних класів, повинні бути оголошені приватними в базовому класі.

Ми можемо контролювати різні типи доступу відповідно до того, хто може отримати до них доступ яким чином

Захищені (protected) члени

Захищені члени або методи класу дуже схожі на приватний член, але відмінність полягає в тому, що доступ до них можна отримати в класах-нащадках, які називаються похідними класами або класами-нащадками.

Це можна перевірити наступним прикладом, де є один дочірній клас SmallBox з батьківського класу Box.

Наступний приклад аналогічний наведеному вище прикладу і тут ширина елемента буде доступна будь-якому методу його похідного класу SmallBox.

```
#include <iostream>
```

```
using namespace std;
class Box {
  protected:
    double width;
};

class SmallBox:Box { // SmallBox клас-нащадок
  public:
    void setSmallWidth( double wid );
    double getSmallWidth( void );
};

// Методи класу -нащадку
double SmallBox::getSmallWidth(void) {
  return width ;
}
void SmallBox::setSmallWidth( double wid ) {
  width = wid;
}
int main() {
  SmallBox box;
  // встановлюємо ширину в box
  box.setSmallWidth(5.0);
  cout << "Width of box : "<< box.getSmallWidth() << endl;
  return 0;
}
```

Результат роботи програми:

```
Width of box : 5
```

Похідний клас успадковує всі методи базового класу з такими винятками:

- конструктори, деструктори та конструктори копіювання базового класу;
- перевантажені оператори базового класу;
- дружні функції базового класу.

Тип успадкування

При виведенні класу з базового класу базовий клас може успадковуватися через **загальне**, **захищене** або **приватне** успадкування. Тип успадкування визначається специфікатором доступу, як описано вище.

На практиці достатньо рідко використовують захищене або приватне наслідування, але достатньо часто загальнодоступне наслідування.

Під час використання іншого типу спадкування застосовуються наступні правила:

- **Public** Inheritance (загальнодоступне наслідування) - при виведенні класу з відкритого базового класу відкриті члени базового класу стають загальнодоступними членами похідного класу, а захищені члени базового класу стають захищеними членами похідного класу. Приватні члени базового класу ніколи не доступні безпосередньо з похідного класу, але можуть бути доступні через виклики загальнодоступним і захищеним членам базового класу.
- **Protected** Inheritance (захищене наслідування) - при виведенні з захищеного базового класу відкриті та захищені члени базового класу стають захищеними членами похідного класу.
- **Private** Inheritance (приватне наслідування) - при отриманні від приватного базового класу відкриті та захищені члени базового класу стають приватними членами похідного класу.

Множинне наслідування

Клас C ++ може успадкувати з більш ніж одного класу, а ось розширений синтаксис -

КЛАС_НАСЛІДНИК: специфікатор-доступу БАЗОВИЙ_КЛАС_1, специфікатор-доступу БАЗОВИЙ_КЛАС_2, ...

або

КЛАС_НАСЛІДНИК: специфікатор-доступу БАЗОВИЙ_КЛАС_1, БАЗОВИЙ_КЛАС_2, ...специфікатор-доступу БАЗОВИЙ_КЛАС_N, ...

Там, де доступ є одним він буде наданий для кожного базового класу, вони будуть розділені комою, як показано вище. Спробуємо наступний приклад

```
#include <iostream>
using namespace std;

// Базовий клас Shape
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};

// Базовий клас PaintCost
```

```
class PaintCost {
  public:
    int getCost(int area) {
      return area * 70;
    }
};

// Клас нащадок
class Rectangle: public Shape, public PaintCost {
  public:
    int getArea() {
      return (width * height);
    }
};

int main(void) {
  Rectangle Rect;
  int area;

  Rect.setWidth(5);
  Rect.setHeight(7);

  area = Rect.getArea();
  // Вивод площі
  cout << "Total area: " << Rect.getArea() << endl;
  // Виведення вартості
  cout << "Total paint cost: $" << Rect.getCost(area) << endl;
  return 0;
}
```

Результат:

```
Total area: 35
Total paint cost: $2450
```

Перезавантаження методів

С ++ дозволяє вказати більше ніж одне визначення для назви функції або оператора в деякій області. Це називається **перевантаженням функції** і **перевантаження оператора** відповідно.

Перевантажена (override) декларація - це декларація, яка оголошена з ***тією ж назвою***, що й раніше оголошена декларація в тій самій області, за винятком того, що обидві декларації мають ***різні аргументи*** і очевидно ***різні визначення*** (реалізації).

Коли ви викликаєте перевантажену функцію або оператор, компілятор визначає найбільш прийнятне визначення для використання, порівнюючи типи аргументів, які ви використовували для виклику функції або оператора з типами параметрів, вказаними в визначеннях. Процес вибору найбільш відповідної перевантаженої функції або оператора називається роздільною здатністю перевантаження.

Зокрема можна створити декілька визначень з одним іменем функції в тій же області. Визначення функції повинно відрізнятися один від одного типами та / або числом аргументів у списку аргументів. Не можна перевантажувати декларації функцій, які відрізняються тільки типом повернення.

Нижче наведено приклад, в якому для друку різних типів даних використовується одна і та ж функція print ()

```
#include <iostream>
using namespace std;
class printData {
  public:
    void print(int i) {
      cout << "Printing int: " << i << endl;
    }
    void print(double  f) {
      cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
      cout << "Printing character: " << c << endl;
    }
};

int main(void) {
  printData pd;
  // Виклик методу print для цілого
  pd.print(5);
  // Виклик методу print для float
  pd.print(500.263);
// Виклик методу print для символу
  pd.print("Hello C++");
  return 0;
}
```

Результат:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Перевантаження Операторів в C ++

Ви можете перевизначити або перевантажити більшість вбудованих операторів, доступних у C ++. Таким чином, програміст може також використовувати оператори з визначеними користувачем типами.

Перевантажені оператори - це функції зі спеціальними іменами: ключове слово "operator", за яким слідує символ, який визначається оператором. Як і будь-яка інша функція, перевантажений оператор має тип повернення і список параметрів.

Приклад.

Box operator+ (const Box &);

Ця декларація оголошує оператор додавання, який можна використовувати для додавання двох об'єктів Box і повертає кінцевий об'єкт Box. Більшість перевантажених операторів можуть бути визначені як звичайні нечленові функції або функції членів класу. У випадку, якщо ми визначимо вищезгадану функцію як нечленовану функцію класу, то нам доведеться передати два аргументи для кожного операнду наступним чином

Box operator+ (const Box &, const Box &);

Нижче наведено приклад, що показує концепцію перевантаження оператору за допомогою функції-члена. Тут об'єкт передається як аргумент, властивості якого будуть доступні за допомогою цього об'єкта, об'єкт, який викликатиме цей оператор, може бути доступний за допомогою цього оператора, як описано нижче

```
#include <iostream>
using namespace std;
class Box {
  public:
    double getVolume(void) {
      return length * breadth * height;
    }
    void setLength( double len ) {
      length = len;
    }
    void setBreadth( double bre ) {
      breadth = bre;
    }
    void setHeight( double hei ) {
      height = hei;
    }

    // Overload + operator to add two Box objects.
    Box operator+(const Box& b) {
      Box box;
      box.length = this->length + b.length;
```

```
        box.breadth = this->breadth + b.breadth;
        box.height = this->height + b.height;
        return box;
      }

    private:
      double length;
      double breadth;
      double height;
};
int main() {
    Box Box1;              // Змінна Box1 класу Box
    Box Box2;              // Змінна Box2 класу Box
    Box Box3;              // Змінна Box3 класу Box
    double volume = 0.0;     // Змінна для обєму

    // box 1 ініціалізація
    Box1.setLength(6.0);
    Box1.setBreadth(7.0);
    Box1.setHeight(5.0);

    // box 2 ініціалізація
    Box2.setLength(12.0);
    Box2.setBreadth(13.0);
    Box2.setHeight(10.0);

    // обєм Box1 1
    volume = Box1.getVolume();
    cout << "Volume of Box1 : " << volume <<endl;

    // обєм box 2
    volume = Box2.getVolume();
    cout << "Volume of Box2 : " << volume <<endl;

    // Додати два обєкти перевантеженим додаванням
    Box3 = Box1 + Box2;

    // Обєм результату 3
    volume = Box3.getVolume();
    cout << "Volume of Box3 : " << volume <<endl;

    return 0;
```

}
Результат роботи програми:
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400

Оператори які можна та неможна перевантажувати
Список операторів, які можна перевантажувати:
+ - * / % ^ & | ~ ! , =<> <= >= ++ -- << >> == != && || += -= /= %= ^= &= |= *= <<= >>= [] () -> ->* new new [] delete delete []
А ось оператори які не можна перевантажувати:
:: .* . ?:

Приклади перевантаження оператора

Ось різні приклади перевантаження оператора, які допоможуть вам зрозуміти концепцію.
Одинарні оператори працюють на одному операнді і наступні приклади операторів:

- Оператори інкременту (++) і декременту (-).
- Унарний мінус (-) оператор.
- Логічний оператор заперечення (!).

Одинарні оператори працюють на об'єкті, за який вони називалися і зазвичай, цей оператор з'являється на лівій стороні об'єкта, як !obj або obj--.
Наступний приклад пояснює, як мінус (-) оператор може бути перевантажений для префікса, а також постфіксного використання

```
#include <iostream>
using namespace std;

class Distance {
  private:
    int feet;           // Фути: 0 to infinite
    int inches;          // Дюйми: 0 to 12
  public:
    // конструктор 1
    Distance() {
      feet = 0;
      inches = 0;
    }
   // конструктор 2
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }
```

```
    // відображення відстані
    void displayDistance() {
      cout << "F: " << feet << " I:" << inches <<endl;
    }
    // перевантажений (overloaded) minus (-) оператор
    Distance operator- () {
      feet = -feet;
      inches = -inches;
      return Distance(feet, inches);
    }
};

int main() {
  Distance D1(11, 10), D2(-5, 11);
  -D1;                // застосуємо оператор -
  D1.displayDistance();   // покажемо D1

  -D2;                // застосуємо оператор -
  D2.displayDistance();   // покажемо D2
  return 0;
}
```

Результат роботи:
F: -11 I:-10
F: 5 I:-11

Бінарні оператори (Binary operators) приймають два аргументи. Частіше за все це оператори додавання (+) , віднімання (-) , множення(*) та ділення (/) operator. В даному випадку оператор (+) перевантажується. Аналогічно перевантажуються оператори (-) та ділення (/) .

```
#include <iostream>
using namespace std;

class Box {
  double length;     // Length of a box
  double breadth;    // Breadth of a box
  double height;     // Height of a box

  public:
  double getVolume(void) {
    return length * breadth * height;
  }
```

```
   void setLength( double len ) {
     length = len;
   }

   void setBreadth( double bre ) {
     breadth = bre;
   }

   void setHeight( double hei ) {
     height = hei;
   }

   // Перевантаження оператору додавання для обєктів Box
   Box operator+(const Box& b) {
     Box box;
     box.length = this->length + b.length;
     box.breadth = this->breadth + b.breadth;
     box.height = this->height + b.height;
     return box;
   }
};

int main() {
   Box Box1;
   Box Box2;
   Box Box3;
   double volume = 0.0;

   // box 1 ініціалізація
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 ініціалізація
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;
```

```
  // volume of box 2
  volume = Box2.getVolume();
  cout << "Volume of Box2 : " << volume <<endl;

  // Додати два обєкти класу:
  Box3 = Box1 + Box2;
   // volume of box 3
  volume = Box3.getVolume();
  cout << "Volume of Box3 : " << volume <<endl;
  return 0;
}
```

Результат роботи:

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Поліморфізм

Слово поліморфізм означає, що щось має багато форм. Як правило, поліморфізм виникає, коли існує ієрархія класів, і вони пов'язані успадкуванням.

Поліморфізм в C ++ означає, що виклик методу призведе до виконання іншої функції в залежності від типу об'єкта, який викликає цей метод.

Розглянемо наступний приклад, коли базовий клас був отриманий іншими двома класами

```
#include <iostream>
using namespace std;
 class Shape {
  protected:
    int width, height;
  public:
    Shape( int a = 0, int b = 0){
      width = a;
      height = b;
    }
    int area() {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
class Rectangle: public Shape {
  public:
    Rectangle( int a = 0, int b = 0):Shape(a, b) { }
```

```
    int area () {
      cout << "Rectangle class area :" <<endl;
      return (width * height);
    }
};

class Triangle: public Shape {
  public:
    Triangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};

int main() {
  Shape *shape;
  Rectangle rec(10,7);
  Triangle  tri(10,5);

  // беремо адресу Rectangle
  shape = &rec;
  // отримуємо його площу
  shape->area();

  // беремо адресу Triangle
  shape = &tri;
    // отримуємо його площу
  shape->area();
}
```

Результат:

```
Parent class area :
Parent class area :
```

Причина неправильного виведення полягає в тому, що виклик функції area () встановлюється компілятором один раз як версія, визначена в базовому класі. Це називається статичним дозволом виклику функції, або статичним зв'язком - виклик функції фіксується до виконання програми. Це також іноді називається раннім зв'язуванням, оскільки функція area () встановлюється під час компіляції програми.

Але тепер давайте зробимо невелику модифікацію в нашій програмі і передуємо декларації area () в класі Shape за допомогою ключового слова virtual, щоб вона виглядала так:

```
class Shape {
  protected:
    int width, height;

  public:
    Shape(int a = 0, int b = 0) {
      width = a;
      height = b;
    }

    // чисто віртуальна функція pure virtual function
    virtual int area() = 0;
};
```

Код = 0 повідомляє компілятору, що функція не має тіла, а вище віртуальна функція буде називатися чисто віртуальною функцією.

Абстрактні класи

Абстрактний клас, концептуально, клас, який не може бути створений і зазвичай реалізується як клас, який має одну або більше чистих віртуальних (абстрактних) функцій.

Чиста віртуальна функція є такою, яка повинна бути перевизначена будь-яким конкретним (тобто, не абстрактним) похідним класом. Це вказується в декларації з синтаксисом "= 0" у декларації функції-члена.

```
class AbstractClass {
public:
 virtual void AbstractMemberFunction() = 0; // Pure virtual function робить
                                  // цей клас Abstract class.
 virtual void NonAbstractMemberFunction1(); // Virtual function – віртуальна функція

 void NonAbstractMemberFunction2();
};
```

Взагалі абстрактний клас використовується для визначення реалізації і призначений для успадкування від конкретних класів. Це спосіб змусити зробити контракт між дизайном класу і користувачами цього класу. Якщо ми хочемо створити конкретний клас (клас, який може бути інстанційованим) з абстрактного класу, ми повинні оголосити і визначити відповідний метод класу для кожної абстрактної функції члена базового класу. В іншому випадку, якщо будь-яка функція-член базового класу залишається невизначеною, ми створимо новий абстрактний клас (іноді це може бути корисним).

Іноді ми використовуємо фразу "чистий абстрактний клас", що означає клас, який має виключно чисті віртуальні функції (і немає даних). Концепція інтерфейсу зіставляється з чистими абстрактними класами в C ++, оскільки в C ++ не існує "інтерфейсної" конструкції так само, як і в Java.

```
class Vehicle {
 public:
    explicit  Vehicle( int topSpeed ) : m_topSpeed( topSpeed ) {}
    int TopSpeed() const {
      return m_topSpeed;
    }

    virtual void Save( std::ostream& ) const = 0;

 private:
    int m_topSpeed;
 };

 class WheeledLandVehicle : public Vehicle {
 public:
    WheeledLandVehicle( int topSpeed, int numberOfWheels )
    : Vehicle( topSpeed ), m_numberOfWheels( numberOfWheels ) {}
    int NumberOfWheels() const {
      return m_numberOfWheels;
    }

    void Save( std::ostream& ) const; // цей метод implicitly virtual

 private:
    int m_numberOfWheels;
 };

 class TrackedLandVehicle : public Vehicle {
 public:
   TrackedLandVehicle ( int topSpeed, int numberOfTracks )
   : Vehicle( topSpeed ), m_numberOfTracks ( numberOfTracks )
   {}
   int NumberOfTracks() const {
     return m_numberOfTracks;
   }
   void Save( std::ostream& ) const; // is implicitly virtual
  private:
   int m_numberOfTracks;
  };
```

У цьому прикладі автомобіль є абстрактним базовим класом, оскільки він має абстрактний метод. Клас WheeledLandVehicle виводиться з базового класу. Він також містить дані, які є загальними для всіх колісних наземних транспортних

засобів, а саме кількість коліс. Клас TrackedLandVehicle - інший варіант класу Vehicle.
Це щось на зразок надуманого прикладу, але воно показує, як ви можете поділитися подробицями реалізації серед ієрархії класів. Кожен клас додатково уточнює концепцію. Це не завжди найкращий спосіб реалізації інтерфейсу, але в деяких випадках він працює дуже добре. В якості орієнтира, для зручності обслуговування і розуміння ви повинні спробувати обмежити наслідування не більш ніж на 3 рівнях. Найчастіше найкращим набором класів є чистий віртуальний абстрактний базовий клас для визначення загального інтерфейсу. Потім потрібно використовувати абстрактний клас для подальшого вдосконалення реалізації для набору конкретних класів і, нарешті, визначення набору конкретних класів.
Приклад використання чистого абстрактного класу

```
class DrawableObject {
 public:
  virtual void Draw(GraphicalDrawingBoard&) const = 0; //малює для GraphicalDrawingBoard
 };

 class Triangle : public DrawableObject {
 public:
  void Draw(GraphicalDrawingBoard&) const; //малює трикутник
 };

 class Rectangle : public DrawableObject {
 public:
  void Draw(GraphicalDrawingBoard&) const; //малює прямокутник
 };

 class Circle : public DrawableObject {
 public:
  void Draw(GraphicalDrawingBoard&) const; //малює коло
 };

 typedef std::list<DrawableObject*> DrawableList;

 DrawableList drawableList;
 GraphicalDrawingBoard drawingBoard;

 drawableList.pushback(new Triangle());
 drawableList.pushback(new Rectangle());
 drawableList.pushback(new Circle());
```

```
for(DrawableList::const_iterator iter = drawableList.begin(),
   endIter = drawableList.end();    iter != endIter;    ++iter) {
  DrawableObject *object = *iter;
  object->Draw(drawingBoard);
}
```

Майте на увазі, що об'єкти, що визивається, не є повністю визначеними (без конструкторів або даних), але вони повинні дати загальне уявлення про силу визначення інтерфейсу. Після того, як об'єкти побудовані, код, який викликає інтерфейс, не знає жодної деталі реалізації викликаних об'єктів, тільки ті з інтерфейсу. Об'єкт GraphicalDrawingBoard є заповнювачем, призначеним для представлення речі, на яку буде малюватися об'єкт, тобто відеопам'яті, буфері малювання, принтера.

Зауважимо, що існує велика спокуса додати конкретні функції-члени і дані до чистих абстрактних базових класів. Якщо таке трапиться, в цілому це знак того, що інтерфейс недостатньо продуманий. Дані і конкретні функції-члени, як правило, мають на увазі конкретну реалізацію і як такі можуть успадковуватися від інтерфейсу, але не повинні бути цим інтерфейсом. Навпаки, якщо існує деяка спільність між конкретними класами, добре робити створення абстрактного класу, який успадковує його інтерфейс від чистого абстрактного класу і визначає загальні дані і функції-функції конкретних класів. Необхідно взяти певну обережність, щоб вирішити, чи слід використовувати спадкування або агрегацію. Занадто багато шарів успадкування можуть ускладнити обслуговування та використання класу. Як правило, максимально допустимі шари успадкування становлять близько 3, якщо вище, зазвичай потрібен рефакторинг класів. Загальним тестом є "є" проти "має", так як на площі є прямокутник, але площа має набір сторін.

Поліморфізм (перевизначення) методів

Таким чином можна додавати нові дані і функції до класу через успадкування. Але як бути, якщо ми хочемо, щоб наш похідний клас успадкував метод від базового класу, але мати іншу реалізацію для нього? Саме тоді мова йде про поліморфізм, фундаментальне поняття в програмуванні ООП. Поліморфізм поділяється на два поняття: статичний поліморфізм і динамічний поліморфізм. Динамічний поліморфізм застосовується в C ++, коли похідний клас замінює функцію, оголошену в базовому класі.

Ми реалізуємо цю концепцію, перевизначивши метод у похідному класі. Але для цього потрібно ввести поняття ***динамічного зв'язування***, ***статичного зв'язування*** і ***віртуальних методів***.

Припустимо, що ми маємо два класи, A і B. B є наслідником A і перевизначає реалізацію методу c (), що знаходиться в класі A. Тепер припустимо, що ми маємо об'єкт b класу B. Як інтерпретувати команду b.c ()?

Якщо b оголошений у стеку (не оголошений як покажчик або посилання), компілятор застосовує статичну прив'язку, це означає, що він інтерпретує (під час компіляції), що ми посилаємося на реалізацію c(), що знаходиться в B.

Однак, якщо ми оголошуємо b як вказівник або посилання класу A, компілятор не може знати, який метод викликати під час компіляції, оскільки b може бути типу A або B. Якщо це зроблено під час виконання, викличеться метод, який буде розташовано в B. Це називається **динамічним зв'язуванням**.
Якщо це буде вирішено під час компіляції, буде викликаний метод, який знаходиться в A. Це **статичне звязування**.

Віртуальні методи

Віртуальні методи є відносно простими, але їх часто неправильно розуміють. Концепція є невід'ємною частиною розробки ієрархії класів щодо класів підкласів, оскільки вона визначає поведінку перевизначених методів у певних контекстах.
Віртуальними методами є функції членів класу, які ***можуть бути перевизначені в будь-якому класі***, отриманому від того, де вони були оголошені. Тіло методу потім замінюється новим набором реалізації в похідному класі.
Примітка:При перевизначенні віртуальних функцій можна змінювати приватний, захищений або відкритий стан доступу до стану методу похідного класу.
Розміщуючи ключове слово virtual перед декларацією методу, ми вказуємо, що коли компілятор має вирішити між застосуванням статичного зв'язування або динамічного зв'язування, він застосує динамічну прив'язку. В іншому випадку буде застосована статичне прив'язка.
Примітка:Хоча в нашому підкласі не потрібно використовувати ключове слово virtual (оскільки функція базового класу є віртуальною, всі заміщення підкласу з неї також будуть віртуальними), це хороший стиль для створення коду для майбутнього використання (для використання поза проекту).
Знову ж таки, це має бути зрозумілішим з прикладом:

```
class Foo{
public:
  void f()  {
    std::cout << "Foo::f()" << std::endl;
  }
  virtual void g()  {
    std::cout << "Foo::g()" << std::endl;
  }
};

class Bar : public Foo{
public:
  void f()  {
    std::cout << "Bar::f()" << std::endl;
  }
  virtual void g()  {
```

```
    std::cout << "Bar::g()" << std::endl;
  }
};

int main(){
  Foo foo;
  Bar bar;

  Foo *baz = &bar;
  Bar *quux = &bar;

  foo.f(); // "Foo::f()"
  foo.g(); // "Foo::g()"

  bar.f(); // "Bar::f()"
  bar.g(); // "Bar::g()"
  // А тепер ....
  baz->f();  // "Foo::f()"
  baz->g();  // "Bar::g()"
  quux->f(); // "Bar::f()"
  quux->g(); // "Bar::g()"
  return 0;
}
```

Наші перші виклики до f () і g () на двох об'єктах прості. Проте цікаві речі з нашим baz pointer, який є вказівником на тип Foo.

f () не є віртуальним і як такий виклик до f () завжди буде викликати реалізацію, пов'язану з типом покажчика, у цьому випадку реалізація з Foo.

Примітка: Пам'ятайте, що перевантаження та перевизначення є різними поняттями.

Виклики віртуальних функцій обчислювально дорожче, ніж звичайні виклики функцій. Віртуальні функції використовують вказівники на вказівки від функцій і вимагають декількох додаткових інструкцій, ніж звичайні функції члена. Вони також вимагають, щоб конструктор будь-якого класу / структури, що містить віртуальні функції, ініціалізував таблицю покажчиків на його віртуальні методи. Всі ці характеристики означатимуть компроміс між продуктивністю та дизайном. Слід уникати попереднього оголошення функцій віртуальними без існуючих структурних потреб. Майте на увазі, що віртуальні функції, які вирішуються лише під час виконання, не можуть бути вбудовані (inline).

Примітка: Деякі потреби у використанні віртуальних функцій можна вирішити за допомогою шаблонів класу.

Чистий віртуальний метод

Є ще одна цікава можливість. Іноді ми взагалі не хочемо забезпечувати реалізацію нашої функції, але хочемо вимагати від людей, які підкласують наш клас, забезпечити реалізацію самостійно. Це називається **чистими віртуальними методами**.

Для позначення чистої віртуальної функції замість реалізації ми просто додаємо "= 0" після оголошення функції.

Коваріантні типи повернення

Коваріантні типи повернення - це можливість для віртуальної функції в похідному класі повернути вказівник або посилання на примірник себе, якщо версія методу в базовому класі робить це.

Приклад.

```
class base{
public:
  virtual base* create() const;
};

class derived : public base{
public:
  virtual derived* create() const;
};
```

Це дозволяє уникнути перетворення типів.

Примітка:Деякі старі компілятори не мають підтримки для коваріантних типів повернення. Для таких компіляторів існують обхідні шляхи.

Віртуальні конструктори

Існує ієрархія класів з базовим класом Foo. Враховуючи панель об'єктів, що входить до ієрархії, бажано виконати наступне:

1. Створіть об'єкт baz того ж класу, що і бар (скажімо, Bar Bar), ініціалізований за допомогою конструктора за замовчуванням класу. Зазвичай використовується синтаксис:

```
Bar * baz = bar.create ();
```

2. Створіть об'єкт baz того ж класу, що і бар, який є копією панелі. Зазвичай використовується синтаксис:

```
Bar * baz = bar.clone ();
```

У класі Foo, методи Foo :: create () і Foo :: clone () оголошуються наступним чином:

```
class Foo{
   // ...
   public:
      // Virtual default constructor: віртуальний конструктор за замовченням
      virtual Foo* create() const;
```

```
    // Virtual copy constructor: віртуальний конструктор копії за замовченням
    virtual Foo* clone() const;
};
```

Якщо потрібно використати Foo як абстрактний клас, методи потрібно зробити чисто віртуальними (pure virtual):

```
class Foo{
  // Віртуальні методи...
   public:
    virtual Foo* create() const = 0;
    virtual Foo* clone() const = 0;
};
```

Для того щоб забезпечити створення ініціалізованого за замовченням об’єкту та створення копій об’єкту кожен клас Bar в ієрархії повинен мати публічний конструктор за замовченням (public default) та конструктор копіювання (copy constructor). Віртуальні конструктори Bar визначені наступним чином:

```
class Bar : ... // Bar наслідник Foo
{
  // ...

  public:
  // Non-virtual default constructor: : невіртуальний конструктор за замовченням
  Bar ();
  //Non-virtual copy constructor: невіртуальний конструктор копії за замовченням
  Bar (const Bar&);

  // Virtual default constructor, inline implementation
  Bar* create() const { return new Foo (); }
  // Virtual copy constructor, inline implementation
  Bar* clone() const { return new Foo (*this); }
};
```

Наведений вище код використовує коваріантні типи повернення. Якщо ваш компілятор не підтримує Bar * Bar :: create (), використовуйте замість Foo * Bar :: create () і аналогічно для clone ().

Під час використання цих віртуальних конструкторів необхідно вручну звільнити об'єкт, створений за допомогою виклику delete baz ;.

Цю неприємність можна уникнути, якщо так званий «розумний покажчик» (наприклад, std :: unique_ptr <Foo>) використовується у зворотному типі замість простого Foo *.

Пам'ятайте, що якщо Foo використовує динамічно виділену пам'ять, ви повинні визначити віртуальний деструктор ~ Foo () і зробити його віртуальним, щоб

подбати про звільнення об'єктів за допомогою покажчиків до батьківських класів.

Віртуальний деструктор

Особливо важливо пам'ятати про визначення віртуального деструктора, навіть якщо він порожній у будь-якому базовому класі, оскільки помилка в ньому може створити проблеми з деструктором, створеним за замовчуванням, який не буде віртуальним.

Віртуальний деструктор не перевизначається при перевизначенні в похідному класі, визначення для кожного деструктора є накопичувальними, і вони починаються з останнього похідного класу до першого базового класу.

Чистий віртуальний деструктор

Кожен абстрактний клас повинен містити декларацію чистого віртуального деструктора.

Чисті віртуальні деструктори - це особливий випадок чистих віртуальних функцій (призначених для перевизначення у похідному класі). Вони завжди повинні бути визначені, і це визначення завжди має бути порожнім.

```
class Interface {
public:
  virtual ~Interface() = 0; //декларація чистого віртуального деструктора
};

Interface::~Interface(){} //визначення чистого віртуального деструктору (завжди повинно бути порожнім)
```

Властивість підпорядкування

Підпорядкування - це властивість, що всі об'єкти, які знаходяться в ієрархії класів, повинні виконувати ті самі методи, при цьому об'єкт базового класу може бути замінений об'єктом, що походить від нього (прямо чи опосередковано).

Приклад. Всі ссавці є тваринами (вони походять від них), і всі кішки є ссавцями. Тому, через властивість підпорядкування ми можемо «лікувати» будь-якого ссавця як тварину, а будь-яку кішку як ссавця. Це означає абстракцію, тому що, коли ми "лікуємо" ссавця як тварину, єдина інформація, яку ми повинні знати про це, що вона живе, вона зростає, і т.д., але нічого не стосується саме ссавців.

Це властивість застосовується в C ++, коли ми використовуємо покажчики або посилання на об'єкти, які знаходяться в ієрархії класів. Іншими словами, покажчик класу тварин може вказувати на об'єкт класу тварин, ссавців або кішок.

Давайте продовжимо наш приклад:

```
//перерахування: перелік відповідних типів
  enum AnimalType {
```

```
    Herbivore,
    Carnivore,
    Omnivore,
};

class Animal {
    public:
        AnimalType Type;
        bool bIsAlive;
        int iNumberOfChildren;
};

class Mammal : public Animal{
    public:
        int iNumberOfTeats;
};

class Cat : public Mammal{
    public:
        bool bLikesFish;  // скоріше за все true
};

int main() {
   Animal* pA1 = new Animal;
   Animal* pA2 = new Mammal;
   Animal* pA3 = new Cat;
   Mammal* pM  = new Cat;

   pA2->bIsAlive = true;     // Correct
   pA2->Type = Herbivore;    // Correct
   pM->iNumberOfTeats = 2;   // Correct

   pA2->iNumberOfTeats = 6;  // Incorrect
   pA3->bLikesFish = true;   // Incorrect

   Cat* pC = (Cat*)pA3;      // Downcast, коректно (але не дуже гарно)
   pC->bLikesFish = false;   // Коректно (хоча й дивно)
}
```

В останніх рядках прикладу є перетворення вказівника на Animal до вказівника на Cat. Це називається перетворення вниз("Downcast"). Downcasts є корисними і повинні бути використані, але спочатку ми повинні переконатися, що об'єкт, який ми перетворюємо, дійсно є типом, яким ми його передаємо. Перетворення базового класу до незв'язаного класу є помилкою. Щоб вирішити цю проблему, слід використовувати оператори перетворення dynamic_cast <> або static_cast

<>. Вони правильно відправляють об'єкт з одного класу в інший і викидають виняток, якщо типи класів не пов'язані між собою. напр. Якщо ви спробуєте:
Cat* pC = new Cat;
motorbike* pM = dynamic_cast<motorbike*>(pC);
Потім програма кине виняток, оскільки кішка не є мотоциклом. Static_cast дуже схожий, тільки він виконує перевірку типу під час компіляції. Якщо у вас є об'єкт, в якому ви не впевнені в його типі, ви повинні використовувати dynamic_cast і бути готовими до обробки помилок під час перетворення. Якщо ви прибираєте об'єкти, де знаєте типи, ви повинні використовувати static_cast. Не використовуйте старі стилі C перетворень, оскільки вони просто дадуть вам помилку доступу, якщо дані типи не пов'язані між собою.

Локальні(анонімні) класи

Локальний клас - це будь-який клас, який визначений у певному блоці оператора, в локальній області, наприклад, усередині функції. Це робиться так, як визначається будь-який інший клас, але локальні класи не можуть звертатися до нестатичних локальних змінних або використовуватись для визначення статичних членів даних. Ці типи класів корисні, особливо в шаблонних функціях, які будуть розглядатися пізніше.

```
void MyFunction(){
  class LocalClass
  {
  // ... members definitions ...
  };

  // ... any code that needs the class ...

}
```

Керування перетворенням типів

Ми вже розглядали автоматичне перетворення типів (неявне перетворення) і згадували, що деякі з них можуть бути визначені користувачем.
Визначене користувачем перетворення з класу в інший клас можна виконати, надавши конструктор у цільовому класі, який приймає клас джерела як аргумент, Target (const Source & a_Class) або надаючи цільовий клас оператором перетворення, як оператор Source().
Іноді потрібно запобігати копіюванню класів. Це потрібно, наприклад, для запобігання проблем, пов'язаних з пам'яттю, які призведуть до того, що конструктор копіювання за замовчуванням або оператор присвоєння за замовчуванням ненавмисно застосований до класу C, який використовує динамічно виділену пам'ять, де конструктор копіювання та оператор присвоєння, ймовірно, перевищують наявні ресурси.
Деякі вказівки щодо стилю пропонують, щоб усі класи не могли копіювати за замовчуванням, а лише коструктором копіювання, якщо це має сенс. Інші (погані) рекомендації говорять про те, що ви завжди повинні чітко писати конструктор копіювання та оператор призначення копій; це насправді погана ідея, оскільки вона додає до зусиль з технічного обслуговування, додає до роботи для читання класу, робить більше помилок, ніж при використанні неявно оголошених конструкторів, і не має сенсу для більшості типів об'єктів. Розумним керівництвом є думати про те, чи копіювання має сенс для типу; якщо це так, то спочатку спробуйте організувати, щоб операції копіювання, створені компілятором, працювали

правильно (наприклад, утримуючи всі ресурси за допомогою класів керування ресурсами, а не через вказівники та посилання), а якщо це не розумно, дотримуйтесь закону трьох. Якщо копіювання не має сенсу, його можна заборонити як показано нижче.
Просто задекларуйте конструктор копіювання і оператор присвоювання, і зробить їх приватними. Не визначайте їх. Оскільки вони не є захищеними або публічними, вони недоступні за межами класу. Використання їх у класі дасть помилку лінкера, оскільки вони не визначені.

```
class C
{
  ...

  private:
    // Not defined anywhere
    C (const C&);
    C& operator= (const C&);
};
```

Пам'ятайте, що якщо клас використовує динамічно виділену пам'ять для членів даних, необхідно визначити процедури випуску пам'яті в деструкторі ~ C (), щоб звільнити виділену пам'ять.
Клас, який оголошує лише ці дві функції, можна використовувати як приватний базовий клас, так що всі класи, які приватно успадковують такий клас, заборонять копіювання.

Лекція 9: Клас рядків

Для роботи з рядками на Сі++ (починаючи з C++98) створений спеціальний клас string, який ще звуть клас роботи з рядком стандартної бібліотеки(STL).
Для порівняння роботи роботи з рядками стандартних ANSI-рядків або рядків, що закінчуються нулевим символом та роботи з рядком класу string розглянемо роботу цих класів на прикладі:

Робота з ANSI-рядком	Робота з класом string

#include <iostream>
 	
#include <cstring> // Required by strcpy()
#include <cstdlib> // Required by malloc()
 	
int main(int argc, char **argv)
{
  char CC[17];// C character string (16 characters + NULL termination)
  char *CC2; // C character string. No storage allocated.
 	
  strcpy(CC,"This is a string");
  CC2 = (char *) malloc(17); // Allocate memory for storage of string.
	#include <iostream>
	#include <string>
	
	int main(int argc, char **argv)
	{
	  std::string SS; // C++ STL string
	  std::string SS2; // C++ STL string
	
	  SS = "This is a string";
	  SS2 = SS;
	
	  std::cout << SS << endl;
	  std::cout << SS2 << endl;
	}

```
    strcpy(CC2, CC);//strcpy(CC2,"This is a string");

    cout << CC << endl;
    cout << CC2 << endl;
}
```

Компіляція: `g++ stringtest.cpp -o stringtest`
Запуск: `./stringtest`
Результат робти обох програм повинен бути ідентичним:

```
This is a string
This is a string
```

Можна побачити, що робота з рядками в С-силі та в стилі С ++ є одночасно допустимими, але неважко побачити, що клас роботи з рядками С ++ забезпечує більше функціональності та зручності. Рядок STL не вимагає попередньо виділеної пам'яті або виділення вручну, значно легше робиться присвоювання та ініціалізація таких рядків.
Клас рядків STL також надає багато корисних методів роботи з рядками.

Для створення, тобто декларації та ініціалізації рядків STL, очевидно потрібно підключити модуль реалізації цього класу #include <string> та викликати конструктор для цього класу.
Для класу string існують наступні варіанти конструкторів:

1. Конструктор по літералу:
string <ім'я_змінної>(<Літерал>);
Приклад:
```
    string x("Literal");
    string sVar1("Програмування на Сі++");
```
2. Конструктор по С-рядку:
char <ім'я_с_рядку>[] ; // Null terminated char
*** /// ім'я_с_рядку ініціалізуєтсья
string <ім'я_змінної>(<ім'я_с_рядку>);
Приклад.
```
    char  cVar[10]= "Hello!";
    string sVar2(cVar);
```

3. Конструктор з декількох симовлів:
string <ім'я_змінної>(<кількість_символів>, <Символ>);

```
string sVar3(5, 'a');              // 5 символів 'a': "aaaaa"
```

4. Конструктор з іншого рядку та початкового символу:
string <ім'я_змінної>(<ім'я_іншого_рядку>, <початок_рядку>);

```
string Var1 ("String Constructor");
string sVar4(Var1, 8);  // Ініціалізує рядок з 8-го символу до кінця рядка  Var1: "Constructor"
```

5. Конструктор за допомогою початкового та кінцевого ітератору іншого рядку STL:
```
 string sVar5(Var1.begin(), Var1.end());
```

Крім того, звичайним чином визначений конструктор копіювання:

string <ім'я_змінної>(<ім'я_іншого_рядку>);

Аналогічно для цього класу існує звичайний деструктор:
~string();

Функція/Операція	**Опис**
Var = *string2* Var.assign("*string-to-assign*")	Присвоєння string (operator=). При присвоюванні С "char*" типу, перевірте наявність NULL для запобагання прблемам (failure/crash). Наприклад, `if( szVar ) sVar.assign( szVar );` де szVar змінна типу С "char *" та sVar типу "string".
Var.swap(*string2*) swap(*string1,string2*)	Метод міняє місцями поточний клас та string2. Функція swap міняє значення аргументів
Var += *string2* Var.append() Var.push_back()	Додає string/символи до кінця рядку(конкатинація). Приклад, string `var("abc");` виклик `var.append("xyz")` створить рядок "abcxyz". Метод push_back() бере у якості аргменту лише символ (char)
Var.insert(size_t position, string) Var.insert(size_t position, char *) Var.insert(size_t position, string, size_t pos1, size_t len) Var.insert(size_t position, char *, size_t pos1, size_t len)	Вставляє рядок на дане місце position: вставляє перед данним місцем. Якщо він дорівнює 0, то вставляє перед рядком pos1: номер позиції першого символу рядку що вставляється len: довжина рядку, що вставляється
Var.erase() Var = ""	Очщує вмість рядку. аргументи на потрібні
+	Конкатинація
==, !=, <, <=, >, >=	Порівнює рядки
Var.compare(*string*) Var.compare(size_t pos1, size_t len, *string*) const; Var.compare(size_t pos1, size_t len1, const *string*, size_t pos2, size_t len2) const;	Порівнює рядки в С-стилі. Повертає int: • 0: якщо рівні. • -1: Не рівні. 1-ше не співпадаюче значення в Var менше ніж у рядку що порівнюється по ASCII таблці. • +1: Не рівні. 1-ше не співпадаюче значення в Var більше ніж у рядку що порівнюється по ASCII таблці. Тут *string* інший STL рядок або null terminated C рядок.
Var.length() Var.size() Var.capacity()	Повертає довжину памяті що зберігає рядок. Методи `length(), size()` та `capacity()` дозволяють отрмати довжину рядку.
Var.max_size()	Повертає максимальний розмір рядку
Var.empty()	Повертає 1 якщо рядок порожній Повретає 0 якщо не порожній.
<<	Виводить в поток виводу
>> getline()	Вводить з потоку вводу
Var.c_str()	Конвертує рядок у С-рядок (що закінчується нулем). Не

Функція/Операція	Опис
	треба звільнювати цей результат!!!
Var.data()	онвертує рядок у С-рядок, який **незакінчується** нулем! Не треба звільнювати цей результат!!!
Var[] Var.at(*integer*)	Квадратні дужки отримують доступ до елементів рядку. Повертає символ на даній позиції (починаючи з 0) . Приклад, `Var("abc");тоді` Var[2] == "c".
Var.copy(char *str,size_t len, size_t index)	str: виділенний буфер для масиву char куди робиться копія len: кількість символів що потрібно зкопіювати index: стартова позиція в рядку (Var) звідки треба копіювати . Відлік починаєтсья з 0 Повертає кількість зкопійований символів
Var.find(*string*) Var.find(*string, positionFirstChar*) Var.find(*string, positionFirstChar, len*)	Знаходить індекс першої появи підряку в рядку. Повертає спеціальне беззнакове ціле. *positionFirstChar - звідки починати пошук в Var* `len` - довжина рядку який потрібно шукати (*string*) Якщо не знайдений підрядок повертає спецконстанту `string::npos` . Приклад `if(Var.find("abc") == string::npos) cout << "Not found" << endl;`
Var.rfind()	Знаходить індекс останньої появи підряку в рядку.
Var.find_first_of(*string*, size_t position) Var.find_first_of(char *str, size_t position) Var.find_first_of(char *str, size_t position, size_t len)	Находить рядки та підрядки тут *string* - STL string `str` - C string. Якщо `position` = 0, починає з початку рядку
Var.find_last_of(*string*, size_t position) Var.find_last_of(char *str, size_t position) Var.find_last_of(char *str, size_t position, size_t len)	Находить рядки та підрядки з кінця. position: початок (тобто в даному рядку кінець рядку пошуку) len: кількість символів що треба знайти
Var.find_first_not_of() Var.find_last_not_of()	
Var.replace(pos1, len1, *string*) Var.replace(itterator1, itterator2, const *string*) Var.replace(pos1, len1, *string*, pos2, len2)	Замінює підрядок новими символами. pos2 та len2 потрібно лише якщо міняється лише підрядок *string*. *string* - інший STL string або С-рядок.
Var.substr(pos, len)	Повретає підрядок тексту від стартової позиції (pos) в рядку та довжині потрібного підрядку.
Var.begin() Var.end()	Ітератори
Var.rbegin() Var.rend()	Обернені ітератори

Iterator types:

- ❿ string::traits_type

- string::value_type
- string::size_type
- string::difference_type
- string::reference
- string::const_reference
- string::pointer
- string::const_pointer
- string::iterator
- string::const_iterator
- string::reverse_iterator
- string::const_reverse_iterator
- string::npos

Інколи потрібно перевести з класу STL рядок в С-рядок. Для цього можна викликати метод

c_str():

string sVar("GGGGG");

char* x = svar.c_str();

Відповідно, для переводу STL рядку в чиловий тип можна скористатись функціями переводу у число С-рядків:

string sVar1("LL1234");

```
int num = stol ( sVar1.c_str(), nullptr);
int num = atoi( sVar1.c_str() );
//if(sscanf(sVar1.c_str(), "%d", &i) != 1)
```

або починаючи з С++11:

```
int num =std::stoi( sVar1 )
```

```
    // object from the class stringstream
    stringstream tmp(s);

    int num = 0;
    tmp >> num;
```

Для переводу числа в тип(клас) STL рядок починаючи з С++11 можна скористатись

res = to_string(number);

або скориставшись класом ostringstream з модулю sstream

#include <sstream> // ostringstream

string int2string1(const int& number){

  ostringstream oss;

```
    oss << number;
    return oss.str();
}
string res =  int2string1(number);
```

або функцією sprintf :

```
   const size_t size=255;
   char text_num[size];
    sprintf(text_num, "%d", number); // Could have printed directly using printf()
    string res1(text_num);
```

Приклади використання функцій рядку:

```
    string a("Preved Medved");
    string b{" i vsem privet"};
    string c;
    cout << a << " " << b << endl; // Output: abcd efg xyz ijk
    cout << "String empty: "<<boolalpha<<c.empty()<< endl;
    // String empty: 1  -  Yes it is empty. (TRUE)
    c = a + b; // concatenation
    cout << c << endl; // abcd efgxyz ijk
    cout <<"String length:"<<c.length()<< endl;
    // String length: 15
    cout << "String size: "<<c.size()<< endl;
    // String size: 15
    cout << "String capacity: "<< c.capacity() << endl;
    // String capacity: 15
    cout << "String empty: "<<c.empty()<<endl;
    // String empty: 0
    // Is string empty? No it is NOT empty. (FALSE)
    string d = c;
```

```
cout << d << endl; // abcd efgxyz ij
cout << "First character: " << c[0] << endl;// First character: a
// Strings start with index 0 just like C.
string f("    Leading and trailing blanks      ");
cout << "String f:" << f << endl;
cout << "String length: " << f.length() << endl;
// String length: 37
cout << "String f:"<< f.append("ZZZ")<< endl;
// String f:    Leading and trailing blanks      ZZZ
cout << "String length: " << f.length() << endl;
// String length: 40
string g("abc abc abd abc");
cout << "String g: " << g << endl;
// String g: abc abc abd abc
cout << "Replace 12,1,\"xyz\",3: " << g.replace(12,1,"xyz",3) << endl;
// Replace 12,1,"xyz",3: abc abc abd xyzbc
cout << g.replace(0,3,"xyz",3) << endl;
// xyz abc abd xyzbc
cout << g.replace(4,3,"xyz",3) << endl;
// xyz xyz abd xyzbc
cout << g.replace(4,3,"ijk",1) << endl;
// xyz i abd xyzbc
cout << "Find: " << g.find("abd",1) << endl;
// Find: 6
cout << g.find("qrs",1) << endl;
string h("abc abc abd abc");
cout << "String h: " << h << endl;
cout << "Find \"abc\",0: " << h.find("abc",0) << endl; // Find "abc",0: 0
cout << "Find \"abc\",1: " << h.find("abc",1) << endl; // Find "abc",1: 4
cout << "Find_first_of \"abc\",0: " << h.find_first_of("abc",0) << endl; // Find_first_of "abc",0: 0
cout << "Find_last_of \"abc\",0: " << h.find_last_of("abc",0) << endl;
// Find_last_of "abc",0: 0
```

```
    cout << "Find_first_not_of \"abc\",0: " << h.find_first_not_of("abc",0) << endl;
    // Find_first_not_of "abc",0: 3
    cout << "Find_first_not_of \" \": " << h.find_first_not_of(" ") << endl;
    // Find_first_not_of " ": 0
    cout << "Substr 5,9: " << h.substr(5,9) << endl;
    // Substr 5,9: bc abd ab
    cout << "Compare 0,3,\"abc\": " << h.compare(0,3,"abc") << endl;
    // Compare 0,3,"abc": 0
    cout << "Compare 0,3,\"abd\": " << h.compare(0,3,"abd") << endl;
    // Compare 0,3,"abd": -1
    cout << h.assign("xyz",0,3) << endl;
    // xyz
    cout << "First character: " << h[0] << endl; // Strings start with 0 // First character: x
```

Результат роботи:

```
Preved Medved  i vsem privet
String empty: true
Preved Medved i vsem privet
String length:27
String size: 27
String capacity: 30
String empty: false
Preved Medved i vsem privet
First character: P
String f:    Leading and trailing blanks
String length: 45
String f:    Leading and trailing blanks      ZZZ
String length: 48
String g: abc abc abd abc
Replace 12,1,"xyz",3: abc abc abd xyzbc
xyz abc abd xyzbc
xyz xyz abd xyzbc
```

```
xyz i abd xyzbc
Find: 6
18446744073709551615
String h: abc abc abd abc
Find "abc",0: 0
Find "abc",1: 4
Find_first_of "abc",0: 0
Find_last_of "abc",0: 0
Find_first_not_of "abc",0: 3
Find_first_not_of " ": 0
Substr 5,9: bc abd ab
Compare 0,3,"abc": 0
Compare 0,3,"abd": -1
xyz
First character: x
```

Рецепт - Парсінг файлу

```
int fun2()
{
   string::size_type posBeginIdx, posEndIdx;
   string::size_type ipos=0;
   string sLine, sValue;
   string sKeyWord;
   const string sDelim(" ");
   string sError;

   ifstream myInputFile(SYS_CONFIG_FILE, ios::in);
   if(!myInputFile){
      sError = "File SYS_CONFIG_FILE could not be opened";
      return -1;// ERROR
   }
   cerr<<"Line";
```

```
    while(getline(myInputFile, sLine)){

        cerr<<sLine;
        if(myInputFile.bad()) break;
        if( !sLine.empty()){
            posEndIdx = sLine.find_first_of(sDelim);
            if(posEndIdx==string::npos) break;
            sKeyWord = sLine.substr( ipos, posEndIdx ); // Extract word
            posBeginIdx = posEndIdx + 1; // Beginning of next word (after ':')
            cout<<posBeginIdx<<sKeyWord<<posEndIdx;
            ipos++;
            }
        }
    return ipos;
}
```