МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЦЕНТРАЛЬНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Механіко-технологічний факультет
Кафедра програмування та захисту інформації

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт з навчальної дисципліни “Об’єктно-орієнтоване програмування” для студентів очної форми навчання за спеціальностями 123 “Комп’ютерна інженерія”, 125 “Кібербезпека”, 6.050102 “Комп’ютерна інженерія”, 6.170103 «Управління інформаційною безпекою»

ЗАТВЕРДЖЕНО
на засіданні кафедри програмування та захисту інформації,
протокол від 21 грудня 2016 року № 11

КРОПИВНИЦЬКИЙ
2016

Методичні вказівки до виконання лабораторних робіт з навчальної дисципліни “об’єктно-орієнтоване програмування” для студентів очної форми навчання за спеціальностями 123 “Комп’ютерна інженерія”, 6.050102 “Комп’ютерна інженерія”, 125 “Кібербезпека”, 6.170103 «Управління інформаційною безпекою»

/ уклад. О. Л. Левошко. — Кропивницький: ЦНТУ, 2016. — 68 с.

Укладач: Левошко О. Л.

Рецензенти: Смірнов О. А., д-р техн. наук, професор;
Дрєєв О. М., канд. техн. наук.

Вступ

1 Особливості C++

У цілому мова C++ є надмножиною мови Сі. Це означає, що можна компілювати програми Сі у середовищі C++, однак компілювати програми C++ у середовищі Сі при наявності в них будь-яких специфічних для C++ конструкцій не можна. Деякі ситуації вимагають спеціальної уваги. Наприклад, одна і та ж сама функція func(), що двічі оголошена в Сі з різними значеннями аргументів, викличе помилку повторення імен. Однак, у C++ func() інтерпретується як перевизначена функція - а те, допустиме це чи ні, залежить від інших обставин.

2 Побудова методичних вказівок

Хоча C++ вводить нові ключові слова та операції для роботи з класами, деякі засоби C++ застосовувані поза контексту класів. Спочатку розглядаються саме ці, використовувані незалежно від класів засоби, а після цього - специфіка роботи з класами та пов'язаними з ними механізмами.

Дані методичні вказівки вмістять у собі 7 лабораторних робіт за наступними темами, які поділені між двома рубіжами:

Лабораторні заняття (рубіж 1)	**Лабораторне заняття 1.** Необ’єктні властивості C++. ЗМ 1-2. При виконанні студенти порівнюють мови програмування С та C++ на основі вивчення операторів cin>>, cout<<, використання коментарів та вивчають операції керування пам’яттю new, delete; перевантаження імен функцій; операції доступу до області бачення; вбудовані функції; аргументи функцій за замовчуванням.
	Лабораторне заняття 2. Програмування класів. Створення об’єктів. Конструктори та деструктори. ЗМ 3. При виконанні студенти вивчають Визначення ключових абстракцій та ознаки складної системи, створюючи приклад абстракції - класи, що містять дані-члени та функції-члени з використанням обмеження доступу до членів класу: специфікаторів public, private, protected. ЗМ 4. При виконанні студенти вивчають взаємозв'язок між класами та об'єктами, а також роль класів та об'єктів в процесі проектування, використовуючи конструктори за замовчуванням та параметризовані конструктори для ініціалізації класу. Створюються глобальні і локальні об’єкти класу. Вивчається індивідуальність об'єктів, час існування об'єктів та операції копіювання об'єктів з використанням конструктора копії.
	Лабораторне заняття 3. Успадкування.

	ЗМ 5-6. При виконанні студенти вивчають приклади ієрархії, відношення між об'єктами, а саме одиночну та множинну спадкоємність, правила виклику конструкторів та деструкторів базових та похідних класів, керування доступом до компонентів класу, віртуальні деструктори.
Лабораторні заняття(рубіж 2)	**Лабораторне заняття 4.** Дружні класи та функції (friend). Правила доступу до членів класу. Поліморфізм. ЗМ 5-7. При виконанні студенти вивчають простий поліморфізм та процес виклику віртуальної функції. Вивчаються спадкоємність та типізація, віртуальні базові класи, абстрактні класи, чисті віртуальні функції.
	Лабораторне заняття 5. Перевантаження операцій. ЗМ 8. При виконанні студенти вивчають функціональну семантику бінарних та унарних операцій, перетворення типів, перевантаження арифметичних операцій, операторів індексування массиву, виклику функції, керування пам'яттю та ін. Вивчаються правила перевантаження операторів.
	Лабораторне заняття 6. Шаблони. Потоки C++. ЗМ 9. При виконанні студенти вивчають методику написання програм, що реалізують родові списки, надійні за типами та визначають якість створеної абстракції. ЗМ 10. При виконанні студенти вивчають засоби реалізації рядково-орієнтованого та файлового введення/виведення та форматування виведення.
	Лабораторне заняття 7. Виключні ситуації.Заміна фукцій unexpected() і terminate(). ЗМ 11. При виконанні студенти вивчають механізм для повідомлень і вживання заходів у випадку виникнення умови виключних ситуацій, а також правила для написання власного коду обробки виключних ситуацій.

Використана література:

1. Майкл И. Хаймен. BORLAND C++ для “ Чайников”. Киев “Диалектика”. 1995 г.
2. C++ Язык программирования. Москва. “И.В.К.- СОФТ”. 1991 г.
3. Д. Рассохин. “От С к С++” . Москва. “Едель” 1993 г.
4. Я. Белецкий. “ Турбо Си++. Новая разработка”. Москва. “Машиностроение”.1994г
5. Справочник по классам BORLAND C++ 3.1/4.Киев. “Диалектика” 1994 г.
6. Ахо А., Хопкрофт Д., Ульман Д. Структуры даннях и алгоритмы. Пер. с англ. Москва. «Вильямс». 2009. 400 с.
7. Керниган Б., Ритчи Д. Язык программирования Си. Пер. с англ. Москва. «Вильямс». 2009. 304 с.
8. Д. Рассохин. “От С к С++” . Москва. “Едель” 1993. 167 с.
9. Кормен Т. и др. Алгоритмы: построение и анализ. Пер. с англ. Москва. «Вильямс». 2011. 1296 с.
10. Р.Вайнер, Л. Пинсон “ C++ изнутри”. Киев. НПИФ “ДиаСофт”. 1993. 283 с.
11. З.Я. Шпак.Програмування мовою С. Львів. Видавництво львівської політехніки. 2011. 436с.
12. Т.Сван. Освоение BORLAND C++ 4.5. Практический курс. Киев “Диалектика”. 1996. 543 с.
13. Т.Сван. Программирование для Windows в Borland C++. Москва “Бином”. 1996г. 561 с.
14. 9.Тэд Фейсон. Объектно-ориентированное программирование на Borland C++ 4.5. Киев. “Диалектика”. 1996 г. 487 с.
15. В.Давыдов. Технологии программирования C++. Санкт-Петербург. “БХВ-Петербург”. 2005 г. 312 с.
16. Borland C++. Programmer’s Guide. Borland International, Inc., 1993. 326 p.
17. Borland ObjectWindows for C++. Reference Guide. Borland International, Inc., 1993. 602 p.
18. Martin D.Carrol, Margaret A.Ellis. Designing and coding reusable. Addison-Wesley., 1995. 317 p.
19. Прата С. Язык программирования С. Лекции и упражнения. Пер. с англ. Москва. «Вильямс». 2009. 960 с.

Ознаки об’єктно-орієнтованих мов програмування. Механізми абстракції, інкапсуляції, успадкування та поліморфізму в С++

Лабораторна робота № 1

Тема : Необ’єктні властивості С++.

Мета: порівняти мови програмуваання С та С++ на основі вивчення операторів cin>>, cout<<, використання коментарів та вивчити оператори керування пам’яттю new, delete; перевантаження імен функцій; операції доступу до області бачення; вбудовані функції; аргументи функцій за замовчуванням.

Теоретичні відомості

1 Основні відмінності С та С++ :

- Додаткові ключові слова: catch, class, delete, inline, operator,private,protected,public, template, this, throw, try,virtual,volatile.
- С++ потребує, щоб прототипи всіх функцій були визначені до їх виклику.
- В С++ здійснюється ретельніша перевірка типів у виразах.
- В С++ тип char - це байт з 8 біт. В С розмір цього типу не визначений.
- В С символьні константи мають тип int. В С++ символьні константи мають тип char.
- Подвійна риска (//) у С++ означає коментар, який займає рядок.
- Введення та виведення здійснюються через потоки cin >> та cout <<

2 Оператори new і delete

Операції new та delete виконують динамічний розподіл та відміну розподілу пам'яті, аналогічно, але з більш високим пріоритетом, ніж стандартні бібліотечні функції родини malloc та free (Див. довідник по бібліотеці).

Спрощений синтаксис:

покажчик_імені = *new* ім'я <ініціалізатор_імені>;
delete покажчик _імені;

Ім'я може бути будь-якого типу, крім "функція, що повертає..." (проте, покажчики функцій тут допустимі).

Операція new намагається створити об'єкт з типом "ім'я", розподіливши (при можливості) sizeof(ім'я) байтів у вільній області пам'яті (яку також називають областю, що динамічно розподіляється). Тривалість існування в пам'яті наданого об'єкту - від моменту його утворення і доти, доки операція delete не відмінить розподілену для нього пам'ять, або до кінця роботи програми.

В разі успішного завершення new повертає покажчик нового об'єкту. Порожній покажчик означає невдале завершення операції (наприклад, недостатній обсяг чи надто велика фрагментація області, що розподіляється). Як і в разі malloc, перш ніж намагатися звертатися до нового об'єкту, треба перевірити покажчик на порожнє значення (якщо тільки не використовується обробник new; докладніше про

це мова піде у наступних розділах). Проте, на відміну від malloc, функція new сама вираховує розмір "ім'я", і явно зазначати операцію sizeof немає необхідності. Далі, покажчик, що повертається, буде мати правильний тип, "покажчик імені", без необхідності явного приведення типів.

```
name  *nameptr  //  name може мати будь-який тип, крім функції
...
if (! (  nameptr  =  new  name ) )
  {
   errmsg ( " Недостатньо пам'яті для  name " );
   exit ( 1 );  }
// використання  *nameptr для  ініціалізації об'єкту  new  name
...
delete  nameptr;  // вилучення  name та відміна розподілу
                        // sizeof (name)  байт пам'яті
```

Примітка: new, являючись ключовим словом, не потребує прототипу.

2.1 Обробка помилок

Ви можете визначити функцію, що буде викликатися, якщо операція new завершується невдало (повертає 0). Щоб повідомити операції new про функцію-обробник new, викличте функцію set_new_handler і передайте їй покажчик на обробник new. Прототипом set_new_handler є наступний (із файлу new.h):

```
void (*set_new_handler( void (*) () )) ();
```

Функція set_new_handler повертає старий обробник new і змінює _new_handler таким чином, щоб він, в свою чергу, вказував на визначений вами обробник new.

2.2 Операція new з массивами

Якщо "ім'я" - це масив, то покажчик, що повертає new, вказує на перший елемент масиву. При створенні за допомогою new багатомірних масивів треба зазначати всі вимірності масиву (хоча ліва вимірність не обов'язково повинна бути константою часу компіляції):

```
mat_ptr = new int [3] [10] [12];       // так можна
mat_ptr = new int [n] [10] [12];       // так можна
mat_ptr = new int [3] [] [12];         // не можна
mat_ptr = new int [] [10] [12];                // не можна
```

2.3 Операція delete з масивами

При вилученні масиву слід використовувати синтаксис: "delete [ ] вираз". Вимірність масиву в квадратних дужках можна зазначати:

```
char *p;
void func ()
 {
  p = new char [10];      // виділити пам'ять під 10 символів
  delete [ ] p;           // видалити 10 символів
 }
```

В Borland C++, коли елемент масиву є класом із деструктором, вимагається тільки []. Проте, гарна манера програмування полягає в тому, щоб завжди зазначати компілятору, що масив видаляється.

2.4 ::operator new

При використанні з об'єктами, що не є класами, new викликає стандартну бібліотечну підпрограму, global ::operator new. Для об'єктів класів типу "ім'я" можна визначити спеціальну операцію ім'я ::operator new. Операція new, стосовно об'єктів класу "ім'я", запускає відповідну операцію ім'я ::operator new; в іншому випадку використовується стандартна операція ::operator new.

2.5.Ініціалізатори з операцією new

Іншою перевагою операції new порівняно з malloc є необов’язковий ініціалізатор (хоч malloc зводить його розподіл пам'яті до нуля). За відсутності явних ініціалізаторів об'єкт, який створюється за допомогою new, містить непередбачені дані ("сміття"). Об'єкти, що розподіляються new, за винятком масивів, можуть ініціалізуватися відповідним висловом у круглих дужках:

```
int_ptr = new int ( 3 );
```

Масиви класів із конструкторами ініціалізуються за допомогою конструктора за замовчуванням. В конструкторах C++ для об'єктів типу класу, операція new, що визначається користувачем, з ініціалізацією, що задає користувач, відіграє ключову роль.

3 Перевантаження імен функцій

Коли деякі функції виконують однакову роботу над об'єктами різних типів, може бути досить зручно дати їм одне й те ж саме ім'я. Використання одного імені для різних дій над різними типами називається перевантаженням (overloading). Цей засіб уже використовується для основних операцій C++: у додавання існує тільки одне ім'я, +, але його можна застосовувати для додавання значень цілих, плаваючих та вказівних типів. Ця ідея легко поширюється на обробку операцій, визначених користувачем, тобто функцій.

Щодо компілятора, єдине спільне, що мають функції з однаковим ім'ям, це ім'я. Таким чином, перевантажені імена функцій - це головним чином зручність запису. Ця зручність значна в разі функцій з узвичаєними іменами на зразок sqrt, print та open. Коли ім'я семантично значуще, як це має місце для операцій типу +, та << і в разі конструкторів, ця зручність стає істотною. Коли викликається перевантажена f(), компілятор повинен зрозуміти, до якої з функцій з ім'ям f треба звернутися. Це виконується шляхом порівняння типів фактичних параметрів з типами формальних параметрів всіх функцій з ім'ям f. Пошук функції, яку слід викликати, здійснюється за три окремих кроки:

1. Шукати точно відповідну функцію, і використовувати її, якщо вона знайдена;
2. Шукати відповідну функцію використовуючи вбудовані перетворення і використовувати будь-яку знайдену функцію;
3. Шукати відповідну функцію, використовуючи перетворення, визначені

користувачем, і якщо множина перетворень єдина, використовувати знайдену функцію.

Наприклад :

```
double print ( double );
int print ( int );
void  f ( ) {
  print ( 1 );
  print ( 1. 0 );
}
```

Правило точної відповідності гарантує, що f надрукує 1 як ціле і 1.0 як число з плаваючою крапкою. Нуль, char або short точно відповідають параметру. Аналогічно, float точно відповідає double.

До параметрів функцій з перевантаженими іменами стандартні C++ правила неявного перетворення типу застосовуються не повністю. Перетворення, що можуть знищити інформацію, не виконуються. Залишаються int в long, int в double, нуль в long, нуль в double і перетворення покажчиків: перетворення нуль у покажчик void*, і покажчик на похідний клас у покажчик на базовий клас.

Наведемо приклад, у якому перетворення необхідне:

```
print ( double );
print ( long );
void f ( int  a ) {
  print ( a );
}
```

Тут a може бути надруковано або як double, або як long. Неоднозначність розв'язується явним перетворенням типу (або print(long(a)) або print(double(a))).

При цих правилах можна гарантувати, що коли ефективність чи точність обчислень для використовуваних типів істотно розрізняються, буде використовуватися найпростіший алгоритм (функція). Наприклад :

```
int  pow ( int,  int );
double  pow ( double,  double );              // із <math.h>
complex  pow ( double,  complex ); // із <complex.h>
complex  pow ( complex,  int );
complex  pow ( complex,  double );
complex  pow ( complex,  complex );
```

Процес пошуку підходящої функції ігнорує unsigned і const.

4 Посилки

У той час як Сі передає аргументи тільки за значенням, C++ дозволяє передавати аргументи як за значенням, так і за посилкою. Типи посилок C++ тісно пов'язані з типами покажчиків і служать для утворення псевдонімів (аліасів) об'єктів та дозволяють здійснювати передачу аргументів функціям за посилкою.

4.1 Прості посилки

Для оголошення посилок поза функцією може використовуватися описувач посилки:

```
int i  = 0;
int  &ir = i;            //  ir є  псевдонімом i
ir  = 2;          //  те  ж,  що i = 2
```

В даному прикладі створюється значення ir, що є псевдонімом i, за умови, що ініціалізатор має той самий тип, що і посилка. Виконання операцій з ir має той самий результат, що і виконання їх з i. Наприклад, ir = 2 привласнює значення 2 змінній i, а &ir повертає адресу i.

Відзначимо, що type& var, type &var та type & var еквівалентні.

4.2 Аргументи типу посилки

Описувач посилки може також використовуватися для оголошення у функції параметрів типу посилки:

```
void func1 ( int i );
void func2 ( int  &ir );       // ir має тип "посилка  на int "
  ...
int sum=3;
func1 ( sum );                 // sum передається за значенням
func2 ( sum );                 // sum передається за посилкою
```

Переданий за посилкою аргумент sum може змінюватися прямо func2. Напроти, func1 одержує тільки копію аргументу sum (переданого за значенням), тому сама змінна sum функцією func1 змінюватися не може.

При передачі фактичного аргументу x за значенням відповідний формальний аргумент у функції приймає копію x. Будь-які зміни цієї копії у тілі функції не відбиваються на самому значенні x. Зрозуміло, функція може повернути значення, що після цього може використовуватися для зміни x, але самостійно змінити напряму параметр, що переданий їй за значенням, функція не може.

Традиційний засіб Сі для зміни x полягає у використанні в якості фактичного аргументу &x, тобто адреси x, а не самого значення x. Хоча &x передається за значенням, функція одержує доступ до x завдяки тому, що їй доступна одержана копія &x. Навіть якщо функції не потрібно змінювати значення x, тим не менш корисно (хоча можливі небажані побічні ефекти) передавати &x, особливо якщо x становить велику за розмірами структуру даних. Передача x безпосередньо за значенням веде до даремних витрат пам'яті на копіювання такої структури даних.

Порівняємо три різних реалізації функції treble:

```
Реалізація 1:
int  treble_1 ( n )
{
  return  3*n;
}
...
int  x,  i  = 4;
x  =  treble_1 (  i );   // тепер  x  = 12,  i  = 4
...
```

```
Реалізація 2 :
void  treble_2 ( int* np )
{
  *np  = (  *np )  *3;
}
...
treble_2 (  int  &i );  // тепер  i  = 12

Реалізація 3 :
void  treble_3 (  int&  n )   //  n має тип посилки
{
 n  =  3*n;
}
...
treble_3 (  i ); // тепер  i  = 36
```

Оголошення формального аргументу type& t (або, що еквівалентно, type &t) встановлює t як ту, що має тип "посилки на тип type". Тому при виклику treble_3 з дійсним аргументом i, і використовується для ініціалізації формального аргументу посилки n. Отже, n відіграє роль псевдоніму i, та n = 3*n також привласнює і значення 3*i.

Якщо ініціалізатор становить константу чи об'єкт непосилочного типу, то Borland C++ створить тимчасовий об'єкт, для якого посилка діє як псевдонім:

```
int& ir = 6;/* створює тимчасовий об'єкт int, з ім'ям псевдоніму  ir, що одержує значення 6*/
float  f;
int& ir2 = f; /*створює тимчасовий об'єкт int, з ім'ям псевдоніму ir2, f перед привласненням перетворюється*/
 ir2 = 2.0            //  тепер ir2 = 2, але f залишається без змін
```

Якщо формальні та фактичні аргументи мають різні (але сумісні за привласненням) типи, автоматичне утворення тимчасових об'єктів дозволяє виконувати перетворення посилочних типів. При передачі за значенням, зрозуміло, проблем з перетворенням типів менше, оскільки перед привласненням формальному аргументу копія фактичного аргументу може бути фізично змінена.

5 Операція доступу до області дії

Операція доступу до області дії (або дозволу області дії) **::** (дві двокрапки підряд) дозволяє здійснювати доступ до глобального (або на час існування файлу) імені навіть в тому випадку, коли це ім'я приховано локальним переоголошенням цього імені:

```
int  i;                  // глобальна  змінна i
...
void  func ( void ) {
      int i=0;           // локальна i "приховує" глобальну змінну i
        i = 3;           // ця i - локальна
```

```
    ::i = 4;              // ця і - глобальна
   printf ("%d", i);      // буде надруковано значення 3
}
```

Примітка: Наведений код працює також, якщо "глобальна" і є статичною змінною на рівні файлу.

При використанні з типами класів операція :: має інший зміст.

6 Вбудовані функції

Специфікатор inline може бути використаний перед оголошенням функції для того, щоб компілятор розташував її код в місці виклику функції. Залежно від реалізації компілятор може виконати цю вимогу чи ні.

7 Аргументи функцій за замовчуванням

Важлива властивість C++ - можливість передачі аргументів у функцію за замовчуванням. Вона використовується, якщо при виклику функції необхідна тільки частина параметрів, а не всі.

Припустимо, нам необхідна функція, що повертає суму чотирьох цілих значень, подібна такій:

```
int sum(int a,int b,int c,int d)
    {  return a+b+c+d;}
```

У даному випадку для виклику функції тільки з двома аргументами необхідно доповнити нулями два відсутні параметри:

```
cout << sum(v1,v2,0,0);
```

Але щоб зробити такий запис необхідно розібратися в документації по функції *sum()* для того, щоб визначити, якими значеннями доповнити невикористані параметри. Аргументи функцій за замовчуванням створені для захисту від використання помилкових значень параметрів при виклику функцій. Вони доповнюють необхідні значення для незаданих аргументів.

Для оголошення значення аргументу за замовчуванням в прототипі функції потрібно завершити параметр функції знаком рівності та необхідним значенням.

```
int sum(int a,int b,int c=0,int d=0);
```

Значення за замовчуванням повинні йти останніми у списку параметрів функції і можуть з'являтися тільки в прототипі функції. Тепер доступні такі оператори:

```
cout << sum(1,2);          //a=1;b=2;c=0;d=0
cout << sum(1,2,3);              //a=1;b=2;c=3;d=0
cout << sum(1,2,3,4);
```

Контрольні питання до л.р.1:

1. Що спільного у мов С та С++? Що нового у С++?
2. Яким чином здійснюється введення та виведення у С++?
3. Що виконує операція new? Доки існує в пам’яті об’єкт створений за допомогою new? Що повертає new у разі успішного виконання?
4. Який засіб називають перевантаженням? Як розуміє компілятор до котрої саме з перевантажених функцій треба звернутися?

5. Що використовують для оголошення посилок поза функцією, для оголошення у функції параметрів типу посилки? У яких випадках корисно використовувати посилки?

6. Що дозволяє здійснювати операція :: (операція розширення області дії)?

7. Які функції називають вбудованими? Чи завжди при використанні inline функція буде вбудованою?

8. Для чого створені аргументи функцій за замовчуванням? Як повинні бути розташовані значення за замовчуванням у списку параметрів функції?

Завдання:

1. Написати програму за допомогою функції до одного з завдань:
   - з'єднання двох рядків;
   - додавання рядка в кінець іншого;
   - визначення довжини рядка;
   - підрахування кількості символів у рядку;
   - переписування рядка у зворотньому напрямі

використовуючи потоки введення/виведення та коментарі C++.

2. Напишіть групу перевантажених функцій, що повертають модуль цілого, довгого цілого та дійсного значения подвійної точності.

3. Прочитати рядок до динамічної змінної, яка зберігається в купі, використовуючи оператори new , << , >>, delete.

4. Написати програму, в якій автоматична змінна r у функціях increment () та main() маскує глобальну змінну r , використовуючи операцію :: .

5. Написати вбудовану функцію min( ) та звичайну функцію imin( ), які повертають менше з двох цілих значень.

6.Написати функцію, що знаходить площу будь-якої фігури. Використовувати значення за замовчуванням.

Завдання для самостійної роботи:

1. В заданому масиві 10 цілих чисел. Змінити порядок слідування його на зворотній без застосування допоміжного масиву.

2. Поміняти місцями символи рядка, симетричні відносно середини. Якщо в рядку непарна кількість символів, то середній залишається на місці.

3. Використовуючи оператори << та >> напишіть програму-фільтр, яка перетворює маленькі літери на великі.

4. Напишіть програму, яка друкує літери a....z та цифри 0....9 та їх цілі значення. Те ж саме виконати для інших друкуємих символів.

5. Написати вбудовану функцію min( ) , яка повертає менше з двох цілих значень. Використовуючи написану функцію, напишіть програму виміру швидкодії, яка повідомляє кількість заощадженого часу порівняно зі звичайним викликом функції.

6.Напишіть функцію, яка виконує обмін значень між двома цілими змінними. Тип аргументу - int*.

7.Напишть іншу функцію обміну, використовуючи тип int&.

8. Порахувати визначник матриці (3x3) застосувавши оператори new і delete та звільнити пам'ять до завершення програми.

9. В матриці 4x4 знайти найбільший елемент та його індекс рядка і

стовпчика.

10. Створити новий масив, видаливши зі старого від’ємні числа.

11. Підрахувати кількість літер у реченні. В програмі змінна і (глобальна) – кількість літер, у функції - змінна і (локальна) – лічильник циклу.

12.Написати групу функцій, що повертають квадрат int, long, double значень.

13.Використовуючи вбудовану функцію вивести на екран цифри, що зустрічаються в реченні.

Лабораторна робота № 2

Тема : Класи. Дані-члени, функції-члени. Оператор розширення області бачення. Специфікації public, private,protected. Конструктори та деструктори: загальні положення. Параметризовані конструктори. Конструктори та ініціалізація класу.

Мета : Усвідомити поняття “ Клас ”, ”дані-члени ” та ”функції-члени ”. Навчитися ними користуватися. Вивчити призначення конструкторів та деструкторів та навчитися ними користуватися

Теоретичні відомості

1 Оголошення класу

Класи C++ передбачають створення розширень системи передвизначених типів. Кожний тип класу становить множину об'єктів та операцій (правил), а також перетворень, використовуваних для утворення, маніпулювання та знищення таких об'єктів. Можуть бути оголошені похідні класи, що успадковують компоненти одного чи більше базових класів (класів, що породжують).

У C++ структури та об'єднання розглядаються як класи з певними замовчуваннями правил доступу.

Спрощений, у "першому наближенні", синтаксис оголошення класу наступний:

ключ_класу ім’я_класу
<:базовий_список> [<список_компонентів>]

Ключ_класу - це **class**, **struct** або **union**.

Необов’язковий базовий_список перераховує базовий клас чи класи, з яких ім’я_класу бере (або успадковує) об'єкти та правила. Якщо оголошені деякі базові класи, то клас "ім’я_класу" називається похідним класом. Базовий список містить специфікатори доступу за замовчуванням та необов’язкове їх перевизначення, які можуть модифікувати права доступу похідного класу до компонентів базових класів.

Необов’язковий список_компонентів оголошує компоненти класу (дані та функції) для імені-класу за замовчуванням та перевизначеннями специфікаторів доступу, що можуть впливати на те, які функції до яких компонентів класу можуть мати доступ.

1.1 Імена класів

Ім’я_класу - це будь-який ідентифікатор, унікальний в межах свого контексту. У класах структур та в об'єднаннях ім'я_класу може бути опущено.

1.2 Типи класів

Оголошення створює унікальний тип, тип класу "ім’я_класу". Це дозволяє вам оголошувати наступні об'єкти класу (або входження класу) наданого типу, а

також об'єкти, що є похідними від цього типу (такі як покажчики, посилки, масиви об'єктів "ім'я_класу" і т. д.):

```
class X {....};
X x, &xr, *xptr, xarray [10];/* чотири об'єкти : типу X, посилка на X, покажчик на X та масив елементів типу X */
struct Y {....};
Y y, &yr, *yptr, yarray [10];
// C дозволяє мати
// struct Y y, &yr, *yptr, yarray [10];
union Z {....};
Z z, &zr, *zptr, zarray [10];
// C дозволяє мати
// union Z z, &zr, *zptr, zarray [10];
```

Визначимо відмінність між оголошенням структур та об'єднань в C і C++: в C ключові слова struct та union обов'язкові, але в C++ вони потрібні тільки в тому випадку, коли імена класів Y і Z приховані (див. наступний розділ).

1.3 Контекст імені класу

Контекст імені класу є локальним, з деякими особливостями, характерними для класів. Контекст імені класу починається з точки його оголошення і закінчується разом з обсяговим блоком. Ім'я класу приховує будь-який клас, об'єкт, нумератор або функцію з тим самим ім'ям в обсяговому контексті. Якщо ім'я класу оголошено у контексті, що містить оголошення об'єкту, функції чи нумератора з тим самим ім'ям, звернення до класу можливе тільки за допомогою уточненого специфікатора типу. Це означає, що з ім'ям класу треба використовувати ключ класу, class, struct або union. Наприклад:

```
struct S {....};
int S (struct S *Sptr);
void func( void )
 {
  S t;                  // неприпустиме оголошення: немає ключа класу і функції S у контексті
  struct S s;// так можна : є уточнення ключом класу
  S ( &s );  // так можна : це виклик функції
}
```

C++ дозволяє також неповне оголошення класу :

```
class X;   // ще немає компонентів!
struct Y;
union Z;
```

Неповні оголошення дозволяють деякі посилання до імен класів X, Y або Z (зазвичай посилання на покажчики об'єктів класів) до того, як класи будуть повністю визначені. Зрозуміло, перш ніж ви зможете оголосити і використовувати об'єкти класів, ви повинні виконати повне оголошення класів з усіма їх компонентами.

2 Об'єкти класів

Об'єкти класів можуть бути привласнені (якщо не було заборонене копіювання), передані як аргументи функції, повернуті функцією (за деякими винятками) і т. д. Інші операції з об'єктами та компонентами класів можуть бути різними способами визначені користувачем, включаючи функції-компоненти та "друзів", а також перевизначення стандартних функцій та операцій при роботі з об'єктами конкретного класу. Перевизначення функцій та операцій називається перевантаженням. Операції та функції, обмежені об'єктами конкретного класу (або взаємозв'язаної групи класів) називаються функціями-компонентами даного класу. C++ має механізм, що дозволяє викликати те ж саме ім'я функції чи операції для виконання іншого завдання, залежно від типу чи числа аргументів чи операндів.

3 Список компонентів класу

Необов'язковий список компонентів становить послідовність оголошень даних (будь-якого типу, включаючи нумератори, бітові поля та інші класи) і оголошень та визначень функцій, кожне з яких може мати необов'язкові специфікатори класу пам'яті та модифікатори доступу. Визначені таким чином об'єкти називаються компонентами класу. Специфікатори класу пам'яті auto, extern та register у даному випадку недопустимі. Компоненти можуть бути оголошені зі специфікаторами класу пам'яті static.

3.1 Функції-компоненти

Функція, яка оголошена без специфікатора friend, називається функцією-компонентом класу. Функція, що оголошена з модифікатором friend, називається "функцією-другом".

Одне й те ж саме ім'я може використовуватися для позначення більш ніж однієї функції, за умови, що вони відмінні по типам або числу аргументів.

3.2 Ключове слово this

Нестатичні функції-компоненти працюють з об'єктами типу класу, з якими вони викликаються. Наприклад, якщо x - це об'єкт класу X, а f - це функція-компонент X, то виклик функції x.f() працює з x. Аналогічним чином, якщо xptr є покажчик об'єкту X, то виклик функції xptr ->f() працює з *xptr. Звідки f може знати, з яким X працювати? C++ передає f покажчик на X, що називається this. This передається як прихований аргумент при виклику нестатичних функцій-компонентів. Ключове слово this становить локальну змінну, доступну в тілі нестатичної функції-компоненту. this не потребує об'явлень, і на нього рідко зустрічаються явні посилки у визначенні функції. Однак, воно неявно використовується у функції для посилки до компонентів. Якщо, наприклад, викликається x.f(y), де y є компонентом X, то this встановлюється на &x, а y встановлюється на this->y, що еквівалентно x.y.

3.3 Функції що вбудовуються (inline)

Функція-компонента може бути оголошена в межах свого класу, але визначена будь-де в іншому місці. І навпаки, функція-компонента може бути і

оголошена, і визначена у своєму класі, і тоді вона називається функцією, що вбудовується. У деяких випадках C++ може зменшити витрати часу на виклик функції, підставляючи замість виклику функції безпосередньо зкомпільований код тіла функції. Цей процес, що називається вбудованим розширенням тіла функції, не впливає на контекст імені функції чи її аргументів. Вбудоване поширення не завжди корисне і бажане. Специфікатор inline становить собою запит (або вимогу) компілятору, в якому ви повідомляєте, що вбудовані поширення бажані. Як і в разі специфікатора класу пам'яті register, компілятор може або задовольнити, або проігнорувати ваше побажання.

Явні чи неявні запити inline краще за все резервувати для функцій невеликих за обсягом та функцій, що часто викликаються, таких як функції типу operator, що реалізують перевантажені оператори. Наприклад, наступне оголошення класу :

```
int i;                                  // global int
class X {
    char* i;
   public:
    char* func (void) {return i;}        // inline за замовчуванням char* i;
       };
```

еквівалентно

```
      inline char* X::func (void) {return i;}
```

func визначається "поза" класом з явним специфікатором inline. Змінна i, яку повертає func( ), є *char* i* класу X.

3.4 Статичні компоненти

Специфікатор класу пам'яті static може бути використаний в оголошеннях компонентів даних та функцій-компонентів класу. Такі компоненти називаються статичними і мають властивості, відмінні від властивостей нестатичних компонентів. В разі нестатичних компонентів для кожного об'єкту класу "існує" окрема копія; у випадку ж статичних компонентів існує тільки одна копія, а доступ до неї виконується без посилки на який-небудь конкретний об'єкт класу. Якщо x - це статичний компонент класу X, то до нього можна звернутися як X::x (навіть якщо об'єкти класу X ще не створені). Однак, можна виконати доступ до x і за допомогою звичайних операцій доступу до компонентів. Наприклад, у вигляді y.x та yptr->x, де y - це об'єкт класу X, а yptr - це покажчик об'єкту класу X, хоч вирази y та yptr ще не обчислені. Зокрема, статична функція-компонент може бути викликана як з використанням спеціального синтаксису виклику функцій-компонентів, так і без нього.

```
class X {
        int member_int;
  public:
        static void func ( int i, X* ptr );
  };
```

```
void g (void);
{
  X obj;
  func (1, &obj);    // помилка, якщо де-небудь ще не визначена глобальна
func ()
X::func (1, &obj); //виклик static func() в X допустимий тільки для статичних
функцій
obj.func (1, &obj)       //те ж саме (допустимо як для статичних, так і
нестатичних функцій)
}
```

Оскільки статична функція-компонент може викликатися без урахування якого-небудь конкретного об'єкту, вона не має покажчика this. Наслідок із цього такий, що статична функція-компонент не має доступу до нестатичних компонентів без явного задання об'єкта за допомогою “.” або “->”. Наприклад, з урахуванням оголошень, зроблених у попередньому прикладі, func() може бути визначена наступним чином:

```
void X::func(int i, X* ptr)
{
  member_int = i;  // на який об'єкт посилається member_int? Помилка!
  ptr->member_int = i;     // так можна : тепер ми знаємо!
}
```

Без урахування вбудованих функцій, статичні функції-компоненти глобальних класів мають зовнішній тип компоновки. Статичні функції-компоненти не можуть бути віртуальними функціями. Неприпустимо мати статичну та нестатичну функції-компоненти з однаковими іменами і типами аргументів.

Оголошення статичного компоненту даних в оголошенні класу не є визначенням, тому де-небудь ще повинно бути визначення, що відповідає за розподіл пам'яті та ініціалізацію. Визначення статичного компоненту даних може бути опущене, якщо діє засіб "ініціалізації нулями за замовчуванням".

Статичні компоненти класу, який оголошений локальним по відношенню до деякої функції, не мають типу компоновки і не можуть бути ініціалізовані. Статичні компоненти глобального класу можуть бути ініціалізовані подібно звичайним глобальним об'єктам, але тільки в контексті файлу. Статичні компоненти підлягають звичайним правилам доступу до компонентів класу, за винятком того, що вони можуть бути ініціалізовані.

```
class X {
  ...
  static int x;
  ...
 };
int X::x = 1;
```

Головне використання статичних компонентів складається в тому, щоб відслідковувати дані, спільні для всіх об'єктів класу, як наприклад, число створених об'єктів, або ресурс, що використовувався останнім з пула, що розділяється всіма подібними об'єктами. Статичні компоненти використовуються також для:

- зменшення числа видимих глобальних імен
- того, щоб зробити очевидним, які саме статичні об'єкти якому класу

належать

- дозволу управління доступом до їх імен.

4 Контекст компоненту

Вираз X::func() у прикладі, що наведений у розділі “Функції, що вбудовуються”, використовує ім’я класу X з модифікатором контексту доступу, що означає, що func(), хоча і визначена "поза" класом, в дійсності є функцією-компонентом X та існує в контексті X. Вплив X:: розповсюджується на тіло визначення цієї функції. Це пояснює, чому значення, що повертає функція, відноситься до X::i, char* і з X, а не до глобальної змінної int i. Без модифікатора X:: функція func представляла б звичайну функцію, що не відноситься до класу і повертає глобальну змінну int i.

Отже, всі функції-компоненти знаходяться у контексті свого класу, навіть якщо вони визначені поза цим класом.

До компонентів даних класу X можна звертатися за допомогою операцій вибору та -> (як і в структурах C). Функції-компоненти можна викликати також за допомогою операцій вибору. Наприклад,

```
class X {
   public:
  int i;
  char name [20];
  X *ptr1;
  X *ptr2;
 void Xfunc (char *data, X* left, X* right);// визначення знаходиться не тут
  };
void f (void)
{
  X x1, x2, *xptr=&x1;
  x1.i = 0;
  x2.i = x1.i;
  xptr->i = 1;
  x1.Xfunc ("stan", &x2, xptr);
}
```

Якщо m є компонентом чи базовим компонентом класу X, то вираз X::m називається кваліфікованим ім'ям; воно має той же тип, що й m, і є виразом, що іменує тільки в тому випадку, якщо виразом, що іменує є m. Ключовий момент тут полягає в тому, що навіть якщо ім'я класу X приховано іншим ім'ям, кваліфіковане ім'я X::m забезпечить доступ до потрібного імені класу, m.

Компоненти класу не можуть додаватися до нього в іншому розділі вашої програми. Клас X не може містити об'єкти класу X, але може містити покажчики чи посилки на об'єкти класу X (відзначимо аналогію з типами структур та об'єднань C).

1 Основні властивості конструкторів та деструкторів

Існує декілька спеціальних функцій-елементів, що визначать, яким чином об'єкти класу створюються, ініціалізуються, копіюються та руйнуються. Конструктори та деструктори є найбільш важливими з них. Вони володіють

більшістю характеристик звичайних функцій-елементів: ви повинні оголосити і визначити їх в межах класу, або оголосити їх у класі, а визначити поза ним. Однак, вони володіють і деякими унікальними властивостями.

1. Вони не мають оголошень значень повернення (навіть void).
2. Вони не можуть бути успадковані, хоча похідний клас може викликати конструктори та деструктори базового класу.
3. Конструктори, як і більшість функцій мови C++, можуть мати аргументи за замовчуванням чи використовувати списки ініціалізації елементів.
4. Деструктори можуть мати атрибут virtual, але конструктори не можуть.
5. Ви не можете працювати з їхніми адресами :

```
main ( )
 {
  ....
  void  *ptr = base::base;     // неприпустимо
  ....
 }
```

6. Якщо конструктори та деструктори не були задані явно, то вони можуть генеруватися C++. Часто вони також можуть запускатися за відсутності явного виклику у вашій програмі. Будь-який конструктор чи деструктор, що створюється компілятором, повинен мати атрибут public.
7. Викликати конструктор таким же чином, як і звичайну функцію, не можна. Виклик деструктора допустимий тільки з повністю уточненим ім'ям.

```
 {...
  X  *p;
  ....
  p->x::~x ( );                // допустимий виклик  деструктора
  X::X ( );             // неприпустимий виклик конструктора
  ....
 }
```

8. При визначенні та знищенні об'єктів компілятор виконує виклик конструкторів та деструкторів автоматично.
9. Конструктори та деструктори при необхідності розподілу пам'яті об'єктові можуть виконувати неявні виклики операцій new та delete.
10. Об'єкт з конструктором чи деструктором не може використовуватися у вигляді елементу об'єднання.

Якщо клас X має один чи більше конструкторів, то один з них запускається кожного разу при визначенні об'єкту x класу X. Конструктор створює об'єкт x та ініціалізує його. Деструктори виконують зворотній процес, руйнуючи об'єкти класу, створені конструкторами.

Конструктори активізуються також при створенні локальних або тимчасових об'єктів даного класу. Деструктори активізуються кожного разу, коли ці об'єкти виходять з області дії.

2 Конструктори

Конструктори відрізняються від інших функцій-елементів тим, що мають те ж саме ім'я, що і клас, до якого вони відносяться. При створенні чи копіюванні об'єкту даного класу відбувається неявний виклик відповідного конструктора.

Конструктори глобальних об’єктів викликаються до виклику функції main(). При використанні стартової директиви pragma для встановлення деякої функції до функції main() конструктори глобальних об’єктів викликаються до функцій початкового завантаження.

Локальні об'єкти створюються, як тільки стає активною область дії змінної. Конструктор також запускається при створенні тимчасового об'єкту даного класу.

```
class  X {
 public :
   X ( );          // конструктор класу  X
};
```

Конструктор класу X не може сприймати X як аргумент:

```
class  X  {
 ....
 public
   X (X);                  // неприпустимо
}
```

Параметри конструктора можуть бути будь-якого типу, за винятком класу, елементом якого є даний конструктор. Конструктор може приймати у вигляді параметра посилку на свій власний клас. В такому випадку він називається конструктором копіювання. Конструктор, що не має параметрів взагалі, називається конструктором, що використовується за замовчуванням.

3 Конструктор, який використовується за замовчуванням

Конструктором, що використовується за замовчуванням для класу X називається такий конструктор, що не має ніяких аргументів: X::X( ). Якщо для класу не існує конструкторів, що визначаються користувачем, то C++ генерує конструктор за замовчуванням. При таких оголошеннях, як X x, конструктор за замовчуванням створює об'єкт x.

Як і всі функції, конструктори можуть мати аргументи, що використовуються за замовчуванням. Наприклад, конструктор:

```
X::X ( int, int = 0 );
```

може мати один чи два аргументи. Якщо даний конструктор буде представлений тільки з одним аргументом, другий аргумент, якого не вистачає, буде прийнятий як int 0. Аналогічним чином, конструктор:

```
X::X ( int = 5, int  = 6 )
```

може мати два аргументи, один аргумент, або не мати аргументів взагалі, причому в кожному випадку використовуються відповідні замовчування. Однак, конструктор за замовчуванням X::X( ) не має аргументів взагалі, і його не треба плутати з конструктором, наприклад, X::X (int = 0), що може або мати один аргумент, або не

мати аргументів.

При виклику конструкторів треба уникати неоднозначностей. У наступному прикладі можливе неоднозначне сприйняття компілятором конструктора, що використовується за замовчуванням та конструктора, що сприймає цілочисельний параметр :

```
class  X {
  public :
     X ( );
     X ( int i = 0 );
 };
main ( )
 {
  X  one ( 10 ); // припустимо: використовується X::X ( int )
  X  two;        ....
 return 0;
}
```

4 Конструктор копіювання

Конструктор копіювання для класу X - це такий конструктор, що може викликатися з одним-єдиним аргументом типу X: X::X (const X&) або X::(const X&, int = 0). У конструкторі копіювання припустимими також є аргументи за замовчуванням. Конструктори копіювання запускаються при копіюванні об'єкту даного класу, зазвичай в разі оголошення з ініціалізацією об'єктом іншого класу:

X x = y;

Якщо такий конструктор необхідний, але у класі X не визначений, C++ генерує конструктор копіювання для класу X автоматично.

5 Перевизначення конструкторів

Конструктори можна перевизначити, що дозволяє створювати об'єкти залежно від значень, що використовувалися при ініціалізації.

```
class  X {
    int  integer_part;
    double  double_part;
 public :
    X ( int i ) { integer_part = i; }
    X ( double d ) { double_part = d }
};
main ( )
 {
  X  one (10); // викликає X::X(int) і встановлює integer_part у значення 10
   X  one (3. 14); // викликає X::X(double)  і встановлює double_part у значення
3. 14
  ....
  return 0;
 }
```

При наявності конструктора класу об'єкти або ініціалізуються, або мають конструктор за замовчуванням. Останній використовується в разі об'єктів без явної ініціалізації.

Об'єкти класів із конструкторами можуть ініціалізуватися за допомогою списків, що задаються у круглих дужках ініціалізаторів. Цей список використовується як список аргументів, що передаються конструкторові. Альтернативним засобом ініціалізації є використання знаку рівності, за яким слідує окреме значення. Це окреме значення може мати тип першого аргументу, що сприймається конструктором даного класу. В цьому випадку додаткових аргументів або не існує, або вони мають значення за замовчуванням. Значення може також бути об'єктом даного типу класу. У першому випадку для створення об'єкту викликається відповідний конструктор. В останньому випадку для ініціалізації об'єкту викликається конструктор копіювання.

```
class  X
 {
    int  i;
  public :
    X ( ); // тіла функцій для ясності опущені
    X (  int  x );
    X (  const  X& );
 };

main ( )
 {
   X  one;           // запуск конструктора за замовчуванням
   X  two(1);        // використовується конструктор X::X (int)
   X  three = 1;     // викликає X::X (int)
   X  four = one;  // запускає X::X (const X&) для копіювання
   X  five(two);     // викликає X::X (cont X&)
   ....
 }
```

Конструктор може привласнювати значення своїм елементам наступним чином: він може приймати значення в якості параметрів і виконувати привласнення елементам змінним власне у тілі функції конструктора :

```
class  X
 {
   int  a,  b;
  public :
    X ( int i, int j )  { a = i; b = j }
 };
```

6 Деструктори

Деструктор класу викликається для звільнення елементів об'єкту до знищення самого об'єкту. Деструктор - це функція-елемент, ім'я якої співпадає з

ім'ям класу, перед яким стоїть символ тильди (~). Деструктор не може приймати ніяких параметрів, а також не оголошує типу або значення, що повертається.

```
class  X
 {
  public :
    ~X ( );          // деструктор класу X
 };
```

Якщо деструктор не оголошений для класу явно, компілятор генерує його автоматично.

7 Виклик деструкторів

Виклик деструктора виконується неявно, коли змінна виходить зі своєї оголошеної області дії. Для локальних змінних деструктори викликаються, коли перестає бути активним блок, в якому вони оголошені. В разі глобальних змінних деструктори викликаються як частина процедури виходу після функції main().

Коли покажчики об'єктів виходять за межі області дії, неявний виклик деструктора не відбувається. Це означає, що для знищення такого об'єкту операція delete повинна задаватися явно.

Деструктори викликаються суворо у зворотній послідовності щодо послідовності виклику відповідних їм конструкторів.

8 atexit, #pragma exit і деструктори

Всі глобальні об'єкти залишаються активними доти, доки не будуть виконані коди в усіх процедурах виходу. Локальні змінні, включаючи ті, що оголошені у функції main(), знищуються при їх виході з області дії. Послідовність виконання в кінці програми Borland C++ у цьому сенсі така:

- виконуються функції atexit у послідовності їх вставлення в програму;
- виконуються функції #pragma exit відповідно до кодів їх пріоритетів;
- викликаються деструктори глобальних змінних.

9 exit( ) і деструктори

При виклику exit( ) з програми деструктори для будь-яких локальних змінних у поточній області дії не викликаються. Глобальні змінні знищуються у звичайній послідовності.

10 abort( ) і деструктори

При виклику abort( ) будь-де з програми деструктори не викликаються, навіть для змінних глобальної області дії.

Деструктор може також викликатися явно, одним з двох способів:

- непрямо, через виклик delete;
- безпосередньо, шляхом завдання повністю уточненого імені деструктора.

delete можна використовувати для знищення тих об'єктів, для яких пам'ять

розподілялася за допомогою new. Явний виклик деструктора необхідний тільки в разі об'єктів, яким за допомогою виклику new розподілялася конкретна адреса пам'яті.

```
class  X  {
 ....
 ~X ( );
 ....
};

void*  operator  new (  size_t  size,  void  *ptr )
{
 return  ptr;
}

char  buffer [ sizeof ( x ) ];

main ( )
{
 X*  pointer  =  new  X;
 X*  exact_pointer;

 exect_pointer  =  new  ( &buffer ) X; //покажчик ініціалізується адресою буферу
 ....
 delete  pointer;                    // delete служить для руйнування покажчика
 exact_pointer->X::~X ( );  }    // прямий виклик для відміни розподілу пам'яті
```

Контрольні питання до л.р. 2:

1. Що називається класом?
2. Які специфікатори доступу ви знаєте?
3. Яку роль відіграє специфікатор доступу?
4. Що треба створити для того, щоб використати клас?
5. Що представляє собою об’єкт класу?
6. За допомогою яких операцій можна звертатись до компонентів даних та функцій класу?
7. Які функції класу називають вбудованими?
8. Для чого потрібні неповні оголошення класу?
9. Які дії можна виконувати над об’єктами класів?
10. Розкажіть про ключове слово this
11. Які компоненти називають статичними?
12. Що представляє собою конструктор?
13. Що представляє собою деструктор?
14. Перелічіть унікальні властивості конструкторів і деструкторів
15. Скільки конструкторів можна оголосити у будь-якому класі?
16. Скільки деструкторів можна оголосити у будь-якому класі?

17. Як викликаються конструктори і деструктори?
18. Коли викликаються конструктори глобальних об'єктів класу і що це забезпечує?
19. Коли викликаються деструктори глобальних об'єктів класу?
20. Як викликаються конструктори та деструктори для автоматичних об'єктів класу оголошених локальними у функції?
21. Який конструктор класу викликається для кожного об'єкту масиву при створенні масивів об'єктів класу?
22. Які механізми безпечного копіювання і привласнення об'єктів класу, що мають члени-покажчики?

Завдання:

1. Визначити класи Triangle та Circle і знайти площі вказаних фігур (трикутника та кола).

2. Визначити клас трикутника де функція-член класу визначає найменшу сторону трикутника (за формулою визначеня відстані між двома точками на координатній площині).

3. За допомогою класу Ttime напишіть DT.Cpp, яка виводить поточні дату та час та заміняє дату в вашому об'єкті на завтрашню.

4. Розробіть клас кнопки, який можна використовувати в моделюванні, де необхідні перевимикачі ввімкнено/вимкнено (дані-члени класу) та функції-члени класу які визначають стан ввімкненої кнопки та вимкненої кнопки.

5. Дана програма:

```
#include <iostream.h>
main( )
{
    cout <<  " Привіт! \n";
     return 0;
}
```

Модифікуйте її так, щоб вона виводила на виході:

Ініціалізація
Привіт!
Очищення

Не змінюйте main( )!!!.

6. Створіть клас, здатний зберігати рядок у купі. Повинна існувати можливість передачі рядка об'єкту вашого класу і потім отримання покажчика на такий самий рядок. Має бути також можливість зміни рядка об'єкту класу. Використовуйте конструктори і деструктор для запровадження всіх автоматичних ініціалізацій та очищення об'єкту.

7. Визначте клас histogram, який підраховує числа у певних інтервалах, що описані як аргументи конструктора histogram. Напишіть функції для виведення гістограми. Передбачте обробку значень, що виходять за межі визначеної області.

8. Є клас з ім'ям Tfruit та об'єкт orange типу Tfruit. Використовуйте orange для ініціалізації нового об'єкту на ім'я grapefruit за допомогою конструктора

копіювання.

Завдання для самостійної роботи:

1. Визначити функцію, що малює лінію, яка з'єднує 2 фігури (кола), відшукуючи 2 найближчі точки дотику.

2.Створити клас для роботи із десятковими числами. Дані-члени: цілі, десяткові; дані-функції: встановлення цілих та десяткових, виведення цілих і десяткових, числа.

3. Розробіть клас траспорт. Дані-члени: назва транспорту, швидкість в км/год, затрати на пальне (л/км). Функції-члени: виведення часу витраченого на подолання шляху та витрат пального. Аргументи функції: назва транспорту та відстань.

4. Розробити клас з наступними членами. Дані-члени: прізвище студента, масив [10] оцінок. За допомогою функції вивести «гарна успішність», якщо к-ть 5 = к-ть 3, а інші 4 або усі 5, інакше вивести «погана успішність».

5. Розробити клас для роботи з рядками. Функції-члени: перетворення першої літери на велику, підрахунок голосних літер, підрахунок інших символів.

6. Розробити клас для шифрування. Дані-член – ключ цифр. Дані-функції: шифратор, дешифратор. До кожної літери речення додати ключ-цифру (шифрування). Дешифрування – зворотній процес.

7. Розробити клас для роботи з паролем. Дані-члени: логін, пароль. Функції-члени: перевірка пароль = логін, наявність букв та цифр у паролі, перевірка довжини пароля, якщо пароль != логін, довжина пароля >=6 та виконується друга умова вивести «безпечний пароль», інакше «змініть пароль».

8. Виконайте завдання 2 так щоб ваша програма використовувала ваш рядковий об'єкт для читання і відображення текстового файлу.

9. Використовуючи створений вами клас TTime, напишіть програму з іменем DAY.CPP, яка повідомляє день тижня для будь-якої дати, що передана в якості параметра командного рядка.

10. Визначте клас для аналізу, запам'ятовування, обчислення та друку простих арифметичних виразів, які складаються з цілих констант та операторів + , - , * , /. Загальний інтерфейс повинен виглядати так:

```
class expr{
public: expr(char*);
        int eval ( );
        void print( );
};
```

Аргумент для конструктора - вираз. Функція expr::eval() повертає значення виразу. Print() виводить вираз на екран.

Наприклад:

```
expr x("123/4 + 123*4 - 3");
cout << "  x= "<< x.eval()<<"\n";
 x.print();
```

11. Розробити клас матриці. Параметри: покажчик на область, кількість рядків та стовпчиків. В класі передбачити 3 конструктори:

- за замовчуванням: створення матриці 3x3 та заповнення випадковими значеннями,

- з одним параметром – створення квадратної матриці та заповнення її,
- з двома – відповідно.

Функція-метод повинна знаходити максимальне і мінімальне число в рядку.

5. Розробити класи Ford та Opel. Конструктори з параметрами (модель, рік випуску авто). Конструктор ініціалізує модель, рік випуску авто, в залежності від марки (передбачити 5+) об'єм двигуна та інші параметри авто. Розробити метод, який виводить інформацію про автомобіль. Приклад:

Марка: Ford

Модель: Fiesta

Рік випуску: 2012

Тип кузова: …

6. Створити клас, що оптимізує купівлю посуду. Параметри конструктора класу: ім'я колекції, ціна предмету. Знайти вартість усіх можливих комплектів з трьох предметів та, як варіант, усі предмети. Розробити функцію-член, яка приймає суму грошей і номер варіанту, а повертає кількість обраних комплектів, що можна купити на дану суму.

12. Створити клас, який має конструктор з параметрами, необхідними для обчислення таких значень для конуса, як: площа основи конуса, площа бокової поверхні, площа поверхні конуса та об'єм. Для кожної шуканої величини розробити методи.

13. Створити клас з конструктором, що отримує всі необхідні аргументи для визначення значення за формулою геометричної прогресії. Використати параметри за замовчуванням.

14. Створити клас з конструктором, що отримує всі необхідні аргументи для визначення значення за формулою Бернулі. Усі параметри мають бути за замовчуванням.

15. Створити клас для авторизації користувачів. В конструкторі використати параметри за замовчуванням (user="user", pass=" ",dateOfBday,tel). Розробити метод перевірки даних, переданих користувачем. Розроблений метод має виводити картку користувача з параметрами переданими конструктору, а якщо ім'я та пароль відповідають адміністраторським, то вивести картку адміністратора.

Лабораторна робота № 3

Тема: Управління доступом до компонентів класу. Одиночна спадкоємність. Віртуальні функції. Віртуальні деструктори.

Мета: Вивчити взаємозв’язок між базовими та похідними класами. Навчитись користуватися модифікаторами доступу до компонентів класів.

Теоретичні відомості

1. Модифікатори доступу

Компоненти класу отримують атрибути доступу або за замовчуванням (залежно від ключа класу та місцезнаходження оголошення), або при використанні якого-небудь із специфікаторів доступу: public, private або protected. Значення цих атрибутів наступні:

public Компонент може бути використаний будь-якою функцією.

private Компонент може бути використаний тільки функціями-компонентами і "друзями" класу, в якому він оголошений.

protected Те ж саме, що для private, але крім того, компонент може бути використаний функціями-компонентами і друзями класів, похідних від оголошеного класу, але тільки в об'єктах похідного типу. (Похідні класи розглядаються у наступному розділі).

Компоненти класу за замовчуванням мають атрибут private, тому для перевизначення даного оголошення специфікатори доступу public або protected повинні задаватися явно.

Компоненти struct за замовчуванням мають атрибут public, але ви можете перевизначити це замовчування за допомогою специфікаторів доступу private або protected.

Компоненти union за замовчуванням мають атрибут public. Перевизначити його не можна. Всі три специфікатори доступу задавати для компонентів об'єднання неприпустимо.

Модифікатор доступу за замовчуванням чи заданий для перевизначення атрибуту доступу залишається дійсним для всіх наступних оголошень компонентів, доки не зустрінеться інший модифікатор доступу. Наприклад:

```
class X {
  int i;          //  X::i за замовчуванням  private
  char ch;        // також
public :
  int j;          // наступні два компоненти - public
  int k;
protected :
  int l;          // X::l - protected
};
struct Y {
  int i;          // Y::i за замовчуванням public
 private:
```

```
  int  j;          // Y::j - private
 public:
  int  k;                  // Y::k - public
};
union  Z  {
  int  i;          // public за замовчуванням, інших варіантів  немає
  double  d;
};
```

Специфікатори доступу можуть бути перераховані та згруповані у будь-якій зручній послідовності. Можна зекономити місце при наборі програми, оголосивши всі компоненти private відразу, і т. і.

2 Доступ до базових і похідних класів

Спочатку дамо визначення базового та похідного класів. Клас, який породжує нові класи, називається базовим. Клас, який успадковує з базового класу дані-члени та функції-члени називається похідним класом.

При оголошенні похідного класу D ви перераховуєте базові класи B1,B2... у базовому_списку, що розділяється комою:

ключ_класу D: базовий_список {<список_компонентів>}

Оскільки сам базовий клас може бути похідним класом, то питання про атрибут доступу вирішується рекурсивно: ви відслідковуєте його доти, доки не дістанетесь "самого" базового класу, що породив всі інші.

D успадковує всі компоненти базових класів. (Перевизначені компоненти базових класів успадковуються, і при необхідності доступ до них можливий за допомогою перевизначень контексту). D може використовувати тільки компоненти базових класів з атрибутами public та protected. Проте, що будуть становити атрибути доступу успадкованих компонентів із точки зору D? D може знадобитися використовувати public-компонент базового класу, але при цьому зробити його для зовнішніх функцій private. Рішення тут полягає в тому, щоб використовувати специфікатори доступу в базовому_списку.

При оголошенні D ви можете задати специфікатор доступу public або private перед класами у базовому_списку:

```
 class  D :  public  B1,  private  B2, ... {
  ...
 }
```

Задати у базовому_списку protected не можна. Об'єднання не можуть містити базових класів і не можуть самі бути використані у вигляді базових класів.

Ці модифікатори не змінюють атрибутів доступу базових членів із точки зору базового класу, хоч і можуть змінити атрибути доступу базових компонентів із точки зору похідних класів.

За замовчуванням є private, якщо D є оголошення класу class, і public, якщо D є оголошення структури struct.

Похідний клас успадковує атрибути доступу базового класу наступним чином:

базовий клас **public**: компоненти public базового класу стають членами public похідного класу. Компоненти protected базового класу стають компонентами protected похідного класу. Компоненти private базового класу залишаються для базового класу private.

базовий клас **private**: і public, і protected компоненти базового класу стають private компонентами похідного класу. Компоненти private базового класу залишаються для базового класу private.

В обох випадках відзначимо, що компоненти private базового класу були і залишаються недосяжними для функцій-компонентів похідного класу доти, доки в описанні доступу базового класу не будуть явно задані оголошення friend. Наприклад,

```
class  X :  A  {
// за замовчуванням для класу -  private  A
          ....
 };
/* клас X є похідним від класу A */
class  Y :  B,  public  C  {       // перевизначає замовчування для C.
  ....
 };
/* клас Y є похідним (множинна спадкоємність) від  B і C. За замовчуванням -
private  B */

struct  S :  D  {            //  за замовчуванням для  struct -  public  D
  ....                       /* struct  S - похідна від  D              */
 }

struct  T :  private  D,  E  {// перевизначає замовчування для D.
//  E за замовчуванням  public
  ....
 }
/* struct  T є похідною (множинна спадкоємність) від D і T. За замовчуванням
-  public  E */
```

Дію специфікаторів доступу в базовому списку можна скоригувати за допомогою кваліфікованого імені в оголошеннях public або protected для похідного класу. Наприклад,

```
        class  B  {
            int  a;                    // за замовчуванням  private
          public :
            int  b,  c;
            int  Bfunc (  void );
         };

        class  X :  private  B  {    // тепер в X a, b, c і Bfunc  private
      int  d;                  // за замовчуванням private. Примітка: f  в X недосяжна
     public :
      B : :  c;                // c була private; тепер вона public
      int  e;
      int  Xfunc (void);
```

```
};

int  Efunc (X& x);                // зовнішня стосовно B та X
```

Функція Efunc може використовувати тільки імена з атрибутом public, наприклад c, e та Xfunc.

Функція Xfunc в X є похідною від private B, тому вона має доступ до :

- "скоригованої-до-типу-public" c
- "private-щодо-X" компонентам B:b i Bfunc
- власним private та public компонентам: d, e та Xfunc.

Проте, Xfunc не має доступу до "private-відносно-B" компоненту a.

3 Порядок виклику конструкторів

У випадку, коли клас використовує один чи більше базових класів, конструктори базових класів запускаються до того, як будуть викликані конструктори похідного класу. Конструктори базового класу викликаються у послідовності їх оголошення. У наступному прикладі:

```
class  Y  {....};
class  X :  public  Y  {....};
X  one;
```

виклик конструкторів відбувається у наступній послідовності:

```
Y ( );                    // конструктор базового класу
X ( );                    // конструктор похідного класу
```

Конструктори елементів масиву запускаються у порядку зростання індексів масиву.

Конструктор може використовувати список ініціалізаторів, що знаходиться до тіла функції:

```
class  X
          {
   int  a,  b;
  public :
   X ( int i, int j ) : a ( i ), b ( j ) {}
 };
```

В цьому випадку ініціалізація X x (1, 2) привласнює значення 1 x::a і значення 2 x::b. Список ініціалізаторів забезпечує механізм передачі значень конструкторам базового класу.

Примітка: Щоб забезпечити можливість їх виклику з похідного класу, конструктори базового класу повинні оголошуватися з атрибутами public або protected.

```
class  base1
 {
   int  x;
  public :
```

```
  base1 ( int i ) { x = i; }
};

class  base2
{
  int  x;
public :
  base2 (  int  i ) :  x (  i )  {}
};

class  top : public base1, public base2
{
  int  a,  b;
public :
  top ( int i, int j ) : base1 ( i*5 ), base2 ( j+i ), a ( i ) { b = j; }
};
```

В разі такої ієрархії класу оголошення top one(1,2) призведе до ініціалізації base1 значенням 5,а base2 значенням 3. Способи ініціалізації можуть комбінуватися один з другим (чергуватися).

Як було описано вище, базові класи ініціалізуються у послідовності опису. Після цього відбувається ініціалізація елементів, також у послідовності їх оголошення, незалежно від їх взаємного розміщення у списку ініціалізації.

```
class  X
{
  int  a,  b;
public :
  X( int i, j ) : a( i ), b( a+j ) {}
};
```

В межах класу оголошення X x (1,1) призведе до привласнення числа 1 x::a і числа 2 x::b.

Конструктори базових класів викликаються перед конструюванням будь-яких елементів похідних класів. Значення похідного класу не можуть змінюватися і після цього впливати на створення базового класу.

```
class  base
        {
          int  x;
public :
  base ( int i ) : x ( i ) {}
};

class  derived : base
{
  int  a;
public :
```

derived (int i):a(i*10), base (a) {}// Зверніть увагу! base буде передане неініціалізоване a

};

У даному прикладі виклик похідного d(1) не призведе до привласнення елементу базового класу x значення 10. Значення, передане конструктору базового класу, буде невизначеним.

Якщо ви хочете мати список ініціалізаторів у невбудованому конструкторі, не розміщуйте цей список у визначенні класу. Замість цього розташуйте його у точку визначення функції:

```
derived::derived ( int i ) : a ( i ) {
 ....
}
```

4 Віртуальні функції

Віртуальні функції можуть бути тільки функціями-членами. В разі віртуальних функцій ви не можете просто змінити тип функції. Отже, неприпустимим є перевизначення віртуальної функції таким чином, щоб вона відрізнялася тільки типом повернення. Якщо дві функції з одним і тим самим ім'ям мають різні аргументи, C++ розглядає їх як різні функції, і механізм віртуальних функцій ігнорує.

Якщо базовий клас B містить віртуальну функцію vf(), а клас D, що є похідним від класу B, містить функцію vf() того ж типу, то якщо функція vf() викликається для об'єкту B або D, виконується виклик D::vf(), навіть якщо доступ визначений через покажчик чи посилку на B.

Наприклад :

```
struct B {
  virtual void vf1 ( );
  virtual void vf2 ( );
  virtual void vf3 ( );
  void f ( );
}

class D : public B {
  virtual void vf ( );  // специфікатор virtual припустимий, але зайвий
  void vf2 ( int ); // не virtual, оскільки тут використовується інший список аргументів
  char  f ( );          // так не можна : змінює тільки тип повернення void f ( );
};

void main ( ){
  D d;              // оголошення об'єкту D
  B* bp = &d;     // стандартне перетворення з D* на B*
  bp->vf1 ( );      // виклик D::vf1
  bp->vf2 ( );      // виклик B::vf2, бо vf2 із D має відмінні аргументи
  bp->f ( );        // виклик B::f ( не віртуальної )
}
```

Перевизначаюча функція vf1 в класі D автоматично стає віртуальною.

Специфікатор virtual може використовуватися в визначенні перевизначаючої функції в похідному класі, але насправді він є в даному випадку зайвим.

Інтерпретація виклику віртуальної функції залежить від типу того об'єкту, для якого вона викликається. В разі виклику невіртуальних функцій інтерпретація залежить тільки від типу покажчика чи посилки, що визначають об'єкт, для якого вона викликається.

Віртуальні функції повинні бути членами деякого класу, але вони не можуть бути статичними членами. Віртуальна функція може бути дружньою (friend) для іншого класу.

Віртуальні функції в базовому класі, як і всі функції-елементи базового класу, повинні визначатися або оголошуватися як функції без побічного ефекту ("чисті" функції).

5 Віртуальні деструктори

Деструктор може бути оголошений як віртуальний (virtual). Це дозволяє покажчику об'єкта базового класу викликати необхідний деструктор у випадку, коли покажчик фактично посилається на об'єкт похідного класу. Деструктор класу, похідного від класу з віртуальним деструктором, сам є віртуальним.

```
class  color {
  public :
     virtual  ~color ( );              // віртуальний  деструктор для  color
  };

class  red : public  color {
  public :
    ~red ( );                    // деструктор для  red також є віртуальним
  };
class  brightred :  public  red {
  public :
    ~brightred ( );           // деструктор для brightred також віртуальний
  };
```

Раніше перелічені у наступних оголошеннях класи:

```
color  *palette [ 3 ];
        palette [ 0 ]  =  new  red;
        palette [ 1 ]  =  new  brightred;
        palette [ 2 ]  =  new  color;
```

дасть наступні результати :

```
        delete   palette  [  0  ];  // Деструктор для   red викликається після деструктора для color
        delete  palette [ 1 ]; // Деструктор для  brightred викликається після  ~red і  ~color
        delete  palette [ 2 ]; // Запуск  деструктора для  color
```

Проте, якщо жоден з деструкторов не був оголошений віртуальним, delete palette [0], delete palette [1] і delete palette [2] викликають тільки деструктор для класу color. Це призведе до неправильного знищення перших двох елементів, що фактично мали тип red і brightred.

Контрольні питання до л.р. 3:

1. Який клас називають базовим?
2. Який клас називають похідним?
3. Які члени успадковує похідний клас з базового класу, а які ні?
4. Чи повинен конструктор похідного класу викликати конструктор базового класу, якщо так, то яким чином?
5. Що означає такий запис: public <базовий клас>?
6. Що означає такий запис: private <базовий клас>?
7. Що означає такий запис: protected <базовий клас>?
8. За допомогою яких дій можна скоригувати дію специфікаторів доступу в базовому списку?
9. З якими атрибутами повинні оголошуватись конструктори базового класу щоб забезпечити можливість їх виклику з похідного класу?
10. Які функції називають віртуальними?
11. Що називають поліморфізмом?

Завдання:

1. Створити базовий клас Car (машина), який характеризується торговою маркою (рядок), числом циліндрів, потужністю. Визачити методи перевизначення зміни потужності. Створити похідний клас Lorry (вантажна машина), який характеризується також вантажопідйомом кузова. Визначити функцію перевизначення марки і зміни вантажопідйомника.
2. Дано:

```
class base {
public: virtual void iam()
{
cout << base\n”;
};
```

Створіть два похідних класи від класу base і для кожного з них визначіть функцію iam() так, щоб вона виводила ім’я класу. Створіть об’єкти цих класів та викличте для них функцію iam(). Використовуючи покажчик base* на базовий клас, викличте функції iam() з похідних класів.

3. Розробіть клас, який здатний запам’ятовувати список об’єктів незаданого типу класу (для будь-якого класу).
4. На основі попередньої задачі напишіть програму, яка здатна запам’ятовувати та відображати зміст каталогу диску в вигляді списку рядкових об’єктів.

Завдання для самостійної роботи

1. Створити базовий клас робітник, потім розширити його, додавши менеджера. Функції-члени: виведення інформації про робітників та менеджерів.
2. Створити базовий клас з функціями для обробки цілих або десяткових чисел. Округлення для int - до десятків (34 = 30). Для double - до десятих (10,367 = 10,4). Відкинути залишок. Параметри функцій: дільник, кількість знаків після коми.

3. Написати програму-діалог, яка генерує код створення класу з успадкованими класами.

4. Базовий клас ліній (координати початку, координати кінця) визначає довжину лінії (сторони). Клас-спадкоємець – прямокутник за допомогою конструктора класу ліній та даних-членів (вершини початків та кінців сторін) визначає площу. Клас-спадкоємець – піраміда визначає об'єм піраміди. Дане-член – висота.

5.Створити клас Pair (пара чисел); визначити методи зміни полів і порівняння пар: пара p1 більше пари p2, якщо (first.p1>first.p2) або (first.p1= first.p2) і (second.p1>second.p2). визначити клас-нащадок Fraction з полями: ціла частина числа і дробова частина числа. Визначити повний набір методів порівняння.

6. Створити клас Liquid (рідина), який має поля назви і густини. Визначити методи перевизначення і змінення густини. Створити похідний клас Alcohol (спирт), який має міцність. Визначити методи перевизначення і змінення міцності.

7. Створити клас Pair (пара чисел); визначити методи змінення полів і обчислення множення чисел. Визначити похідний клас Rectangle (прямокутник) з полями-сторонами. Визначити методи обчислення периметра та площі прямокутника.

8. Створити клас Man (людина), з даними: ім'я, вік, стать і вага. Визначити методи перевизначення імені, віку і ваги. Створити похідний клас Student, який має додаткове дане: рік навчання. Визначити методи перевизначення і збільшення року навчання.

Лабораторна робота № 4

Тема: Дружні класи та функції (friend). Правила доступу до членів класу. Поліморфізм.

Мета: Вивчити техніку пізнього зв'язування на прикладі використання віртуальних функцій. Навчитися користуватися можливістю "дружби " з класом**.**

Теоретичні відомості

1 Дружні функції та дружні класи

"Друг" F класу X - це функція чи клас, що, не являючись функцією-елементом X, має тим не менш повні права доступу до елементів X private та protected. В усіх інших відносинах F - це звичайна з точки зору області дії, оголошень та визначень функція.

Оскільки F не є елементом X, вона не лежить в області дії X і тому не може викликатися операціями вибору x.F і xptr->f (де x - це об'єкт X, а xptr -це покажчик на об'єкт X).

Якщо в оголошенні чи визначенні функції в межах класу X використовується специфікатор friend, то така функція стає "другом" класу X.

Дружня функція, визначена в межах класу, підпорядковується тим самим правилам вбудовування, що і функції-елементи класу. Дружні функції не залежать від їх позиції у класі чи специфікаторів доступу. Наприклад:

```
class  X
{
int  i;                                  // private щодо X
friend void friend_func(X*,int);//friend_func не є private, хоч вона і оголошена у розділі private
  public:
void member_func (int);
 };
// визначення для обох функцій відзначимо доступ до  private  int  i
void  friend_func (X* xptr, int a)  { xptr->i = a; }
void  X::member_func (int a)  { i = a; }
X xobj;
// відзначимо різницю у викликах функцій
friend_func ( &xobj, 6 );
xobj.member_func ( 6 );
```

Ви можете зробити всі функції класу Y дружніми для класу X в одному оголошенні:

```
class  Y;              // неповне оголошення
class  X  {
    friend  Y;
            int  i;
    void  member_funcx ();
 };
class  Y  {
    void  friend_x1 (  X& );
```

```
    void  friend_x2 (  X* );
  ....
        };
```

Функції, що оголошені в Y, є дружніми для X, хоч вони і не мають специфікаторів friend. Вони мають доступ до приватних елементів X (private), таким як **i** та member_funcx.

Крім того, окремі функції-елементи класу X також можуть бути дружніми для класу Y:

```
class  X  {
  ....
    void  member_funcx ( );
 }
class  Y  {
    int  i;
    friend  void  X::member_funcx ( );
  ....
 };
```

"Дружність" класів не транзитивна: якщо X є дружнім для Y, а Y - дружній для Z, це не означає, що X - дружній Z. Однак, "дружність" успадковується

2 Чисті віртуальні функції

```
class B {
virtual void vf ( int ) = 0; // = 0
}
```

означає "чисту" функцію. В класі, похідному від такого базового класу, кожна "чиста" функція має бути визначена чи переоголошена в такій якості.

Якщо віртуальна функція визначена в базовому класі, то немає необхідності її перевизначення в похідному класі. При викликах буде просто викликана відповідна базова функція.

Віртуальні функції змушують певним чином розплачуватись за свою універсальність: кожен об'єкт похідного класу повинен містити покажчик на таблицю функцій з тим, щоб під час виконання програми викликати потрібну (пізнє зв'язування).

3 Абстрактні класи

Абстрактним називається клас із як мінімум однією чистою віртуальною функцією. Віртуальна функція задається як "чиста" за допомогою відповідного специфікатора.

Абстрактний клас може використовуватися тільки у вигляді базового класу для інших класів. Об'єкти абстрактного класу створюватися не можуть. Абстрактний клас не може використовуватися як тип аргументу чи як тип повернення функції. Проте, допускається оголошувати покажчики на абстрактний клас. Допустимі посилки на абстрактний клас за умови, що при ініціалізації не вимагається утворення тимчасового об'єкту.

Наприклад :

```
class shape {              // абстрактний клас
  point center;
...
public :
where ( ) { return center; }
  move ( point p ) { center = p; draw ( ); }
  virtual void rotate ( int ) = 0;   // " чиста " віртуальна функція
  virtual void draw ( ) = 0;         // " чиста " віртуальна функція
  virtual void hilite ( ) = 0 ;      // " чиста " віртуальна функція
          ...
};
shape x;          // помилка: спроба утворення об'єкту абстрактного класу
shape* sptr;      // покажчик на абстрактний клас припустимий
shape f ( );              // помилка: абстрактний клас не може бути типом повернення
int g ( shape s ); // помилка: абстр.клас не може бути типом аргументу функції
hape& h ( shape& );  // посилка на абстрактний клас у вигляді типу повернення
                     //або аргументу функції припустима
```

Припустимо, що клас D є похідним безпосередньо від абстрактного базового класу B. Тоді для кожної чистої віртуальної функції pvf в B, якщо D не забезпечує визначення для pvf, то pvf стає "чистою" функцією елементом D, а сам D буде абстрактним класом.

Наприклад, з використанням показаного вище класу shape :

```
class circle:public shape { // circle є похідним від абстрактного класу
  int radius;
public :
  void rotate ( int ) { }// визначається віртуальна функція: дії по обертанню кола відсутні
  void draw ( );   // circle::draw() має бути де-небудь визначена
};
```

Функції-елементи можуть оголошуватися в конструкторі абстрактного класу, але виклик чистої віртуальної функції безпосередньо чи непрямо з такого конструктора приводить до помилки під час виконання.

4 Віртуальні базові класи

При множинній спадкоємності базовий клас не може задаватися в похідному класі більш ніж один раз :

```
class B {... };
class D : B, B {... } : // неприпустимо
```

Проте, базовий клас можна передавати похідному класу більш одного разу непрямо :

```
class X : public B {... }
class Y : public B {... }
class Z : public X, public Y {... } // допустимо
```

В даному випадку кожен об'єкт класу Z буде мати два підоб'єкти класу B.

Якщо це створює проблеми, до специфікатору базового класу може бути додано ключове слово virtual.

Наприклад:

```
class X : virtual public B {... }
class Y : virtual public B {... }
class Z : public X, public Y {... }
```

Тепер B є віртуальним базовим класом, а клас Z має тільки один підоб'єкт класу B. В разі множинних базових класів :

```
class X : public Y, public Z {....};
X one;
```

конструктори викликаються у послідовності їх оголошення :

```
Y ( );        // першими слідують  конструктори базового класу
Z ( );
X ( );
```

Конструктори віртуальних базових класів запускаються до будь-яких невіртуальних базових класів. Якщо ієрархія містить множинні віртуальні базові класи, то конструктори віртуальних базових класів запускаються у послідовності їх оголошення. Після цього, перед викликом конструкторів похідного класу, конструюються невіртуальні базові класи.

Якщо віртуальний клас є похідним від невіртуального базового класу, то для того, щоб віртуальний базовий клас був сконструйований правильно, ця невіртуальная база повинна бути першою. Код:

```
class X : public Y, virtual public Z
X one;
```

дає наступну послідовність:

```
Z ( );        // ініціалізація  віртуального базового класу
Y ( );        // невіртуальний базовий клас
X ( );        // довільний клас
```

Або, у більш складному випадку,

```
class base;
class base2;
class level1 : public base2, virtual public base;
class level2 : public base2, virtual public base; class
toplevel : public level1, virtual public level1;
toplevel view;
```

Порядок конструювання перегляду view буде наступним:

base( ); // старший в ієрархії віртуальний базовий клас конструюється тільки раз

base2(); //невіртуальна база віртуальної бази level2 викликається для конструювання level2

```
level2 ( ); // віртуальний базовий клас
base2 ( ); // невіртуальна база для  level1
level1 ( ); // інша невіртуальна база
toplevel ( );
```

В разі, коли ієрархія класу містить множинні входження віртуального класу, даний базовий клас конструюється тільки один раз. Однак, якщо існують і віртуальні, і невіртуальні входження базового класу, то конструктор класу запускається один раз для всіх віртуальних входжень і один раз для всіх

невіртуальних входжень базового класу.

Контрольні питання до л.р. 4:

1. Які функції називають дружніми для будь-якого класу?
2. Які класи називають дружніми для будь-якого класу?
3. Що треба зробити, щоб оголосити клас дружнім іншому класу?
4. Які класи називають взаємно дружніми?
5. Які функції називають чистими віртуальними?
6. Який клас називають абстрактним?
7. Що таке множинна спадкоємність?
8. Що таке віртуальні базові класи?
9. Як викликаються конструктори віртуальних базових класів?

Завдання:

1. Напишіть код, що моделює роботу двигуна внутрішнього згоряння. Придумайте 2 класи, один з іменем TEngine (двигун), інший з іменем TFuel (пальне), в якому є закритий дійсний член подвійної точності level, що показує, скільки пального залито в баки. За допомогою друзів оголосіть ваші класи так, щоб клас TEngine мав безпосередній доступ до закритого члена level класу TFuel.
2. Використовуючи множинну спадкоємність напишіть клас, який здатен відобразити зміст каталога та запам’ятати рядок, котрий містить спецсимволи. Наприклад, ініціалізація нового об’єкта-каталога рядком “*.cpp” призведе до відображення всіх файлів з розширенням .cpp, що розміщуються в поточному каталозі, та до запам’ятовування самого цього рядка для наступних посилань.
3. Напишіть кілька зв’язаних між собою класів для програми, яка вводить дані з екрану, на зразок бази даних, що містить імена та адреси. Створіть класи, які запитують та повертають значення різних типів даних у визначеному місці екрану.
4. Створити абстрактний базовий клас Figure з віртуальними методами вирахування площі і периметра. Створити похідні класи: Rectandle (прямокутник), Circle (коло), Trapezium (трапеція) зі своїми функціями площі і периметра. Самостійно визначити, які поля необхідні, які з них можна задати в базовому класі, а які – в похідних. Площа трапеції: S=(a+b)*h/2.

Завдання для самостійної роботи

1. Створити абстрактний базовий клас Body (тіло) з віртуальними функціями обчислення площі поверхні і об’єма. Створити похідні класи: Parallelepiped (параллелепіпед) і Ball (куля) зі своїми функціями площі поверхні та об’єму.
2. Створіть базу даних зі 100 випадкових цілих значень за допомогою класу TDataBase. Відобразьте значення об’єктів в 8-ми символьних стовбчиках.
3. Створити абстрактний базовий клас Currency (валюта) для роботи з грошовими сумами. Визначити віртуальні функції переведення в гривні і

виведення на екран. Реалізувати похідні класи: Dollar (долар) і Euro (євро) з своїми функціями переведення і виведення на екран.

4. Створити абстрактний базовий клас Triangle для представлення трикутника з віртуальними функціями обчислення площі і периметра. Поля даних повинні включати дві сторони і кут між ними. Визначити класи-нащадки: прямокутний трикутник, рівнобедрений трикутник, рівносторонній трикутник зі своїми функціями обчислення площі і периметра.

5. Створити абстрактний базовий клас Root (корінь) з віртуальними методами обчислення коренів чисел і виведення результату на екран.

Лабораторна робота № 5

Тема: Перевантаження операцій

Мета: Навчитися перевантажувати операції **+ , -,*, /, ( ), [ ]** та оператори, визначені користувачем.

Теоретичні відомості

C++ дозволяє перевизначати дію більшості операцій, щоб при використанні з об'єктами конкретного класу вони виконували задані функції. Як і в разі перевизначених функцій C++ в цілому, компілятор визначає різницю в функціях за контекстом виклику, за числом і типом аргументів операндів.

Перевизначення (перевантаження) можна виконати для всіх операцій, за винятком таких операцій : . : : ? :

Також не можна перевизначати символи препроцесора # і ##.

Ключове слово operator, за яким слідує символ операції, називається ім'ям функції-операції. При визначенні нової (перевизначеної) дії операції воно використовується як звичайне ім'я функції.

Операція-функція, що викликається з аргументами, поводить себе як операція, що виконує певні дії з операндами. Операція-функція не може змінювати число аргументів чи правила пріоритету та асоціативності операцій порівняно з її нормальним використанням. Розглянемо клас complex:

```
class complex {
  double real, imag;     // за замовчуванням private
public :
                    ...
  complex ( ) { real = imag = 0; }     // вбудований конструктор
  complex ( double r, double i = 0 ) { // ще один конструктор
   real = r; imag = i;
  }
  ...
};
```

Примітка: Даний клас був створений тільки для завдань ілюстрації. Він відрізняється від класу complex з бібліотеки підтримки.

Ми можемо легко розробити функцію для додавання комплексних чисел, наприклад:

```
complex Addcomplex ( complex c1, complex c2 );
```

проте буде більш природним мати можливість записати :

```
complex c1 (0, 1), c2 (1, 0), c3;
c3 = c1 + c2;
```

замість:

```
c3 = Addcomplex (c1,c2);
```

Операцію + можна легко перевизначити, якщо включити до класу complex наступне оголошення:

```
friend complex operator + (complex c1, complex c2);
```

і визначити його (можливо, inline) таким чином :

```
complex operator + (complex c1, complex c2){
return complex (c1.real + c2.real, c1.imag + c2.imag );
```

}

Операції-функції можна викликати безпосередньо, хоч зазвичай вони викликаються непрямо, при використанні перевизначених (перевантажених) операцій:

```
c3 = c1.operator+(c2);    // те  ж саме, що c3 = c1 + c2
```

На відміну від від new і delete, що мають свої власні правила, операція-функція має бути або нестатичною функцією-членом, або мати як мінімум один аргумент типу класу. Операції-функції =,( ),[ ] і -> повинні бути нестатичними функціями-членами. За винятком операції-функції привласнення = ( ) всі операції-функції для класу X успадковуються класом, похідним від X, згідно зі стандартними правилами дозволу для перевизначених функцій. Якщо X є базовим класом для Y, перевизначену операцію для класу X можна далі перевизначити для класу Y.

Перевизначення new і delete

Операції new і delete можуть перевизначатися таким чином, щоб давати альтернативні варіанти управління вільною пам'яттю (динамічною областю). Якщо визначається користувачем операція new, вона повинна повертати void* і мати в якості першого аргументу size_t. Якщо визначається користувачем операція delete, вона повинна мати тип void і перший аргумент void*. Другий аргумент типу size_t не є обов'язковим.

Тип size_t визначається в файлі stdlib.h.

Наприклад:

```
#include <stdlib. h>
class X {
...
public:
  void* operator new (size_t size ) {
        return newalloc (size );
  }
  void operator delete (void* p) {
        newfree (p);
  }
  X ( ) {              /* ініціювання*/}
  X (char ch) {       /* тут теж/ }
  ~X ( ){             /* очищення*/ }
  ...
};
```

Аргумент size задає розмір об'єкту, що створюється, а newalloc і newfree визначаються користувачем функції розподілу та знищення розподілу пам'яті. Виклики конструктора і деструктора для об'єктів класу X (або об'єктів, похідних від X, для яких не існує власних перевизначених операцій new і delete) призведе до запуску відповідних операцій, що визначаються користувачем X::operator new ( ) і X::operator delete ( ), відповідно.

Операції-функції X::operator new( ) і X::operator delete( ) є статичними елементами X, як при явному оголошенні їх static, так і без нього, тому вони не можуть бути віртуальними функціями.

Стандартні, визначені (глобальні) операції new і delete можуть при цьому

також використовуватися в області дії X, як явно з операціями глобальної області дії (::operator new і ::operator delete), так і неявно, при утворенні і руйнуванні об'єктів класів, відмінних від класу X і що не є похідними від класу X. Наприклад, можна використовувати стандартні операції new і delete при визначенні перевантажених версій:

```
void* X: : operator new ( size_t ){
  void* ptr = new char [ s ];  // виклик стандартної new...
  return ptr;
}

void X::operator delete ( void* ptr ){
  ...
  delete ( void* ) ptr;          // виклик стандартної delete
}
```

Причиною того, що розмір аргументу визначається size, є те, що класи, похідні від X, успадковують X::operator new. Розмір об'єкту похідного класу може істотно відрізнятися від розміру, що визначається базовим класом.

Перевизначення унарних операцій

Перевантаження префіксної або постфіксної унарної операції виконується за допомогою оголошення нестатичної функції-члену, функції, яка не приймає ніяких аргументів, або за допомогою оголошення функції, що приймає один аргумент. Якщо @ становить унарну операцію, то @x і x@ можна інтерпретувати як x.operator@( ) і operator@(x), відповідно, залежно від оголошення. Якщо оголошення було зроблено в обох формах, то й дозвіл неоднозначності залежить від переданих при виклику операції стандартних аргументів.

В C++ пізніх версій, коли operator++ або operator-- оголошується як функція-член без параметрів або як функція, що не є членом, з одним параметром, вона лише перевизначає префіксну операцію ++ або операцію --. Постфіксну операцію ++ або операцію -- можна перевизначити лише визначивши її як функцію-елемент із одним параметром int або як функцію, що не є членом класу, яка приймає один клас і один параметр int. Додамо, наприклад, до попереднього прикладу такі рядки :

```
operator++ ( int );
operator-- ( int );
```

Коли перевизначається тільки префіксний варіант operator++ або operator-- і операція застосовується до об'єкта класу як постфіксна, то компілятор видає попередження. Після цього він викликає префіксну операцію, дозволяючи компілювати коди версії 2.0. В результаті компіляції поперднього прикладу видаються такі попередження:

```
Warning : Overloaded prefix 'operator ++' used as postfix
operator in function func ( )
" Перевизначена префіксна операція 'operator ++'  використовується в функції func ( ) як постфіксна операція "
Warning : Overloaded prefix 'operator --' used as postfix
operator in function func ( )
" Перевизначена префіксна операція 'operator --'  використовується в функції func ( ) як постфіксна операція "
```

Перевизначення бінарних операцій

Перевизначення бінарної операції виконується за допомогою оголошення нестатичної функції-члена класу, що приймає один аргумент, або за допомогою оголошення функції не члена класу (зазвичай friend), що приймає два аргументи. Якщо @ є бінарною операцією, то x@y можна інтерпретувати або як x.operator@ (y), або як operator@ (x, y), в залежності від виконаних оголошень. Якщо оголошені обидві форми, то дозвіл неоднозначності залежить від переданих при виклику операції стандартних аргументів.

Перевизначення операції привласнення =

Операцію присвоєнння = можна перевизначити тільки за допомогою оголошення нестатичної функції-члена. Наприклад:

```
class String {
        ......
  String& operator = ( String& str );
        ......
  String ( String& )
  ~String ( );
}
```

Даний код, спільно з оголошенням String::operator=( ), дозволяє виконувати рядкові привласнення str1 = str2, як це робиться в інших мовах програмування. На відміну від інших функцій-операцій, функція-операція привласнення не може успадковуватися похідними класами. Якщо для будь-якого класу X не існує операції, що визначається =, то операція = визначається за замовчуванням як поелементне привласнення елементів класу X :

```
X& X::operator = ( const X& source ){    // поелементне привласнення     }
```

Перевизначення операції виклику функції ( )

Виклик функції:

первісний_вираз (<список_виразів>)

розглядається у вигляді двійкової операції з операндами "первісний_вираз " і "список_виразів" (можливо, порожній). Відповідна функція-операція - це operator(). Надана функція може визначатися користувачем для класу X (і будь-яких похідних класів) тільки у вигляді нестатичної функції-члена. Виклик x(arg1,arg2), де x становить об'єкт класу X, інтерпретується в такому випадку як x.operator( )(arg1, arg2).

Перевизначення операції індексування [ ]

Аналогічним чином, операція індексування:

первісний_вираз [вислів]

розглядається як двійкова операція з операндами "первісний_вираз" і "вислів". Відповідна операція-функція - це operator[ ]. Вона може визначатися користувачем

для класу X (і будь-яких похідних класів) тільки шляхом нестатичної функції-члена. Вислів x[y], де x є об'єктом класу X, інтерпретується в даному випадку як x.operator[ ] (y).

Перевизначення операції доступу до елементу класу ->

Доступ до елементу класу за допомогою :

первісний_вираз -> вираз

розглядається як унарна операція. Функція operator-> повинна розглядатися як нестатична функція-член. Вислів x -> m, де x - це об'єкт класу, інтерпретується як (x.operator -> ( ))->m, таким чином, що operator-> ( ) повинен або повертати покажчик на об'єкт даного класу, або повертати об'єкт класу, для якого визначається operator->.

Контрольні питання до л.р. 5:

1. Яку можливість відкриває перевантаження операторів?
2. Які оператори не можуть перевантажуватись?
3. Якими двома способами можна оголосити перевантаження операторів?
4. Що відбувається, якщо функція, яка перевантажує оператор є дружньою функцією?
5. Скільки параметрів матиме перевантажена функція, якщо вона є членом класу, об’єктами якого оперує ?
6. Як може бути визначена унарна операція?
7. Що служить ознакою того, що функція викликається для операції у постфіксному варіанті?
8. Як можна визначити свої власні правила перетворення типів?
9. Які існують правила перевантаження операторів?

Завдання:

1. Перевантажити оператори + для комплексних чисел:

А)Зробити функцію перевантаження оператору “+” дружньою класу.

Б) Зробити функцію перевантаження оператору “+” - членом класу.

2. Створити 3 класи, що описують одиниці вимірювання температури за Цельсієм, Фаренгейтом та Кельвіном. Реалізувати перетворювач одиниць вимірювання температури, використовуючи перевантаження операторів для **приведення типів**.

3. Для класу String перевантажити операції + += == [ ] таким чином, щоб забезпечувалося виконання таких операцій:

String+String
String+”Text”
“Text”+String
String += “Text”
String += ‘c’
String==“Text”
String[Pos]

Завдання для самостійної роботи:

1. Реалізувати клас TInteger, який дозволяв будувати та використовувати великі цілі числа > 32 біт, використовуючи оператори =, +, ++, +=, -, --, -=, *, *=, /, /=, <<, >>, <, >, ==, !=, <=, >=, |, |=, &, &=, ~.

```
class TInteger: {
    void TInteger(int Size);      //Size - розмір цілого числа в байтах
    TInteger& operator+= ( TInteger );
    TInteger& operator+= ( Int );
    TInteger& operator+  ( TInteger );
    TInteger& operator+  ( Int );
        .....
}
```

2. Визначити тип Vec4 як вектор із чотирьох елементів типу float. Перевантажити оператор [ ] для Vec4. Визначити оператори + ,-, * ,/, = ,+=,-= для комбінацій векторів та чисел з плаваючою крапкою.

3. Визначити клас Vector, аналогічний Vec4, розмір якого передається як аргумент конструктору Vector::vector(int).

4. Визначити клас Mat4 як вектор з чотирьох Vec4. Для цього класу визначіть operator[ ], який повертає Vec4. Для цього типу перевантажити звичайні матричні операції.

5. Визначити клас Matrix, аналогічний Mat4, розміри якого передаються як аргументи конструктору Matrix::matrix(int,int).

6. Створити клас Money для роботи з грошовими сумами. Число повинно бути представлено двома полями: типу long для гривень і типу unsigned char – для копійок. Дробова частина (копійки) при виведенні на екран повинна бути відокремленя від цілої частини комою. Реалізувати додавання, віднімання та ділення сум.

7. Раціональний (нескорочуваний) дріб представляється парою чисел (a,b), де a – чисельник, b – знаменник. Створити клас Rational для роботи з раціональними дробами. Обов’язково повинні бути реалізовані наступні операції:

- Додавання add, (a,b) + (c,d) = (ad+bc, bd);
- Віднімання sub, (a,b) – (c,d) = (ad–bc, bd);
- Множення mul, (a,b) x (c,d) = (ac, bd);
- Ділення div, (a,b) /(c,d) = (ad, bc);

8. Створити клас Fraction для роботи з дробовими числами. Число повинно бути представлено двома поля: цілою частиною – довге ціле зі знаком, дробовою частиною – беззнакове коротке ціле. Реалізувати арифметичні операції: додавання, віднімання, множення і операції порівння.

Лабораторна робота № 6

Тема: Шаблони. Потоки C++.

Мета: Набути навичок при користуванні шаблонними класами. Оволодіти вмінням користуватися потоками C++.

Теоретичні відомості

Шаблони, також називаються родовими або параметризованими типами, дозволяють створювати родину зв'язаних функцій чи класів. В цьому розділі ми спочатку наведемо базовий принцип, а після цього разглянемо деякі специфічні моменти.

Синтаксис:

оголошення_шаблону :
    шаблон <список_аргументів_шаблону> оголошення

список_аргументів_шаблону :
    аргумент_шаблону
    <список_аргументів_шаблону>, аргумент_шаблону

аргумент шаблону:
    аргумент_типу
    оголошення_аргументу

список_аргументів_шаблону :
    аргумент_шаблону
    <список_аргументів_шаблону>, аргумент_шаблону

аргумент шаблону:
    аргумент_типу
    оголошення_аргументу

аргумент_типу:
    class ідентифікатор

ім`я_класу_шаблону:
    і`мя_шаблону <список_арг_шаблону>

список_арг_шаблону:
    арг_шаблону
    список_арг_шаблону, арг_шаблону

арг_шаблону:
    вираз
    ім`я_типу

Шаблони функцій

Розглянемо функцію max(x,y), котра повертає більший з двох своїх аргументів. x і y можуть мати тільки той тип, який володіє властивістю порівняння. Але він очікує на те, що типи параметрів x і y будуть оголошені під час компіляції. Без використання шаблонів потрібні численні перевизначення версій функцій, по одній для кожного підтримуємого типу даних, причому навіть в тому випадку, якщо коди для кожної версії практично ідентичні.

Наприклад :

```
int max ( int x, int y )

      { return ( x > y )? x : y;

    }
long max ( long x, long y )

      { return ( x > y )? x : y;

    }....
```

(далі йдуть інші версії функції max).

Одним із засобів роз’вязання цієї проблеми є використання макрокоманд :

```
#define max ( x, y ) ( ( x > y )? x : y )
```

Проте, в разі використання #define обминається механізм перевірки типів, котрий дає перевагу C++ над Ci. Насправді, таке використання макрокоманд в C++ майже не використовується. Ясно, що max (x, y) потрібна для порівння сумісних типів. На жаль, використання макрокоманди допускає порівняння між int і struct, тобто, типами, які несумісні.

Іншою проблемою, що виникає в разі макрокоманд, є підстановка, яка виконується тоді, коли вона небажана :

```
class Foo
     {
public :
   int max ( int, int );  // дає синтаксичну помилку; проходить розширення!!
};
```

Якщо ж використовується шаблон, то ви можете визначити зразок для родини зв'язаних перевизначених функцій, дозволивши типу даних бути параметром:

```
template <class T>              // Визначення
T max ( T x, T y )
```

```
{                                   // шаблону
  return ( x > y )? x : y;
};                                  // функції
```

Тип даних представлений аргументом шаблону <class T>. При використанні в програмі компілятор генерує потрібну функцію в відповідності з типом даних, реально використовуваним при викликові:

```
int i;
Myclass a, b;
int j = max ( i, 0 );               // аргументи - цілі
Myclass m = max ( a, b );           // аргументи мають тип Myclass.
```

У вигляді <class T> може використовуватися будь-який тип даних (не тільки клас). Компілятор сам викликає потрібну операцію operator> (), так що ви можете використовувати max з аргументами будь-якого типу, для якого визначена операція operator> ( ).

Перевизначення шаблонної функції

Попередній приклад називається шаблоном функції (або узагальненою функцією). Конкретна реалізація шаблону функції називається шаблонною функцією. Ви можете перевизначити генерацію шаблонної функції для конкретного типу за допомогою нешаблонної функції :

```
#include <string. h>
char *max ( char *x, char *y )
        {
return ( strcmp ( x, y ) >0 )? x : y;
        }
```

Якщо ви викликаєте цю функцію з рядковими аргументами, вона виконується замість автоматичної шаблонної функції. В цьому випадку виклик функції дозволяє уникнути безглуздого порівняння двох покажчиків.

Шаблонні функції, згенеровані компілятором, виконують тільки тривіальні перетворення аргументів.

Тип (типи) аргументів шаблонної функції повинні використовувати всі формальні аргументи шаблону. Якщо це не так, то при викликові функції не можна визначити фактичні значення для невикористовуємих.

Явні і неявні шаблонні функції

При виконанні дозволу перевизначення (після кроків по пошуку точного збіга) компілятор ігнорує шаблонні функції, згенеровані неявно компілятором.

```
template<class T> T max ( T a, T b )
        {
 return ( a > b )? a : b;
       }
```

```
    void f ( int i, char c )
          {
max ( i, i );              // викликає max ( int, int )
max ( c, c );       // викликає max ( char, char )
max ( i, c );       // викликає max ( int, char )
max ( c, i );       // викликає max ( char, int )
        }
```

Цей код дає такі повідомлення про помилки :

Could not find a match for 'max (int, char) ' in function f (int, char)

Could not find a match for 'max(char,int) ' in function f (char, int)

" Не знайдено відповідність для... в функції... "

Якщо користувач явно визначає шаблонну функцію, то ця функція, з іншого боку, буде повністю брати участь у дозволі перевизначення.

Наприклад :

```
template<class T> T max ( T a, T b )
          {
               return ( a > b )? a : b;
           }
int max ( int, int );      // явно оголошує max ( int, int )
    void f ( int i, char c )
          {
               max ( i, i );     // викликає max ( int, int )
               max ( c, c );    // викликає max ( char, char )
               max ( i, c );     // викликає max ( int, char )
               max ( c, i );     // викликає max ( char, int )
          }
```

Шаблони класів

Шаблон класу (що також називається родовим класом або генератором класів) дозволяє вам задавати зразок для визначень класів. Гарними прикладами в цьому сенсі є родові класи container. Розглянемо наступний приклад класу векторів (одномірний масив). Коли ви маєте вектор із цілих чисел чи даних будь-якого іншого типу, базові операції, що виконуються з типом, є одними і тими ж (вставлення, вилучення, індексуваня і т.і.). В разі типу елементу, розглядаємого як параметр type для класу, система буде на ходу генерувати визначення класу, надійного за типами:

```
#include <iostream. h>
template <class T>              // Визначення
class Vector
{                                // шаблону класу
  T *data;
  int size;
 public :
   Vector ( int );
   ~vector ( ) {delete [ ] data; }
   T& operator [ ] ( int i ) {return data [ i ]; }
};
```

```
// Зверніть увагу на синтаксис для зовнішніх визначень
    template <class T>
    Vector<t> : : Vector ( int n )
            { data = new T [ n ];
      size = n;
        }
main ( )
            {
Vector<int> x ( 5 );       // Згенерувати вектор із цілих
       for ( int i = 0; i < 5; ++i )
         x [ i ] = i;
       for ( i = 0; i < 5; ++i )
         cout << x [ i ] << ' ';
       cout << '\n';
       return 0;
        }  // на виході буде 0 1 2 3 4
```

Як і в разі шаблонів функцій, для перевизначення автоматичного визначення для даного типу може бути реалізовано явне визначення шаблонного класу :

```
    class Vector<char *> {... };
```

Символ Vector завжди повинен супроводжуватись типом даних в наріжних дужках. Він не може з'являтися окремо, за винятком деяких випадків у визначенні оригінального шаблону.

Повна реалізація класу векторів знаходиться в файлі vectimp.h в вихідних кодах бібліотеки класів container. ці коди знаходяться в підкаталозі \borland\classlib\include.

Аргументи

Хоча в цих прикладах використовується тільки один шаблонний аргумент, допускається і декілька аргументів. Крім типів даних, шаблонні аргументи можуть визначатись також як значення :

```
    template<class T, int size = 64> class Buffer {... };
```

Шаблонні аргументи, що не є типами, такі як size, можуть мати аргументи, приймаємі за замовчуванням. Значенням для таких аргументів може бути вираз-константа :

```
    const int N = 128;
    int i = 256;
    Buffer<int, 2*n> b1;          // правильно
    Buffer<float, i> b2;          // помилка : і не є  константою
```

Так як кожне предписання значення для шаблонного класу насправді є класом, то він приймає свою власну копію статичних елементів. Аналогічним чином, шаблонні функції одержують свої власні копії статичних локальних змінних.

Наріжні дужки

Треба дотримуватися акуратності при використанні правої наріжної дужки при предписанні значення :

```
    Buffer<char, ( x > 100? 1024 : 64 ) > buf;
```

В цьому прикладі, якщо опустити круглі дужки навколо другого аргумента, то > між x і 100 буде достроково зачиняти список шаблонних аргументів.

Родові списки, надійні за типами

В загальному випадку, коли треба запрограмувати безліч дуже схожих речей, згадайте про шаблони. Проблеми з перевизначенням наступного класу, класу родових списків:

```
class Glist
     {
public :
   void insert ( void );
   void *peek ( );     //....
  };
```

є в тому, що він не є надійним за типами і загальні рішення потребують повторних визначень класу. Оскільки перевірка типів при вставленні відсутня, то ви не маєте засобу довідатися, що ж ви одержите назад. Проблему надійності за типами можна вирішити, створивши клас повертання

```
class Foolist: public Glist
     {
public :
   void insert ( Foo *f ) { Glist : : insert ( f );
}
   Foo *peek ( )
{ return ( Foo ) Glist : : peek ( ); }
     //....
  };
```

Цей клас є надійним за типами. insert буде лише брати аргументи типу покажчик_на_foo або об'єкт_отриманий_із_foo, тому внутрішній контейнер буде тільки вміщувати покажчики, котрі самі зазначають, що той тип є Foo. Це означає, що приведені типи Foolist::peek завжди безпечні і створюють правильний Foolist. Тепер, щоб зробити те ж саме для Barlist, Bazlist і т.д., потрібні повторні визначення окремих класів. Для вирішеня проблеми повторних визначень класів з точки зору безпеки типів, знов використаємо шаблони.

```
template <class T> class List : public Glist
     {
public :
  void insert ( T *t ) { Glist : : insert ( t ); }
  T *peek ( ) {return ( T ) Glist : : peek ( ); }
    //...
  }
List<foo> flist;   // створити клас Foolist і екземпляр з  ім'ям flist
List<bar> blist; // створити клас Barlist і екземпляр з ім'ям blist
List<baz> zlist; // створити клас Bazlist і екземпляр з ім'ям zlist
```

Виняток покажчиків

Зручним засобом є включення фактичних об'єктів, що робить непотрібними

покажчики, а також зменшує число викликів віртуальних функцій, бо компілятор знає фактичні типи об'єктів. Цей засіб дає велику користь, якщо віртуальні функції достатньо малі для того, щоб бути ефективно вбудованими. При виклику через покажчики вбудуваних функцій компілятор не знає фактичні типи об'єктів, що зазначаються.

Нижче наводиться визначення шаблону, що виключає покажчики :

```
template <class T> abase
        { //...
  private
   T buffer;
};
class anobject : public asubject, public abase<afilebuf>
        { //...
};
```

Всі функції в abase можуть викликати функції, визначенні в afilebuf, безпосередньо, без необхідності звертання до покажчика. Якщо якась із функцій в afilebuf може бути вбудована, то ви одержите великий вигриш по швидкості, бо шаблони допускають, щоб вони були вбудованими.

Потоки. Загальні положення

Потоки введення-виведення в C++ (зазвичай позначаються як iostreams, але найчастіше – як streams) забезпечують всі можливості, що надаються в мові Сі бібліотекою stdio.h. Вони використовуються для перетворення типізованих об'єктів на текст, що читається, і навпаки. Потоки можуть також читати і записувати двійкові дані. Мова C++ дозволяє визначати або перевизначати функції введення-виведення і оператори, що після цього автоматично використовуються стосовно відповідних визначених користувачем типів.

Що таке потік?

Потоком називається абстрактне поняття, що відноситься до будь-якого переносу даних від джерела (або постачальника даних) до приймача (або споживача) даних. Коли мова йде про введення символів від джерела, використовуються також синоніми витягнення, прийом та одержання і вставлення, розміщення чи запам’ятовування, коли мова йде про виведення символів у приймач. У якості джерел та приймачів даних (або і того і іншого) існують класи для підтримки консольного виведення (constrea.h), буферів пам'яті (iostream.h), файлів (fstream.h) і рядків (strstream.h).

Бібліотека iostream

Бібліотека iostream (визначена у файлі iostream.h) містить дві паралельні родини класів: класи, що є похідними (породженими) зі streambuf, та класи, похідні з ios. Обидва ці класи є класами нижнього рівня, і кожен з них виконує різний набір задач. Один із цих двох класів є базовим класом для всіх класів потоків. Доступ із класів, що базуються на ios, до класів, що базуються на streambuf, здійснюється через покажчик.

Виведення

Потокове виведення здійснюється за допомогою операції вставлення, чи занесення, <<. Для операцій виведення перевизначається стандартна операція зсуву ліворуч <<. Її лівий операнд становить об'єкт типу ostream. Правий операнд може мати будь-який тип, для якого визначене виведення потоком (тобто, фундаментальний тип чи будь-який з перевизначених для нього типів). Наприклад:

```
cout << " Hello! \n ";
```

записує рядок " Hello! " у cout (стандартний потік виведення, що зазвичай спрямований на екран), після чого слідує новий рядок.

Операція << володіє асоціативністю зліва і повертає посилку на об'єкт ostream, для якого вона викликалася. Це дозволяє організувати каскадні вставлення:

```
int i = 8;
double d = 2. 34;
 cout << " i= " << i << ", d= " << d << " \n ";
```

Це викличе виведення чого-небудь на зразок:

i = 8, d = 2. 34

на стандартний пристрій виведення.

Вбудовані типи

До вбудованих типів даних, що підтримуються безпосередньо, відносяться: char, short, int, long, char* (що розглядається як рядок), float, double, long double та void*. Цілочисельні типи перетворюються за правилами, за замовчуванням діючим для функції printf (якщо тільки ці правила не змінені шляхом встановлення різноманітних прапорців ios). Наприклад, якщо задані оголошення int i; long l;, то наступні два оператори:

```
cout << i << " " << l;
printf ( " %d %ld, i, l );
```

призведуть до одного і того ж результату.

Вставлення покажчику (void*) також предвизначене:

```
int i = 1;
cout << &i;      // покажчик виводиться на дисплей у шістнадцятковому форматі
```

Інші функції виведення описані в класі ostream.

Форматування виведення

Форматування введення і виведення визначається різноманітними прапорцями станів формату, переліченими у класі ios. Прапорці формату визначаються наступним чином:

```
public :
  enum
{
skipws,     // пропуск пробільного символу на введенні
left,       // виведення з лівим вирівнюванням
```

```
    right,      // виведення з правим вирівнюванням
    internal,   // заповнювач після знаку чи покажчика системи числення
    dec,        // десяткове перетворення
    oct,        // вісімкове перетворення
    hex,        // шістнадцяткове перетворення
    showbase,   // показати на виході покажчик системи числення
    showpoint,  // показати позицію десяткової точки (на виході)
    uppercase,  // виведення шістнадцякових значень літерами верхнього регістру
    showpos,    // показати знак "+" для позитивних чисел
  scientific,//використовувати запис чисел з плаваючою крапкою з виведенням
  експоненти E
        // наприклад,  12345E2
    fixed, // використовувати запис чисел з плаваючою крапкою типу 123.45
    unitbuf, // скидання на диск всіх потоків після вставлення
    stdio, // скидання на диск stdout і stderr після вставлення
      };
```

Ці прапорці читаються і встановлюються функціями елементами flags, setf та unsetf.

Заповнення і доповнення

Символ-заповнювач і напрямок доповнення залежать від встановлення внутрішніх прапорців, відповідаючих за ці параметри.

За замовчуванням символом-заповнювачем є прогалина. Змінити це значення за замовчуванням дозволяє функція fill:

```
int  i = 123;
cout.fill ("*");
cout.width (6);
cout  <<  i;                  //  на дисплеї буде виведено  ***123
```

За замовчуванням, встановлюється вирівнювання по правому краю (доповнення символами-заповнювачами ліворуч). Ці замовчування (а також інші форматні прапорці) можна змінювати за допомогою функцій setf та unsetf:

```
int  i  = 56;
....
cout.width (6);
cout.fill ('#');
cout.setf ( ios::left, ios::adjustfield );
cout  <<  i;          //  на дисплей буде виведено  56####
```

Другий аргумент, ios::adjustfield, повідомляє setf, що біти повинні встановлюватися. Перший аргумент, ios::left, повідомляє setf, у які саме значення встановлюються ці біти. Альтернативно можна використовувати маніпулятори setfill, setiosflags і resetiosflags, що дозволяють модифікувати символ-заповнювач та напрямок доповнення при форматуванні. Список масок, використовуваних setf, наводиться в описі елементів даних ios.

Введення

Введення потоком аналогічне виведенню, але використовує перевизначену

операцію зсуву праворуч, >>, і називається операцією витягнення, або витягненням. Лівий операнд операції >> є об'єктом типу класу istream. Як і для виведення, правий операнд може бути будь-якого типу, для якого визначене виведення потоком.

За замовчуванням операція >> опускає пробільні символи (як визначено функцією isspace у ctype.h), а після цього зчитує символи, що відповідають типу об'єкту введення. Пропущення пробільних символів керується прапорцем ios::skipws у перелічуваній змінній стану. Прапорець skipws зазвичай встановлює пропущення пробільних символів. Очищення цього прапорця (наприклад, за допомогою setf) вимикає пропущення пробільних символів. Визначимо також спеціальний маніпулятор "приймача", ws, що дозволяє ігнорувати пробільні символи.

Розглянемо наступний приклад:

```
int  i;
double  d;
cin >> i >> d;
```

Останній рядок викликає пропущення пробільних символів. Цифри, що зчитуються зі стандартного пристрою введення (за замовчуванням це клавіатура), перетворюються після цього на внутрішній двійковий формат і записуються у змінну **i**. Після цього знов пропускаються пробільні символи, і нарешті зчитується число з плаваючою точкою, що перетворюється і записується у змінну **d.**

Для типу char (signed або unsigned) дія операції >> складається у пропущенні пробільних символів і запису наступного (непробільного) символу. Якщо вам потрібно прочитати наступний символ, незважаючи на те, чи є він пробільним, чи ні, то можна використовувати одну з функцій елементів get.

Для типу char* (що розглядається як рядок) дія операції >> складається у пропущенні пробільних символів і запису наступних (непробільних) символів доти, доки не зустрінеться наступний пробільний символ. Після цього додається завершуючий нульовий (0) символ. Треба бути обережним і уникати "переповнення" рядка. Ширина за замовчуванням, дорівнює нулю (означає, що граничне значення не задане), може бути змінена за допомогою setw наступним чином:

```
char array [Size];
cin.width ( sizeof (array) );
cin >> array;                    // дозволяє уникнути переповнення
```

В разі будь-якого введення вмонтованих типів, якщо кінець введення зустрінеться раніше першого непробільного символу, у приймач buf нічого записано не буде, а стан istream буде встановлено рівним "відмові". Таким чином, якщо приймач не був ініціалізований, то він і залишиться неініціалізованим.

Введення типів, визначених користувачем

Для введення або виведення своїх власних, визначених вами типів, ви повинні перевизначити операції вилучення та вставлення. Це можна зробити наступним чином:

```
#include  <iostream.  h>
struct  info
{
  char  *name;
```

```
        double  val;
        char  *units;
  };
//Ви можете перевизначити << для виведення наступним чином:
ostream&operator<<(ostream& s, info& m)
        {
          s << m.name << " " << m.val << " " << m.units;
         return s;
       };
// Ви можете перевизначити >> для введення наступним чином:
istream&operator>>(istream& s, info& m)
        {
          s >> m.name >> m.val >> m.units;
          return s;
  };

 main ( )
 {
  x.name = new char [15];
  x.units = new char [10];
  cout << "\n input name, value and units: ";
  cin >> x;
  cout << "\n my input: " << x;
  return 0;
 }
```

Контрольні питання до л.р. 6:

1. Що можна назвати шаблоном?
2. Як оголошують шаблони функцій?
3. Що представляє собою шаблон класу?
4. Який клас називають контейнерним класом?
5. Як можна вирішити проблему надійності за типами?

Завдання:

1. Створити структуру родового зв’язаного списку, використовуючи шаблони.

2. Прочитайте файл, який складається з чисел з плаваючою крапкою, складіть із пар прочитаних чисел комплексні числа і запишіть їх у файл, використовуючи потоки.

3. Напишіть програму яка приймає з клавіатури довільну кількість рядків та записує їх у файл. Ім’я файлу задається при запуску програми першим параметром командного рядка. Роботу програми закінчити при отриманні з клавіатури умови EOF.

4. Визначити тип name_and_adress. Перевантажити для нього оператори << та >>. Скопіювати потік об’єктів name_and_adress.

Завдання для самостійної роботи

1. Створити шаблон для створення функцій, які виводять масив типу int і типу float.

2. Створити шаблон в одному масиві з n елементів та вирахувати: кількість елементів ==0, суму елементів масиву, розташовану після мінімального елемента. Впорядкувати елементи масиву за зростанням їх модулей.

3. Розробити програму калькулятор з налаштуванням кількості десяткових знаків та відформатованим виглядом (вирівнення по правому краю).

4. Записати вихідний файл, додати до кожного числа останнє число файлу.

5. Записати вихідний файл, помножити кожне третє число на подвоєну суму першого і останнього від'ємного числа.

6. Записати вихідний файл, помножити кожне парне число на перше від'ємне число файлу.

7. Дан символьний файл f, який складається з малих латинських літер і розділових знаків. Переписати файл, заміняючи маленькі літери великими. Вивести на екран файл до і після перетворення.

Лабораторна робота № 7

Тема: Виключні ситуації. Заміна фукцій *unexpected()* і *terminate()*.

Мета: Навчитися використовувати техніку обробки виключних ситуацій.

Теоретичні відомості.

Виключна ситуація - це умова виключної ситуації, що потребує спеціальної обробки. В.с. найкраще використовувати для обробки помилок, що виникають при виконанні, але їхнє застосування на цьому не обмежується.

Для збудження в.с. оператор посилає об'єкт, що описує суть в.с. Об'єкт м.б. літеральним значенням, рядком, об'єктом класу або будь-якого іншого об'єкта. Об'єкт в.с. не обов'язково має бути об'єктом класу.

Для обробки в.с. деякий оператор перехоплює умову, яка послана іншим процесом. Оператори, що перехоплюють в.с., називаються обробниками в.с.

Програми готуються до перехоплення в.с., випробуючи один або декілька процесів, що збуджують в.с. Для використання в.с. випробовується один або декілька операторів і перехоплюється будь-яка в.с., що збуджується цими операторами.

Виявивши стан помилки, функція може збудити в.с., що викликає такі наслідки:

Функція оголошує, що виникла умова в.с. Це м.б. як помилкою, так і іншою обставиною, що потребує обробки.

Функція запитує вирішення проблеми обробки в.с. Обробник, якщо він існує, викликається автоматично у відповідь на посилку об'єкта в.с.

У програмі збуджується в.с. за допомогою виконання оператора *throw*, зазвичай всередині функції:

```
throw “Overflow”;
```

У іншому місці програми обробник рядкових в.с. може впіймати і відобразити на екрані посланий об'єкт. В обробнику задається тип об'єкта у виразі:

```
catch(const char *message)
{
  cout << “Error! ” << message;
  ....
}
```

Оператор *catch* ловить послані об'єкти рядкових в.с. і відображає їх оператором виведення в потік. Якщо подальші дії не визначені в блоці *catch*, то програма продовжить свою роботу після *catch*. Можна також аварійно перервати виконання програми, викликати іншу функцію або продовжити цикл для повторного виконання дій, що викликали проблему.

В.с. - механізм для повідомлень і вживання заходів у випадку виникнення умови в.с. В.с. не нав'язують необхідних дій. Їхня обробка цілком залежить від програміста.

Функції в.с. не обмежені обробкою помилок. Наприклад, порожній обліковий об'єкт може повідомляти про те, що він порожній, шляхом збудження в.с. Відбулася помилка або щось інше, залежить тільки від того, як програміст призначить аварійне переривання виконання програм.

Але оскільки використання в.с. може призвести до двозначності в програмах, їх найкраще використовувати для вживання заходів у випадку виникнення дійсних помилок, що можуть змусити програму перервати процес і призвести до аварійного завершення або невірних результатів.

Функції можуть збуджувати одну або декілька в.с. різних типів, що представляють різноманітні умови в.с.

Збудження в.с. негайно завершує виконання функції, у якій виконується оператор *throw*.

Виключні ситуації забезпечують альтернативний механізм повернення для функцій.

Оголошення в.с.

За допомогою альтернативної форми оголошення функції можна задавати типи в.с., які дозволяється збуджувати в даній функції. Наприклад, функція *AnyFunction()* може задати типи своїх в.с. таким чином:

```
void AnyFunction() throw(Error);
```

Наступний за ім'ям функції і списком параметрів вираз *throw()* вказує, що функція *AnyFunction()* може збуджувати в.с. типу *Error*. Це оголошення вказує компілятору, що функції *AnyFunction* не дозволяється збуджувати в.с. інших типів. Для завдання функції, що збуджує в.с. декількох типів, варто перерахувати типи локальних об'єктів в.с.:

```
void AnyFunction() throw(Error, char*, OtherType);
```

Необроблені в.с.

Необроблені в.с. передаються нагору по ланцюжку викликів одних функцій іншими доти, доки не зустрінеться відповідний їм оператор catch або доки більше не залишиться неоглянутих обробників в.с. Якщо відбудеться останнє, для обробки в.с. викликається одна з трьох спеціальних функцій, які автоматично приєднуються до кожної програми на C++, що використовує в.с. Ці функції мають імена

unexpected(), terminate(), abort().

Вони викликаються відповідно до правил:

Якщо в програмі виникли в.с., що обробляються оператором catch, то викликається функція unexpected*()*. В.с., які не обробляються жодним оператором catch, називаються непередбаченими в.с. За замовчуванням unexpected*() викликає* функцію terminate*()*. Непередбачені в.с. можуть збуджуватися програмою при виявленні ушкодження стеку дескриптором класу, внаслідок чого викликається функція *terminate()*. За замовчуванням функція *terminate()* викликає функцію *abort()*.

Функція *abort()* негайно перериває виконання програми. Вона ніколи не викликається безпосередньо. Якщо не оброблені всі можливі в.с. і не вжиті заходи для перепрограмування функцій *unexpected()* і *terminate()*, необроблена в.с. аварійно перерве виконання програми шляхом звертання до *abort()*.

Можна замінити функції *unexpected()* і *terminate()* своїм кодом для обробки необроблених в.с. Наприклад, має сенс замінити функцію *unexpected()* для повідомлення користувача про будь-які необроблені в.с. на етапі розробки програми. Це може допомогти виявити відсутні обробники помилок у програмі. У

інших випадках, можна замінити *terminate()* діагностуючим кодом для перегляду пам'яті, щоб виявити оператори, які руйнують купу.

Замінити функцію *abort()* не можна. Її виклик завжди призводить до завершення програми.

Заміна функцій *unexpected()* і *terminate()*

Для завдання адреси власної функції - обробника непередбачених в.с. варто використовувати функцію *set_unexpexted()*, оголошену в заголовочному файлі EXCEPT.H. Функція повинна мати тип *unexpected_function*, не мати аргументів і нічого не повертати.

Обробник користувача непередбачених в.с. може збуджувати виключну ситуацію. У цьому випадку пошуки оператора *catch* починаються з того місця, звідки спочатку було викликано обробник.

Для завдання власної функції-обробника завершення програми варто використовувати функцію *set_terminate()*, також оголошену в заголовочному файлі EXCEPT.H. Функція повинна мати тип *terminate_function*, не мати параметрів, нічого не повертати. Обидві функції *set_expected()* і *set_termilate()* повертають адресу поточної функції-обробника. Можна зберегти і потім відновити існуючі обробникі, запам'ятавши їхні адреси в змінних, а потім передаючи їх обернено функціям. Наприклад, спочатку оголошується прототип функції-обробника: *void unexpectedHandler();* Потім встановлюється обробник:

```
set_unexpected(unexpectedHandler);
```

Зберігання і відновлення адреси старої функції-обробника:

```
unexpected_function savedAddress;
savedAddress = set_unexpecred(unexpectedHandler); //новий обробник
set_unexpected(saveAddress); //відновлення
//новий обробник більше не використовується
```

Функція користувача *terminate()* встановлюється аналогічно, тільки замість *set_unexpected()* викликається функція *set_terminate();* що повертає адресу *muny terminate*_function.

Приклад програми встановлення обробників непередбачених в.с. і завершення. У програмі також демонструється прийнятний засіб обробки непередбачених в.с. невідомих типів, що можуть збуджуватися в погано документованих бібліотечних функціях.

```
#include<iostream.h>
#include<except.h>
#define MAXERR 10
class MaxErr{};
class Error{
public:
  Error();
  void Say();
private:
  static int count;
};
void Run() throw(Error);
void trapper();
```

```
void zapper();
int Error::count;
void main()
{
 set_unexpected(trapper);
 set_terminate(zapper);
 for(;;)
 {
  try{Run();}
  catch(Error e){e.Say();
 }
}
void Run() throw(Error)
{
//throw Error();
 throw “Невідомий тип об'єкта”;
}
void trapper()
{
 cout << “Обробник непередбачених ситуаций.  .”;
 throw Error();
}
void zapper()
{
 cout << “Обробник завершення функції”;
 exit(-1);
}
Error::Error()
{
 count++;
 if(count>MAXERR)
 throw MaxError();
}
void Error::Say()
{
 cout << count << “\n”;
}
```

При запуску unexpected.cpp виведеться 10 повідомлень про помилки перед завершенням.

У програмі використовуються два класи в.с. Об'єкт класу *MaxError*, що не має даних і функцій, посилається, коли число помилок перевищує *константу* MAXERR, задану рівною 10.

Конструктор класу *Error* інкрементує статичний член класу *count*. Оскільки член *count* - статичний, то існує тільки один екземпляр цього значення і він існує доти, доки програма не завершиться. Член *count* - закритий член класу, отже він не зміниться в операторах програми поза класом. Функція-член *Say()* відображає значення лічильника помилок програми.

Якщо член *count >= MAXERR*, конструктор класу *Error* збуджує в.с. типу

MaxError. При збудженні в.с. об'єкт класу не створюється, тому створення об'єкта класу *Error* переривається.

У функції *main()* встановлюються функції-обробники, що заміщають за замовченням *функції unexpected()* і *terminate()*. Потім виконується нескінчений цикл *for*. Всередині циклу в блоці *try* викликається функція *Run()* і оператор *catch* перехоплює усі об'єкти класу *Error*, що посилає *Run()*. При збудженні в.с. *catch* відображає поточне значення лічильника помилок шляхом звертання до функції *Say()* посланого об'єкту.

Функція *Run()* завжди збуджує в.с., моделюючи виникнення декількох помилок. Користувацький обробник непередбачених в.с. *trapper()* виводить повідомлення про виклик цієї функції. Це відбувається після збудження 10 в.с. Обробник непередбачених в.с. може збудити ще одну в.с. для продовження програми. У прикладі оператор *throw Error();* посилає новий екземпляр класу *Error*. Створення об'єкта класу *Error* змушує конструктор цього класу збудити ще одну ситуацію типу *MaxError*. Цей тип помилки не підтримується в операторі *catch*, тому викликається обробник завершення програми *zapper()*. Функція завершення не може збудити в.с., а також не може виконати оператор *return*.

Перепрограмуємо функцію *Run()* так, щоб у ній збуджувалася в.с. невідомого типу. Наприклад, у якості невідомого типу може виступити рядок.

Коли програма зкомпілює і запустить обробник непередбачених ситуацій, *trapper()*, буде викликатись для кожної виключної ситуації, *збуджуваною* функцією Run(). Це відбувається тому, що не існує оператора *catch* для об'єктів в.с. типу *char** або *const char**. Функція *trapper()*, проте, транслює нерозпізнані об'єкти в.с., посилаючи об'єкт відомого типу, у даному випадку, Error. Нова збуджена в.с. продовжує виконання програми в операторі *catch* всередині функції *main()*, що обробляє трансльовану виключну ситуацію.

Зрештою, у програмі максимальне число помилок буде перевищено, і об'єкт класу MaxError буде *посланий* конструктором Error. У результаті викличеться обробник завершення програми шляхом звертання до бібліотечної функції *exit()*.

Лістинг результату в 1м випадку (у *Run()* використовується *throw Error();*)

```
1
2
...
10
Обробник непередбаченої помилки.
Обробник завершення функції.
```

У 2му випадку (у *Run()* - рядок):

```
Обробник невідомої помилки.
1
Обробник невідомої помилки.
2
...
10
Обробник невідомої помилки.
Обробник завершення функції.
```

Контрольні питання до л.р. 7:

1. Що представляє собою виключна ситуація?
2. Що називають обробниками виключних ситуацій?
3. Що представляє собою блок try?
4. На що вказує наступний за ім`ям функції і списком параметрів вираз throw()?
5. Які в. с. називають непередбаченими?
6. Що відбувається з необробленими виключними ситуаціями?
7. Як викликається функція abort(), що вона виконує, чи можна її замінити своїм кодом?
8. Для чого використовують функції set_unexpected(), set_terminate(), скільки вони мають аргументів і що повертають?

Завдання

1. Наберіть текст програми наведеного прикладу обробки виключних ситуацій та налагодьте його (розгляньте 2 випадки роботи програми). Поясніть отримані результати у першому та другому випадку.
2. Напишіть програму, в якій користувачеві пропонують кілька разів ввести число. Кожного разу виводити його корінь. Якщо вводиться від'ємне число, виводити повідомлення про помилку. Використати техніку виключних ситуацій.

Завдання для самостійної роботи

Функції, які реалізовуються в завданнях обовязково повинні виконувати перевірку параметрів, що передаються і генерувати виключення у випадках помилок.

1. Функція обчислює площу трикутника по трьом сторонам S=sqrt(p(p – a)(p – b)(p – c)), де p=(a+b+c)/2.
2. Функція обчислює корінь лінійного рівняння ax+b=0.
3. Функція обчислює корінь квадратного рівняння $ax^2+bx+c=0$.
4. Функція перевіряє, чи є рядок, що передається, поліндромом.