В. М. Овсяннік, О. К. Погудіна

МОВА С++ НЕ ДЛЯ ЧАЙНИКІВ

2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

В. М. Овсяннік, О. К. Погудіна

МОВА С++ НЕ ДЛЯ ЧАЙНИКІВ

Навчальний посібник

Харків «ХАІ» 2020

УДК 004.43 (075.8)
О34

Рецензенти: д-р техн. наук, с.н.с. Г. А. Кучук,
д-р техн. наук, професор М. Л. Угрюмов

Овсяннік, В. М.

О34 Мова С++ не для чайників [Електронний ресурс] : навч. посіб./В. М. Овсяннік**,** О. К. Погудіна. – Харків : Нац. аерокосм. ун-т ім. М. Є. Жуковського «Харків. авіац. ін-т», 2020. – 130 с.

Подано теоретичний і практичний матеріал із дисципліни «Об’єктно-орієнтоване програмування». Викладено основні складові мови програмування С++, що стосуються функцій, структур і класів. Посібник складено з розділів, що доповнюють лекційний курс і містять приклади розв’язання практичних задач об’єктно-орієнтованого програмування.

Для студентів вищих навчальних закладів, що вивчають курси «Об’єктно-орієнтоване програмування», «Сучасні технології програмування», «Інструментальні засоби візуального програмування». Може бути корисним для спеціалістів галузі інформаційних технологій.

Іл. 6. Бібліогр.: 9 назв

УДК 004.43 (075.8)

ЗМІСТ

ПЕРЕДМОВА

У посібнику розглянуті структури, функції та об’єктно-орієнтоване програмування (ООП) на мові програмування С++. Матеріал посібника викладений у порядку природнього освоєння мови і розрахований на читача, що бажає самостійно підвищити рівень своїх знань і долучитися до когорти програмістів-професіоналів. Виклад матеріалу припускає знання основ мови та наявність деякого досвіду програмування. Посібник має численні приклади програм, оскільки освоєння мови С++, як і будь-якої іншої мови програмування теж, неможливо без написання й налагодження програм на комп'ютері. У роботі розглядається переважно перший стандарт мови С++ ISO/IEC 14882, неформально відомий як С++98. Приклади програм тестовано в інтегрованому середовищі (ІС) Microsoft Visual Studio 2015 (MVS-2015).

Особлива увага приділяється об'єктно-орієнтованому програмуванню як сучасної технології розроблення програм, без якої неможливо розроблення практично цінних програм у будь-якому середовищі програмування і на будь-якій мові програмування.

Безсумнівно, що в наш час С++ є найкращою універсальною мовою програмування *для професіоналів*. Це означає, що в тому разі, коли програмування яких-небудь завдань є Вашим основним заняттям, то ця мова для Вас.

Історія мови С++ нараховує вже майже чотири десятиліття (уважаючи з 1980 р.), а лежачої в її основі мови С – ще на одне десятиліття більше. Не вдаючись в історичний огляд і обговорення всіх гідностей мови С++, відзначимо лише деякі історичні віхи й важливі моменти.

Мова С була розроблена на початку 70-х років минулого століття Деннісом Рітчі (Dennis Ritchie), що у той час працював програмістом у компанії Bell Telephone Laboratories (США). Головною метою автора була розроблення такої мови програмування, яка б поєднувала в собі міць машинно-орієнтованої мови (асемблера) з лаконічністю мови високого рівня. Така мова є надзвичайно корисним інструментом для розроблення складних програмних продуктів (ПП), таких як операційні системи (ОС) й компілятори. І, треба сказати, ця мета була досягнута: Unix для електронно-

обчислювальної машини (EOM) PDP-7 стала першої ОС, написаної не на асемблері, а на мові високого рівня.

На початку 80-х років минулого століття Бьярн Страуструп (Bjarne Stroustrup) розробив мову C++, суттєво поліпшивши синтаксис мови С, у бік більшої наочності мовних конструкцій та сучасних уявлень про стиль розроблення програм, й доповнивши її класами, тобто засобами для розроблення програм з використанням концепцій ООП. З того часу для мов С и С++ були розроблені безліч компіляторів для різних апаратних і програмних платформ. У цей час існує, діє і продовжує удосконалюватися стандарт мови С++ [6], якому повинні слідувати розробники компіляторів і інших інструментальних засобів. Мова С++ широко використовується при розробці операційних систем: Windows в основному написана на цій мові.

У посібнику наведені численні приклади програм і їх фрагментів, які компілювалися в IC MVS-2015. Тим не менше, автори не боги і не можуть гарантувати відсутність будь-яких алгоритмічних помилок. З міркувань лаконічності при написанні прикладів програм робилися деякі відмови від типових рекомендацій зі стилю програмування:

- описи класів іноді виконуються в тому ж файлі, у якому перебуває функція **main()**, хоча доцільно опис класу розміщати в окремому заголовному файлі (**.h**), а реалізацію його методів – у файлі реалізації (**.cpp**);
- у деяких прикладах у конструкторах, деструкторах і інших методах класів виконується введення або вивід даних, чого в реальних програмах робити категорично не рекомендується.

У посібнику тексти програм або їх окремі фрагменти наведені з використанням шрифту **Times New Roman**. а коментарі – шрифту *Times New Roman*.

Посібник призначений для студентів, що навчаються за спеціальностями «Комп'ютерні науки» та «Сучасні технології програмування», а також для початківців, що вже мають деякий досвід розроблення програм.

1. ФУНКЦІЇ

1.1. Посилання

Визначення

ім'я_типу &ім'я_посилання ініціалізатор

уводить посилання на об'єкт, яке стає іншим іменем, тобто синонімом об'єкта. Після такого визначення ім'я посилання може використовуватися так само, як ім'я об'єкта: з його допомогою можна отримувати доступ до значення об'єкта з метою його одержання або модифікації.

Як ініціалізатор можна використовувати:

- ім'я функції;
- лівосторонній (**lvalue**) вираз, що вказує на об'єкт, який має адресу пам'яті, наприклад змінна.

Приміром, після таких визначень посилань

```
int v;
int &ref1=v;
int &ref2=v;
```

можна використовувати такі оператори:

```
v=123;
cout<<ref1; // виводить 123
cout<<ref2; // виводить 123
ref2=321;
cout<<v; // виводить 321
cout<<ref1; // виводить 321
```

Хоча посилання і є синонімом об'єкта, воно може бути після ініціалізації "націлине" на інший об'єкт, виявляючи властивості покажчика. Цікаві результати дає, наприклад, програма:

```
// Demoref.cpp – ілюстрація властивостей посилань
void main()
{
  double v1 = 111, v2 = 222;
  double &ref1 = v1; // визначення й ініціалізація посилання
  cout << ref1 << endl; // виводить       111
  ref1 = v2; // "настроювання" посилання на іншу змінну !
  cout << ref1 << endl;    // виводить    222
  cout << v1 << endl;      // виводить    222 !
  cout << v2 << endl;      // виводить    222
  cout << &ref1 << endl;  // виводить 0032F7B0
  cout << &v2 << endl;     // виводить 0032F7A0
}
```

Допускається визначати посилання на константу:

const float E = 2.71; // *визначення константи*
float const &refe = E; // *визначення посилання на константу*
const float &refe = E; // *еквівалентно попередньому операторові-виразу*

У цьому випадку за допомогою посилання змінити значення константи не можна, тобто, наприклад, оператор **refe=0.0;** викличе помилку компілятора.

Зауваження. Якщо Ви раніше не зустрічалися з посиланнями й Вам незрозуміле призначення посилань – не зневіряйтеся, так і повинно бути. Дивіться наступні підрозділи, у яких обговорюються функції

Допускається також використовувати посилання на функції (подібно покажчикам на функції) і посилання, що є результатом, тобто ті, що повертаються функцією. В останньому випадку можна одержати дуже цікаві розв'язки, наприклад, використовувати функцію ліворуч від знака присвоєння, тобто в як лівостронній вираз (**lvalue**). Приклади використання посилань як параметрів функцій, що повертають значення, розглядаються нижче.

1.2. Опис функцій

Функцію можна визначити як деякий особливим чином оформлений фрагмент програми, що містить у загальному випадку довільну кількість оголошень і операторів. Значення функцій у будь-якій мові програмування, у тому числі й у C++, переоцінити неможливо. Грамотне використання функцій дозволяє:

- структурувати програму, тобто розділити її на окремі блоки, кожний з яких виконує одну специфічну дію, наприклад, знаходить корені рівняння другого ступеня;
- полегшити розроблення й підвищити читабельність програми, оскільки зрозуміти призначення однієї функції невеликого розміру значно простіше, ніж розібратися в аналогічному фрагменті коду чималої програми;
- забезпечити вільний обмін між програмістами результатами їх роботи, поданими у вигляді окремих функцій або їх наборів, тобто бібліотек. При цьому програмістові-користувачеві немає потреби аналізувати текст функції – досить знати тільки формат виклику функції;
- забезпечити програміста стандартними бібліотеками функцій загального й спеціального призначення, тим самим підвищивши продуктивність його праці;

- підвищити якість і надійність програми завдяки тому, що для розроблення деякого складного алгоритму можна залучити фахівців найвищої кваліфікації, після чого всі інші зможуть просто використовувати результати їх праці.

Наявність функцій дозволяє також зробити мову програмування порівняно легко переносимою на різні апаратні й програмні платформи, оскільки специфічні особливості платформ «ховаються» у функціях. Так, наприклад, уведення й виведення даних є специфічним для кожної платформи, і для забезпечення можливості розроблення переносимої програми він реалізується у вигляді функцій, однакових за форматом виклику, але таких, що мають специфічний для платформи код.

Одержати уявлення про те, як можуть виглядати опис, визначення й застосування функції, можна із прикладу програми, що ілюструє застосування бібліотечної функції двійкового пошуку **bsearch()**:

```
/* Прототип функції bsearch
   void *bsearch(const void *key, const void *base, size_t nelem,
    size_t width, int (*fcmp)(const void*, const void*))
*/

typedef  int (*fptr)(const void*, const void*); // покажчик на функцію
int numarray[] = {123, 145, 512, 627, 800, 933};
int N= sizeof(arr) / sizeof(arr[0]);
int numeric (const int *p1, const int *p2)
 {
   return(*p1 - *p2);
 }

 int lookup(int key)
 {
   int  *itemptr;
   itemptr = (int *) bsearch (&key, numarray, N, sizeof(int), (fptr)numeric);
   return (itemptr != NULL);
 }
void main()
 {
   if (lookup(512)) cout<<"512 is in the table\n";
   else  cout<<"512 isn't in the table\n";
}
```

Вивчивши матеріал цього розділу, можна навчитися розробляти такі функції.

Загалом визначення будь-якої функції має такий формат:

```
модифікатор тип_функції ім'я_функції
(специфікації_формальних_параметрів)
{ тіло_функції }
```

Перший рядок наведеного визначення функції називається її *заголовком*. Тіло функції завжди має бути блоковим оператором, можливо порожнім.

Заголовок функції, після якого зазначено крапку з комою, – це оголошення функції, що зазвичайназивають *прототипом*. Наприклад:

int Max(int a, int b); // *прототип функції*
inline float Cube(float x); // *прототип функції, що підставляється*
int Min(int, int); // *прототип функції, у якому опущені імена параметрів*
float Abs(float x) {return x>0?x:-x; } // *визначення функції*

Термін *сигнатура функції* означає тип значення, що повертається функцією, а також кількість і типи формальних параметрів функції. Інакше кажучи, дві функції мають однакову сигнатуру, якщо їх заголовки відрізняються тільки іменами функцій. У наведеному вище прикладі функції **Max** і **Min** мають однакову сигнатуру. Поряд з термінами *оголошення функції* й *визначення функції* будемо вживати й термін *опис функції*, що означає оголошення або визначення, у тих випадках, коли це не має суттєвого значення.

Як необов'язковий *модифікатор* можна використовувати ключові слова **inline**, **const, cdecl, pascal** і деякі інші, призначення яких стосовно до функцій тут не розглядається.

Тип функції – це тип значення, що повертається функцією. Якщо функція не повертає ніякого значення, то як її тип треба вказати ключове слово **void**. Якщо ж функція повертає яке-небудь значення, то вона обов'язково повинна має оператор **return**, за яким повинен слідувати вираз, що визначає значення, яке повертається функцією.

Для головної функції **main()** тип функції зазвичай **void**, однак він може бути й типом **int**. В останньому випадку значення, що повертається функцією **main()**, можна проаналізувати у батьківським процесом, тобто іншою програмою або операційною системою.

Ім'я функції – це ідентифікатор, який має бути унікальним у своєму просторі імен, тобто серед імен об'єктів, визначених поза функціями. Ім'я функції за замовчуванням має клас пам'яті **extern** і статичну тривалість існування. Ім'я функції за певних умов доступно в будь-якому місці файла або файлів, з яких складається програма.

Специфікація формальних параметрів (СФП) задає кількість формальних параметрів, їх типи й порядок проходження. Якщо функція не має формальних параметрів, то порожні дужки після імені функції все одно треба зазначати. У цьому випадку також у дужках може бути зазначено ключове слово **void**, наприклад:

void Blank(); // *функція без параметрів і нічого не повертає*
void Otherblank(void); // *функція без параметрів і нічого не повертає*
float * Otherblank(void); /**функція без параметрів, що повертає покажчик*

*на дійсний тип******/*

У загальному випадку СФП можна подати таким чином:

- порожні дужки або зарезервоване слово **void**, що означають відсутність формальних параметрів (ФП);
- список специфікацій формальних параметрів, що закінчується, можливо, трьома крапками.

Специфікація кожного ФП функції має вигляд:

[модифікатор] тип_ФП ім'я_ФП [=значення_за_замовчуванням]

де квадратні дужки використано для того, щоб показати необов'язковість модифікатора і значення за замовчуванням. Крім того, допускається в опису функції не вказувати ім'я ФП, якщо він у ній не використовується.

Як *модифікатор* можуть бути використані ті ж ключові слова мови, що й в описі змінних.

Якщо для будь-якого ФП задано значення за замовчуванням, то воно має бути задано також для всіх інших ФП, розташованих у списку *праворуч*. Якщо надалі при виклику функції відповідні фактичні параметри будуть опущені, замість них будуть використані значення за замовчуванням. Опускати фактичні параметри знов-таки можна тільки з кінця, тобто пропускати їх не можна. Зазначимо, що можливість використання значень параметрів за замовчуванням не еквівалентна можливості використання функцій зі змінною кількістю параметрів.

Функції зі значеннями параметрів за замовчуванням і змінною кількістю параметрів будуть розглянуті нижче.

Тіло функції завжди береться у фігурні дужки, тобто має являти собою блоковий оператор, але допускається вказувати й порожні фігурні дужки, які часто використовують у процесі розроблення програми у функціях-заглушках. Якщо функція має тип **void**, то вона не повертає ніякого значення й у ній наявність оператора **return** не є обов'язковою.

Якщо тип функції відрізняється від **void**, уживання оператора **return** є обов'язковим і за ним повинен бути вираз, що задає значення, яке повертається функцією, того ж типу, що й тип функції. В обох випадках у функції можна використати будь-яку кількість операторів **return**.

Прототип функції необхідний у тому випадку, якщо виклик функції здійснюється до її визначення, що зазвичай так і буває. Визначення функції за наявності прототипу може розташовуватися в тому ж файлі, де здійснюється її виклик, або в іншому. Прототип функції, як зазначалося вище, відрізняється від заголовка функції тільки наявністю крапки з комою. Крім того, у прототипі в специфікаціях ФП допускається не вказувати їхні імена, оскільки в цьому випадку компілятору вони не потрібні. Наприклад, при такому визначенні функції

float Summa(float mas[], int n)

```
{
  float sum=0.0;
  for(int i=0; i<n; sum+=mas[i++]);
  return sum;
}
```

прототип може мати вигляд

float summa(float [], int);

1.3. Виклики функцій

Виклик функції є виразом вигляду:

ім'я_функції(список_фактичних_параметрів)

Значенням цього виразу є значення, що повертається функцією, тип якого відповідає типу функції. Якщо функція повертає якесь значення (має тип не **void**), то виклик функції зазвичай включається у вираз того ж типу, що й тип значення, яке повертається функцією, наприклад:

double Tan=sin(x)/cos(x); // *sin() і cos() – загально відомі математичні функції*

Якщо функція не повертає ніякого значення, то її виклик являє собою, зазвичай, оператор-вираз, наприклад:

system("color f0"); puts("Xo -Xo-Xo”);

Навіть якщо функція повертає яке-небудь значення, то його можна ігнорувати, викликаючи цю функцію так, начебто вона не повертає значення, наприклад:

sin(x); // *сенсу немає, але припустимо*
_getch(); /* *припинення виконання програми до натискання якої-небудь клавіші* */

Мова С++ потребує, щоб список фактичних параметрів відповідав формальним параметрам за кількістю, типами і послідовністю переліку, за винятком функцій зі значеннями параметрів за замовчуванням і функцій зі змінною кількістю параметрів. Разом з тим, про що мова йтиме нижче, типи фактичних параметрів можуть відрізнятися у деяких випадках від типів формальних параметрів. Наприклад, формальний параметр може мати цілочисловий тип, а фактичний – дійсний або символьний тип. Інакше кажучи, усі числові типи, до яких належать цілочислові, символьні, логічний тип і перерахування, є сумісними. У деяких випадках це може призвести до втрати точності або значення, але за це відповідає вже програміст. Кількість фактичних параметрів може бути менше кількості формальних, якщо для останніх вказані значення за замовчуванням.

Відповідність фактичних параметрів формальним перевіряється компілятором на етапі компіляції програми.

Передача фактичного параметра за адресою виконується в тих випадках, коли:

- параметр є масивом або покажчиком;
- формальний параметр описаний як посилання, тобто перед ним зазначено символ &.

Це означає, наприклад, що структурні типи передаються у функцію за значенням, тобто копіюються їхні значення.

Проілюструємо ці положення прикладами програми. Передамо у функцію структуру:

```
struct Tstr{double d;};
void Func(Tstr stc)
{
    stc.d=1.23;
}
void main()
{
    Tstr strct={-1.0};
    Func(strct);
    cout<<"strct.d=="<<strct.d<<endl; // виводить -1
}
```

З іншого боку, при описі параметра функції як посилання ситуація буде іншою:

```
void Func(Tstr & stc)
{
    stc.d=1.23;
}
void main()
{
    Tstr strct={-1.0};
    Func(strct);
    cout<<"strct.d=="<<strct.d<<endl; // виводить 1.23
}
```

Як приклад створимо функцію **InputVec()**, яка має запросити у користувача кількість елементів масиву, ввести значення елементів масиву й повернути масив і кількість його елементів.

```
/*Demoref.cpp - ілюстрація використання параметра-посилання у функції*/
void InputVec(double Vec[], /*масив, значення елементів якого має
                                повернути функція */
              int &n) //кількість елементів масиву, що повертає  функція

 {
cout<<"Уведи кількість елементів масиву>";
cin>>n;
```

```
for (int i=0;i<n;i++)
{
  cout<<"Уведи "<<i<<"-й елемент масиву>";
  cin>>Vec[i];
}
}
void main()
{
  double Arr[200];
  int nf=0; // початкове значення задаємо виключно з ілюстративною метою
  InputVec(Arr,nf);
  cout<<nf<<endl;
}
```

У програмі після виклику функції **Inputvec()** буде виведено ту кількість елементів масиву **n**, яку введе користувач у функції **Inputvec()**. Якщо заголовок функції описати як

```
void InputVec(double Vec[], int n)
```

то елементи масиву функція буде вводити, але значення змінної **nf** залишиться нульовим, що є *алгоритмічною* помилкою.

А чому елементи масиву будуть передані у функцію **main(),** якщо формальний параметр **double Vec[]** не описано як посилання? Відповідь проста: за значенням у функцію передається ім'я масиву, тобто його адреса, і, отже, до самих елементів масиву функція має безпосередній доступ.

Положення буде іншим, якщо масив буде передаватися у функцію як покажчик. Припустимо, потрібно написати таку функцію, яка створювала б масив у динамічній пам'яті.

```
/*Demoref2.cpp - передача у функцію покажчика за посиланням*/
void InputVecp(double *&Vec, int &n)
{
  cout<<"Уведи кількість елементів масиву>";
  cin>>n;
  Vec=new double [n]; // резервуємо пам'ять для масиву
  for (int i=0;i<n;i++)
  {
    cout<<"Уведи "<<i<<"-й елемент масиву>";
    cin>>Vec[i];
  }
}
void main()
{
  double *Arr;
  int nf=0;
  InputVecp(Arr,nf);
```

```
  for(int i=0;i<nf;i++) cout<<Arr[i]<<endl;
  // ...
  delete Arr;
}
```

За задумом, функція **InputVecp()** має ввести кількість елементів масиву, зарезервувати динамічну пам'ять для масиву й повернути за допомогою параметрів і кількість елементів масиву, і їх значення. Для цього покажчик треба описати як посилання: **double *&Vec**. Якщо ж цього не зробити, тобто описати покажчик як **double *Vec**, то програма компілюватися буде без помилок, але очікувані значення елементів масиву у функції **main()** не побачимо. У найкращому разі будуть виведені випадкові значення, а в найгіршому – виконання програми буде перервано операційною системою при спробі виведення елементів масиву у функції **main()**.

Чому так відбувається? Якщо формальний параметр не описано як посилання, то функція все одно може його змінювати – покажчику **Vec** присвоюється адреса динамічної області пам'яті, але це значення не присвоюється відповідному фактичному параметру – покажчику **Arr** – оскільки функція **InputVecp()** працює з його копією.

В обговорюваній програмі **Demoref2** показано також, як легко й просто в мові C++ можна реалізувати «динамічний» масив, фактична кількість елементів якого визначається на етапі виконання програми.

Якщо (формальний) параметр функції описано як масив або покажчик, як відповідний фактичний параметр можна передати посилання на змінну, наприклад:

```
int k;
  // …
Writeln(&k,1);
```

Звісно, що в такому разі функція **Writeln()** може опрацьовувати масив, який містить тільки один елемент.

1.4. Повернення значень функціями

Якщо функція повертає який-небудь тип, що відрізняється від **void**, то в ній обов'язково має бути присутнім, принаймні один, оператор **return**, що повертає значення того ж типу, що й тип функції. Функція може повертати значення майже будь-якого типу, а саме:

- будь-якого фундаментального типу;
- перелічування **enum**;
- структуру (**struct**), об'єднання (**union**) або клас (**class**);
- посилання;
- покажчик на будь-який тип, у тому числі й покажчик типу **void**.

Якщо за своїм призначенням функція має щось повернути, то найкраще це зробити саме шляхом повернення значення за допомогою **return**, а не за допомогою модифікації значення параметра. Чому? Оскільки так функцію простіше й наочніше використовувати, оскільки сам контекст її виклику підказує її призначення. Якщо ж функція змінює значення якого-небудь параметра, то цей факт користувач функції може й не помітити.

Отже, якщо функція повертає значення, то оператор **return** повинен мати такий формат:

return вираз;

Як вираз можна використовувати будь-який вираз, що має той же тип, що й тип функції. Якщо тип виразу не збігається з типом функції, то компілятор виконує неявне перетворення типу, якщо таке перетворення можливе. Неявне перетворення виконується за тими же правилами, що й при ініціалізації змінних, оскільки семантика оператора **return** така ж, що й семантика ініціалізації змінної.

Як приклад розглянемо варіанти реалізації функції, яка повертає суму двох параметрів.

```
int Sum(int a, int b) { return a+b; } /*повертає значення виразу*/
int & Sumref1(int a, int b)
{ int Local; return Local=a+b; } /*повертає посилання на локальну змінну,
                                   пам'ять для якої виділена у стеку */
int & Sumref2(int a, int b)
{ int *ploc=new int; return *ploc=a+b; } /* повертає посилання на локальну
        змінну, пам'ять для якої виділена у купі */
```

Нагадаємо, що купою (heap) зазвичай називають динамічну ділянку пам'яті комп'ютера.

> **Зауваження**. Приклади реалізації функцій **Sumref1()** і **Sumref2()** є сугубо ілюстративними й жодним чином не можуть бути зразками для бездумного наслідування.

Наведена вище реалізація функції **Sumref1()** потенційно небезпечна тим, що вміст стекової пам'яті постійно змінюється, внаслідок чого значення локальної змінної буде дуже скоро загублене. Розглянемо результат виконання такого фрагмента програми:

```
int &Ref=Sumref1(3,2);
cout<<Ref<<endl; // виводить 5
cout<<Ref<<endl; // виводить випадкове ціле число (?)
```

Значення змінної **Ref** «зненацька» змінилося при її повторному використанні. У той же час такий виклик функції

```
int Res=Sumref1(3,2);
```

цілком безпечний, оскільки значення локальної змінної **Local**, що повертається функцією, копіюється у змінну **Res**.

Стосовно реалізації функції **Sumref2()** пропонується відповісти на таке запитання: як можна звільнити пам'ять, що була виділена у купі для покажчика **ploc**?

> **Запитання**. А яку користь можна витягти з того, що функція, яка повертає посилання, може бути використана так?
> **int r3= Sumref2(300,200)=Sumref2(3,2)+100;**

1.5. Покажчики і масиви як параметри функцій

У масивів і покажчиків більше спільних властивостей, ніж відмінностей. Нагадаємо, що масив є, по суті, константним покажчиком. Це твердження стосується також і покажчиків на покажчики й багатовимірних масивів. Наприклад, за наявності визначень

```
double *Ptr, Vec[20];
double **Ptr2,Vec2[20][10];
```

цілком припустимі вирази

```
Ptr[i]=*Vec; // еквівалентно *(Ptr+i)=Vec[0];
Ptr[0]=*(Vec+i); // еквівалентно *Ptr=Vec[i];
**Vec2=Ptr2[6][7]; // еквівалентно Vec2[0][0]=*(*(Ptr2+6)+7);
```

Константність же масиву виявляється в тому, що:

```
Ptr=Vec; // цілком припустимо
Vec=Ptr; Vec2[i] = *Ptr2; /* обидва оператори викликають помилку
                             компіляції */
```

Коли покажчик або масив передається у функцію як параметр, то немає ніякої можливості визначити (у функції) фактичну кількість елементів масиву або кількість об'єктів, на які посилається покажчик. Цю кількість треба передавати у функцію у вигляді окремого параметра або яким-небудь іншим способом. Виключення становлять рядки із завершальним нулем, які передаються як масиви або покажчики на символ, наприклад:

```
void F(char * s, char Mas[5])
{
  cout<<"Довжини:   "<<strlen(s)<<','<<strlen(Mas)<<endl;
  cout<<"Розміри: "<<sizeof( s)<<','<<sizeof( Mas)<<endl;
}

// …
F("три","більше п'яти"); // виводить Довжини:   3, 11 Розміри: 4, 4
```

Зауважимо, що зазначення будь-якої кількості елементів масиву в його описі як формального параметра функції ніякого значення не має: компілятору потрібно тільки вказати, що параметр є масивом – і більш нічого. Крім того, слід пам'ятати, що операція **sizeof** для параметра масиву у функції повертає розмір покажчика, а не кількість елементів масиву.

1.6. Рекурсивні функції

Рекурсивна функція, як відомо, це така, у тілі якої виконується виклик самої себе. Така рекурсія називається прямою рекурсією. Непрямою називають рекурсію, при якій функція **A** містить у явній або неявній формі виклик функції **B()**, яка у свою чергу викликає функцію **A()**.

У чому переваги й недоліки рекурсії?

Перевагою рекурсивних функцій є лаконічність реалізації таких задач, в яких рекурсія закладена в самій структурі даних, що опрацьовуються. Прикладами таких структур даних можуть бути зв'язні списки, дерева та інші ієрархічні структури. Як конкретний приклад можна назвати задачу перегляду всіх каталогів файлової системи, яка являє собою типовий приклад деревоподібної структури даних.

Недоліками рекурсивних функцій є підвищені вимоги до розміру стеку, потенційна небезпека «зациклення» і, зазвичай, алгоритмічна складність її реалізації, наочності і розуміння алгоритму.

Рекурсію неефективно застосовувати в тих випадках, коли можна розв'язати задачу із застосуванням звичайного ітераційного алгоритму. «Класичний» приклад рекурсивної функції – обчислення факторіала, оскільки рекурсія закладена в самому визначенні факторіала:

```
double Fact(int n)
{ return n>1? n*Fact(n-1) : 1; }
```

Виглядає красиво, однак «не все золото, що блищить». Менш витончений варіант реалізації функції

```
double Fact(int n)
{
  if(n<=1) return 1;
  double temp=1;
  for(int i=2;i<=n;temp*=i++);
  return temp;
}
```

не «з'їдає» стековий простір і, якщо не наділяти компілятор суперінтелектом, буде працювати значно швидше і безпечніше рекурсивної функції.

1.7. Функції, що підставляються

Функції, що підставляються або вбудовуються, визначаються так само, як і звичайні функції, за винятком того, що для них додатково вказується ключове слово **inline**, наприклад:

```
inline double Cube(double x)
{ return x*x*x; }
```

Головна перевага функцій, що підставляються, полягає в тому, що компілятор у місце їх виклику вбудовує їх код, а не здійснює виклик функції за допомогою відповідних команд процесора. Застосування функцій, що підставляються, дозволяє одержати більш "швидкий" код завдяки можливому збільшенню розміру програми. Найбільш імовірними кандидатами в функції, що підставляються, є короткі функції.

Не кожна функція, оголошена як **inline**, стане такою насправді. Незважаючи на ключове слово **inline**, компілятор не реалізує функцію, що підставляється, якщо вона:

- містить будь-який із операторів циклу, перемикач, умовний оператор або оператор безумовного переходу;
- є рекурсивною.

Інша причина, яка може унеможливити трактування функції, що підставляється, – це контекст її виклику.

Якщо компілятор не може реалізувати функцію, що підставляється, то він видає попередження **«**Functions containing … are not expanded inline**»**.

1.8. Перевантаження функцій

Перевантажені функції – це дві функції або більше, загальним для яких є тільки ім'я. Перевантажені функції мають відрізнятися одна від одної кількістю параметрів і/або типами параметрів. Перевантажені функції можуть мати різний код.

Яка саме з перевантажених функцій буде викликатися, компілятор визначає за її викликом, точніше, за типами і кількістю фактичних параметрів. Простий приклад – це функція, що повертає абсолютне значення виразу будь-якого типу. Візьмемо приклад програми **Overload**, що ілюструє опис і використання перевантажених функцій.

```
/* Overload.cpp – перевантажені функції */
int Abs(int x) {cout<<"int "; return x>0?x:-x;}
long int Abs(long int x) { cout<<"long int ";return x>0?x:-x; }
float Abs(float x) { cout<<"float ";return x>0?x:-x; }
void main()
{
 cout<<Abs(-123)<<endl; // виводить int 123
```

```
  cout<<Abs(-123/3)<<endl; // виводить int 41
  cout<<Abs(-1234567L)<<endl; // виводить long int 1234567
  cout<<Abs(-10./3.)<<endl; /*викликає повідомлення компілятора про
                              помилку*/
  float x=-100000;
  cout<<Abs(x/3)<<endl; // виводить float 33333.332031
}
```

Спроба виклику функції **Abs(-10./3.)** призводить до виведення компілятором помилки «Ambiguity between 'Abs(int)' and 'Abs(float)'» (неоднозначність між 'Abs(long int)' і 'Abs(float)'). Таке ж повідомлення про помилку з'явиться і при спробі виклику функції **Abs(double(-10./3.))**. Причина полягає в тому, що внаслідок застосування неявного перетворення типів у компілятора виникає проблема вибору між двома варіантами функцій: **long int Abs(long int x)** і **float Abs(float x)**.

1.9. Шаблони функцій

Шаблони функцій (**template**), на відміну від їхнього перевантаження, призначені для розв'язання задачі автоматичного створення компілятором функцій, які можуть обробляти дані різних типів, але алгоритм їх роботи при цьому однаковий. Шаблон функції описується один раз, але особливим чином, і на основі його опису компілятор генерує необхідну функцію залежно від типів фактичних параметрів, зазначених у її виклику. Кількість генерованих компілятором різних функцій дорівнює кількості викликів функції з різними типами параметрів.

Формат опису шаблону функції наведемо на прикладі все тієї ж функції, яка повертає абсолютне значення параметра:

```
template <class Type>
Type Abs(Type x) { return x>0?x:-x;}
```

Після такого визначення шаблону функція **Abs()** може бути викликана з параметром будь-якого типу, який може бути використаний в операції **"x>0?x:-x"**. Останнє зауваження істотне, оскільки спроба викликати цю функцію з параметром типу покажчик призведе до помилки (якій саме?).

Як видно з прикладу, шаблон функції – це свого роду її параметризоване визначення. У наведеному прикладі таким параметром є ідентифікатор **Type**, замість якого при генерації конкретного визначення функції компілятор підставить ім'я типу фактичного параметра, зазначеного при виклику функції. Наприклад, за наявності операторів

```
int x; double  y;
/*...*/
cout<<Abs(x)<<Abs(y);
```

компілятор згенерує такі визначення функцій:

```
int Abs(int x) {return x>0?x:-x;}
double Abs(double x) {return x>0?x:-x;}
```

1.10. Функції зі змінною кількістю параметрів

У мові C++ є можливість розроблення таких функцій, які мають певну кількість обов'язкових параметрів і ще будь-яку кількість необов'язкових параметрів будь-якого типу, не обумовленого в оголошенні функції. Такі функції порівняно складні в написанні, однак дозволяють досягти певної гнучкості в їх використанні. Прикладом такої функції може бути бібліотечна функція **printf()**, призначена для виведення довільного числа значень різних типів. Наприклад, виклик функції **printf(**"Здрастуй, читач!"**);** з одним (обов'язковим) параметром просто призводить до виведення вітання на екран, а в цьому фрагменті функція викликається з двома додатковими параметрами:

```
int i=11; float f=1.23;
printf("Ціле число=%5d Дійсне число=%5.3f", i, f); /*виводить Ціле число
=11 Дійсне число =1.230 */
```

Формат опису функції зі змінною кількістю параметрів такий:

тип_функції ім'я_функції(специфікація_явних_параметрів, ...);

Три крапки наприкінці списку явних формальних параметрів указують на те, що виклик функції може мати довільну кількість додаткових параметрів. У всіх випадках додаткові (необов'язкові) параметри мають вказуватися наприкінці списку фактичних параметрів.

Описи явних параметрів мають задовольняти звичайні вимоги. При виклику функції компілятор перевіряє відповідність типів фактичних параметрів до типів явно заданих формальних параметрів, а необов'язкові фактичні параметри ніяк не перевіряються.

Кількість необов'язкових фактичних параметрів може бути довільною, у тому числі й нульовою.

Позаяк чудеса зустрічаються рідко, то задачею розпізнавання фактичної кількості параметрів і одержання їх значень має займатися програміст. Для одержання доступу до необов'язкових параметрів використовується той факт, що значення фактичних параметрів передаються через стек. При стандартному (модифікатор функції cdecl), прийнятому за замовчуванням, способі передачі фактичних параметрів вони розміщуються в стеку в тому ж порядку, у якому вони були перелічені в списку формальних параметрів. Як ілюстрацію одного зі способів одержання значень необов'язкових параметрів наведемо текст програми **Getoptn**.

```
/* Getoptn - ілюстрація витягу фактичних параметрів за допомогою
```

```
   покажчика. Модифікатор cdecl указувати не обов'язково – він
   передбачається за замовчуванням*/
void cdecl Demoparm(int p1, double p2, int p3)
{
/* настроюємо покажчик ip на перший параметр*/
int *ip=&p1;
/* настроюємо покажчик dp на другий параметр*/
double *dp=(double *)(ip+1);
cout<<"p1="<<*ip<<" p2="<<*dp;
/* настроюємо покажчик ip на третій параметр*/
ip=(int *)(dp+1);
cout<<" p3="<<*ip<<endl;
}
void main()
{
  Demoparm(1, 2.34, 5); // виводить p1=1 p2=2.34 p3=5
}
```

Як видно з прикладу, у функції **Demoparm()** для доступу до значень обов'язкових параметрів потрібна тільки інформація про кількість і типи параметрів. Отже, для одержання значень необов'язкових параметрів можна зробити так, як у наведеному вище прикладі.

Як інші можливі способи розв'язання задачі одержання значень необов'язкових параметрів можна назвати такі:

1. Якщо всі параметри одного типу, то можна в єдиному обов'язковому параметрі передати їхню кількість.
2. В останньому фактичному параметрі передавати спеціальне значення, що вказує на те, що цей параметр є останнім.
3. Якщо параметри мають різні типи, то потрібно яким-небудь способом передавати тип кожного з них, наприклад, як це робиться у функції **printf()**.
4. Скористатися макросами, наявними у файлі **stdarg.h**

Останній спосіб – використання макросів – найкращий, оскільки дозволяє писати надійні програми, які не залежать від конкретного компілятора й апаратної платформи.

1.11. Покажчики на функції

Використання такого типу, як покажчик на функцію, дозволяє ефективно програмувати розв'язання таких задач, у яких потрібно:

- викликати за допомогою одного виразу різні функції з певної сім'ї;
- викликати функції з бібліотек, що динамічно підключаються (dynamic link library – DLL) при завантаженні бібліотек за допомогою функції **LoadLibrary()**;

- передавати функції як параметри в інші функції.

Використання покажчиків на функції виглядає так. Після опису покажчика він зв'язується з потрібною функцією і потім виконується її виклик за допомогою покажчика. Функції, які передбачається викликати за допомогою одного покажчика, повинні мати однакову сигнатуру. Нагадаємо, що термін «сигнатура функції» визначає специфікацію формальних параметрів функції, тобто їхні типи й порядок перелічення, а також тип значення, що повертається функцією.

Визначення покажчика на функцію має такий формат:

тип_функції (*ім'я_покажчика)(специфікація_ параметрів) ініціалізатор;

Тут тип функції – це тип значення, що повертається функцією. Ім'я покажчика – ідентифікатор, за допомогою якого буде здійснюватися виклик функції. Круглі дужки й символ розіменування * є обов'язковими елементами опису.

Специфікація формальних параметрів визначає сигнатуру функції, яка може бути пов'язана з даним покажчиком і викликана.

Ініціалізатор є необов'язковим елементом визначення й дозволяє зв'язати покажчик з функцією безпосередньо при його описі. Визначення обов'язково має закінчуватися крапкою з комою.

Наприклад, визначення

char (*fptr)(float *, int)=func1;

уводить покажчик на функцію, яка повертає значення символьного типу й має два формальні параметри: покажчик на **float** і цілочислового типу. Тут **func1()** – це ім'я функції, з якою зв'язується покажчик. Значенням імені функції **func1()** є адреса початку функції в пам'яті. Функція **func1()** має повертати значення типу **char** і мати два формальні параметри того типу, який зазначено в описі покажчика на функцію.

Формат виклику функції за допомогою покажчика на функцію може бути одним з таких двох рівноцінних:

ім'я_покажчика(список_фактичних_параметрів);
(*ім'я_покажчика)(список_фактичних_параметрів);

Для наведеного вище визначення покажчика **fptr** виклик функції за допомогою покажчика може мати вигляд:

fptr(mas,5); /* *або*/ **(*fptr)(mas,5);**

Зрозуміло, що попередньо покажчик необхідно зв'язати з функцією,що викликається наприклад:

fptr=func; // *func - ім'я функції*

Як перший приклад наведемо програму **Demofptr**, яка використовує покажчик на функцію для того, щоб по черзі викликати чотири прості функції.

```
// Demofptr.cpp – використання покажчика на функцію
float add(float a, float b)  { return a+b;}
float sub(float a, float b)  { return a-b; }
float mult(float a, float b) { return a*b; }
float div(float a, float b)   { return a/b; }
float (*oper)(float, float); // оголошення покажчика на функцію
void main()
{
 float c1=10,c2=20;
 char ch='+';
 while(ch!=' ')
 {
   cout<<c1<<ch<<c2<<"=";
   switch (ch)
  {
    case '+': oper=add; ch='-'; break;
    case '-': oper=sub; ch='*'; break;
    case '*': oper=mult; ch='/'; break;
    case '/': oper=div; ch=' '; break;
  }
  cout<<oper(c1,c2)<<endl;
 }
}
```

Покажчики на функції незамінні тоді, коли потрібно, щоб яка-небудь функція могла викликати іншу функцію, передану їй як параметр. Наприклад, якщо програмуємо обчислення певного інтеграла яким-небудь числовим методом для будь-якої функції, необхідно мати можливість виконати виклик цієї функції, переданої як параметр. Зробити це можна з використанням покажчика на функцію.

Задамо прототип функції обчислення інтеграла на відрізку **a,b**:

```
float integral(покажчик_на_функцію, float a, float b);
```

Для того, щоб використовувати покажчик на функцію як параметр іншої функції, спочатку визначимо його тип за допомогою ключового слова **typedef**:

```
typedef  float (*Tuserfunc) (float);
```

Тепер прототип функції обчислення певного інтеграла набуде вигляду:

```
float integral(Tuserfunc Userfunc, float a, float b);
```

Програма **Funcpar** ілюструє передачу функції як параметра в іншу функцію.

```
// Funcpar.cpp - передачі функції в іншу функцію як параметр
/* Тестова функція func()  повертає значення інтегрованої  функції
    у точці  x*/
double func(double x)
{ return sin(x);}
typedef double (*Tuserfunc)(double);
/* Функція integral() обчислює й повертає інтеграл від функції Userfunc на
інтервалі a..b методом трапецій */
double integral(Tuserfunc Userfunc, double a, double b)
{
  const int n=1000; /*кількість відрізків, на які розділяється інтервал a..b.
        Більші значення цієї константи дають більш високу точність*/
  double integ=0.0, dx=(b-a)/(n-1);
  for (double x=a;x<=b;x+=dx) integ+=Userfunc(x);
  return integ*dx;
}
void main()
{
   cout<<integral(func, 0, M_PI)<<endl;
}
```

У результаті виконання програми одержимо 1.999998 при **n=1000** і 2 – при **n=2000**. Слід звернути увагу на константу **M_PI**, визначену у файлі **math.h**.

2. СТРУКТУРИ Й ОБ'ЄДНАННЯ

2.1. Структури

Структури спільно з масивами, класами й об'єднаннями входять до групи структурних типів мови, які, у свою чергу, входять до групи похідних типів, тобто типів, що створює програміст для своїх потреб. У мові C++ структури й об'єднання можуть використовуватися так, як вони були визначені у мові С, а можуть також використовуватися як класи в ООП. Розглянемо структури й об'єднання у трактуванні мови С, тобто не як класи.

Структуру можна визначити як безліч різнотипних елементів, об'єднаних у єдине ціле. Поняття структури у цьому сенсі подібне поняттю масивів, однак відрізняється від них саме можливістю об'єднання елементів різних типів. Таким чином, структуру можна трактувати як масив, елементи якого мають власні імена, а не індекси.

У загальному випадку визначення структури має вигляд:

struct ім'я_типу_структури {тип_1 ім'я_1; тип_2 ім'я_2; /*...*/
тип_n ім'я_n;} ім'я_екземпляра_структури ініціалізатор;

Тут **struct** є ключовим словом мови C++. Ім'я типу структури позначає те, що можна назвати іменем даного структурного типу й може бути використане надалі для визначення екземплярів структури, наприклад змінних, покажчиків на структури і т. д. Тип_1, тип_2 і т. д. є іменами типів компонент (членів) структури, тобто тих елементів, які поєднує структура, а ім'я_1, ім'я_2 та інші є іменами цих компонентів. Ім'я екземпляра структури – це, наприклад, ім'я змінної структурного типу, для якої виділяється відповідний розмір пам'яті. Ініціалізатор, зазвичай дозволяє задати початкові значення для компонентів структури.

Опис і визначення структур, які будуються за наведеним вище форматом, досить різноманітні і у них можуть бути відсутні деякі з наведених елементів опису. Розглянемо це докладніше за допомогою прикладів.

Нехай маємо таке визначення структури:

struct TStudent {char *fam, *name, *sname; int mark[10]; char contract;}
anyst;

Цей структурний тип, що має ім'я **TStudent**, поєднує п’ять компонентів різних типов. **anyst** – змінна структурного типу, для якої виділяється стільки пам'яті, скільки необхідно для зберігання всіх її п’яти компонентів, а саме:

sizeof(char*)*3 + sizeof(int)*10+ sizeof(char)==4*3+4*10+1==53 байт

Ім'я типу **TStudent** може бути надалі використане для опису інших об'єктів цього структурного типу, наприклад:

TStudent *ptr_st; // *покажчик на структуру*

TStudent group[25]; // *масив структур*

Опис кожного компонента, у тому числі останнього, має закінчуватися крапкою з комою. При такому описі структури

struct TStudent {char *fam, *name, *sname; int mark[10]; char contract;};

уводиться тільки опис типу структури **TStudent** і не створюється жодного об'єкта. Конкретні об'єкти – екземпляри структури – можна тоді описати так, як це вже було показано вище:

TStudent *ptr_st; // *покажчик на структуру*
TStudent group[25]; // *масив структур*

При визначенні змінних структурного типу можна виконати їхню ініціалізацію:

struct {char *fam, *name, *sname; int mark[10]; char contract;}
any_st={"Безпрізвищний"**,**"Іван"**,**"Петрович"**, {3,4,5,4,3,4,5,4,3,4},**'Y'**};**

Звертання до компонентів структури виконується за допомогою так званих уточнених (кваліфікованих) імен:

ім'я_структури.ім'я_компонента

де ім'я структури – це ім'я змінної структурного типу (але не ім'я структурного типу!). Якщо змінна структурного типу є покажчиком на структуру, то при звертанні до компонента структури можна рівноправно використовувати одну з таких форм уточненого імені:

(*ім'я_покажчика).ім'я_компонента
ім'я_покажчика->ім'я_компонента

Наприклад, при такому визначенні змінних структурного типу

struct TStudent {char *fam, *name, *sname; int mark[10]; char contract;}
any_st, *ptr_st, group[25];

припустимими будуть такі уточнені імена:

any_st.fam
(*ptr_st).mark[0]
ptr_st->contract
group[1].name

До змінних одного структурного типу можна застосовувати операцію присвоєння, наприклад:

any_st=*ptr_st;

У результаті виконання цієї операції всім компонентам структури **any_st** присвоюються значення відповідних компонентів структури ***ptr_st**. Ніякі інші операції над структурами в цілому виконувати не можна.

2.2. Об'єднання

Об'єднання **union** (суміші) багато в чому схожі за своїми властивостями на структури **struct**, але відрізняються від них тим, що компоненти об'єднання "накладаються" один на одного, займаючи те саме місце пам'яті. Наприклад, структура

struct st_type {char cmas[4]; double fvar; int int1,int2;} st1;

займає 4 + 8 + 4 * 2 = 20 байт пам'яті і її компоненти розташовуються в пам'яті послідовно один за одним. Розмір цієї структури можна одержати й за допомогою операції **sizeof**:

sizeof(st_type)==20 sizeof(st1) ==20

Тепер замінимо слово **struct** на **union**, визначивши у такий спосіб об'єднання:

union un_type {char cmas[4]; double fvar; int int1,int2;} un1;

Розмір об'єднання дорівнює розміру максимального з його компонентів і у цьому випадку маємо **sizeof(un_type)==8**.

Одержати доступ до кожного з компонентів об'єднання можна точно так само, як і для структур:

ім'я_об'єднання.ім'я_компонента
(*ім'я_покажчика).ім'я_компонента
ім'я_покажчика->ім'я_компонента

Наприклад:

un1.cmas[0]='a'; un1.int1=123;

Об'єднання можуть бути, подібно структурам, ініціалізовані, але при цьому початкове значення можна задати тільки для першого компонента, наприклад:

union un_type{char cmas[4]; double fvar; int int1,int2;} un1={'a','b','c','d'};

3. УВЕДЕННЯ У КЛАСИ

3.1. Основні принципи об'єктно-орієнтованого програмування

Поняття класу як розширення поняття структури є головною відмінною рисою мови C++ від її попередниці C. Назва C++ з'явилася не відразу, а як більш удала заміна назви "C с класами". Така назва мови – C++ – була запропонована Ріком Масситті (Rick Mascitti) у 1983 р. і закріпилася, оскільки образно відтворила родовід цієї мови. Автором мови C++ уважається Бьярн Страуструп (Bjarne Stroustrup).

Класи в C++ надають програмістам можливість розробляти програми відповідно до принципів ООП. Ідеї ООП у наш час є найбільш передовими й широко використовуються в багатьох мовах програмування.

Головними ідеями ООП є **інкапсуляція**, **успадкування** й **поліморфізм**. Деякі автори додають до цих трьох «китів» ще й **абстракцію**, і вони праві.

Інкапсуляція – це об'єднання в єдине ціле даних і функцій, які їх обробляють. Таке об'єднання дозволяє забезпечити нерозривність програмного коду і даних, що, у свою чергу, дає такі вигоди, як підвищення надійності програм, можливість їх доробок не шляхом модифікації, а завдяки розширенню, тобто додаванню нових функцій. Об'єднання даних і методів їх оброблення в об'єктах дозволяє максимально ізолювати дані від іншої частини програми, тим самим забезпечуючи їх цілісність і можливість простого використання в інших програмах.

Успадкування є властивістю одних об'єктів породжувати інші. Об'єкт-нащадок (descendant) автоматично успадковує від об'єкта батька (parent) дані й функції їх оброблення й може як доповнювати об'єкти новими даними й функціями, так і перекривати дані і функції об'єкта батька своїми, у такий спосіб ніби замінюючи їх. Властивість успадкування вирішує гостру проблему модифікації «чужого» коду й надає ООП у цілому виняткову гнучкість, оскільки дозволяє повторно використовувати наявний код і розширювати його не шляхом модифікації, а шляхом заміщення успадкованих функцій і додаванням своїх. При розробленні нової програми програміст підбирає найбільш підходящий об'єкт і створює одного або декілька нащадків від нього, які вміють робити те, чого не може батьківський об'єкт.

Поліморфізм – це властивість об'єктів-родичів, тобто об'єктів однієї ієрархії, яка дозволяє вирішувати ті самі задачі різними методами, що враховують і реалізують специфічні особливості конкретного класу. У результаті функція з тим самим іменем використовується багатьма об'єктами однієї ієрархії, але в кожному класі вона має свою реалізацію. Поліморфізм дозволяє також розробляти такі функції, які можуть бути

використані й майбутніми, ще навіть не спроектованими класами і будуть працювати так, як того потребує новий об'єкт.

Бібліотеки всіх сучасних інструментальних середовищ розроблення програм та різні операційні системи надають свої сервіси програмістові в переважній більшості випадків у формі об'єктів, унаслідок чого розроблення сучасних програм неможливе без знання ООП.

3.2. Поняття класу

Поняття класу в C++ є найбільш близьким до поняття структури і, можна сказати, є подальшим її розвитком. Клас в C++ дозволяє визначити сукупність даних і функцій їх оброблення, а також визначати такі операції над цими структурованими даними, які не визначено в мові, що становить суть перевантаження стандартних операцій.

У найпростішому випадку опис класу можна зробити так:

ключ_класу ім'я_класу {компоненти_класу};

Тут *ключ_класу* – будь-яке із ключових слів **class, struct** або **union**. *Ім'я_класу* – довільно вибираний ідентифікатор, що являє собою ім'я типу. *Компоненти_класу* – це опис і/або визначення типів, даних, функцій, інших класів, перелічень, дружніх функцій і бітових полів.

Для спрощення з початку будемо вважати, що компонентами класу є компонентні дані й функції їх оброблення. Компонентні дані називають також член-даними класу, атрибутами або полями, а компонентні функції – член-функціями або методами. В описі класу виділяють *заголовок* і *тіло*. Тіло класу – це укладений у фігурні дужки список компонентів, а заголовок – це те, що передує тілу:

заголовок_класу {тіло_класу};

Наведений формат опису є тільки оголошенням типу і на його основі не створюється ніяких об'єктів, яким виділялася б пам'ять. Для визначення *екземпляра класу*, який зазвичай називають *об'єктом*, його ім'я можна додати до опису класу:

ключ_класу ім'я_класу {компоненти_класу} ім'я_об'єкта;

або визначити окремо, що є важливішим:

ім'я_класу ім'я_об'єкта;

Нехай, наприклад, маємо опис такого класу:

```
struct
{
  int x,y; unsigned char ch;
  void Show()
  {
    gotoxy(x,y);// переводить курсор у позицію вікна з координами x, y
```

```
      putch(ch);   // виводить символ
    }
    void Hide() { gotoxy(x,y); putch(' '); }
    void MoveTo(int newx, int newy)
    {
     Hide(); x=newx; y=newy; Show();
    }
  } point1, point2;
```

Зверніть увагу на те, що поля класу не передаються методам класу як фактичні параметри, а використовуються ними безпосередньо. Крім того, методи класу можуть викликати один одного, причому порядок опису даних і методів у класі значення не має, оскільки клас має свою власну область видимості описів. Ідентифікатори компонентів класу мають бути унікальними тільки в межах опису класу, тобто в його тілі.

Визначено також два об'єкти **point1** і **point2** цього класу. Використання цих об'єктів ілюструє програма **Point1.cpp**.

```
/* Point1.cpp - перший варіант використання класу "точка"*/
int Random(int num) // повертає псевдовипадкове число з діапазону 0..num-1
{
 int N = ((double)rand() / RAND_MAX * num - 1);
 return N;
}

void main()
{
    srand(time(0)); // ініціалізація датчика псевдовипадкових чисел
    point1.x=60; point1.y=12; point1.ch='*'; point1.Show();
    point2.x=40; point2.y=12; point2.ch='#'; point2.Show();
    while(!kbhit()) // доки не натиснута будь-яка клавіша
    {
        point1.MoveTo(Random(80),Random(25)); Sleep(200);
        point2.MoveTo(Random(80),Random(25)); Sleep(200);
    }
    point1.Hide(); point2.Hide();
}
```

Програма переміщує випадковим чином два символи на екрані до натискання будь-якої клавіші. У ній використовується доступ до полів і методів об'єктів шляхом указання кваліфікованих імен.

Як і для структур, можна визначити покажчик на об’єкт класу:

```
struct {/* тіло класу*/} *pointptr;
```

і тоді доступ до полів і методів можна буде одержувати так:

```
pointptr->x=123; pointptr->Hide() ; /*або*/ (*pointptr).x=123;
(*pointptr).Hide();
```

Зрозуміло, що попередньо необхідно виділити динамічну пам'ять для об'єкта, що буде розглянуто пізніше.

Використання ключового слова **union** замість **struct** дозволяє створити клас, поля якого мають ті ж властивості, що й компоненти об'єднання. Такий клас, однак, не може мати нащадків і на практиці зустрічається рідко.

Використання ключового слова **class** створює клас, поля й методи якого недоступні для зовнішніх функцій. Такий клас навряд чи є корисним і тому для забезпечення доступу до компонентів класу використовуються специфікатори доступу **protected** (захищений) або **public** (загальнодоступний), які будуть розглянуті нижче.

Наведений у програмі **Point1** клас має істотний недолік: доступ до значень полів реалізується шляхом безпосереднього звертання до них, що позбавляє програму гнучкості. Зовнішня програма, тобто програма, що використовує даний клас, залежить, зокрема, від ідентифікаторів і типів полів і при їхньому зміненні з великою ймовірністю також має буде піддатися змінам. Крім того, задання початкових значень необхідно робити завжди, для кожного екземпляра (об'єкта) класу, воно є громіздким, затінюючи суть розв'язуваної задачі. Шляхи усунення цих недоліків розглянуто у наступному підрозділі.

Конструктори і розмежування доступу до компонентів класів

Поля й методи класу, розглянутого в попередньому підрозділі, є доступними як усередині класу – методи звертаються до полів безпосередньо й можуть також викликати один одного, – так і для зовнішньої програми, що використовує цей клас. Дотримуючись принципу приховання даних в інтересах самої ж зовнішньої програми, можна обмежити безпосередній доступ до компонентів класу двома способами:

- шляхом використання ключа класу **class** замість **struct**;
- шляхом використання усередині класу специфікаторів доступу **public, private** і **protected.**

Якщо в опису класу вказати ключ **class** замість **struct**, то всі поля й методи класу стають недоступними для програми й при спробі звернутися до них викликають помилку компілятора "::x is not accessible" – «компонент x недоступний». Наприклад, при такому визначенні класу:

```
class CFoo
{
int x,y;
    void Show() { cout<<"x="<<x<<" y="<<y;}
}obj;
```

оператор **obj.x=0** або **obj.Show()** викличе помилку компілятора. Отже, такий клас може мати тільки обмежене застосування, оскільки зовнішня програма може тільки створювати екземпляри такого класу й виконувати операції над об'єктами в цілому, наприклад копіювати їх.

Специфікатор доступу **public** (загальнодоступний) робить усі розташовані після нього поля й методи доступними для зовнішньої програми. Специфікатор **private** (особистий, власний) має прямо протилежну дію: усі розташовані після нього поля й методи стають недоступними для зовнішньої програми. Специфікатор **protected** (захищений) має особливий сенс при побудові ієрархії класів і його вживання у «звичайному» класі еквівалентне дії специфікатора **private**. Якщо клас визначено із ключем класу **class** і в ньому не використано жоден зі специфікаторів доступу, то за замовчуванням використовується специфікатор **protected**.

В описі класу після кожного зі специфікаторів доступу ставиться двокрапка. Область дії будь-якого специфікатора поширюється від місця його появи до кінця опису класу або до іншого специфікатора. Будь-який специфікатор може бути вжитий в описі класу багаторазово.

Наприклад, при такому описі класу:

```
class CFoo
{
 public:
    int x;
 private:
    int y;
 public:
    void Show(){ cout<<"x="<<x<<" y="<<y;}
}obj;
```

поле **x** і метод **Show()** є загальнодоступними, а поле **y** – особистим. Зрозуміло, наведений приклад використання специфікаторів є тільки ілюстрацією властивостей специфікаторів: на практиці саме таке розмежування доступу навряд чи має сенс. Більше того, з міркувань доцільності й наочності рекомендується спочатку перелічити відкриті члени, а потім – закриті.

Таким чином, за допомогою специфікаторів доступу можна зробити недоступними для зовнішньої програми деякі поля й методи для того, щоб зробити програму більш незалежною від реалізації класу. Очевидний сенс полягає в тому, щоб заборонити зовнішній програмі звертатися безпосередньо до полів класу і завдяки цьому досягтися таких цілей:

- уникнути можливих помилок через некоректну модифікацію даних;
- зробити програму, що використовує клас, незалежною від даних класу;

- надати можливість розробнику класу модифікувати його дані без необхідності внесення змін у програму, що використовує клас.

Треба зазвичай надати зовнішній програмі можливість установлювати значення полів і одержувати їх. Зробити це можна за допомогою спеціально для цього призначених методів, що й ілюструється в програмі **Point2**, яка містить опис удосконаленого класу **TPoint2**.

```
/* Point2.cpp - використання ключових слів private і public для обмеження
доступу до даних і використання функції Init() для присвоєння значень
атрибутам  об'єкта */
struct TPoint2
{
    private: // особисті компоненти
       int x,y; unsigned char ch;
    public: // загальнодоступні компоненти
            void Init(int ix, int iy, unsigned char c) {x=ix; y=iy; ch=c;}
            void Show()  { gotoxy(x,y); putch(ch);}
            void Hide()   { gotoxy(x,y); putch(' ');}
            void MoveTo(int newx, int newy)
            {
                  Hide(); x=newx; y=newy; Show();
            }
           int getx() {return x;}
           int gety() {return y;}
 }point1,point2;

void main()
{
    srand(time(0)); // ініціалізація датчика випадкових чисел
    point1.Init(40,10,'*'); point1.Show();
    point2.Init(20,12,'#'); point2.Show();
    while(!kbhit())
    {
      point1.MoveTo(Random(80), Random(25));Sleep(200);
      point2.MoveTo(Random(80), Random(25));Sleep(200);
    }
    _getch();
    gotoxy(point1.getx()+1,point1.gety());
    cout<<"x="<<point1.getx()<<" y="<<point1.gety();
    gotoxy(point2.getx()+1,point2.gety());
    cout<<"x="<<point2.getx()<<" y="<<point2.gety(); _getch();
    point1.Hide(); point2.Hide();
 }
```

Метод **Init()** дозволяє зовнішній програмі встановити значення полів, а методи **getx()** і **gety()** – одержати їх. Включення цих методів до класу, які,

як може здатися на перший погляд, не мають ніякого сенсу, дозволяє досягти дуже важливого результату: тепер зовнішня програма стала незалежною принаймні від типів цих полів та їх імен. Завдяки наявності цих методів можна змінити типи полів будь-яким чином, наприклад увести покажчики або масиви, а зовнішню програму змінювати не доведеться.

Як видно з тексту програми, використання спеціального методу **Init()** для задання початкових значень компонентних даних робить програму більш стислою і наочною.

Можна також задати початкові значення для формальних параметрів методу **Init()**:

```
void Init(int ix=40, int iy=12, unsigned char c='*')
{x=ix; y=iy; ch=c;}
```

Зазначимо, що метод **Init()** можна викликати і з указанням частини фактичних параметрів **point2.Init(20,12)**, оскільки всі угоди щодо звичайних (некомпонентних або, як іноді кажуть, глобальних) функцій з початковими значеннями формальних параметрів поширюються й на компонентні функції класів.

Присвоїти початкові значення всім або деяким полям класу й виконати будь-які інші дії з ініціалізації об'єкта можна також за допомогою спеціального методу, який називається конструктором. Конструктор описується як функція з іменем, що збігається з іменем класу:

```
ім'я_класу (список_формальних_параметрів)
{ /* тіло_конструктора */}
```

Конструктор, на відміну від інших функцій, не може повертати значення. Список формальних параметрів конструктора може бути порожнім, але зазвичай в ньому вказують формальні параметри, інколи зі значеннями за замовчуванням, за допомогою яких виконують ініціалізацію полів об'єкта. Головною рисою й відмінністю конструктора від «звичайної» компонентної функції є те, що конструктор викликається автоматично при визначенні об'єкта – екземпляра класу. Якщо клас має хоча б один конструктор, то він буде обов’язково викликаний на етапі виконання програми.

Якщо формальні параметри конструктора не мають значень за замовчуванням, то при будь-якому визначенні об'єкта для нього необхідно вказувати фактичні параметри.

Явний виклик конструктора здійснюється одним із трьох нееквівалентних способів:

- **ключ_класу ім'я_класу {тіло_класу} ім'я_об'єкта (фактичні_параметри);**
- **ім'я_класу ім'я_об'єкта (фактичні_параметри);**
- **ім'я_класу (фактичні_параметри).**

За першим способом конструктор викликається при описі класу й визначенні об'єкта класу.

Другий спосіб використовується при визначенні нового об'єкта описаного раніше класу.

Третій спосіб використовується при визначенні так званого безіменного об'єкта, який можна використовувати у виразі відповідного типу, наприклад як фактичний параметр якої-небудь функції.

В одному класі можна визначити декілька конструкторів подібно тому, як це робиться при перевантаженні функцій, але тільки один з них може мати усі параметри зі значеннями за замовчуванням.

Програма **Point3.cpp** ілюструє застосування конструктора і його явний і неявний виклики.

```
/* Point3.cpp - використання конструктора з параметрами за замовчуванням
*/
void main()
{
  struct TPoint3
  {
  private:
    int x,y; unsigned char ch;
  public:
    TPoint3(int ix=40, int iy=12, unsigned char ich='*')
    { x=ix; y=iy; ch=ich;}
    void Show() { gotoxy(x,y); putch(ch);}
    void Hide() { gotoxy(x,y); putch(' ');}
    void MoveTo(int newx, int newy) { Hide(); x=newx; y=newy; Show();}
    int getx() {return x;}
    int gety() {return y;}
} point1; /* тут виконується неявний виклик конструктора зі значеннями
             параметрів за замовчуванням для об'єкта point1*/
TPoint3 point2(20,20,'#'); /*визначення нового об'єкта і  явний виклик
                             конструктора*/
cout<<" x1="<<point1.getx()<<" y1="<<point1.gety()<<endl;
cout<<" x2="<<point2.getx()<<" y2="<<point2.gety();
_getch();
}
```

Третій спосіб явного виклику конструктора при використанні безіменного об'єкта можна проілюструвати на прикладі передачі об'єкта як параметра функції. Нехай є деяка функція з таким прототипом:

```
void Func(TPoint3 apoint);
```

Можна спочатку визначити об'єкт і передати його у функцію:

```
TPoint3 Obj(20,20,'#'); // явний виклик конструктора
Func(Obj); // передача об'єкта у функцію
```

Якщо ж цей об'єкт потрібний тільки для його передачі у функцію **Func()**, то два попередні оператори можна замінити одним:

Func(TPoint3(20,20,'#')); **/*** *створення безіменного об'єкта й передача його у функцію* *******/*

Конструктори можна використовувати не тільки для задання початкових значень полів класу, але й для виконання будь-яких інших операцій, які необхідно завжди виконувати при створенні нового екземпляра об'єкта. Якщо, наприклад, для полів об'єкта має бути виділена пам'ять у динамічній області, то цю операцію доцільно включити до конструктора, забезпечивши тим самим обов'язкове її виконання.

Конструктор, зазвичай, має бути визначений зі специфікатором доступу **public** для того, щоб його можна викликати явним або неявним способом. Якщо конструктор оголосити зі специфікатором **protected**, то такий конструктор можна буде викликати тільки у похідному класі.

> **Зауваження**. До параметрів конструктора можна додати ім'я об'єкта для того, щоб у процесі налагодження програми з декількома об'єктами їх можна було ідентифікувати.

3.3. Види конструкторів і особливості їх використання

У цьому підрозділі мова йтиме про різні види конструкторів і використання в класі декількох конструкторів.

Конструктор може бути:

- конструктором з формальними параметрами, що не мають значень за замовчуванням;
- конструктором з початковими, тобто за замовчуванням, значеннями формальних параметрів;
- конструктором за замовчуванням, який не має формальних параметрів взагалі;
- конструктором копіювання.

Якщо для формальних параметрів конструктора не задано значення за замовчуванням, то при його явному або неявному виклику мають бути обов'язково задані фактичні параметри. Наприклад, таке визначення екземпляра класу point1

```
struct TPoint3
{ //...
  TPoint3 (int ix, int iy, unsigned char c) { x=ix; y=iy; ch=c;}
  // …
} point1;
```

викличе повідомлення компілятора про помилку "Could not find a match for 'TPoint3:: TPoint3 ()' ", яке можна перевести приблизно як "Не можу знайти нічого підходящого для 'TPoint3::TPoint3()'", з указанням на останній рядок наведеного фрагмента програми.

Клас може мати декілька конструкторів, які залежно від деяких умов будуть ініціалізувати екземпляр класу тим чи іншим чином. На конструктори класу в цьому випадку поширюються ті ж угоди, що й на перевантажені функції. У наведеному вище прикладі **Point3.cpp** використано один конструктор: з параметрами за замовчуванням. Приклади інших конструкторів:

```
TPoint3 (){ x=1;y=1;ch=' ';} //конструктор за замовчуванням
TPoint3 (const TPoint3 &org) //конструктор копіювання
{ /*оператори "глибокого" копіювання*/}
```

Конструктор копіювання, якщо він не визначений явно, створюється компілятором автоматично й викликається при створенні об'єктів-копій таким способом:

```
TPoint3 p1(1,1,'*');
TPoint3 p2(p1); /* для створення p2 викликається конструктор
                   копіювання*/
TPoint3 p3=p1; /*для створення p3 викликається конструктор копіювання */
```

Об'єкти **p2** і **p3** будуть мати ті ж значення полів, що й об'єкт **p1**. Зверніть увагу, що для об'єктів **p2** і **p3** параметри не вказуються.

Конструктор копіювання викликається також при виконанні оператора **return** у функції, тип якої – клас (див. приклад в кінці цього підрозділу).

Треба зазначити, що конструктор копіювання *не викликається* при виконанні операції присвоєння, наприклад **p2=p1**.

Необхідно відзначити важливу деталь: конструктор копіювання, який створюється за замовчуванням, виконує так зване *поверхневе копіювання*, при якому копіюються тільки значення полів об'єктів. Якщо поля класу є покажчиками, то копіюються, природно, значення покажчиків, тобто адреси, але не виділяється динамічна пам'ять під зберігання значень об'єкта-копії, на які настроєні покажчики. Отже, у цьому випадку в конструкторі копіювання необхідно, крім копіювання значень полів не покажчиків, виділити динамічну пам'ять для полів покажчиків і лише після цього скопіювати ті дані, на які вказують покажчики об'єкта оригіналу. Ці операції так званого *глибокого копіювання* програміст повинен виконати самостійно.

У програмі **Point4.cpp** ілюструється застосування конструктора за замовчуванням без параметрів і власного конструктора копіювання.

```
/*Point4.cpp – використання конструктора за замовчуванням і власного
             конструктора копіювання */
struct TPoint4
{
 private:
```

```
  int x,y; unsigned char ch;
    public:
  // конструктор за замовчуванням
  TPoint4 () { x=1; y=1; ch='?';}
  // власний конструктор копіювання
  TPoint4 (const TPoint4 &org)
  {
    x=org.x<80?org.x+1:1;
    y=org.y<25?org.y+1:1;
    ch=org.ch+1;
  }
  void Show() { gotoxy(x,y); putch(ch);}
  void Hide() { gotoxy(x,y); putch(' ');}
  void MoveTo(int newx, int newy)
  { Hide(); x=newx; y=newy; Show();}
  int getx() {return x;}
  int gety() {return y;}
  };

  void main()
  {
     TPoint4 point1; /* тут виконується неявний автоматичний виклик
                        конструктора для об'єкта point1 */
     TPoint4 point2=point1; /* визначення нового об'єкта point2 з неявним
                         викликом  конструктора копіювання */
     point1.Show();  point2.Show();
     _getch();
  }
```

Порада. Можливо, Вам захочеться довідатися, коли саме викликається який-небудь конструктор або деструктор і чи викликається він взагалі. Можна це зробити шляхом встановлення контрольних точок (breakpoint) і в режимі налагодження трасувати програму. Також можна звернутися до такого відомого способу налагодження: скористатися макросами **TRACE** або **ATLTRACE** для виведення необхідної інформації у вікно **Output** середовища програмування MVS. Приклади використання цих макросів наведені у подальших підрозділах.

При визначенні декількох конструкторів в одному класі не повинно виникати неоднозначності (ambiguity) при їхньому виклику. Наприклад, при такому визначенні класу і його об'єктів:

```
  class TDemoCons
  {
    public:
      int x,y;
```

```
    // конструктор за замовчуванням
    TDemoCons() {x=1; y=1;}
    /* конструктор з параметрами, для яких задані значення за
       замовчуванням */
    TDemoCons(int ix=20, int iy=20)
    {x=ix; y=iy;}
};
void main()
{
    TDemoCons Obj1(1,1);
    TDemoCons Obj 2;
}
```

компілятор зможе однозначно визначити конструктор для об'єкта **Obj1** – конструктор з параметрами за замовчуванням, але видасть повідомлення про помилку при визначенні об'єкта **Obj2**, оскільки не може однозначно визначити, який з конструкторів треба використовувати.

3.4. Деструктори

Іншим спеціальним методом класу, крім конструктора, є деструктор, який визначається так:

~имя_класу() { */* тіло деструктора*/***}**

Деструктор викликається автоматично при знищенні об'єкта, але також може бути викликаний явно. Знищення об'єкта відбувається тоді, коли, наприклад, відбувається вихід із блока або функції, де був створений (визначений) об'єкт, тобто при виході з області дії опису даного об'єкта. Деструктор необхідний і зручний тоді, коли при знищенні об'єкта мають бути виконані певні дії. Наприклад, якщо поля об'єкта були розміщені в динамічній пам'яті, то цю пам'ять необхідно звільнити перед знищенням об'єкта, оскільки в протилежному випадку вона буде загублена для програми. На відміну від конструктора, клас може мати тільки один деструктор.

Зауваження. Якщо завершення роботи програми виконується шляхом виклику функції **exit()**, то викликаються деструктори тільки статичних об'єктів, тобто оголошених у глобальній області або з модифікатором **static**. Для локальних об'єктів деструктори у випадку виклику функції **exit()** не викликаються. Якщо в програмі був визначений покажчик на об'єкт і об'єкт створювався за допомогою операції **new**, то виклик деструктора відбувається при звільненні пам'яті, виділеної для об'єкта, за допомогою операції **delete.**

Деструктор не може мати значення, що повертається, і формальних параметрів. Ім'я деструктора – це завжди ім'я класу, безпосередньо перед яким зазначений знак тильда ~. Деструктор, як і конструктор, має бути визначений зі специфікатором доступу **public**, інакше він не може бути викликаний явним або неявним способом.

У програмі **Point5.cpp** деструктор просто викликає метод **Hide()**, унаслідок чого виведені крапки стираються з екрана при завершенні програми, коли відбувається автоматичне знищення об'єктів класу **TPoint5**.

```
/* Point5.cpp – явний і неявний виклики деструкторів */
struct TPoint5
{
  private:
    int x,y; unsigned char ch;
  public:
    TPoint5 (int ix=40, int iy=12, unsigned char c='*')
    { x=ix; y=iy; ch=c;}
    void Show() { gotoxy(x,y); putch(ch);}
    void Hide() { gotoxy(x,y); putch(' ');}
    void MoveTo(int newx, int newy)
    { Hide(); x=newx; y=newy; Show(); }
    ~TPoint5() {Hide();} // деструктор
};

void main()
{
  TPoint5 point1;
  TPoint5 point2(20,20,'#'); /* визначення нового об'єкта і явний виклик
                                конструктора*/
  point1.Show(); point2.Show();
  _getch();
  point1.~TPoint5(); // явний виклик деструктора для point1
  _getch();
}  /* тут відбувається неявний повторний виклик деструктора для об’єкта
      point1 і виклик деструктора для point2 */
```

Зверніть увагу на те, що при завершенні роботи програми – виході з функції **main()** – викликаються деструктори для об'єктів **point1** і **point2**. Таким чином, явний виклик деструктора **point1.~TPoint5()** у даному контексті не є необхідним. Більше того, для реального об'єкта повторний виклик деструктора швидше за все призведе до помилки (наприклад, при повторному звільненні динамічної пам'яті). Внаслідок цього явний виклик деструктора потрібно виконувати тільки тоді, коли це дійсно необхідно відповідно до алгоритму програми. Аналіз подібних випадків виходить за межі даної роботи, але його можна знайти у роботі [1].

Деструктор автоматично викликається тоді, коли об'єкт припиняє своє існування, наприклад, коли потік виконання програми виходить із області опису об'єкта. Наприклад,, якщо об'єкт визначений у блоці або функції

```
{ // початок блока або тіла функції
   TPoint5 point2(20,20,'#');
   // …
}// кінець блока або тіла функції
```

то виклик деструктора буде виконаний у момент виходу із блока або функції, у тому числі при виконанні оператора **return**.

3.5. Розміщення екземплярів класу в динамічній пам'яті

Динамічна пам'ять є найбільш зручним сховищем екземплярів класу в тих випадках, коли об'єкти потрібні не на всьому протязі виконання програми, а тільки на деяких етапах. У цьому випадку використання динамічної пам'яті для розміщення об'єктів найбільш доцільне з міркувань мінімізації розміру пам'яті, необхідної для програми.

При визначенні покажчика на екземпляр класу й використанні операції **new** для виділення динамічної пам'яті явний або неявний виклик конструктора проводиться при виконанні операції **new**:

```
TPoint *point1,*point2, *point3;
/* резервування пам'яті й виклик конструктора з фактичними
   параметрами*/
point1=new TPoint (1,1,'1');
/* резервування пам'яті й виклик конструктора з параметрами
  за замовчуванням*/
point2=new TPoint;
/* резервування пам'яті для трьох екземплярів класу й виклик конструктора
   з параметрами за замовчуванням*/
point3=new TPoint [3];
// виклики компонентних функцій
point1->Show();
(*point1).Show();
point3[i].Hide();
```

Таким чином, формат використання операції **new** для виділення динамічної пам'яті екземпляра класу такий:

```
екземпляр_класу = new  ім'я_типу_класу (фактичні параметри)
[кількість_екземплярів_класу];
```

Якщо пам'ять потрібно виділити тільки для одного об'єкта, то кількість екземплярів класу вказувати не потрібно.

Виклик компонентних функцій при визначенні покажчика на екземпляр класу виконується, як видно з прикладу, з використанням операції доступу до члена класу **->** або операції розіменування *****.

Для того щоб було можливим виділяти пам'ять відразу для декількох екземплярів класу, клас має обов'язково мати конструктор за замовчуванням (конструктор без параметрів) або не мати жодного конструктора. А якщо ні, то компілятор видає повідомлення про помилку. За необхідності розміщення в динамічній пам'яті масиву об'єктів необхідно подбати про ініціалізацію кожного з екземплярів класу окремо, наприклад за допомогою окремої функції.

Звільнення пам'яті, виділеної для екземплярів класу, виконується за допомогою операції **delete**, наприклад:

delete point1; delete point3;

При виконанні цієї операції автоматично викликається деструктор класу, якщо він визначений. Звільняти пам'ять, виділену для екземпляра класу, у самому деструкторі не варто, оскільки це може призвести до помилки.

3.6. Статичні компоненти класу

Статичні компоненти класу на відміну від «звичайних», розглянутих дотепер, є належністю типу клас, а не конкретного екземпляра класу. Інакше кажучи, ці компоненти для кожного типу класу, в якому вони оголошені, існують тільки в одному екземплярі на відміну від інших компонентних даних, копії яких має кожен екземпляр класу.

Доступ до статичних компонентів можливий з використанням такого синтаксису:

ім'я_класу:: ім'я_стат_компонента
ім'я_об'єкта. ім'я_класу::ім'я_стат_компонента
ім'я_об'єкта. ім'я_стат_компонента

Статичні компонентні функції можна використовувати як для доступу до статичних компонентних даних, так і для інших цілей. Розглянемо простий приклад.

```
class Base
{
     static int Common;// визначаємо статичний компонент класу
    public: // дозволяємо доступ до функцій класу
     static int get_Common (){ return Common;}
     static void set_Common(int NewValue) {Common=NewValue;}
     void ShowCommon(){cout<<"Common="<<Common<<endl;}
};
int Base::Common;
```

```
void main()
{
    Base::set_Common(222);
    cout<<Base::get_Common(); // виводить 222
    Base::ShowCommon();  // виклик нестатичної функції
    Common=234; ::Common=234; /* ці оператори викликають помилку
                                 компілятора*/
}
```

Зверніть увагу на такі моменти.

По-перше, статичний компонент має бути описаний як у тілі класу, так і поза ним, причому в глобальній області видимості. Якщо перенести опис **int Base::Common** у функцію **main(),** одержимо повідомлення компілятора про помилку. Причина цього самоочевидна: оскільки статичний компонент належить класу, то він має бути видимий скрізь, де видимий опис цього класу. Опис статичного компонента – поза тілом класу – може включати ініціалізатор:

```
int Base::Common=123;
```

По-друге, доступ до статичного компонента можна одержати за допомогою статичних функцій і без визначення екземпляра класу, що й зроблено у програмі. При цьому у кваліфікованому імені компонентної функції використовується не ім'я об'єкта, а ім'я самого класу: **Base::ShowCommon()**.

З іншого боку, можливий, але не бажаний, і прямій доступ до статичного компонента, якщо він є доступним для зовнішніх функцій, тобто оголошений з використанням специфікатора доступу **public**, наприклад:

```
class Base
{
  public:
  static int Common;
  // …
};
int Base::Common;
void main()
{
   // …
  cout<<Base::Common;
}
```

Доступ до статичних компонентів можна одержати із нестатичних компонентних функцій, у тому числі й з конструктора, використовуючи як кваліфікатор ім'я об'єкта:

```
class Base
{
static int Common;
```

```
  public:
void Show() { cout<<Common<<endl; }
};
int Base::Common=111; /*Ініціалізувати можна тільки в глобальній
                        області або в  компонентній функції – статичній
                        або нестатичній */
void main()
{
  Base Ins;
  Ins.Show();
  Base::Show(); /* викликає помилку компілятора: не можна викликати
                   нестатичну  компонентну функцію з використанням
                   імені класу*/
}
```

З іншого боку, неможливо також одержати доступ до нестатичного компонента зі статичної функції:

```
class Base
{
   int Local;
  public:
   static void BadIdea() { cout<<Local; } // викликає помилку компілятора
};
```

Це цілком логічно, оскільки статична функція може бути викликана й у контексті, де екземпляр класу не існує і, отже, може не існувати й нестатичного (екземплярного) компонента.

У чому ж полягає практичний зміст статичних компонентів і статичних функцій, яку можна мати з них користь?

По-перше, статичні компоненти корисні тоді, коли всі екземпляри класу повинні мати деякі загальні дані, доступні для всіх об'єктів. Наприклад, якщо є клас вікно, то розумно, приміром, роздільну здатність екрана монітора у пікселах, яку встановлено на даний момент, зберігати у статичних компонентах, оскільки роздільна здатність екрана може бути тільки одною для всіх об'єктів-вікон. У цьому випадку можна змінювати роздільну здатність екрана без закриття всіх відкритих вікон, якщо організувати посилання кожному вікну повідомлення про зміну роздільної здатності. Для цього можна список відкритих вікон і посилання на них зберігати й підтримувати в статичних компонентних даних.

По-друге, застосування статичних компонентів є деякою альтернативою глобальним змінним, широке використання яких у програмах великого розміру загрожує неприємностями аналогічного розміру.

3.7. Приклад класу TVector

Як добре відомо, властивості масивів у мові C++ такі ж, як і у мові C. Нагадаємо, що при визначенні масиву треба визначити кількість його елементів обов'язково як вираз константного типу, наприклад:

```
const int max=100;
int N=100;
double Vec1[max*2], // припустимо
Vec2[100], // теж припустимо
Vec3[N], // неприпустимо
Vec4[]={7,1,2020}; /* теж припустимо, компілятор визначить масив як
                      Vec4[3] і присвоїть його елементам вказані початкові
                      значення*/
```

Головною перевагою такого способу визначення масивів є велика ефективність доступу до елементів масивів. Разом з тим масиви у мові C++ мають і декілька суттєвих недоліків:

- індекси масивів можуть бути тільки цілочисловими і завжди починаються з нуля;
- вихід індексу за межі масиву в один чи інший бік не контролюється і призводить до катастрофічних помилок під час виконання програми;
- кількість елементів масиву під час виконання програми змінити неможливо.

Ці недоліки можна усунути, якщо розробити такий клас-масив, що не тільки не буде мати перелічених недоліків, а ще й надасть користувачеві інши корисні властивості.

Наведений нижче клас **TVector** має такі властивості:

- тип елементів масиву задається зовні класу за допомогою ідентифікатора **ElemType**, що потім дозволить легко створити шаблонний клас;
- користувач класу може додавати нові елементи у кінець масиву, вставляти елементи у будь-яке місце масиву, видаляти будь-які елементи масиву, а також вставляти в існуючий масив інший;
- кількість елементів масиву довільна: масив ніби «розширюється» при додаванні нових елементів і, навпаки, «звужується» при вилученні існуючих елементів. Таке «чудо» можливе за рахунок того, що клас зберігає елементи масиву у динамічній пам'яті і перерозподіляє цю пам'ять при зміненні розміру масиву.

Зауваження. У стандартній бібліотеці C++ є шаблонний класс **vector**, що є потужнішим, ніж клас **TVector**, але на цьому прикладі можна побачити ті можливості, які надає сама по собі концепція ООП і як саме ці можливості можна використати.

Відомо, що будь-який клас має свої недоліки. Одним з недоліків класу **TVector** є той факт, що для доступу до елементів масиву неможливо використовувати квадратні дужки, але цей недолік буде виправлений у наступному підрозділі.

Наведемо приклад, реалізований у проекті **Vector**, що є консольним додатком. Зміст заголовного файла TVector.h (директиви препроцесора пропущені) такий:

```
typedef double ElemType;
class TVector
{
 public:
    TVector();
    TVector(int iSize, int iGain=0, ElemType InitValue=0);
    ~TVector();
    bool SetCapacity(int NewCapacity);
    bool SetElem(const ElemType &Value, int Index);
    bool GetElem(ElemType &Value, int Index) const;
    bool InsertElem(const ElemType &Value, int Index);
    bool DeleteElem(int Index);
    void AddElem(const ElemType &Value);
    int Size() const {return VectorSize;}
    int Capacity() const {return VectorCapacity;}
    bool InsertVec(TVector &SubVector, int Index);
 protected:
    ElemType * pVec; /*тут зберігаються елементи масиву*/
    int VectorSize, /*поточна кількість  існуючих елементів масиву */
VectorCapacity,  /*кількість елементів масиву, для яких зарезервована
                   пам'ять.  VectorCapacity завжди дорівнює VectorSize
                   або більше*/
Gain; /*це та кількість елементів, на яку збільшується VectorCapacity
        за необхідності перерозподілу пам'яті при додаванні елементів до
        масиву*/
};
```

Зміст файла реалізації TVector.cpp (директиви препроцесора пропущені):

```
TVector::TVector()
{
 pVec = 0;
 VectorSize = VectorCapacity = Gain = 0;
}

TVector::TVector(
```

```
  int iSize, /*початкова кількість елементів масиву*/
  int iGain, /*дивись призначення Gain*/
  ElemType InitValue /*початкове значення елементів  масиву*/
)
{
  Gain = iGain;
  if (iSize <= 0)
  {
    pVec = 0;
    VectorCapacity = VectorSize = 0;
    return;
  }
  pVec = new ElemType[iSize];
  VectorCapacity = VectorSize = iSize;
  for (int i = 0; i<VectorSize; pVec[i++] = InitValue);
}
TVector::~TVector()
{
  if (pVec) delete[] pVec;
}
/* SetElem – задає нове значення елемента масиву з індексом Index */
bool TVector::SetElem(const ElemType &Value, int Index)
{
  if ((Index<0) || (Index>VectorSize - 1)) return false;
  pVec[Index] = Value;
  return true;
}
/* SetCapacity – змінює розмір пам'яті, зарезервованої для масиву*/
bool TVector::SetCapacity(int NewCapacity)
{
  if (NewCapacity<0) return false;
  int OldCapacity = VectorCapacity;
  VectorCapacity = NewCapacity;
  ElemType *temp = new ElemType[VectorCapacity];
  for (int i = 0; i<OldCapacity; i++) temp[i] = pVec[i];
  delete[] pVec; pVec = temp;
  return true;
}
/* AddElem – додає новий елемент зі значенням Value у кінець масиву*/
void TVector::AddElem(const ElemType &Value)
{
  if (++VectorSize>VectorCapacity)
  {
    int NewCapacity;
    if (Gain>0) NewCapacity = VectorCapacity + Gain;
    else NewCapacity = VectorCapacity + 1;
    SetCapacity(NewCapacity);
```

```
  }
  pVec[VectorSize - 1] = Value;
}
/*GetElem – повертає значення елемента массиву Value з індексом Index */
bool TVector::GetElem(ElemType &Value, int Index) const
{
  if ((Index<0) || (Index>VectorSize - 1)) return false;
  Value = pVec[Index];
  return true;
}
/*InsertElem – вставляє значення Value у позицію масиву з індексом Index;
    розмір масиву при цьому збільшується на 1*/
bool TVector::InsertElem(const ElemType &Value, int Index)
{
  if ((Index>VectorSize - 1) || (Index<0)) return false;
  if (VectorSize >= VectorCapacity)SetCapacity(VectorCapacity + 1);
  for (int i = VectorSize - 1; i >= Index; i--) pVec[i + 1] = pVec[i];
  pVec[Index] = Value;
  VectorSize++;
  return true;
}
/* DeleteElem – видаляє елемент масиву з індексом Index; розмір масиву при
   цьому зменшується на 1*/
bool TVector::DeleteElem(int Index)
{
  if ((Index>VectorSize - 1) || (Index<0)) return false;
  for (int i = Index; i<VectorSize - 1; i++) pVec[i] = pVec[i + 1];
  VectorSize--;
  return true;
}
/* InsertVec – вставляє новий масив SubVector у позицію масиву з індексом
    Index; розмір масиву при цьому збільшується на кількість елементів
    масиву SubVector */
bool TVector::InsertVec(TVector &SubVector, int Index)
{
  if ((Index<0) || (Index + SubVector.Size()>VectorSize - 1))
    return false;
  if (VectorSize + SubVector.Size()>VectorCapacity)
    SetCapacity(VectorSize + SubVector.Size());
  VectorSize += SubVector.Size();
  for (int i = VectorSize - 1; i >= Index; i--) pVec[i] = pVec[i - SubVector.Size()];
  /*!!!*/    for (int i = Index; i<Index + SubVector.Size(); i++) pVec[i] =
SubVector.pVec[i - Index];
  return true;
}
```

Тестування класу.

```
void Print(TVector &Vec)
```

```
{
  cout << "=========================" << endl;
  cout << "Vec.Capacity==" << Vec.Capacity() << " Vec.Size==" << Vec.Size()
       << endl;
  for (int i = 0; i<Vec.Size(); i++)
  {
    cout.width(5); cout.precision(2); cout.fixed;
    ElemType temp;
    Vec.GetElem(temp, i);
    cout << temp;
  }
  cout << endl << "=========================" << endl;
}

void main()
{
  TVector MyVec(3, 5, -1.1);
  int N = 5, Index, i;
  ElemType Val = 0.1;
  for (i = 0; i<N; i++)
  {
    MyVec.AddElem(Val);
    Val += 1;
  }
  Print(MyVec);         cout << endl;

  TVector sVec(3, 5, 4.5);
  do
  {
    cout << "1.Set" << endl
      << "2.Get" << endl
      << "3.Insert" << endl
      << "4.Delete" << endl
      << "5.Add" << endl
      << "6.Insert vector" << endl
      << "7.Exit" << endl << '>';
    float a = 0.5;
    switch (_getch() - '0')
    {
    case 1: cout << endl << "Val, Index>"; cin >> Val >> Index;
      cout << (MyVec.SetElem(Val, Index) ? "true" : "false") << endl;
      break;
    case 2: cout << endl << "Index>"; cin >> Index;
      cout << (MyVec.GetElem(Val, Index) ? "true" : "false") << endl;
      cout << Val << endl;
      break;
    case 3: cout << endl << "Val, Index>"; cin >> Val >> Index;
```

```
      cout << (MyVec.InsertElem(Val, Index) ? "true" : "false") << endl;
      break;
    case 4: cout << endl << "Index>"; cin >> Index;
      cout << (MyVec.DeleteElem(Index) ? "true" : "false") << endl;
      break;
    case 5: cout << endl << "Val>"; cin >> Val;
      MyVec.AddElem(Val);
      break;
    case 6:
      MyVec.InsertVec(sVec, 1);
      break;
    case 7: return;
    }
    Print(MyVec);
  } while (1);
  _getch();
}
```

3.8. Клас TVector з перевантаженими операціями

Забігаючи наперед, наведемо приклад використання такого засобу мови C++, як *перевантаження операцій*. Хотілося б при роботі з масивом, тобто класом TVector, мати можливість використовувати операцію [] для доступу до елемента масиву за індексом для того, щоб одержати значення елемента масиву або модифікувати його. Доцільно, щоб при цьому клас перевіряв коректність індексу для того, щоб не було спроб виходу за межі масиву. Ну й ще б хотілося мати можливість виведення значень усіх елементів масиву за допомогою стандартного потоку **cout**. Усе це і багато чого іншого дійсно можливо виконати за допомогою перевантаження операцій. Для того щоб усі ці мрії втілити в життя, треба в оголошення класу (файл TVector.h) додати такі описи:

```
class TVector
{
public:
    // …
    ElType & TVector::operator [](int Index)
{
       if((Index>=0)&&(Index<VectorSize)) return pvec[Index];
}
friend ostream & operator <<(ostream &out,const TVector &Vec)
    {
          if(!Vec.pVec) return out;
          for(int i=0;i<Vec.VectorSize; i++)
          {
                out.width(5); out.precision(2); out.fixed;
```

```
            out<<Vec.pVec[i];
        }
        return out;
    }
// …
};
```

Тепер у програмі, що використовує даний клас, можна використовувати вирази такого виду:

```
TVector MyVec(3,5,-1);
cout<<MyVec; // виведення усіх елементів масиву
MyVec[k]=12.3; // присвоєння значення елементу масиву з індексом k
ElType Val= MyVec[k]; /* одержання значення елемента масиву
                         з індексом k */
```

Якщо Ви уважно подивитеся на функцію перевантаження операції **[]**, то у Вас, напевно, виникне питання: а що функція поверне, якщо індекс буде поза межами масиву? А невідомо що, тобто випадкове значення, що не є добре. Ну і як же бути в цьому випадку? Прийнятним виходом є генерація виняткової ситуації (виключення), яку може перехопити функція, що використовує вирази типу **MyVec[k]**. Текст функції перевантаження операції **[]** можна у цьому випадку записати так:

```
ElType & TVector::operator [](int Index)
{
    if((Index<0)||(Index>=VectorSize)) throw "Bad index";
    return pvec[Index];
}
```

У цьому випадку функція, що використовує вирази типу **MyVec[k],** може використати такий код:

```
    try
    {
// …
        MyVec[i]=0;
// …
    }
    catch(char * Mes) {
        AfxMessageBox((CString)Mes);
    }
```

Якщо при обчисленні виразу **MyVec[i]** індекс буде поза діапазоном, виконається оператор, зазначений у блоці **catch**, тобто в цьому випадку на дисплеї буде показано вікно з текстом "Bad index". Природно, що в реальній програмі для оброблення такої помилки треба реалізувати щось доцільніше, ніж виведення повідомлення про помилку користувачеві програми.

4. УСПАДКУВАННЯ КЛАСІВ І ПОЛІМОРФІЗМ

4.1. Успадкування класів

4.1.1. Визначення похідного класу

Як було зазначено вище при розгляді трьох основних принципів ООП, успадкування є можливістю створення нових класів не шляхом модифікації наявних класів, а шляхом створення таких нових класів, які успадковують поля й методи існуючих. Існуючий клас називають батьківським класом, а побудовані на його основі класи – похідними або породженими класами, а також нащадками або спадкоємцями. Самий перший клас ієрархії називають ще базовим класом.

Кожний з похідних класів може, у свою чергу, бути батьківським класом, у результаті чого може бути отримана ієрархія класів, яку можна відобразити у вигляді деревоподібної структури. На рис. 4.1 базовим класом ієрархії є CParent, а всі інші – похідні.

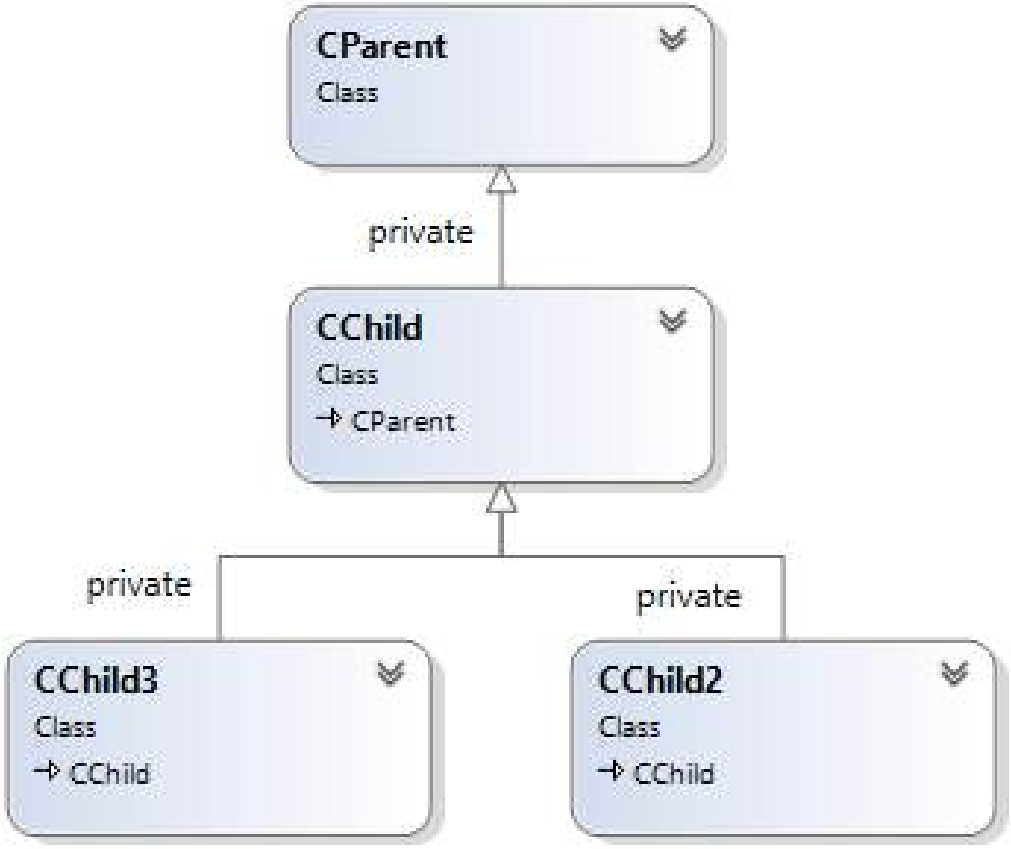

Рис. 4.1. Приклад діаграми, що відображає ієрархію класів

Похідний клас одержує доступ до всіх полів і методів батьківского класу відповідно до статусу доступу, але при цьому поля й методи батьківского класу не копіюються, а залишаються його належністю.

Похідний клас може перекривати компонентні дані й компонентні функції, тобто може включати компоненти з тими ж іменами, що й компоненти батьківського класу. У цьому випадку він включає як свої власні компоненти, так і компоненти батьківського класу (класів). Для звертання до таких компонентів доведеться використовувати кваліфіковані (уточнені) імена. Простий приклад:

```
struct Ta
{
int Var;
Ta() {Var=1;}
void Print() {cout<<"Ta.Print Var="<<Var<<endl;}
}ia;
struct Tb: Ta
{
int Var;
Tb() {Var=2;}
void Print()
{
        cout<<"Tb.Print Var="<<Var<<endl;
        cout<<"Tb.Print Ta::Var="<<Ta::Var<<endl;
}
}ib;
// ...
ia.Print();        // виводить Ta.Print Var=1
ib.Print();        // виводить Tb.Print Var=2 і Tb.Print Ta::Var=1
ib.Ta::Print(); // виводить Ta.Print Var=1
ia.Tb::Print(); /* викликає помилку компілятора:  'Tb' is not a public base
                   class of 'Ta' */
```

Оскільки в обох класах оголошена функція **Print()**, то для виклику функції базового класу з використанням об'єкта похідного класу використовують кваліфіковане ім'я **ib.Ta::Print()**. Зверніть увагу на той факт, що опис у похідному класі компонента **Var** з тим же іменем, що й у батьківському класі, припустимий і призводить до того, що в похідному класі будуть існувати обидва компоненти, що ілюструє функція **Print** класу **Tb**. У цьому випадку для доступу до компонента батьківського класу треба використовувати уточнене ім'я **Ta::Var**.

У загальному випадку визначення похідного класу має такий формат:

```
ключ_класу   ім'я_похідного_класу : статус_доступу
ім'я_базового_класу {тіло_похідного_класу};
```

Від опису базового класу воно відрізняється наявністю двокрапки, після якої вказується необов’язковий статус доступу й ім'я базового класу. Якщо базовий клас не один (множинне успадкування), то імена базових класів розділяються комами, наприклад:

```
class a{/* тіло_класу */};
```

class b{/* *тіло_класу* ****/};**
class c: private a, public b {/* *тіло_класу* ****/};**

Статус доступу задається, як і при указанні доступу до полів і методів класу, за допомогою тих же специфікаторів **public, private** і **protected**. В ієрархії класів діють такі угоди щодо доступності їх компонентів.

Особисті (**private**) компоненти доступні тільки усередині того класу, де вони визначені. Захищені (**protected**) компоненти доступні усередині класу, в якому вони визначені, і в похідних класах, а для зовнішньої програми такі компоненти недоступні.

На доступність компонентів впливає також ключ класу – **class** або **struct**. Зазначимо, що ні батьківський, ні похідний клас не можуть бути об'єднаннями **union**. Якщо перед іменем батьківського класу статус доступу не зазначений, то похідний клас одержує доступ до компонентів базового класу, що мають статуси доступу **public** або **protected**. У похідному класі ці поля й методи одержують статус доступу **private**, якщо (похідний) клас визначений за допомогою ключа **class**, і статус доступу **public**, якщо клас визначений із ключем **struct**.

Змінити статус доступу при спадкуванні можна шляхом указання специфікаторів доступу **public, private** і **protected**, які можна вказувати перед іменем кожного похідного класу.

Важливо зазначити, що змінення статусу доступу в похідному класі можливе тільки у бік його обмеження: якщо який-небудь компонент базового класу має, наприклад, статус доступу **protected**, то він ніяк не може одержати статус **public** у похідному класі. Інакше кажучи, компонент базового класу зі статусом доступу **private** недоступний у похідному класі незалежно від того, який специфікатор доступу зазначений перед іменем батьківського класу в описі похідного класу. З іншого боку, компонент батьківського класу зі статусом доступу **public** стане недоступним для програми, що використовує похідний клас, якщо при описі похідного класу для базового класу зазначено статус доступу **private** або **protected**.

Якщо перед іменем батьківського класу статус доступу не зазначений, то передбачається **private**, якщо визначення похідного класу виконане із ключем **class**, і **public –** якщо із ключем **struct**.

Навіщо взагалі потрібні статуси доступу при спадкуванні та ще з такими непростими правилами взаємодії специфікаторів доступу батьківських класів і похідного? Грамотне використання цих специфікаторів дозволяє зменшити ступінь залежності похідних класів від деталей реалізації батьківських класів. Якщо розробнику похідного класу надати повний доступ до всіх компонентів батьківського класу, то будемо мати такі погані наслідки.

По-перше, цілком можлива некоректна модифікація полів батьківського класу в похідному, оскільки розробник похідного класу може не знати усіх деталей реалізації батьківського класу. А відповідно до одного

із законів Мерфі «та неприємність, яка може трапитися, обов'язково трапиться, причому в самий невідповідний час і з максимально можливою шкодою».

По-друге, похідний клас одержує більшу залежність від батьківських класів, і модифікація батьківського класу з великою ймовірністю призведе до необхідності модифікації й похідного класу.

4.2. Поліморфні об'єкти й віртуальні функції

Віртуальні функції – це такі компонентні функції, заголовок яких починається із ключового слова **virtual**. Ці функції використовуються в ієрархії класів і дозволяють елегантно вирішити проблему заміщення функцій батьківських класів у похідних. Інакше кажучи, якщо в кожному із класів ієрархії є функція, що виконує ту саму дію, але специфічним для кожного класу способом, то цю функцію треба оголосити віртуальною. Об'єкти таких класів називають поліморфними (від грец. poly багато + morphē форма). У цьому випадку компілятор забезпечує коректний виклик функції для об'єкта будь-якого класу ієрархії.

Поліморфізм можна також визначити як можливість працювати з об'єктами похідних класів, використовуючи інтерфейс їх батьківського класу. Але перш ніж розглядати механізм віртуальних функцій, розглянемо сумісність об'єктів ієрархії класів.

4.2.1. Сумісність об'єктів

Об'єкти однієї ієрархії класів мають сумісність знизу вверх за присвоєнням (**Polymorph.sln**):

```
struct TA
{
        int a;
        TA(int ia)
        {a=ia;}
        void Calloutput() {Output();}
        void Output() {cout<<"TA::a="<<a<<endl;}
};
struct TB:TA
{       int b;
        TB(int ib):TA(ib)
        {b=ib;/*TA::TA(ib);*/}
        void Output() {cout<<"TA::a="<<a<<" TB::b="<<b<<endl;}
};
struct TC:TB
{
```

```
        int c;
        TC(int ic):TB(ic)
        {c=ic;}
        void Output()
        {
          cout<<"TA::a="<<a<<" TB::b="<<b<<" TC::c="<<c<<endl;
        }
    };
void main()
{
    TA aobj(1); // TA::TA
    TB bobj(2); // TA::TA  TB::TB
    TC cobj(3); // TA::TA  TB::TB TC::TC
    aobj.Output(); // TA::a=1
    bobj.Output(); // TA::a=2 TB::b=2
    cobj.Output(); // TA::a=3 TB::b=3 TC::c=3
    aobj=bobj;
    aobj.Output(); // TA::a=2
    aobj=cobj;
    aobj.Output(); // TA::a=3
    bobj=cobj;
    bobj.Output(); // TA::a=3 TB::b=3
    //  bobj=aobj; cobj=bobj; cobj=aobj; неприпустимі операції
}
```

Припустимі присвоєння наведено у програмі. Виконання операції присвоєння одного об'єкта іншому означає присвоєння полям класу, зазначеного ліворуч, значень відповідних полів класу, зазначеного праворуч від операції присвоєння. Зворотні присвоєння неприпустимі: викликають помилку компілятора.

Сумісність за присвоєнням поширюється й на сумісність фактичних і формальних параметрів функцій, що мають тип класу. Розглянемо приклад програми, що ілюструє сумісність параметрів (**Polymorph.sln**).

```
struct TA
{
    void Output() {cout<<"TA::Object"<<endl;}
};
struct TB:TA
{
    void Output() {cout<<"TB::Object"<<endl;}
};
struct TC:TB
{
    void Output() {cout<<"TC::Object"<<endl;}
};
void GFunc1(TA  Obj) { Obj.Output(); }
```

```
void GFunc2(TA *Obj) { Obj->Output();}
void main()
{
    TA aobj; TC cobj;
    GFunc1(aobj); // виводить TA::Object
    GFunc1(cobj); // виводить TA::Object
    TB *bobj=new TB;
    GFunc2(bobj); // виводить TA::Object
    delete bobj;
}
```

Як видно з наведених у коментарях результатів виконання програми, отриманий об'єкт глобальні функції **GFunc1()** і **GFunc2()** завжди трактують як такий, що має тип **TA**. Чому? Тому що, коли компілюються функції **GFunc1()** і **GFunc2(),** компілятор має підставити адресу компонентної функції **Output()**, і він підставляє адресу саме функції **TA::Output()** відповідно до типу формального параметра. Інакше і бути не може, оскільки тип фактичного параметра, переданого у функції **GFunc1()** і **GFunc2()**, з'ясовується тільки на етапі виконання програми. Таку підстановку адреси функцій, що викликаються на етапі компіляції, прийнято називати *раннім зв'язуванням* (early binding – зв'язування на етапі компіляції, статичне зв'язування). У нашому випадку потрібно визначити адресу методу, що викликається, на етапі виконання програми, що називається *пізнім зв'язуванням* (late binding – динамічне, відкладене зв'язування). Для реалізації пізнього зв'язування функцію **Output()** треба оголосити *віртуальною*.

4.2.2. Віртуальні функції

Перепишемо фрагмент тексту програми з попереднього підрозділу, визначивши компонентні функції **Output()** як віртуальні (**Polymorph.sln**).

```
struct TA
{
    void Print(){Output();}
    virtual void Output() {cout<<"TA::Object"<<endl;}
};
struct TB:TA
{
    virtual void Output() {cout<<"TB::Object"<<endl;}
};
struct TC:TB
{
    virtual void Output() {cout<<"TC::Object"<<endl;}
};
void GFunc1(TA  Obj)   { Obj.Print(); }
void GFunc2(TA *Obj)  { Obj->Print();}
void GFunc3(TA &Obj) { Obj.Print();}
```

```
void main()
{
    TA aobj; TC cobj;
    GFunc1(aobj); // виводить TA::Object
    GFunc1(cobj); // виводить TA::Object
    TB *bobj=new TB;
    GFunc2(bobj); // виводить TB::Object
    GFunc3(cobj); // виводить TC::Object
    GFunc3(*bobj); // виводить TB::Object
    TA *pobj;
    pobj=new TC; /* Увага, для створення об'єкта типу TA
                    використовується конструктор класу TC! */
    pobj->Print(); // виводить TC::Object
    *pobj=*bobj;
    pobj->Print(); // виводить TC::Object
    cout<<sizeof(*pobj)<<endl; /* виводить 4. Якщо додати до класу TA
                                  цілочисловий компонент, то sizeof поверне 8 */
    delete bobj;
    delete pobj;
}
```

Зверніть увагу на те, що механізм віртуальних функцій не працює належним чином у тому випадку, коли об'єкт передається у функцію **GFunc1()** за значенням. Причина полягає в тому, що в цьому випадку компілятор виконує неявне зведення типу фактичного параметра **cobj** до типу формального параметра **TA Obj**. У випадку опису формального параметра як покажчика в **GFunc2()** або посилання в **GFunc3()** віртуальні функції забезпечують очікувану поведінку об'єкта.

Синтаксичні правила опису віртуальних функцій нескладні. Для того щоб функція стала віртуальною, вона має:

- бути оголошеною із ключовим словом **virtual**, причому досить це зробити тільки в базовому класі, точніше, у класі, де вперше з'являється віртуальна функція. Інакше кажучи, можна вилучити ключове слово **virtual** при визначенні функції в класах **TB** і **TC**;
- мати те саме ім'я й однакову сигнатуру у класах ієрархії, причому не обов'язково в усіх.

Крім того, для коректного функціонування віртуальних функцій потрібно дотриматися певних правил щодо конструкторів класів. Така вимога пов'язана з тією обставиною, що конструктор базового класу забезпечує неявним чином побудову *таблиці віртуальних методів* (virtual method table або vtable). Ця таблиця містить адреси віртуальних функцій і використовується на етапі виконання програми для виклику потрібної

віртуальної функції відповідно до типу об'єкта. Правила, що стосуються конструкторів, будуть ще розглянуті й проілюстровані нижче.

Яку користь можна отримати із прийнятих правил сумісності об'єктів і властивостей віртуальних функцій? Користь ілюструється функціями **GFunc2()** і **GFunc3()**: завдяки тому, що формальний параметр функцій має тип базового класу, як фактичний параметр в цій функції може бути переданий об'єкт будь-якого похідного класу цієї ієрархії. Тип переданого об'єкта буде в цих функціях трактуватися коректно й об'єкт буде функціонувати відповідно до свого типу, а не типу формального параметра. Такий об'єкт, тобто той, що має віртуальні функції, і прийнято називати *поліморфним*.

4.2.3. Віртуальні функції й конструктори

Наведемо ще один приклад, що наочно ілюструє ті можливості, які досягаються завдяки віртуальним функціям. Прототипом до нього постав загально відомий приклад з посібника з мови Turbo Pascal 5.5 фірми Borland, який є досить наочним.

Припустимо, потрібно розробити класи для зображення фігур у графічному режимі. Який клас слід прийняти як базовий? При побудові ієрархії класів, яка є непростим завданням, доцільно спочатку виявити те загальне, що є в усіх сутностях, які потрібно розробити. Ієрархія класів має місце там, де є природна ієрархія сутностей реального світу, які повинні моделювати класи. Загальним для всіх графічних фігур є, принаймні, їх місце розташування – координати на екрані. Найпростішою графічною фігурою є точка, яку візьмемо як базовий клас. Природно припустити також, що істотною властивістю будь-якої графічної фігури є також її колір, але цим, як і іншими практично корисними властивостями, зневажатимо для того, щоб не затінювати основну ідею. Таким чином, у базовому класі визначаємо ті компоненти – дані й функції, які є загальними для всіх похідних класів графічних фігур.

Отже, визначимо базовий клас «точка» і похідний клас «коло» спочатку без віртуальних функцій.

```
/* Figures1.cpp – ілюстрація класів «точка» і «коло» без
                  віртуальних функцій */
class TPoint
{
  protected:
int x,y;
  public:
TPoint (int ix=0, int iy=0) { x=ix; y=iy;}
/* Show() – зображує точку */
void Show()
{
    // рисування точки
```

```
}
/* Hide() – ховає точку */
void Hide()
{
    // ховання точки, що включає виклик Show()
}
/* MoveTo() – переміщає точку в нове місце на екрані,
              задане координатами newx і  newy */
void MoveTo(int newx, int newy)
{
  Hide(); x=newx; y=newy; Show();
}
};
class TCircle: public TPoint
{
 protected:
int Radius;
 public:
TCircle(int ix=0, int iy=0, int irad=0) :TPoint(ix,iy) // !!!
{ Radius=irad; }
void Show()
{
// рисування кола
}
void Hide()
{
    // ховання кола, що включає виклик Show()
}
void MoveTo(int newx, int newy)
{
     Hide(); x=newx; y=newy; Show();
}
};// кінець опису класу TCircle

void main()
{
    TPoint p1(100,100);
    p1.Show(); _getch();
    TCircle *c1=new TCircle(100,100,50);
    c1->Show(); _getch();
    p1.MoveTo(400,200);  _getch();
    c1->MoveTo(400,200); _getch();
    delete c1;
}
```

Якщо програму запустити на виконання, то будуть показані точка і коло, які при натисненні будь-якої клавіші переміститься в нове положення.

Те, що об'єкт коло визначений як покажчик на клас, істотного значення в цьому контексті не має. Зверніть увагу на трохи незвичайну форму виклику конструктора базового класу в конструкторі класу «коло»:

TCircle(int ix=0, int iy=0, int irad=0) :TPoint(ix,iy)

Такий формат виклику конструктора базового класу є найкращим. Знов-таки, у цьому випадку це особливого значення не має, але для класу з віртуальними функціями буде мати. Конструктор базового класу міг би бути викликаний і в тілі конструктора похідного класу:

```
TCircle(int ix=0, int iy=0, int irad=0)
{ TPoint :: TPoint(ix,iy); Radius=irad; }
```

Зрештою, чому б не записати конструктор **TCircle** просто так:

```
TCircle(int ix=0, int iy=0, int irad=0)
{ x=ix; y=iy; Radius=irad; }
```

Програма й у цьому випадку буде виконуватися коректно, але так діяти не слід. Головна причина полягає в тому, що тепер конструктор класу **TCircle** повторює код конструктора класу **TPoint** і, отже, при зміненні конструктора **TPoint** потрібно не забути змінити й конструктор **TCircle** (і конструктори всіх інших похідних класів теж!).

З подібної причини функція **Hide()** має ховати точку шляхом виклику функції **Show()** з установленням як колір точки кольору фону, а не шляхом повторного виклику функції зображення точки. Завдяки цьому вся специфіка рисування точки буде локалізована у функції **Show()** і для іншого зображення точки потрібно змінити тільки код цієї функції. Аналогічний підхід використано й у функції **MoveTo()**, що буде корисним уподальшому при розробленні похідних класів.

Чи не є зайвим повторення коду функцій **MoveTo()** і **Hide()** у класі **TCircle**? Якщо так, то спробуйте вилучити ці функції з класу **TCircle**. Програма буде успішно компілюватися, оскільки при виконанні оператора **c1->MoveTo(400,200)** компілятор буде спочатку шукати функцію **MoveTo()** у класі **TCircle**, а якщо не знайде – то в батьківському класі. Програма також буде виконуватися, але замість кола переміщатися буде точка! Чому? Про причину Ви вже, напевно, здогадалися: коли компілюється функція **TPoint::MoveTo()**, то в ній викликаються, природно, функції **TPoint::Hide()** і **TPoint::Show()**. Підставлення адресів функцій, що викликаються на етапі компіляції, називається, як про це вже говорилося в підрозд. 4.2.1, раннім зв'язуванням. У нашій програмі необхідно, щоб функції **TPoint::MoveTo()** і **TPoint::Hide()** викликали функцію **TPoint::Show()** або **TCircle::Show()** залежно від того, який об'єкт їх викликав: типу **TPoint** або **TCircle**. А це в загальному випадку може бути зроблено тільки на етапі виконання програми, що й називається пізнім зв'язуванням. Для реалізації пізнього

зв'язування функції **TPoint::Show()** і **TCircle::Show()** треба зробити віртуальними, що й ілюструє програма **Figures2.cpp**.

```
/* Figures2.cpp – ілюстрація класів «точка» і «коло» з
                  віртуальними функціями */
class TPoint
{
  protected:
int x,y;
  public:
TPoint (int ix=0, int iy=0) { x=ix; y=iy;}
/* Show() – зображує точку */
virtual void Show()
{
    // рисування точки
}
/* Hide() – ховає точку */
void Hide()
{
    // ховання точки, що включає виклик Show()
}
/* MoveTo() – переміщує точку в нове місце на екрані,
               задане координатами newx і  newy */
void MoveTo(int newx, int newy)
{
   Hide(); x=newx; y=newy; Show();
}
};
class TCircle: public TPoint
{
  protected:
     int Radius;
  public:
     TCircle(int ix=0, int iy=0, int irad=0) :TPoint(ix,iy) // !!!
     { Radius=irad; }
     virtual void Show()
     {
       // рисування кола
     }
};
void main()
{
     TPoint p1(100,100);
     p1.Show(); _getch();
     TCircle *c1=new TCircle(100,100,50);
     c1->Show(); _getch();
     p1.MoveTo(400,200); _getch();
```

```
    c1->MoveTo(400,200); _getch();
    delete c1;
}
```

Будь-яким чином реалізуйте зображення точки та кола, випробуйте цю програму – і побачите, що переміщуються точка й коло. Додайте за аналогією з колом ще який-небудь клас, наприклад, трикутник і функції **Hide()** і **MoveTo()** будуть ховати й переміщати Вашу фігуру!

Зауваження. Тепер повернемося до понять поліморфізму й віртуальних функцій. Поліморфізм – це присвоєння дії імені, яка розділяється вгору і вниз класами ієрархії, причому кожний клас ієрархії реалізує цю дію специфічним для нього чином. Будь-який клас ієрархії графічних фігур повинен мати метод рисування фігури, який у всіх класах називається однаково (у нашому прикладі **Show()**), але кожний клас має реалізувати його по-своєму. Саме такий метод і повинен бути оголошений віртуальним.

4.2.4. Конструктори в ієрархії класів

Для коректної роботи механізму віртуальних функцій на етапі виконання має бути в наявності таблиця віртуальних методів, яку створює конструктор базового класу. Таким чином, якщо класи ієрархії мають конструктори (а зазвичай так воно і є), то при визначенні будь-якого об'єкта ієрархії класів має бути забезпечений виклик конструкторів усієї ієрархії класів, обов'язково включаючи базовий клас. Код, що створює таблицю віртуальних функцій, автоматично включається в конструктор базового класу, тобто першого класу ієрархії. Суттєво, що виклик конструктора батьківського класу повинен виконуватися саме так, як це зроблено в класі **TCircle**:

```
TCircle(int ix=0, int iy=0, int irad=0) :TPoint(ix,iy)
{/* власний код конструктора  TCircle */}
```

Як видно з прикладу, виклик конструктора батьківського класу, що включає фактичні параметри, якщо вони є, має розміщатися між заголовком конструктора і його тілом. Саме в цьому випадку гарантується виклик конструктора батьківського класу до початку виконання коду конструктора похідного класу.

Базовий клас, тобто найперший клас ієрархії класів, може також взагалі не мати ніякого конструктора або мати конструктор за замовчуванням, тобто конструктор без параметрів.

Якщо базовий клас ієрархії взагалі не має конструктора, то компілятор автоматично створює такий конструктор за замовчуванням (без

параметрів). Конструктор за замовчуванням може викликатися конструктором похідного класу або не викликатися зовсім: у будь-якому разі його виклик буде виконано автоматично.

Для похідних класів конструктор за замовчуванням також створюється компілятором автоматично й тому програміст може його не створювати, якщо він фактично не потрібний. Однак, як тільки в одному із класів ієрархії буде створено конструктор з параметрами, то й у всіх похідних класах конструктори також повинні бути створені й вони повинні викликати конструктори батьківських класів.

Як приклад наведемо програму **Constr.cpp**, яка ілюструє використання конструкторів за наявності віртуальних функцій.

```
/*Constr.cpp - ілюстрація використання конструкторів для класів
                 із віртуальними функціями */
struct Base
{
    int N;
    Base() {} // конструктор без параметрів і тіла
    void Print() { Output();}
    virtual void Output()
    { N=1234; cout<<" Base::N="<<N<<endl; }
};
struct Child :Base
{
    Child(int initn):Base()  { N=initn; }
    virtual void Output()
    { cout<<" Child::N="<<N<<endl; }
};
void main()
{
    Base B;
    B.Print(); // виводить: Base::N=1234
    Child C(4321);
    C.Print(); // виводить: Child::N=4321
}
```

Звернемо увагу на те, що функція **Print()** класу **Base** викликає функцію **Output()** похідного класу **Child**. Робота програми не зміниться в жодному з таких варіантів її запису:

- забрати конструктор із класу **Base**, залишивши його виклик у конструкторі класу **Child**;
- забрати виклик конструктора **Base** у конструкторі **Child**, не забираючи конструктор із класу **Base**;
- забрати виклик конструктора **Base** у конструкторі **Child** і забрати конструктор із класу **Base**;
- забрати взагалі конструктори з обох класів.

Таким чином, можна сформулювати два основні правила визначення й виклику конструкторів в ієрархії класів при використанні віртуальних функцій.

По-перше, для будь-якого класу ієрархії має бути забезпечений виклик конструкторів усього ланцюжка батьківських класів аж до базового.

По-друге, виклик конструктора батьківського класу має виконуватися за тілом конструктора похідного класу за схемою:

Похідний_клас(/* *формальні параметри* ***/) :**
Батьківський_клас(/* *фактичні параметри* ***/)**
{/* *тіло конструктора похідного класу* ***/ }**

Ще раз зазначимо, що якщо клас має конструктор з параметрами, то конструктор за замовчуванням компілятором не створюється.

4.2.5. Деструктори в ієрархії класів

Незалежно від наявності конструктора класи ієрархії можуть мати деструктори, причому деструктор викликається автоматично при знищенні об'єкта й може бути також викликаний явно. На відміну від конструкторів у деструкторі похідного класу не потрібно викликати деструктор базового класу. Тестова програма, яка дозволяє перевірити особливості виклику деструкторів:

```
/*Destr.cpp - ілюстрація використання деструкторів в ієрархії класів */
struct TParent
{
    ~TParent() {cout<<"TParent::~TParent()"<<endl;}
};
struct TSon : TParent // деструктор відсутній
{ };
struct TGrandSon : TSon
{
    ~TGrandSon(){cout<<"TGrandSon::~TGrandSon()"<<endl;}
};
void main()
{
    { TSon Son; } // виведення на екран: TParent::~TParent()
{ TGrandSon Grandson; } /* виведення на екран:  TGrandSon::~TGrandSon()
                                TParent::~TParent() */
}
```

Як випливає із наведеного тексту програми, виклик деструкторів для випадку ієрархії класів виконується автоматично, причому спочатку виконується код деструктора похідного класу, а вже потім код деструктора

батьківського класу. Слід зауважити, що відсутність явно оголошеного деструктора в класі **TSon** ніяк не впливає на виклик деструктора базового класу **TParent**.

4.3. Абстрактні класи

Абстрактним називається клас, який має абстрактні функції. («Ви не підкажете, де знаходиться цирк?» – Біля театру.– «А де знаходиться театр?» – Біля цирку…). Абстрактною називається така віртуальна функція, яка не має реалізації, не може її мати й не може бути викликана. Віртуальна функція оголошується абстрактною за допомогою спеціального синтаксису, що схоже на ініціалізацію нулем:

```
struct TAbsClass //абстрактний клас
{
    int n;
    virtual void Func1()=0; // абстрактна функція
    virtual char Func2(int, int)=0; // теж абстрактна функція
    virtual void Func3(int)  {} // не абстрактна функція
};
```

Спроба визначення екземпляра абстрактного класу

```
TAbsClass obj;
```

викликає помилку компілятора. Абстрактний клас може мати компонентні дані, не абстрактні віртуальні й не віртуальні функції, конструктор і деструктор, статичні компоненти. Абстрактним класом можна скористатися тільки шляхом оголошення похідного від нього класу й у ньому (обов'язково!) перекрити абстрактні віртуальні функції не абстрактними:

```
struct TRealClass:TAbsClass
{
  virtual void Func1(){}
  virtual char Func2(int a, int b) {}
}Obj; // припустимо
//…
   Obj.Func1(); // теж припустимо
```

Абстрактний клас використовується як база для побудови ієрархії класів. Сенс опису абстрактного класу полягає в тому, щоб програміст, який використовує цей клас, не зміг створити об'єкт цього класу, а повинен був би обов'язково створити похідний клас і реалізувати в ньому методи, оголошені абстрактними в базовому класі. Цей спосіб передачі інформації програмісту-користувачу класу значно надійніше указання на необхідність реалізації методів у документації, оскільки в цьому випадку роботу програміста буде контролювати компілятор, що нічого не забуває.

Проілюструємо застосування абстрактного класу на тому ж прикладі із графічними фігурами.

```
/* Figures3.cpp - варіант програми Figures2.cpp, що ілюструє застосування
                  абстрактного класу*/
class TLocation // абстрактний клас
{
 protected:
    int x,y;
 public:
    // TLocation(){}
    virtual void Show()=0;
    virtual void Hide()=0;
};
class TPoint : public TLocation
{
 protected:
    bool Visible;
 public:
    TPoint (int ix=0, int iy=0)//:TLocation()
    {x=ix; y=iy; Visible=false;}
    virtual void Show();
    virtual void Hide();
    void MoveTo(int newx, int newy)
    {Hide(); x=newx; y=newy; Show();}
};
class TCircle: public TPoint
{
 protected:
    int Radius;
 public:
    TCircle(int ix=0, int iy=0, int irad=0):TPoint(ix,iy)
    { Radius=irad;}
    void Show()
    { circle(x,y,Radius); Visible=1;}
};
void TPoint::Show()
{
    // рисуємо точку
    Visible=true;
}
void TPoint::Hide()
{
    // ховаємо точку, викликаючи при цьому Show()
    Visible=false;
}
void main()
{
    TPoint p1(100,100);
    TCircle *c1=new TCircle(100,100,50); p1.Show();
```

```
    _getch();
    c1->Show();
    _getch();
    p1.MoveTo(400,200);
    _getch();
    c1->MoveTo(400,200);
    _getch();
}
```

Зауважимо, що абстрактний клас не містить конструктора, але разом з тим механізм віртуальних функцій працює коректно: конструктор за замовчуванням створюється компілятором автоматично і автоматично ж викликається. Спробуємо забрати коментар з рядка

TPoint (int ix=0, int iy=0)//:*TLocation()*

таким чином викликаючи «неіснуючий» конструктор базового класу **TLocation**. Помилки не буде і програма буде працювати коректно. Можна, природно, додати конструктор у клас **TLocation** і не викликати його в класі **TPoint**: він однаково буде викликаний автоматично, якщо тільки буде конструктором без параметрів.

5. КОНТЕЙНЕРНІ КЛАСИ

Клас-контейнер містить як член-дані об'єкти іншого або інших класів. Клас-контейнер називають ще зовнішнім класом, а клас, який є його член-даним, – внутрішнім класом. Внутрішній клас можна визначити у контейнері різними способами:

- член-даним покажчиком;
- «звичайним» член-даним, тобто не покажчиком;
- масивом об'єктів.

Залежно від цього створення об'єкта(ів) внутрішнього класу в контейнері має виконуватися по-різному. Наприклад, для випадку опису внутрішнього класу як член-даного не покажчика, треба вибрати коректний спосіб виклику конструктора внутрішнього класу.

5.1. Контейнер із внутрішнім класом член-даним

Наведемо найпростіший приклад класу-контейнера:

```
struct CInner
{
    int k;
    CInner()
    {
        k=0;
        TRACE("Constructor CInner was called\n");
    }
    CInner(int ik)
    {
        k=ik;
        TRACE("Constructor CInner with parameter was called\n");
    }
    ~CInner()
    {
        TRACE("Destructor CInner was called\n");
    }
};
struct CContainer
{
    int m;
    CInner Inobj;
    CContainer()
    {
        TRACE("Constructor CContainer was called\n");
        m=0;
    }
    CContainer(int im){m=im;}
```

```
    ~CContainer()
    {
        TRACE("Destructor CContainer was called\n");
    }
}Cont;
```

При такому опису класу **CContainer** для створення внутрішнього об'єкта **Inobj** буде викликаний (автоматично) конструктор без параметрів, причому він буде викликаний ще до виклику конструктора класу **CContainer** – перевірте! З іншого боку, при знищенні об'єкта **Cont** спочатку буде викликаний деструктор класу **CContainer**, а потім – деструктор **CInner**.

А яким чином можна викликати конструктор з параметрами для об'єкта **Inobj**? При такій реалізації класу **CContainer:**

```
struct CContainer
{
    int m;
    CInner Inobj(5); // неприпустимо!
    // ...
};
```

виникне помилка компіляції.

А ось така реалізація класу цілком припустима:

```
struct CContainer
{
    int m;
    CInner Inobj;
    CContainer():Inobj(5) // дивись сюди
    {
        TRACE("Constructor CContainer was called\n");
        m=0;
    }
    CContainer(int im):Inobj(5) // дивись сюди
    {
        m=im;
        TRACE("Constructor CContainer with parameter was called\n");
    }
    ~CContainer()
    {
        TRACE("Destructor CContainer was called\n");
    }
}Cont1, Cont2(123);
```

Звернемо увагу, що для об'єкта **Cont2** буде викликаний конструктор з параметрами.

5.2. Контейнер з покажчиком на внутрішній клас

Наведемо такий приклад:

```
struct CInner   // як і в попередньому прикладі
{
    int k;
    CInner()
    {
        k=0;
        TRACE("Constructor CInner was called\n");
    }
    CInner(int ik)
    {
        k=ik;
        TRACE("Constructor CInner with parameter was called\n");
    }
    ~CInner()
    {
        TRACE("Destructor CInner was called\n");
    }
};
struct CContainer
{
    int m;
    CInner *pobj;
    CContainer()
    {
        pobj=new CInner;
        // або pobj=new CInner(3);
        m=0;
        TRACE("Constructor CContainer was called\n");
    }
    CContainer(int im)
    {
        pobj=new CInner;
        // або pobj=new CInner(3);
        m=im;
        TRACE("Constructor CContainer with parameter was called\n");
    }
    ~CContainer()
    {
        if(pobj)  delete  pobj;
        TRACE("Destructor CContainer was called\n");
    }
}Cont1, Cont2(123);
```

Як видно з реалізації контейнерного класу, створити об'єкт внутрішнього класу програміст може там, де він вважає за потрібне. У наведеному прикладі це зроблено у конструкторі контейнерного класу, а його знищення – у деструкторі. Якщо простежити за черговістю виклику конструкторів і деструкторів, то можна виявити, що конструктор внутрішнього класу викликається *після* конструктора контейнерного класу, як і деструктор.

Хоча наведений спосіб створення об'єкта внутрішнього класу і є доцільним, оскільки він забезпечує обов'язковий виклик конструктора й деструктора об'єкта внутрішнього класу, він не є єдино можливим. Ніхто й ніщо не заважає створювати внутрішній об'єкт у будь-яких член-функціях контейнерного класу, якщо це доцільно для розв'язуваної задачі, але тоді користувач такого контейнера повинен стежити за коректним порядком використання таких член-функцій.

У наведеному прикладі контейнерний клас сам створював і знищував внутрішній об'єкт, однак на практиці може бути і випадок, коли «внутрішній» об'єкт існує «ззовні» контейнерного класу. У цьому випадку до завдання контейнера не входять створення й знищення об'єкта внутрішнього класу. Як ілюстрацію цього випадку розглянемо такий приклад програми:

```
////////////// Контейнер з покажчиком на зовнішній об'єкт //////////
struct CInner
{
    int k;
    CInner()
    {
        k=0;
        TRACE("Constructor CInner was called\n");
    }
    CInner(int ik)
    {
        k=ik;
        TRACE("Constructor CInner with parameter was called\n");
    }
    int  getnumber() { return k; }
};
struct CContainer
{
    int m;
    CInner *pobj;
    CContainer(int im, CInner *iobj)
    {
        pobj=iobj;
        m=im;
        TRACE("Constructor CContainer with parameter was called\n");
    }
    void Print(){cout<<"CInner::k="<<pobj->getnumber()<<endl;}
```

```
};

void main()
{
    CInner *pinn=new CInner(123);
    //використання pinn
    CContainer Cont(12,pinn);
    Cont.Print(); // 123
    //деякий код
    CInner Inn(321);
    CContainer Cont2(12,&Inn);
    Cont2.Print(); // 321
    //деякий код
    delete pinn;
}
```

5.3. Менш абстрактний приклад контейнерного класу

Розглянемо ще один приклад, менш абстрактний. Клас **CDate** призначено для того, щоб зручно та безпечно оперувати з датами: день, місяць і рік. У конструкторі класу за замочуванням доцільно передбачити встановлення якої-небудь коректної дати, наприклад поточної. У конструкторі класу з параметрами можна встановити будь-яку дату, але треба в ньому передбачити перевірку коректності встановлюваної дати: наприклад, не буває 30 лютого або 31 квітня і т. д. Саме таким чином для обчислення нової дати шляхом, наприклад додавання певної кількості днів до поточної дати за допомогою функції **Adddays()** треба коректно обчислити нові день, місяць і рік. Алгоритми подібних обчислень вже давно існують – дивіться літературу.

```
// Файл Date.h
#pragma once
class CDate
{
    int m_nday;
    int m_nmonth;
    int m_nyear;
public:
    CDate(void);
    CDate(int iday, int imonth,int iyear);
    ~CDate(void);
    int Adddays(int Days);
    int Addmonths(int Months);
    int Addyears(int Years);
    bool setday(int newday);
    bool setmonth(int newmonth);
    bool setyear(int newyear);
```

```
    int getday();
    int getmonth();
    int getyear();
};

// Файл Date.cpp
#include "Stdafx.h"
#include "Date.h"
CDate::CDate(void): m_nday(0), m_nmonth(0), m_nyear(0)
{
    // деякий код
    TRACE("Constructor CDate was called\n");
}
CDate::CDate(int iday,  int imonth, int iyear)
{
    // деякий код
    TRACE("Constructor CDate was called\n");
}

CDate::~CDate(void)
{
    TRACE("Destructor ~CDate was called\n");
}
bool CDate::setday(int newday){ /* деякий код*/ }
bool CDate::setmonth(int newmonth){ /* деякий код*/}
bool CDate::setyear(int newyear){ /* деякий код*/}
int CDate::getday(){return m_nday;}
int CDate::getmonth(){return m_nmonth;}
int CDate::getyear(){return m_nyear;}
bool Adddays(int Days){ /* деякий код*/}
bool Addmonths(int Months){ /* деякий код*/}
bool Addyears(int Years){ /* деякий код*/}
```

Уявімо, що потрібно зберігати та оброблювати дані про деяких службовців, обчислювати їх зарплатню, строки надання відпусток тощо. Тоді створимо клас **CPerson**, що буде містити необхідні дані про службовців, у тому числі різноманітні дати службовця: дати народження, зарахування на роботу, відпусток і т. д.

========== **Person.h**==========

```
// Файл Person.h
#pragma once
#include "date.h"
class CPerson
{
    char *m_sfname;
    char *m_ssname;
    CDate m_dbirthdate;
```

```
    CDate * m_dschooldate;
public:
    CPerson(void);
    CPerson(char *Fname, char *Sname,int iday, int imonth, int iyear);
    ~CPerson(void);
    char * getfname(void);
    char * getsname(void);
    void setfname(char * newfname);
    void setsname(char * newsname);
    void setbirthdate(int Day, int Month, int Year);
    void setschooldate(int Day, int Month, int Year);
    int getbirthday();
    int getbirthmonth();
    int getbirthyear();
    // інші функції, у тому числі для доступу до членів m_dschooldate
};
```

========== **Person.cpp**==========

```
// Файл Person.cpp
#include "Person.h"
CPerson::CPerson(void)
: m_sfname(NULL)
, m_ssname(NULL)
, m_dbirthdate(0,0,0)
{
    m_dschooldate=new CDate();
    TRACE("Constructor CPerson was called\n");
}
CPerson::CPerson(char *Fname, char *Sname,int iday, int imonth, int iyear)
: m_dbirthdate(iday, imonth, iyear)
{
    // ініціалізуємо m_sfname і m_ssname значеннями Fname і Sname
    m_sfname=new char [strlen(Fname)+1];
    strcpy_s(m_sfname,strlen(Fname)+1,Fname);
    m_ssname=new char [strlen(Sname)+1];
    strcpy_s(m_ssname,strlen(Sname)+1,Sname);
    m_dschooldate=new CDate();
    TRACE("Constructor CPerson with parameters was called\n");
}

CPerson::~CPerson(void)
{
    if(m_sfname) delete [] m_sfname;
    if(m_ssname) delete [] m_ssname;
    if(m_dschooldate) delete m_dschooldate;
    TRACE("Destructor ~CPerson was called\n");
```

```
}

char * CPerson::getfname(void)
{
    return m_sfname;
}
char * CPerson::getsname(void)
{
    return m_ssname;
}
void CPerson::setfname(char * newfname)
{
    m_sfname=new char [strlen(newfname)+sizeof(char)];
    strcpy(m_sfname,newfname);
}
void CPerson::setsname(char * newsname)
{
    m_ssname=new char [strlen(newsname)+sizeof(char)];
    strcpy(m_ssname,newsname);
}
void CPerson::setbirthdate(int Day, int Month, int Year)
{
    m_dbirthdate.setday(Day);
    m_dbirthdate.setmonth(Month);
    m_dbirthdate.setyear(Year);
}
int CPerson::getbirthday()
{
    return m_dbirthdate.getday();
}
int CPerson::getbirthmonth()
{
    return m_dbirthdate.getmonth();
}
int CPerson::getbirthyear()
{
    return m_dbirthdate.getyear();
}
void CPerson::setschooldate(int Day, int Month, int Year)
{
    m_dschooldate->setday(Day);
    m_dschooldate->setmonth(Month);
    m_dschooldate->setyear(Year);
}
```

Найпростіший приклад використання об’єктів класу CPerson наведено нижче.

```
/* Файл Democontainerclass.cpp Ілюстрація використання контейнерного
  класу */
void main()
{
    // ...
    setlocale(LC_ALL,"rus");
    CPerson *ppers;
    ppers=new CPerson("Акакій","Непийпиво",26,02,2000);
    cout<<ppers->getsname()<<' '<<ppers->getbirthday()<<'.'
          <<ppers->getbirthmonth()<<'.'<<ppers->getbirthyear()<<endl;
    ppers->setschooldate(31,05,2015);
    delete ppers;
    ppers=new CPerson;
    delete ppers;
    cout<<"Натискай any key, хутко!";
    _getch();
}
```

6. ДРУЖНІ ФУНКЦІЇ Й КЛАСИ

6.1. Дружні функції

Припустимо, у програмі використовується кілька класів різних типів, тобто таких, що не належать до однієї ієрархії. Тоді спільне оброблення компонентних даних цих класів може бути виконано двома способами: шляхом безпосереднього звертання до компонентних даних або шляхом непрямого звертання до них через компонентні функції класів, призначених спеціально для цієї мети. Згідно з правилами "гарного тону" в ООП не рекомендується використовувати перший спосіб, а саме безпосереднє звертання до компонентних даних із зовнішної програми. Ігнорування цього принципу, по-перше, робить програму залежною від компонентних даних і, по-друге, ускладнює розроблення, налагодження й супровід програми внаслідок того, що простежити за змінами цих даних буде набагато складніше. Безпосередній доступ до компонентних даних значною мірою нейтралізує одну із безсумнівних переваг ООП: забезпечення цілісності даних завдяки їх приховуванню від програми, що використовує об’єкти класів.

Отже, залишається другий шлях – звертання до даних за допомогою компонентних функцій класів. Крім того, програмісту надається ще одна можливість, яка реалізується за допомогою так званих *дружніх* функцій і *дружніх* класів. Розглянемо спочатку дружні функції.

Дружньою називається така функція, яка має доступ до всіх компонентів класу, у тому числі власних (**private**), захищених (**protected**) і відкритих (**public**), і при цьому не є компонентною функцією цього класу. Насамперед зазначимо, що функція не може стати дружньою "без відома" самого класу: вона має бути оголошена в ньому зі специфікатором **friend** (друг).

У наведеній нижче програмі ілюструється застосування двох дружніх функцій до деякого класу **TMaster**.

```
/* Проект Friend - приклад опису дружніх функцій*/
class TMaster
{
   // public:
   int a,b,c;
   friend int FriendFunc1(TMaster &m)
   {return m.a+m.b+m.c;}
   friend int FriendFunc2(TMaster &m);
public:
   TMaster(int ia,int ib, int ic)
   {a=ia; b=ib; c=ic;}
   void Print()
   {
    cout<<"a="<<a<<" b="<<b<<" c="<<c<<endl;
   }
```

```
    // інші компонентні функції класу
};
int FriendFunc2(TMaster &m)
{
  m.Print(); // можна викликати функції класу
  return m.a*m.b*m.c;
}
/*
int Func3(TMaster &m)
{return (m.a+m.b+m.c)/3;}
*/
void main()
{
  TMaster M(1,2,5);
   cout<<"Сума      ="<<FriendFunc1(M)<<endl; // виводить 8
   cout<<"Добуток=" <<FriendFunc2(M)<<endl; // виводить 10
  _getch();
}
```

Як видно з прикладу, дружня функція може бути визначена як у тілі класу (функція **FriendFunc1()**), так і поза нього (функція **FriendFunc2()**), однак у другому випадку в класі обов'язково має з'явитися прототип цієї функції.

Угоди щодо дружніх функцій такі:

1. Дружня функція має обов'язково одержувати екземпляр класу як параметр.
2. Дружня функція має доступ до всіх компонентів класу, у тому числі приватних (**private**) і захищених (**protected**), за допомогою кваліфікованих імен.
3. Оскільки дружня функція не є компонентною функцією, вона не має безпосереднього доступу до компонентних даних, навіть якщо вони оголошені як загальнодоступні (**public**).

Зазначимо, що "недружня" класу функція **Func3** не має доступу до приватних компонентів класу: якщо вилучити коментар з визначення цієї функції, то одержимо повідомлення компілятора про помилку «компоненти **m.a, m.b, m.c** недоступні». Однак цієї помилки не буде, якщо компонентні дані класу зробити загальнодоступними: вилучити коментар зі специфікатора **public** у визначенні класу.

Місце розміщення прототипу або визначення функції в класі не має значення й права доступу функції до компонентних даних ніяк не залежать від специфікаторів доступу, оскільки дружня функція не є компонентною функцією класу.

Дружня функція не може бути одночасно й компонентною функцією того класу, стосовно якого вона є дружньою, але вона може бути компонентною функцією іншого класу.

Проілюструємо останнє твердження прикладом:

```
class Two; /* попередній опис класу Two, необхідний для опису
              функції Func у класі One */
class One
{
  public:
void Func(Two &t) {/* тіло функції*/}
};
class Two
{
friend void One::Func(Two &t);
};
```

Як видно з прикладу, спочатку необхідно описати той клас, стосовно якого функція є компонентною, а потім вже описати клас, для якого функція є дружньою. Ще раніше слід зробити так званий попередній (випереджальний) опис класу (у прикладі опис **class Two**), оскільки інакше неможливо вказати тип параметра функції **Func()** при її опису як компонентної функції класу **One**.

Функція може бути оголошена дружньою відразу до декількох класів, наприклад так:

```
class Two; /* попередній опис класу */
class One
{
    friend void Func(Two &b, One &a);
};
class Two
{
    friend void Func(Two &b, One &a);
};
void Func(Two &b, One &a)
{/* тіло функції*/}
```

Отже, дружні функції можна використовувати в тих випадках, коли потрібно виконати спільне оброблення даних декількох різних класів. Крім того, вони знаходять також застосування при перевантаженні операцій, що буде розглянуто нижче. Без них у мові C++ складно було б перевантажити яку-небудь операцію, операндами якої мають бути різнотипні, можливо, неспоріднені, класи.

6.2. Дружні класи

Будь-який клас **B** може бути оголошений як дружній іншому класу **A**. Завдяки "дружнім відносинам" компонентні функції класу **B** одержують доступ до всіх компонентних даних і функцій класу **A**, причому незалежно від їхнього статусу доступу. Для того щоб це було можливо, дружній клас **B** має одержати екземпляр класу **A** як параметр. Функції класу **B** не потрібно описувати як дружні класу **A**.

Як приклад розглянемо програму **Friend_2.cpp**.

```
/* Friend_2.cpp - приклад опису дружнього класу */
class One
{
    int a;
    void Onefunc()
    { cout<<"Функція Onefunc викликана...\n";}
    friend class Two; // оголошення дружнього класу
  public:
    One(int ia=1) {a=ia;}
};
class Two // цей клас дружній класу One
{
 public:
    void Twofunc(One &one)
    {
        cout<<"one.a="<<one.a<<'\n';
        one.Onefunc();
    }
// інші компоненти класу
};
void main()
{
    One First(5);
    Two Second;
    Second.Twofunc(First); _getch();
}
```

Результат виконання програми:

```
one.a=5
Функція Onefunc викликана...
```

Як видно з прикладу програми, компонентна функція **Twofunc()** класу **Two**, який оголошений дружнім класу **One**, для одержання доступу до компонентних даних і функцій класу **One** має одержати екземпляр класу **Two** як параметр. Слід звернути увагу на те, що компоненти класу **One**, до яких звертається функція **Twofunc()**, є приватними компонентами цього класу.

Якщо з опису класу **One** вилучити оголошення дружнього класу (**friend class Two**), то при компіляції програми одержимо повідомлення про помилку: компоненти **One::a i One::Onefunc()** є недоступними. З іншого боку, якщо ці компоненти оголосити як загальнодоступні, то програма буде компілюватися й виконуватися так само, як і наведена вище. Інакше кажучи, оголошення якого-небудь класу дружнім іншому класу зводиться до того, що дружній клас одержує доступ до всіх компонентів іншого класу незалежно від статусу доступу.

7. ПЕРЕВАНТАЖЕННЯ ОПЕРАЦІЙ

7.1. Поняття й основні правила перевантаження операцій

У мові C++, як і в інших мовах програмування, є набір стандартних операцій, які можна застосовувати до базових і похідних типів даних, причому правила застосування цих операцій регламентовано. Разом з тим мова C++ надає програмісту можливість поширювати дії стандартних операцій на такі типи даних, для яких ці операції в мові не визначено.

Як типи даних, для яких програміст перевизначає операції, можуть бути як визначені в мові типи даних, так і ті, що визначаються програмістом як нові типи даних. Наприклад, можна визначити правила виконання бінарної операції додавання **x+y** для рядків символів, масивів та інших типів. Оскільки чудес у програмуванні не буває, то зрозуміло, що ці правила програмісту, зрештою, доведеться якимось чином визначити.

Дифірамб. Перевантаження операцій у мові C++ є настільки потужним і витонченим засобом мови, що переоцінити його просто неможливо. Фактично, завдяки цій можливості можна визначити для свого конкретного завдання або предметної області такий похідний тип даних, який буде поводитися подібно визначеним (базовим) у мові типам даних. Наприклад, відсутність у мові вбудованого контролю індексів масиву досить просто можно замінити шляхом розроблення відповідного класу з усіма бажаними і необхідними для конкретного завдання властивостями. Таким чином, завдяки саме перевантаженню операцій з'являється можливість суттєво, якщо не сказати безмежно, розширити можливості мови C++.

Можливість розширення дії (перевантаження – overload) стандартної операції на інший тип даних досягається тільки в тому випадку, якщо хоча б один з операндів операції є екземпляром класу. Визначення операції, що перевантажується, оформляється у вигляді спеціальної операції-функції, у заголовку якої вказується ключове слово **operator**. Ця функція має такий формат:

тип_функції operator знак_операції (список формальних параметрів)
{ тіло операції-функції }

Тут *тип_функції* – це, як і для звичайних функцій, тип значення, що повертається функцією. Зазвичай формальні параметри мають тип клас, однак у деяких випадках вони можуть мати і будь-який із визначених у мові типів. У деяких випадках, що буде застережено нижче, операція-функція може взагалі не мати формальних параметрів.

Як і будь-яка інша функція, операція-функція може мати прототип. Наприклад, якщо необхідно визначити операцію додавання для екземплярів **X** і **Y** класу **Newtype**, то прототип операції-функції може мати такий вигляд:

Newtype operator +(Newtype X, Newtype Y);

Використання перевантаженої операції буде мати звичайний вигляд, наприклад **X+Y**, що інтерпретується як виклик операції-функції **operator +(X,Y)**.

Основні правила перевантаження операцій

Для забезпечення можливості перевантаження будь-якої операції операція-функція має задовольняти одну з таких вимог:

- бути компонентною функцією класу;
- бути дружньою класу функцією;
- бути глобальною функцією й мати принаймні один параметр типу клас.

Кількість параметрів операції-функції залежить від кількості операндів – арності – операції, що перевантажується, і від способу її оформлення: компонентна, дружня або глобальна функція.

Інші правила, що стосуються перевантаження операцій, такі.

Не допускається вводити власні позначення операцій, що не збігаються зі стандартними операціями мови. Наприклад, неможливо визначити операцію @, оскільки вона у мові C++ відсутня.

Більшість операцій мови C++ можуть бути перевантажені за винятком таких:

- «крапка» як прямий вибір компонента структурного типу (**клас.компонент**);
- вибір компонента через покажчик на нього .* (**клас.*компонент**);
- сфера дії області видимості (**::**);
- трійкова або умовна операція **(?:)**;
- операція **sizeof**;
- **typeid**;

- **throw**;
- **dynamic_cast**;
- **static_cast**;
- **const_cast**;
- **reinterpret_cast**.

Не можуть бути перевантаженими препроцесорні операції # і ##, оскільки вони і не є операціями мови, а також неможливо перевантажити роздільники, наприклад крапку з комою.

При перевантаженні стандартних операцій неможливо змінити їхній пріоритет і синтаксичні правила обчислення виразів. Наприклад, неможливо ввести бінарну операцію прирощення (++) або унарну операцію присвоєння (=).

Перевантаження бінарних операцій

Будь-яка бінарна операція для деякого класу може бути реалізована двома різними способами: як компонентна функція з одним параметром або як глобальна, можливо, дружня, функція з двома параметрами.

Якщо операція-функція оформляється як компонентна, то як перший операнд використовується екземпляр класу, для якого викликана операція-функція, а як другий – параметр компонентної операції-функції. Наприклад, якщо **A** і **B** – екземпляри класу, для якого перевантажена операція **"+"** з використанням глобальної або дружньої функції, то вираз **A+B** трактується як такий виклик операції-функції: **operator +(A,B)**. Якщо ж операція-функція оформлена як компонентна функція, той же вираз трактується як **A.operator+(B)**.

Інакше кажучи, при оформленні операції-функції як компонентної функції класу **A** як перший операнд використовуються компоненти класу **A**, а які другий – компоненти класу-параметра **B.**

Для перевантаження бінарних операцій **=, [], (), ->, new, delete** можуть бути використані тільки нестатичні компонентні операції-функції для того, щоб гарантувати, що першими операндами обов'язково будуть лівосторонні вирази (lvalue - left value). Виключення становлять операції **new** і **delete**, які за замовчуванням є статичними (**static**).

Перевантаження унарних операцій

Для перевантаження унарної операції може використовуватися компонентна функція без параметрів, звичайна або дружня функція з одним параметром.

Інші правила перевантаження операцій

Як відомо, у мові C++ є такі операції над базовими типами даних мови, які є комбінаціями інших операцій. Наприклад, операція ***=** є комбінацією операцій множення і присвоєння. Якщо операції **=** і ***** перевантажені, то із цього не випливає можливість використання операції ***=**: вона повинна перевантажуватися окремою операцією-функцією.

Перевантаження стандартної операції для будь-якого класу не змінює правил і змісту виконання цієї операції для виразів, у які не входять екземпляри класу. Наприклад, правила обчислення виразу **A+5.2** для змінної **A** типу **float** не змінюються і змінити їх неможливо. З іншого боку, якщо **A** є екземпляром класу, для якого перевантажена операція +, то виконується перевантажена операція, якщо вона передбачена для такого випадку.

Останній випадок, а саме використання в одному виразі класу (з перевантаженою операцією) і якого-небудь базового типу потребує більш детального розгляду.

Припустимо, що **A** і **B** є екземплярами класу, для якого перевантажена бінарна операція **+**. У цьому випадку для обчислення виразів **A+B** операція-функція може бути оформлена як компонентною функцією, так і глобальною, можливо, дружньою, функцією. Те ж саме можна сказати і про обчислення виразів типу **A+5**.

Якщо ж передбачається можливість використання виразів типу **5+A**, то операція-функція не може бути оформлена як компонентна функція з тієї причини, що обчислення виразу **5+A** еквівалентно виразу **5.operator +(A)**, який є помилковим. Отже, для цього випадку необхідно визначати операцію-функцію як глобальну або дружню. Є й інший варіант: зробити явне зведення типу константи до типу класу за допомогою конструктора класу, що буде розглянуто на прикладі перевантаження операцій для комплексних чисел.

При перевантаженні бінарних операцій необхідно враховувати всі можливі комбінації типів операндів. Наприклад, при перевантаженні операції додавання комплексних чисел необхідно враховувати, і це повинен зробити програміст, такі можливі випадки:

- додавання двох комплексних чисел;
- додавання комплексного числа до дійсного;
- додавання дійсного числа до комплексного.

Завдяки угодам про перетворення стандартних типів параметрів можна не враховувати той факт, що дійсне число, що складається, може мати ще й довільний числовий тип.

У мові C++ діють також правила, що дозволяють програмісту визначати операції-функції для префіксного й постфіксного вживання унарних операцій ++ і --, які у цьому посібнику не розглядаються.

7.2. Приклад перевантаження операцій для комплексних чисел

У наведеній нижче програмі **Cmplx_1.cpp** зроблено опис класу **TComplex**, який призначений для оброблення комплексних чисел. Клас надає такі можливості:

- задавати початкові значення для комплексних чисел за допомогою конструктора **TComplex(double ire=0, double iim=0)**;
- одержувати дійсну й уявну частини комплексного числа за допомогою компонентних функцій **get_Re** і **get_Im()** відповідно;
- виводити на екран значення комплексного числа за допомогою компонентної функції **Printcmplx()**;
- виконувати перевантажені операції додавання й множення двох комплексних чисел або дійсного числа й комплексного.

> **Зауваження**. До складу бібліотек мови входить заголовний файл **complex.h**, що містить опис класу **complex**. Подивіться на його реалізацію після розбору тексту наведеної нижче програми. У стандартній бібліотеці мови C++ визначений шаблонний клас **complex**.

В ілюстративних цілях перевантажена операція-функція додавання оформлена у вигляді компонентної функції, а операція множення – у вигляді дружньої функції.

```
/* Cmplx_1.cpp – ілюстрація класу TComplex, що опрацьовує комплексні
                 числа */
#include <Conio.h>
#include <Iostream.h>
class TComplex
{
  double Re,Im;
public:
  TComplex(double ire=0, double iim=0)
  { Re=ire; Im=iim; }
  double get_Re(){ return Re;}
  double get_Im(){ return Im;}
  TComplex operator +(TComplex X)
     { return TComplex(Re+X.Re,Im+X.Im);}
  friend TComplex operator *(TComplex X,TComplex Y);
  void Printcmplx()
  { cout<<"real="<<Re<<",imag="<<Im<<endl; }
};
TComplex operator *(TComplex X,TComplex Y)
{ return TComplex(X.Re*Y.Re-X.Im*Y.Im, X.Re*Y.Im+Y.Re*X.Im);}

void main()
```

```
{
  TComplex a(1.23,4), b, c;
  b=TComplex(1,-1);
  cout<<"Число a: "; a.Printcmplx(); // Число a: real=1.23,imag=4
  cout<<"Число b: "; b.Printcmplx(); // Число b: real=1,imag=-1
  c=a+b; cout<<" a+b : "; c.Printcmplx(); // a+b : real=2.23,imag=3
  c=a*b; cout<<" a*b : "; c.Printcmplx(); // a*b : real=5.23,imag=2.77
  c=a+5; cout<<" a+5 : "; c.Printcmplx(); //a+5 : real=6.23,imag=4
  c=a*5; cout<<" a*5 : "; // a*5 : real=6.15,imag=20
  c.Printcmplx();
  c=5*a; cout<<" 5*a : "; // 5*a : real=6.15,imag=20
  c.Printcmplx();
  // c=3+b; cout<<" Сума чисел: ";
  _getch();
} // кінець файлу Cmplx_1.cpp
```

Пояснимо програму. Конструктор класу

```
TComplex(double ire=0, double iim=0)
{ Re=ire; Im=iim; }
```

що має формальні параметри зі значеннями за замовчуванням *спеціально для того*, щоб мати можливість комбінувати у виразах комплексні й дійсні числа. Якщо вилучити з конструктора значення формальних параметрів за замовчуванням, то одержимо повідомлення компілятора про помилку "illegal structure operation" – неприпустима операція над структурою, що вказує на вирази, в яких змішуються екземпляри класу й дійсні числа: **a+5**, **a*5**, **5*a**. Справа в тому, що за наявності у конструктора формальних параметрів зі значеннями за замовчуванням вираз **a+5** обробляється компілятором як **a+TComplex(5)**, тобто компілятор може виконати (і робить це) неявне зведення типу з використанням конструктора **TComplex**. Якщо ж конструктор не має значень за замовчуванням для формальних параметрів, то він не може бути викликаний неявно.

Те ж повідомлення про помилку буде отримано при спробі вилучити коментар з оператора **c=3+b**. У цьому випадку причина помилки полягає в тому, що для обчислення виразу **3+b** виклик компілятором операції-функції додавання має вигляд **3.operator +(b)**. Оскільки константа **3** не є екземпляром класу **TComplex**, такий виклик, зазвичай, неприпустимий. Для того, щоб такі вирази могли бути використані, можна запропонувати:

- оформити операцію-функцію додавання як некомпонентну (дружню або глобальну) функцію із двома параметрами;
- вираз **3+b** переписати у вигляді **TComplex(3)+b**, скориставшись конструктором для перетворення числа **3** у комплексне число, точніше, в екземпляр класу **TComplex**.

Таким чином, компілятор неявно використовує конструктор **TComplex** для зведення типу числа **5** до типу **TComplex** при обчисленні виразу **a+5**, завдяки чому, власно, і можливий запис таких виразів.

При зведенні числа **5** до типу **TComplex** неявний виклик конструктора має вигляд **TComplex(5,0)**, так що дійсна частина комплексного числа стає такою, що дорівнює **5**, а уявна – **0**. Якби треба було задати комплексне число з нульовою дійсною частиною й ненульовою уявною, то довелося б використовувати явний виклик конструктора, наприклад **TComplex(0,123)**. Вираз вигляду **TComplex(0,123)** являє собою безіменний екземпляр класу **TComplex**.

Слід звернути увагу також на те, що як операнди розглянутих операцій можна використовувати цілі числа знов-таки завдяки тому, що компілятор виконує неявне зведення цілого типу до дійсного.

У класі **TComplex** було б не зайво виконати перевантаження операції виведення значення комплексного числа на монітор за допомогою класу **ostream**, який входить до складу стандартної бібліотеки, що поставляється разом з компілятором.

Як знак операції виведення доцільно вибрати, але зовсім не обов'язково, усе ту ж операцію зсуву вліво **<<**, яка вже використана в класі **ostream**. Операцію-функцію оформимо у вигляді дружньої функції, додавши її в клас **TComplex**:

```
class TComplex
{
     // колишні компоненти – дивись  програму Cmplx_1.cpp
  friend ostream & operator <<(ostream &Out, TComplex &X)
  { Out<< '('<<X.Re<<"+i"<<X.Im<<')'; return Out; }
// інший варіант:
  friend ostream & operator <<(ostream &Out, TComplex &X)
  { return Out<< '('<<X.Re<<"+i"<<X.Im<<')';}
};
```

У новій версії класу **TComplex** виведення комплексних чисел на монітор набуває такого простого вигляду:

```
cout<<"Число a: "<<a<<endl;
```

Оформити перевантаження операції виведення можна у вигляді такої операції-функції:

```
friend void operator <<(ostream &Out, TComplex &X)
{ Out<< '('<<X.Re<<"+i"<<X.Im<<')';}
```

Різниця у цьому випадку полягає в тому, що цей варіант операції-функції не повертає ніякого значення, у той час як у попередньому варіанті функція мала тип значення **ostream &** (посилання на клас), що повертається. Що із цього випливає? Якщо спробувати за цим варіантом операції-функції записати оператор

```
cout<<"Число a: "<<a<<endl;
```

то одержимо повідомлення компілятора про помилку «Not an allowed type» (неприпустимий тип), причому курсор при цьому буде вказувати на маніпулятор **endl**!

У той же час використання операторів начебто зовсім подібних

```
cout<<"Число a: "<<a;
cout<<endl;
```

не викликає повідомлення про помилку і результат їх виконання коректний. Отже, повернення функцією посилання на клас **ostream** необхідно для того, щоб цю операцію можна було використовувати в ланцюжку, тобто щоб після виведення значення комплексного числа можна було включати в потік **cout** й інші дані.

Додамо в клас **TComplex** перевантажені операції - (відняти) і -= (відняти й присвоїти):

```
class TComplex
{
    // колишні компоненти – дивись програму Cmplx_1.cpp
  public:
     TComplex operator -(TComplex X);
     TComplex operator -=(TComplex X)
     {Re-=X.Re; Im-=X.Im; return *this; }
};
TComplex TComplex::operator -(TComplex X)
{ return TComplex(Re-X.Re, Im-X.Im);}
```

Для ілюстративних цілей визначення операції віднімання винесено з тіла класу. Після додавання перевантажених операцій віднімання можна записати такі вирази:

```
a=b-c;
a-=c;
```

Добре, а чи припустимі такі вирази:

```
a=1.23;  b=0;  a-=b=c;  a=a*b+c;  //ma  інші?
```

Перевірте. Перевантажену операцію «відняти й присвоїти» можна оформити й так:

```
TComplex& operator -=(TComplex X)
{Re-=X.Re; Im-=X.Im; return *this; }
```

Таким чином нова реалізація операції повертає посилання на клас. Що із цього випливає? Запишемо такий вираз:

```
(a-=b)=0;
```

Якщо операція повертає посилання, то значення **a** після обчислення виразу дорівнюватиме **0**, а якщо ні, то **a-b**.

Можна написати глобальну операцію-функцію **/** (ділення двох комплексних чисел):

```
void operator /(TComplex X,TComplex Y) { return; }
```

можна навіть у програмі використовувати оператор-вираз вигляду

```
a/b;
```

але алгоритмічного змісту в цьому немає, оскільки функція не повертає результат і не може, оскільки має тип **void**.

> **Нагадування**. Якщо Ви прочитали цей підрозділ і не набирали текст програми, випробовуючи її на практиці, – то це тільки дуже поверхневе ознайомлення з перевантаженням операцій, а не їх вивчення та освоєння.

7.3. Деякі особливості перевантаження операцій на прикладі комплексних чисел

При реалізації операцій-функцій, особливо для класів, член-дані яких розміщуються в динамічній пам'яті, корисно проаналізувати порядок викликів конструкторів, конструкторів копіювання й деструкторів (проект Cmplx).

Із цією метою, по-перше, додамо в клас власний конструктор копіювання винятково для того, щоб відстежити момент його виклику. По-друге, додамо в конструктори й деструктор макроси **TRACE**, за допомогою яких будемо відслідковувати виклики конструкторів і деструктора.

Розглянемо спочатку клас комплексних чисел:

```
class Complex
{
    double Re,Im;
    char * Objname; // для зберігання імені об'єкта
 public:
    Complex(double ire=0, double iim=0, char * Objname="")
    {
          Re=ire; Im=iim; this->Objname=Objname;
          TRACE("Complex Objname=%s  this=%d\n",Objname, (int) this);
    }
    ~Complex()
   {
          if((!Objname) || ((int)Objname==0xcccccccc)) Objname="Bad_Ptr";
```

```
        TRACE("~Complex Objname=%s  this=%d\n",Objname,(int)this);
    }
    Complex(const Complex & Org)
    {
        TRACE("Copyctr Org.Objname=%s this=%d\n", Org.Objname,
                (int)this);
        Re=Org.Re; Im=Org.Im; Objname=Org.Objname;
    }
    Complex & operator=(const Complex & Org)
    {
      TRACE("operator= Org.Objname=%s this=%d\n", Org.Objname,
            (int) this);
      Re=Org.Re; Im=Org.Im;
      return *this;
    }
    Complex operator +(Complex &X)
    {
        return Complex(Re+X.Re,Im+X.Im,"Return operator+");
    }
    Complex operator -(Complex X);
    Complex operator -=(Complex X)
    { Re-=X.Re; Im-=X.Im; return *this;}
    friend Complex operator *(Complex X,Complex Y);
    friend ostream & operator <<(ostream &Out, Complex X)
    { return Out<< '('<<X.Re<<"+i"<<X.Im<<')';}
};
Complex Complex::operator -(Complex X)
{return Complex(Re-X.Re,Im-X.Im);}

Complex operator *(Complex X,Complex Y)
{
    return Complex(X.Re*Y.Re-X.Im*Y.Im, X.Re*Y.Im+Y.Re*X.Im);
}
```

Перевірка коректності значення покажчика **Objname** у деструкторі виконана для того, щоб уникнути помилки часу виконання в деяких складних випадках.

Проаналізуємо результати виконання такого програмного коду. У коментарях наведено результати роботи макросів **TRACE**.

```
void main()
{
 {
    Complex a(1.23,4,"a"), b(2.34,5,"b"), c(0,0,"C");
                //Complex Objname=a  this=1244980
                //Complex Objname=b  this=1244948
                //Complex Objname=C  this=1244916
      c=a+b;
```

```
                    //Copyctr Org.Objname=b  this=1244604
                    //Complex Objname=Return operator+  this=1244628
                    //~Complex Objname=b  this=1244604
                    //operator= Org.Objname=Return operator+  this=1244824
                    //~Complex Objname=Return operator+  this=1244628
                    //Copyctr Org.Objname=C  this=1244624
                    //~Complex Objname=C  this=1244624
      /* Якщо параметр операції-функції додавання оголосити посиланням
         Complex operator +(Complex &X),  то одержимо таку послідовність
         викликів методів:
              Complex Objname=Return operator+  this=1244648
              operator= Org.Objname=Return operator+  this=1244916
              ~Complex Objname=Return operator+  this=1244648
              Copyctr Org.Objname=C  this=1244644
              ~Complex Objname=C  this=1244644
   */

      /* Якщо операцію-функцію присвоєння оголосити як
         Complex & operator=(const Complex & Org)
         (і при цьому операцію-функцію додавання оголосити як
         Complex operator +(Complex &X),
         то одержимо таку послідовність викликів методів:
              Complex Objname=Return operator+  this=1244680
              operator= Org.Objname=Return operator+  this=1244916
              ~Complex Objname=Return operator+  this=1244680
    */
        cout<<c<<endl;
              //Copyctr Org.Objname=C  this=1244624
              //~Complex Objname=C  this=1244624
      }
              //~complex Objname=C  this=1244916
              //~complex Objname=b  this=1244948
              //~complex Objname=a  this=1244980
  }
```

Для того щоб відстежити порядок виклику конструкторів і деструкторів, необхідно виконати трасування програми, причому із заходом у функції.

Які основні висновки можна зробити з результатів виконання цієї програми, точніше, реалізації операцій-функцій?

По-перше, якщо об'єкт передається у функцію як параметр-значення, то його копія створюється за допомогою конструктора копіювання. Це означає, що якщо реалізація конструктора копіювання за замовчуванням не влаштовує, то потрібно розробити його коректну реалізацію. Наприклад, якщо серед член-даних класу є покажчики й для них виділяється динамічна пам'ять, то потрібно розробити коректний конструктор так званого «глибокого копіювання». Також слід зазначити, що конструктор копіювання

викликається й при поверненні об'єкта з функції за допомогою оператора **return**.

Якщо операція -= (відняти й присвоїти) оформлена так:

```
Complex  operator -= (Complex X)
{ Re-=X.Re; Im-=X.Im; return *this;}
```

то при обчисленні виразу **a -= b** одержимо таку послідовність викликів конструкторів і деструкторів:

```
Complex Objname = a  this = 4455536
Complex Objname = b  this = 4455504
Copyctr Org.Objname = b  this = 4455124
Copyctr Org.Objname = a  this = 4455176
~complex Objname = b  this = 4455124
~complex Objname = a  this = 4455176
~complex Objname = b  this = 4455504
~complex Objname = a  this = 4455536
```

Зверніть увагу, що деструктори об'єктів викликаються начебто двічи: знайдіть цьому пояснення.

І ще один приклад. Якщо операцію -= оформити таким чином:

```
Complex  & operator -= (Complex &X)
{ Re-=X.Re; Im-=X.Im; return *this;}
```

то при обчисленні виразу **a -= b** одержимо таку послідовність викликів конструкторів і деструкторів:

```
Complex Objname = a  this = 4455536
Complex Objname = b  this = 4455504
~complex Objname = b  this = 4455504
~complex Objname = a  this = 4455536
```

Як говориться, відчуйте різницю!

7.4. Приклад класу масиву з перевантаженими операціями

Як ще один приклад розробимо клас **CArr**, призначений для оброблення одновимірних масивів.

```
class CArr
{
    double *Arr;
    int N;
    char * ObjName;
public:
    CArr(int N, char * Name="");
    CArr (const CArr &Org);
    ~CArr();
```

```
    bool getElem(int i, double &Elem);
    bool setElem(int i, const double Elem);
    friend ostream & operator <<(ostream &Out, const CArr &Obj);
    double & operator [](int Index);
    CArr * operator +(const CArr &B);
    CArr   operator -(const CArr &B);
    CArr & operator =(CArr &X);
};
CArr::CArr(int N, char * Name)
{
    this->N=N;
    Arr=new double [N];
    ObjName=Name;
    TRACE("CArr %s %d\n", ObjName, (int) this);
}
CArr::CArr (const CArr &Org)
{
    this->ObjName="CopyCTR";
    TRACE("CopyCTR Org=%s this=%s\n", Org.ObjName,  ObjName);
    N=Org.N;
    Arr=new double [N];
    for(int i=0;i<N;i++)
        Arr[i]=Org.Arr[i];
}
CArr::~CArr()
{
    TRACE("~CArr %s %d\n",ObjName,(int)this);
    delete []Arr;
}
bool CArr::getElem(int i, double &Elem)
{
    if((i>N-1)||(i<0)) return false;
    Elem=Arr[i];
    return true;
}
bool CArr::setElem(int i, const double Elem)
{
    if((i>N-1)||(i<0)) return false;
    Arr[i]=Elem;
    return true;
}
ostream & operator <<(ostream &Out, const CArr &Obj)
{
    Out.precision(2);
    for(int i=0;i<Obj.N;i++)
    {
        Out.width(5); Out<<Obj.Arr[i];
```

```
    }
    return Out;
}
double & CArr::operator [](int Index)
{
    return Arr[Index];
}
CArr * CArr::operator +(const CArr &B)
{
    CArr *Res;
    Res=new CArr(N);
    for(int i=0;i<N;i++)
        Res->Arr[i]=this->Arr[i]+B.Arr[i];
    return Res;
}
CArr CArr::operator -(const CArr &B)
{
    CArr Res(N,"Res");
    for(int i=0;i<N;i++)
        Res.Arr[i]=this->Arr[i]-B.Arr[i];
    return Res;
}
CArr & CArr::operator =(CArr &X)
{
    this->ObjName=X.ObjName;
    TRACE("= Org=%s this=%d\n", X.ObjName, (int)this);
    delete []Arr;
    N=X.N;
    Arr=new double [N];
    for(int i=0;i<N;i++)
        Arr[i]=X.Arr[i];
    return *this;
}
```

Реалізація операції віднімання масивів, як й інших корисних на практиці операцій, віддана на ваш розсуд. Після виконання цієї роботи використання об'єктів класу може бути реалізовано у такий спосіб:

```
void main()
{
    setlocale(LC_ALL,"rus");
    srand( (unsigned)time( NULL ) );
    int N=7;
    CArr A(N,"A");
    CArr B(N,"B"), *C;
    for(int i=0;i<N;i++)
    {
        double Val=(double)rand()/RAND_MAX*10;
```

```
            A.setElem(i,Val);
            Val=(double)rand()/RAND_MAX*10;
            B.setElem(i,Val);   // можна так
            B[i]=Val;           // а можна й так
        }
        cout<<"Масив A"<<endl<<A<<endl<<endl;
        cout<<"Масив B"<<endl<<B<<endl<<endl;
        C=A+B;
        cout<<"Масив C=A+B"<<endl<<*C<<endl<<endl;
        delete C;
        CArr D(N,"D");
        D=A-B;
        cout<<"Масив D=A-B"<<endl<<D    <<endl<<endl;
        cout<<"Масив A-B"  <<endl<<(A-B)<<endl<<endl;
}
```

8. ШАБЛОННІ КЛАСИ

8.1. Визначення шаблонного класу

Перш ніж розглядати шаблонний клас, створимо, як приклад, «звичайний» клас стек у самому найпростішому вигляді:

```
// Файл Stack.h
#pragma once
typedef  int  ElType;
class CStack
{
    enum {Max=5}; /*кількість елементів масиву*/
    ElType Vec[Max]; /*містить елементи стека*/
    int Top; /*індекс останнього елемента стека*/
public:
    CStack(void) { Top=-1;}
/* Push - Поміщає елемент Elem у стек.  Повертає true, якщо стек не
        заповнений до кінця*/
    bool Push(const ElType & Elem)
    {
        if(Top<(Max-1)){Vec[++Top]=Elem; return true;}
        return false;
    }
/* Pop - Витягає елемент Elem зі стека. Повертає true, якщо стек не
        порожній*/
    bool Pop(ElType & Elem)
    {
        if(Top<0) return false;
        else    { Elem=Vec[Top--]; return true;}
    }
/* IsEmpty - Повертає true, якщо стек порожній*/
    bool IsEmpty(){return Top<0; }
    /* IsFull - Повертає true, якщо стек заповнений*/
    bool IsFull(){ return Top>=Max-1;}
};
```

Для спрощення тексту програми реалізація функцій класу поміщена безпосередньо в тіло класу. Для первісного тестування класу досить виконати такий програмний код:

```
    CStack stk;
    ElType elem=1;
    cout<<Ukr("Поміщаємо в стек дані: ");
    /*Визначення функції Ukr наведено у додатку*/
    while(stk.Push(elem))
    {
        cout<<elem<<' ';   elem*=2;
```

```
}
cout<<Ukr("\n і вилучаємо їх: ");
while(stk.Pop(elem))     cout<<elem<<' ';
```

Результати виконання наведеного фрагмента програми:

Поміщаємо в стек дані: 1 2 4 8 16
і вилучаємо їх: 16 8 4 2 1

Як сховище класу використано статичний масив, що у багатьох випадках є недоліком. Цей недолік можна усунути, якщо як сховище даних стеку використовувати покажчик і динамічно перерозподіляти пам'ять.

Другий момент. Природньо, що хотілося б мати такий клас, який би дозволяв зберігати у стеку дані будь-якого типу. Для того щоб підвищити незалежність класу **CStack** від типу даних, уведено визначення типу

```
typedef  int  ElType;
```

Завдяки цьому для указання конкретного типу даних, що зберігаються у стеку, досить змінити тільки одне визначення, наприклад

```
typedef  CMyClass  ElType;
```

Поряд з деяким позитивним ефектом цей підхід має той недолік, що за необхідності мати в одній програмі два і більш стеків з різними типами даних, доведеться створювати копії класу **CStack** з різними іменами.

Усунути цей істотний недолік можна шляхом створення шаблонного класу стеків Шаблонний клас можна визначити як такий опис класу, на основі якого компілятор автоматично згенерує код конкретного класу. Шаблонний клас, подібно шаблонам функцій, являє собою параметризований клас. Конкретний клас створюється шляхом задання фактичних параметрів, кількість яких довільна. Шаблонні класи створюються у тих випадках, коли необхідно створювати довільну кількість класів, що мають однаковий програмний код, але різні типи оброблюваних даних. Шаблонні класи реалізують один з видів поліморфізму – *поліморфізм типів*.

> **Зауваження.** Страуструп [1] розрізняє два види поліморфізму: поліморфізм часу виконання й поліморфізм часу компіляції. Поліморфізм часу виконання забезпечується віртуальними функціями: віртуальна функція в похідному класі може бути замінена на іншу з тим же іменем, але з іншим кодом. Шаблони класів забезпечують поліморфізм часу компіляції.

Визначимо шаблонний клас стеків (проект Stack_v2).

```
// Файл Stack.h
```

```
#pragma once
template <class  ElType>
class CStack
{
    // колишній опис класу без змін
};
```

Як випливає з опису шаблонного класу, він має починатися із ключового слова мови **template** (шаблон), слідом за яким у кутових дужках мають бути зазначені формальні параметри шаблону. Тепер створимо конкретний клас і протестуємо його так само, як і вище:

```
Cstack <double> stk; // визначення конкретного класу
double elem=0.5;
cout<<Ukr("Поміщаємо в стек дані: ");
while(stk.Push(elem)) {  cout<<elem<<' '; elem*=2; }
cout<<Ukr("\nі вилучаємо їх: ");
while(stk.Pop(elem)) cout<<elem<<' ';
```

Конкретний клас створюється компілятором шляхом підстановки фактичного параметра шаблону – у даному прикладі **double** – замість формального параметра шаблону **ElType**.

Якщо функція шаблонного класу визначається поза тілом класу, то шаблон класу має повторюватися і вслід за іменем класу ім'я параметра шаблону має дублюватися, наприклад:

```
template <class ElType>
bool Cstack<ElType>::IsEmpty() { return Top<0; }
```

Зазвичай, як і для випадку шаблонних функцій, шаблонний клас може мати довільну кількість параметрів, наприклад:

```
template <class Type_1, class Type_2>
class CExample
{  /*...*/ };
```

Замість ключового слова **class** можна використовувати **typename**:

```
template <typename Type_1, typename Type_2>
class CExample
{ /*...*/ };
```

У загальному випадку формальний параметр шаблону може оголошуватися двома способами:

- ✓ class ідентифікатор_параметризованого_типу;
- ✓ ім'я_типу ідентифікатор.

У шаблонному класі **CStack** оголошувався параметр першого типу **ElType**: **<class ElType>.**

Як фактичний параметр у цьому випадку потрібно вказати ім'я типу, у даному прикладі **double**:

```
CStack <double> stk;
```

За другим способом формальний параметр шаблону оголошується, наприклад так:

```
<int Num>
```

У цьому випадку фактичний параметр шаблону має бути константою зазначеного типу, для даного прикладу – цілочислового типу. Тоді в тілі класу цей параметр трактується як константа. Така можливість дозволяє визначити шаблон класу **CStack** таким чином:

```
// Файл Stack.h
#pragma once
template <class ElType, int Max>
class CStack
{
//    enum {Max=5}; /*кількість елементів масиву*/
      ElType Vec[Max]; /*містить елементи стека*/
      int Top; /*індекс останнього елемента стека*/
public:
      // колишній опис функцій класу без змін
};
```

Слід зазначити, що перелічення **enum {Max=5}** тепер не потрібно і воно закоментовано. Використання нової версії шаблонового класу може виглядати так:

```
CStack <double, 7> stk;
double elem=0.5;
cout<<Ukr("Поміщаємо в стек дані: ");
while(stk.Push(elem)) {  cout<<elem<<' '; elem*=2; }
cout<<Ukr("\n і витягаємо їх: ");
while(stk.Pop(elem)) cout<<elem<<' ';
```

Як бачимо, нова версія шаблонного класу значно зручніша у використанні, оскільки дозволяє не тільки задати тип даних, що зберігаються у стеку, але й максимальну кількість даних, які можуть зберігатися у ньому.

8.2. Приклад шаблонного класу масиву

Як ілюстрацію використання шаблонного класу розглянемо приклад дуже простої реалізації шаблонного класу масив **CArr** (проект **Vector**). У чому полягає головна відмінність класу **CArr** від класу **TVector**, наведеного вище у підрозділі 3.8? Щоб використовувати клас **TVector** для іншого типу даних, треба перевизначити тип **ElType** і виконати компіляцію цього класу для свого власного типу даних. Якщо в додатку треба використовувати масиви декількох типів, то доведеться створювати копії класу **TVector** для кожного типу даних. Неважко уявити собі також увесь той кошмар, який прийдеться пережити за необхідності модифікації класу **TVector**. У той же час використання шаблону класу не потребуватиме перевизначення типів або створення різних варіантів реалізації класу масиву: компілятор сам згенерує стільки класів масиву з різним типом елементів, скільки вам знадобиться. Тут має місце практично повна аналогія з використанням перевантажених і шаблонних функцій.

Опис шаблонного класу **CArr**:

```
template <class ElType, int Size>
class CArr
{
        ElType Mas[Size];
    public:
        CArr() { for(int i=0;i<Size; i++) Mas[i]=0;}
        bool Put(const ElType & Elem, int Index);
        bool Get(ElType & Elem, int Index);
        ElType operator [] (int index) {return Mas[index];}
};

template <class ElType, int Size>
bool CArr<ElType,Size>::Put(const ElType & Elem, int Index)
{
    if (Index>=0 && Index<Size)
    {
        Mas[Index]=Elem;
        return true;
    }
    else return false;
}

template <class ElType, int Size>
bool CArr<ElType,Size>::Get(ElType & Elem, int Index)
{
    if (Index>=0 && Index<Size)
```

```
    {
        Elem=Mas[Index];
        return true;
    }
    else return false;
}
```

Приклад використання шаблону класу **CArr**.

```
CArr <float, 5> fmas1, fmas2; /* 5 – максимальна кількість елементів
    масиву  */
int i=0;
float F=1.0;
while(fmas1.Put(F,i++))F+=0.5;
i=0;
cout<<"fmas1 : ";
while(fmas1.Get(F,i++)) cout<<F<<' '; // 1 1.5 2 2.5 3
cout<<endl;
fmas2=fmas1;
cout<<"fmas2: ";
for(i=0;i<5;i++) cout<<fmas2[i]<<' '; // 1 1.5 2 2.5 3
cout<<endl;
fmas2[0]=-1;
cout<<"fmas2: ";
for(i=0;i<5;i++) cout<<fmas2[i]<<' '; // -1 1.5 2 2.5 3
cout<<endl;

cout<<"fmas1: ";
for(i=0;i<5;i++) cout<<fmas1[i]<<' '; // 1 1.5 2 2.5 3
cout<<endl;
```

Для того щоб можна було використовувати вираз **fmas2[i]** як лівосторонній вираз (lvalue), тобто мати можливість присвоювання значення елементам масиву з використанням перевантаженої операції **[]**, треба цю перевантажену операцію визначити так, щоб вона повертала посилання на **ElType**:

```
ElType & operator [] (int index) {return Mas[index];}
```

Зазвичай у такому шаблоні використовується найпростіший спосіб зберігання елементів масиву – просто в масиві, розмір якого задається при визначенні екземпляра класу. (Разом з тим цей найпростіший спосіб має й максимальну ефективність). Для розширення функціональності в «дорослому» шаблоні класу треба було б увести багато вдосконалень:

- розширити кількість функцій і операцій, які надаються класом;
- реалізувати контроль над індексами елементів масиву;
- додати генерацію виключень при виявленні помилок;

- додати можливість автоматичного виділення пам'яті при розширенні масиву тощо

Слід зазначити, що при генерації реального класу на основі шаблону генеруються тільки ті функції, які реально використовуються в програмі.

8.3. Шаблон класу TVector

Усі описи треба розмістити у заголовному файлі Vector.h.

```
template <class ElType>
class TVector
{
protected:
    ElType * pvec;
    int VectorSize, VectorCapacity, Gain;
public:
    TVector();
    TVector(int isize, int igain=0, ElType Initvalue=0);
    virtual ~TVector();
    bool setCapacity(int NewCapacity);
    bool setElem(const ElType &Value, int Index);
    bool getElem(ElType &Value, int Index) const;
    bool Insertelem(const ElType &Value, int Index);
    bool Deleteelem(int Index);
    void AddElem(const ElType &Value);
    int Size() const {return VectorSize;}
    int Capacity() const {return VectorCapacity;}
    bool InsertVec(TVector &Subvector, int Index);
    ElType &  operator [](int Index);
    friend ostream & operator <<(ostream &out, const TVector<ElType>
                                        &Vec)
    {
        if(!Vec.pvec) return out;
        for(int i=0;i<Vec.VectorSize; i++)
        {
            out.width(5); out.precision(2); out.fixed;
            out<<Vec.pvec[i];
        }
        return out;
    }
};

template <class ElType>
TVector<ElType>::TVector(int isize, int igain, ElType Initvalue)
{
    Gain=igain;
```

```
    if(isize<=0)
    {
        pvec=0;
        VectorCapacity=VectorSize=0;
        return;
    }
    pvec=new ElType [isize];
    VectorCapacity=VectorSize=isize;
    for(int i=0;i<VectorSize; pvec[i++]=Initvalue);
}
// …
template <class ElType>
bool TVector<ElType>::InsertVec(TVector<ElType> &Subvector, int Index)
{
    if((Index<0)||(Index+Subvector.Size()>VectorSize-1))
     return false;
    if(VectorSize+Subvector.Size()>VectorCapacity)
      setCapacity(VectorSize+Subvector.Size());
    VectorSize+=Subvector.Size();
    for(int i=VectorSize-1;i>=Index; i--) pvec[i] = pvec[i-Subvector.Size()];
/*!!!*/    for(int i=Index; i<Index+Subvector.Size(); i++)
                pvec[i]=Subvector[i-index];
    return true;
}
```

Приклад використання шаблону класу **TVector**:

```
void Print(TVector<double> &Vec)
{
    cout<<"==========================="<<endl;
    cout<<"Vec.Capacity=="<<Vec.Capacity()<<
          " Vec.Size=="<<Vec.Size()<<endl;
    cout<<Vec;
    cout<<endl<<"==========================="<<endl;
}
void main()
{
    TVector <double> MyVec(3,5,-1.1);
    double Val=0.1;
    int N=10;
    for(int i=0;i<N; i++)
    {
        MyVec.AddElem(Val);
        Val+=1;
    }
    Print(MyVec);
}
```

9. ЗВ'ЯЗНІ СПИСКИ, СТЕКИ, ЧЕРГИ Й ІНШІ СТРУКТУРИ ДАНИХ

9.1. Поняття зв'язних списків

Зв'язний список — базова динамічна структура даних, що складається з вузлів, кожний з яких містить як дані, так і одне або два посилання (покажчика) на наступний і/або попередній вузол списку. Принциповою перевагою зв'язного списку перед масивом є структурна гнучкість: порядок елементів зв'язного списку може не збігатися з порядком розташування елементів даних у пам'яті комп'ютера, а порядок обходу списку завжди явно задається його внутрішніми зв'язками. Вузли (англ. node) зв'язного списку називають ще елементами.

Розрізняють такі види зв'язних списків:

- однозв'язний (односпрямований зв'язний список);
- двозв'язний (двоспрямований зв'язний список);
- кільцевий зв'язний список;
- Xor-зв'язний список;
- розгорнутий зв'язний список.

Перші два види списків найбільш прості й поширені, як і кільцеві списки, а останні два більш складні й застосовуються рідше. Поза сумнівом, можна придумати і реалізувати ще більш екзотичні види списків.

Переваги зв'язних списків такі:

- висока ефективність (малий час і вимоги до розміру необхідної пам'яті) додавання й видалення елементів списку;
- кількість вузлів списку обмежена тільки розміром доступної динамічної пам'яті програми;
- на відміну від масивів або покажчиків кількість вузлів списку сугубо динамічна і може легко змінюватися.

Основні недоліки списків:

- на відміну від масивів доступ до елемента списку згідно з його порядковим номером у списку значно менш ефективний і більш складний;
- на поля-покажчики витрачається додаткова пам'ять, яка не потрібна в масивах;
- робота зі списком повільніша, ніж з масивами, оскільки до будь-якого елемента списку можна звернутися, тільки пройшовши всі попередні йому елементи;
- елементи списку зазвичай розташовані в несуміжних комірках пам'яті, що позначається на ефективності її використання.

У роботі [9] можна знайти корисний опис динамічних структур даних та алгоритмів на графах з прикладами на мові Borland Pascal.

9.2. Найпростіший алгоритм реалізації однозв'язного списку

Для того щоб реалізувати таку структуру даних як зв'язний список, зручно розробити спеціальний структурний тип. Припустимо, нам потрібно обробляти довільну, тобто заздалегідь невідому, кількість структур, у яких зберігаються ПІБ й рік народження деяких персон. Тоді можемо оголосити такий структурний тип:

```
struct TNode
{
    char * Name;   // ПІБ
    int BirthYear;  // рік народження
    TNode *Next;  // покажчик на наступний вузол списку
};
```

Поле **Next** використовується для того, щоб зв'язати елементи зв'язного списку – структури типу **TNode** – у єдиний ланцюжок. Робиться це в такий спосіб. Опишемо покажчик **First** на перший елемент списку й виділимо для нього пам'ять:

```
TNode *First;
First=new  TNode;
```

Тепер можна присвоїти яким-небудь чином значення полям вузла, наприклад, присвоюванням:

```
char *s1="Bjarne Stroustrup";
First->Name=new char [strlen(s1)+1];
strcpy(First->Name,s1);
First->BirthYear=1950;
```

Графічно отриманий результат відображено на рис. 9.1.

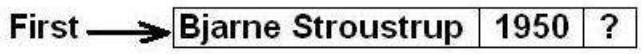

Рис. 9.1. Створення першого елемента списку

Поки вміст покажчика **First.Next** не визначено, він має нульове або випадкове значення, а має вказувати на наступний елемент списку. Для досягнення цієї мети опишемо додатковий покажчик **Temp** і створимо наступний елемент списку за допомогою операторів:

```
TNode * Temp=new TNode;
First->Next=Temp;
```

У результаті виконання цих операторів буде виділена область пам'яті під наступний елемент списку і адреса початку цієї області буде присвоєна полю **Next** вузла **First**, як показано на рис. 9.2.

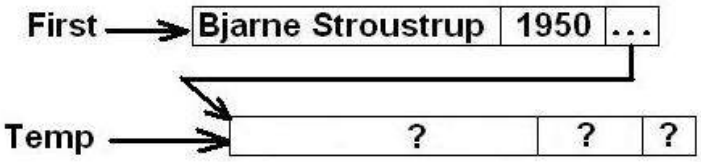

Рис. 9.2. Створення другого елемента списку

Тепер можна присвоїти значення полям **Name** і **BirthYear**:

```
s1="Dennis Ritchie";
Temp->Name=new char [strlen(s1)+1];
strcpy(Temp->Name,s1);
Temp->BirthYear=1941;
```

Графічно результат виконання наведеного коду подано на рис. 9.3.

Рис. 9.3. Список з двох елементів

Подібним чином можна створювати третій елемент списку, четвертий і т. д. Неважко здогадатися, що такий підхід не годиться для створення списку з довільною кількістю вузлів. Формально все правильно, і такі оператори й подібні їм не є помилковими. Але на практиці кількість елементів списку є невизначеною і, мабуть, треба організувати який-небудь підходящий цикл, наприклад такий:

```
char Buf[123];
TNode *First = NULL, *Temp;
while (1)
{
  /*Визначення функції Ukr наведено у додатку*/
 cout << Ukr("Уведи ПІБ відомого комп'ютерного фахівця>");
 cin.getline(Buf, sizeof(Buf), '\n');
 if (Buf[0] == 0) break; /*Завершуємо введення елементів списку,
                  якщо користувач увів порожній рядок*/
 if (!First)
 {
  First = new TNode; Temp = First;
```

```
    }
    else
    {
      Temp->Next = new TNode;
      Temp = Temp->Next;
    }
    Temp->Name = new char[strlen(Buf) + 1];
    strcpy(Temp->Name, Buf);
    cout << Ukr("Уведи його рік народження>");
    cin.getline(Buf, sizeof(Buf), '\n'); // уводимо ціле число як рядок символів
    Temp->BirthYear = atoi(Buf);   /* перетворюємо рядок символів у ціле
                                      число */
    Temp->Next = NULL;
  }
```

Оператор **Temp->Next=NULL** призначено для того, щоб можна було ідентифікувати кінець списку при «проходженні» по його елементах.

Як видно з наведеного фрагмента програми, для роботи зі списком цілком достатньо мати всього два покажчики: покажчик **First** на перший елемент списку і **Temp** – робочий покажчик. Значення покажчика **First**, який указує на перший елемент списку, втрачати не можна в жодному разі, оскільки при цьому буде загублено увесь список.

Само собою зрозуміло, що в «дорослій» програмі треба починати заповнення зв'язного списку прямісінько з наведеного вище «нескінченного» циклу **while(1)**.

Особливість зв'язного списку полягає в тому, що на відміну від масиву доступ до його елементів можна одержати тільки послідовно починаючи з першого елемента й закінчуючи останнім (для однозв'язного списку). Наприклад, виведення змісту елементів списку на монітор може бути виконано у такий спосіб:

```
    /* Виведення член-даних вузлів списку на монітор*/
    int num=0;
    Temp=First;
    while(Temp)
    {
        cout<<++num<<' '<<Temp->Name<<' '<<Temp->BirthYear<<endl;
        Temp=Temp->Next;
}
```

Зрозуміло, що перед завершенням роботи програми виділену для елементів зв'язного списку пам'ять треба звільнити. У нашому прикладі пам'ять виділялась як для елементів (ще говорять *вузлів* – node) зв'язкового списку, так і для даних – рядків символів (ПІБ), що зберігаються у вузлах. Як і будь-яку іншу задачу, цю можна розв'язати багатьма способами, наприклад так:

```
/* Звільнення динамічної пам'яті шляхом почергового видалення всіх
   елементів зв'язного списку, у тому числі рядків символів*/
Temp=First;
while(Temp)
{
        if(Temp->Name) delete Temp->Name;
        Temp=First->Next;
        delete First;
        First=Temp;
}
```

Зазвичай розумно було б реалізувати зв'язний список як клас і звільнити пам'ять у деструкторі.

У наведених фрагментах програмного коду не перевіряється наявність вільної пам'яті при її виділенні, що, безсумнівно, призведе до помилок. Якщо ви не прагнете бути облаяними потенційними користувачами вашої програми, варто було б діяти, наприклад, так:

```
if(!(Temp->Next=new TNode)) Караул!;
if(!(Temp->Name=new char [strlen(Buf)+1])) Кінець світу!;
```

9.3. Операції над елементами зв'язного списку

Однією з переваг зв'язного списку порівняно з масивами (або покажчиками, що те саме) є той факт, що такі операції над вузлами зв'язного списку, як видалення або вставлення вузлів, виконуються значно більш ефективно, ніж над елементами масивів. Разом з тим при цьому потрібно реалізувати більш складні алгоритми. Це ж стосується і сортування елементів списку і, певною мірою, пошуку елементів.

У шаблонових класах **CList** з бібліотеки Microsoft Foundation Class (MFC) і **list** з Standard Template Library (STL) передбачено такі операції над списками:

- додавання в початок списку нового вузла або нового списку;
- додавання в кінець списку нового вузла або нового списку;
- видалення всіх елементів списку;
- видалення першого елемента списку;
- видалення останнього елемента списку;
- одержання елемента із заданим порядковим номером;
- видалення елемента із заданим порядковим номером;
- заміна елемента із заданим порядковим номером на новий;
- вставка елемента перед зазначеним порядковим номером;
- вставка елемента після зазначеного порядкового номера;
- пошук елемента із заданим змістом;
- пошук усіх елементів, що задовольняють деяку умову, і повернення їх у вигляді нового списку;

- видалення всіх елементів, що задовольняють деяку умову;
- присвоювання заданого значення заданій кількості елементів списку;
- обмін місцями двох елементів списку;
- одержання кількості елементів списку.

9.4. Стеки й черги

Створимо файл Lib.h з визначеннями найпростішої реалізації таких загальновизнаних і широко використовуваних структур даних, як стеки й черги, основані на однозв'язному списку.

```
#pragma once
namespace Stack  /*LIFO (last in — first out, останнім прийшов — першим
                                                    вийшов)*/
{
 struct Node
 {
  int Num;
  Node *Next;
 };
 void Push(Node *&Stc, int Val) // розміщення елемента  у стеку
 {
  if(!Stc) // стек порожній
  {
     Stc=new Node; Stc->Num=Val; Stc->Next=NULL; return;
  }
  Node * Temp=new Node;
  Temp->Next=Stc; Temp->Num=Val; Stc=Temp;
 }
 bool Pop(Node *&Stc, int &Val) // вилучення елемента зі стеку
 {
  if(!Stc) return false; // стек порожній
  Node *Temp=Stc->Next; Val=Stc->Num;
  delete Stc;
  Stc=Temp;
  return true;
 }
}

namespace Queue  /*FIFO (first in — first out, першим прийшов — першим
вийшов)*/
{
 struct Node
 {
    int Num;
    Node *Next;
```

```
 };
 void Push(Node *&Kju, int Val)
 {
  if(!Kju)
  {
    Kju=new Node; Kju->Num=Val; Kju->Next=NULL; return;
  }
  Node * Last = Kju;
  while(Last->Next) Last = Last->Next;
  Node * Temp = new Node; Temp->Next=NULL; Temp->Num=Val;
  Last->Next=Temp;
 }
 bool Pop(Node *&Kju, int &Val)
 {
    if(!Kju) return false;
    Node *Temp=Kju->Next; Val=Kju->Num;
    delete Kju;
    Kju=Temp;
    return true;
 }
}
```

Проілюструємо використання цих структур даних такими функціями:

```
//////////////// Використання стеку ////////////////
void StackDemo()
{
 using namespace Stack;
 Node * Mystc=NULL;
 // Поміщаємо у стек числа 1  2  4  8  16
 for (int i = 0; i<5; i++)
  Push(Mystc,1<<i);
 // виведення елементів списку
 Node * Temp=Mystc;
 cout<<"List: ";
 while(Temp)
 {
  cout<<Temp->Num<<' ';
  Temp=Temp->Next;
 }// 16  8  4  2  1
 cout<<endl<<"==============="<<endl;
 // Вилучаємо елементи зі стеку
 while(Mystc)
 {
    int n;
    if(Pop(Mystc,n)) cout<<n<<' '; else break;
 } // 16  8  4  2  1
 cout<<endl;
```

```
  Push(Mystc,-1); Push(Mystc,-11); //-1 -11
  int n;
  Pop(Mystc,n); cout<<n<<' ';  //-11
  Pop(Mystc,n); cout<<n<<endl; //-1
}

//////////////// Використання черги ////////////////
void QueueDemo()
{
  using namespace Queue;
  Node * Mykju=NULL;
  // Поміщаємо в чергу числа 1  2  4  8  16
  for (int i = 0; i<5; i++)
    Push(Mykju,1<<i);
  // виведення елементів списку
  Node * Temp=Mykju;
  cout<<"List: ";
  while(Temp)
  {
     cout<<Temp->Num<<' ';
     Temp=Temp->Next;
  } // 1  2  4  8  16
  cout<<endl<<"==============="<<endl;

  // Вилучаємо елементи із черги
  while(Mykju)
  {
     int n;
     if(Pop(Mykju,n)) cout<<n<<' '; else break;
  } // 1  2  4  8  16
  cout<<endl;

  Push(Mykju,-1); Push(Mykju,-11); //-1 -11
  int n;
  Pop(Mykju,n); cout<<n<<' ';  //-1
  Pop(Mykju,n); cout<<n<<endl; //-11
}
```

Слід звернути особливу увагу на послідовність вилучення елементів зі стеку та черги.

9.5. Двійкові дерева

Двійковим (бінарним) деревом називають таку ієрархічну структуру даних, у якій кожний вузол має не більш двох нащадків (рис. 9.4). Двійкове

дерево доцільне будувати на основі зв'язного списку, кожний вузол якого містить дані й два покажчики: на «лівого» і «правого» нащадків, наприклад:

```
struct Node
{
  int Num;
  Node *Left, *Right;
};
```

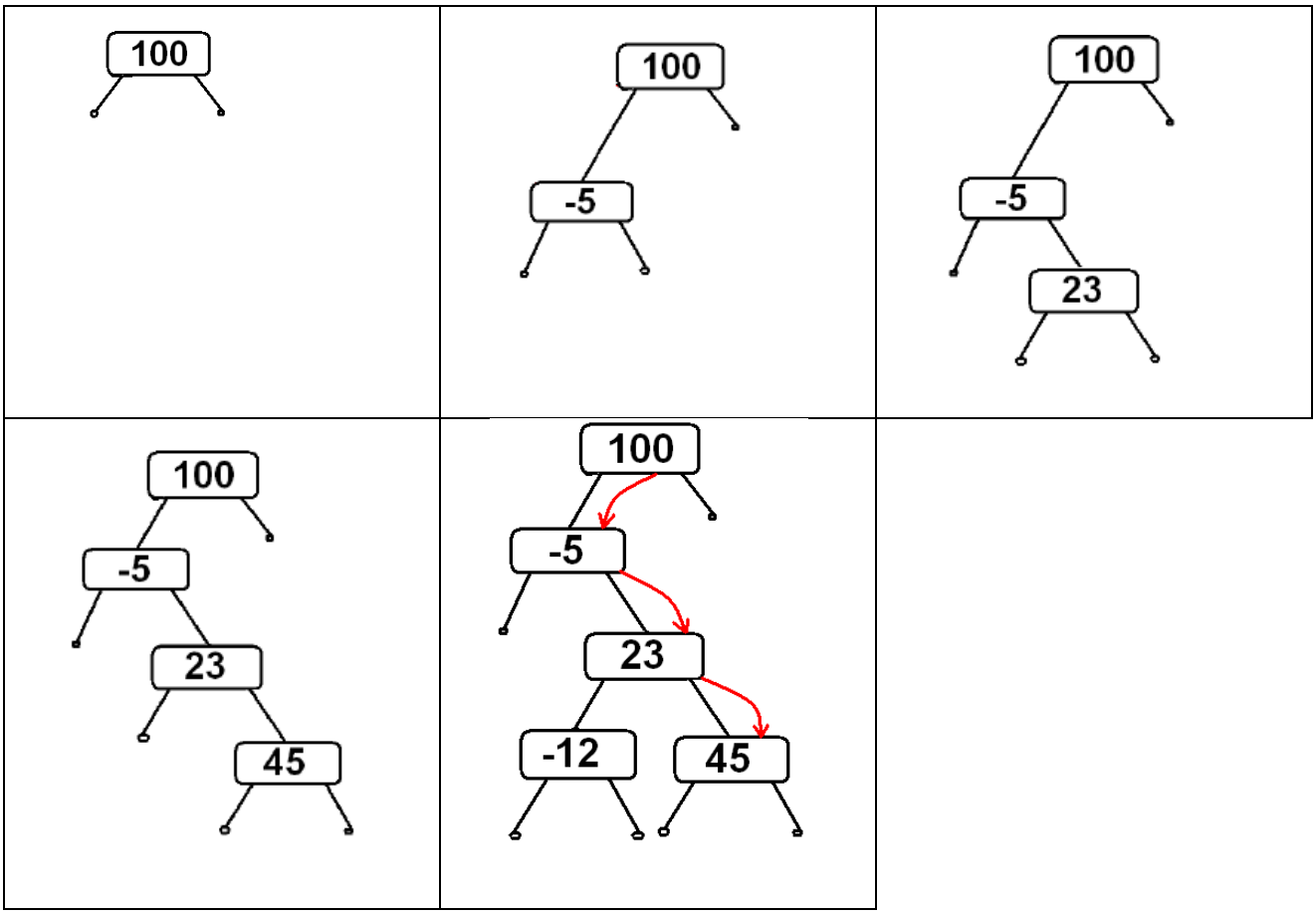

Рис. 9.4. Послідовність додавання значень у двійкове дерево

Перший, самий верхній вузол дерева називають батьківським вузлом або коренем, а гілки– лівим і правим нащадками. Двійкове дерево можна подати як звичайне дерево, стовбур якого перебуває вгорі, а гілки, відповідно, внизу. Крім того, кожна гілка такого дерева може містити не більш двох піддерев.

Кожний вузол дерева має містити деяке значення, яке називається ключем. Нащадок, який містить менше значення ключа, розташовується в дереві зліва, а більше – відповідно справа. Передбачається, що однакових значень ключів у дереві не існує, тобто всі ключі мають оригінальні значення.

Будь-яке двійкове дерево або є порожнім, або складається з даних і двох піддерев, кожне з яких може бути порожнім. Очевидним, але важливим

для розуміння фактом є те, що кожне піддерево у свою чергу теж є деревом. Якщо у деякого вузла кожне із піддерев порожнє, то він називається листовим вузлом (листовою вершиною) або термінальним елементом.

Наведемо тексти функцій, призначених для опрацювання двійкового дерева, у вузлах якого зберігається деяке цілочислове значення (ключ) і, звісно, два покажчики:

```
namespace BinaryTree  /**/
{
 struct Node
 {
   int Num;
   Node *Left, *Right;
 };
 void Push(Node *&BTree, int Val)
 {
  if (BTree==NULL) //Якщо гілки  ще немає
  {
   BTree = new Node; // Виділяємо пам'ять для першого вузла
   BTree->Num = Val; //Запам'ятовуємо значення
   TRACE("BTree = %d Val = %d\n",BTree,BTree->Num);
   BTree->Left = BTree->Right = NULL;
   return;
  }
 //Дерево є
 if (Val>BTree->Num)
 {
   Push(BTree->Right,Val); /*Якщо значення Val більше, ніж значення Num
          у поточному вузлі, поміщаємо  його у праву гілку дерева*/
   TRACE("BTree = %d BTree->Left=%d BTree->Right=%d Val = %d\n",
              BTree,BTree->Left, BTree->Right,  BTree->Num);
 }
 else
 {
   Push(BTree->Left,Val); //Інакше поміщаємо його у ліву  гілку дерева
   TRACE("BTree = %d BTree->Left=%d BTree->Right=%d  Val = %d\n",
              BTree, BTree->Left,  BTree->Right, BTree->Num);
  }
}
void TreePrint(Node *&BTree, char * LR)
{
 if (BTree==NULL) return; //Якщо дерево порожнє
 else
 {
  TreePrint(BTree->Left,"Left");/* За допомогою рекурсії
                 відвідуємо ліве піддерево*/
  cout<<LR<<':'<<BTree->Num<<endl; // Виводимо елемент
 }
```

```
    TreePrint(BTree->Right,"Right"); /* За допомогою рекурсії
                    відвідуємо праве піддерево*/
   }
  }
```

Ілюстрація використання наведених вище функцій для опрацювання двійкового дерева:

```
  void BinaryTreeDemo()
  {
    using namespace Binarytree;
    Node * BTree=NULL;
    int Numbers[]={100,-5,23,45,-12};
    int n=sizeof(Numbers)/sizeof(Numbers[0]);
    cout<<Ukr("Розміщуємо дані у дереві:");
    for(int i=0;i<n;i++)
      cout<<Numbers[i]<<"  "; //100  -5  23  45  -12
    cout<<endl<<"_____________________"<<endl;

    for (int i = 0; i<n; i++)
      Push(BTree,Numbers[i]);
    TreePrint(BTree,"Right");
   //Left:-12
   //Left:-5
   //Right:23
   //Right:45
   //Right:100
  }
```

Функція пошуку значення у дереві:

```
  bool TreeFind(Node *&BTree, int Fval)
  {
    if (!BTree) return false; //Якщо дерево порожнє
    if(BTree->Num==Fval) return true;
    if(Fval<BTree->Num) Treefind(BTree->Left,Fval);
    else Treefind(BTree->Right,Fval);
  }
```

Використання функції пошуку Treefind():

```
  void BinaryTreeDemo2()
  {
    using namespace BinaryTree;
    Node * BTree=NULL;
    int Numbers[]={100,-5,23,45,-12};
    int n=sizeof(Numbers)/sizeof(Numbers[0]);
    cout<<" Поміщаємо дані у дерево:";
    for(int i=0;i<n;i++)
      cout<<Numbers[i]<<"  "; //100  -5  23  45  -12
```

```
   cout<<endl<<"_____________________"<<endl;

   for (int i = 0; i<n; i++)
     Push(BTree,Numbers[i]);
   TreePrint(BTree,"Right");
//Left:-12
//Left:-5
//Right:23
//Right:45
//Right:100
  int Numbers2[]={300,100,-5,23,400,45,-12,500};
  n=sizeof(Numbers2)/sizeof(Numbers2[0]);
  for(int i=0;i<n;i++)
    cout<<"Значення "<<Numbers2[i]<<(Treefind(BTree, Numbers2[i])?
        " - знайдено":" - не знайдено ")<<endl;
// Значення 300 - не знайдено
// Значення 100 - знайдено
// Значення -5 - знайдено
// Значення 23 - знайдено
// Значення 400 - не знайдено
// Значення 45 - знайдено
// Значення -12 - знайдено
// Значення 500 - не знайдено
}
```

9.6. Реалізація зв'язного списку як класу

Уявіть собі, що треба розробити програму, яка буде використовувати кілька зв'язних списків, у кожному з яких будуть зберігатися різні дані як за типами, так і за кількістю. Її можна написати, але це буде складна й малокорисна робота. Природнім виходом є реалізація алгоритмів роботи зі зв'язним списком як класу або класів.

Насамперед створимо клас **CNode** – вузол зв'язного списку:

```
class CNode /* Вузол зв'язного списку */
{
    char * Name;
    int BirthYear;
    CNode *Next;
  public:
    CNode();
    ~CNode();
    void setNextNode(CNode * nextnode);
    CNode * getNextNode();
    void setName(char * newName);
    char* getName();
```

```
    void setBirthYear(int newBirthYear);
    int getBirthYear();
    void operator =(const CNode & op);
};
```

Екземпляр такого класу являє собою один-єдиний вузол списку й усі його член-функції призначено тільки для присвоєння і одержання значень член-даних вузла, тобто так звані аксесори (accessors). Реалізація цих функцій, а також конструктора й деструктора така:

```
#include "StdAfx.h"
#include "MyList.h"
/* Усі член-функції класу CNode оперують тільки з одним вузлом списку
   і нічого не знають про інші вузли списку */
CNode::CNode():Name(0), Next(0)
{
}
CNode::~CNode()
{
    if(Name) delete Name;
}
/*CNode::setNextNode - установлює нове значення для покажчика Next
                       вузла списку  */
void CNode::setNextNode(CNode * nextnode)
{
    Next=nextnode;
}
/*CNode::getNextNode – повертає значення Next вузла списку*/
CNode * CNode::getNextNode()
{
    return Next;
}
/*CNode::setName – установлює нове, можливо, первісне,  значення
                   компонента Name вузла списку */
void CNode::setName(char * newName)
{
    if(this->Name) delete this->Name;
    int len=strlen(newName)+1;
    this->Name=new char [len];
    strcpy_s(this->Name,len,newName);
}
char* CNode::getName()
{
    return Name;
}
/*CNode::BirthYear - установлює нове,  можливо, первісне,
      значення компонента BirthYear вузла списку*/
void CNode::setBirthYear(int newBirthYear)
```

```
{
    BirthYear=newBirthYear;
}
int CNode::getBirthYear()
{
    return BirthYear;
}
void CNode::operator =(const CNode & op)
{
    BirthYear=op.BirthYear;
    if(this->Name) delete this->Name;
    int len=strlen(op.Name)+1;
    this->Name=new char [len];
    strcpy_s(this->Name,len,op.Name);
}
```

Код цього класу не має ніяких функцій, призначених для створення вузлів списку, переміщення за списком і т. д. Такі функції мають бути реалізовані в іншому класі – **CMylist**, який буде оперувати з покажчиками на вузли списку.

```
class CMyList
{
    CNode * First, * Current; /*Покажчики на перший та поточний вузли
                                списку*/
    int NodeCount; /*Кількість вузлів списку*/
    CNode * GetLastNode();
public:
    CMyList();
    ~CMyList();
    void AddNode(const CNode &node);
    CNode * GetCurNode();
    bool MoveToNextNode();
    bool MoveToFirstNode();
};
```

Член-функції класу **CMylist** оперують з усіма вузлами списку й дозволяють додавати нові вузли списку в його кінець, переміщатися по вузлах списку в напрямку від його початку до кінця, одержувати дані поточного вузла списку. Реалізацію член-функцій цього класу наведено нижче:

```
/*CMyList::CMyList - ініціалізуємо порожній список*/
CMyList::CMyList()
{
    First=Current=NULL; NodeCount=0;
}
```

```
/*CMyList::~CMyList - звільняє пам'ять, виділену для списку в цілому.
Пам'ять, яка, можливо, була виділена для даних вузла списку, звільняється
деструктором класу вузла списку  */
CMyList::~CMyList()
{
    /* звільнення пам'яті шляхом почергового  видалення усіх вузлів зв'язного
       списку */
    Current=First;
    while(Current)
    {
        Current=First->getNextNode();
        delete First;
        First=Current;
    }
}
/*CMyList::AddNode - найважливіша функція, яка дозволяє додавати
                    у кінець списку новий вузол */
void CMyList::AddNode(const CNode &node)
{
    if(!First) Current=First=new CNode; /* створюємо перший елемент
                                           списку */
    else /* створюємо черговий елемент списку */
    {
        Current=GetLastNode();
        Current->setNextNode(new CNode);
        Current=Current->getNextNode();
    }
    *Current=node; /* додаємо у кінець списку новий вузол */
    ++NodeCount;   /* збільшуємо кількість вузлів списку */
}
/*GetCurNode - повертає значення покажчика на поточний вузол списку,
        який може використовуватися для доступу до даних вузла списку */
CNode * CMyList::GetCurNode()
{
    return Current;
}

/*CMyList::MoveToNextNode - переміщує поточний покажчик Current
   на наступний вузол. Якщо поточний покажчик указує на  останній вузол
   списку – повертає значення false*/
bool CMyList::MoveToNextNode()
{
    if(Current->getNextNode())
    {
        Current=Current->getNextNode();
        return true;
    }
```

```
    Current=NULL; return false;
}
/*CMyList::MoveToFirstNode – переміщує поточний покажчик на перший
                                    вузол списку*/
bool CMyList::MoveToFirstNode()
{
    if(First)
    {
        Current=First; return true;
    }
    else
    {
        Current=NULL; return false;
    }
}

/*CMyList::GetLastNode – повертає покажчик на останній вузол списку
        і при цьому не змінює значення поточного покажчика Current
*/
CNode * CMyList::GetLastNode()
{
    if(!First) return NULL; /*якщо список порожній*/
    if(!(Current->getNextNode())) return Current; /*якщо Current
        вказує на останній вузол списку, то повертаємо Current*/
  /* Інакше переміщуємося за списком від першого до останнього вузла */
    CNode * Temp=First;
    for(int i=0;i<NodeCount-1;i++) Temp=Temp->getNextNode();
    return Temp;
}
```

У наведеній нижче функції **DemoClasscMyList** ілюструється використання класу **CMyList** для створення списку, що містить довільну кількість вузлів.

```
void PrintList(CMyList &mylist, int code)
{
    /* Виведення змісту списку на монітор */
    int num=0;
    if(!(mylist.MoveToFirstNode()))
        cout<< Ukr ("Список порожній!")<<endl;
    else
    {
        cout<<"============================"<<endl;
        cout<<endl<<Ukr("Зміст списку:")<<endl;
    }
    while(mylist.GetCurNode())
    {
        switch(code)
```

```
            {
              case 1251: cout<<++num<<' '<<Ukr(mylist.GetCurNode()
                         ->getName());
                         break;
              case 866:  cout<<++num<<' '<<mylist.GetCurNode()->getName();
                         break;
            }
            cout<<' '<<mylist.GetCurNode()->getBirthYear()<<endl;
        /* альтернативний спосіб виведення даних, що зберігаються у вузлах
           списку:
            tmp_node=*mylist.GetCurNode();
            cout<<++num<<' '<<tmp_node.getName()<<
                  ' '<<tmp_node.getBirthYear()<<endl;
        */
            if(!(mylist.MoveToNextNode())) break;
        }
        cout<<"============================"<<endl;
    }

    void DemoClassCMyList()
    {
        CMyList mylist;
        char s1[123], s2[10];
        CNode tmp_node; // "робочий" вузол
        /* У циклі while(1) створюється довільна кількість вузлів списку*/
        while(1)
        {
            cout<<Ukr("\nУведи ПІБ відомого комп'ютерного фахівця "
                "[порожній рядок - вихід])>");
            cin.getline(s1,sizeof(s1),'\n');
            if(s1[0]==0) break; /*Закінчуємо уведення вузлів
                  списку, якщо користувач уведе порожній рядок */
            cout<<Ukr("Уведи його рік народження>");
            cin.getline(s2,sizeof(s2),'\n'); /*Уводимо
                                        число як рядок символів*/
            int BirthYear=atoi(s2); /* перетворюємо рядок
                                      символів у ціле число*/
            tmp_node.setName(s1);
            tmp_node.setBirthYear(BirthYear);
            mylist.AddNode(tmp_node);
        }
        PrintList(mylist,866);
    }
```

Не забудьте додати функцію **Ukr**.

Зверніть увагу на те, що компоненти класу **CMylist** незалежать від даних, які зберігаються у вузлах списку **CNode**. Якщо буде потрібно

використовувати **CMylist** для роботи з іншими вузлами (тобто з вузлами, що зберігають інші дані), то досить буде змінити або написати новий клас **CNode**.

Максимального ж ефекту можна добитися, якщо перетворити клас **CMylist** у шаблонний.

Додаток

ПЕРЕКОДУВАННЯ СИМВОЛІВ КИРИЛИЦІ

Оброблення рядків символів у мові C++ є одним із самих складних завдань. Це пов'язано з низкою таких складних обставин:

- по-перше, наявність численних кодувань: **ASCII** (7 або 8 бітів), **ANSI** (8 бітів), **Multibyte Character Sets** (**Mbcss** – 16 бітів), **Unicode** (**UTF-7, UTF-8,UTF-16,UTF-32, UTF-32 Big-Endian**);
- по-друге, наявність великої кількості різних способів визначення й оброблення рядків символів: бібліотечних функцій і цілих класів.

Дуже корисний і докладний опис особливостей кодування символів можна знайти в роботі [9, розд. 2], а тут увагу буде приділено тільки деяким окремим проблемам оброблення символів, що виникають при розробленні консольних додатків Windows. У додатках із графічним інтерфейсом користувача проблем зазвичай не виникає завдяки гарній вбудованій підтримці кодування **Unicode**, зазвичай **UTF-16**.

Інтегроване середовище MVS за замовчуванням зберігає файли програми в кодуванні **Cyrillic Windows 1251** (рис. Д.1), хоча можливо вибрати й інше кодування. Для цього необхідно в діалоговому вікні **«Save File As»** викликати вікно **«Advanced Save Options»** за допомогою команди **«Save with Encoding»**, яка у свою чергу викликається за допомогою трикутника, розташованого на кнопці **Save** (див. рис. Д.1).

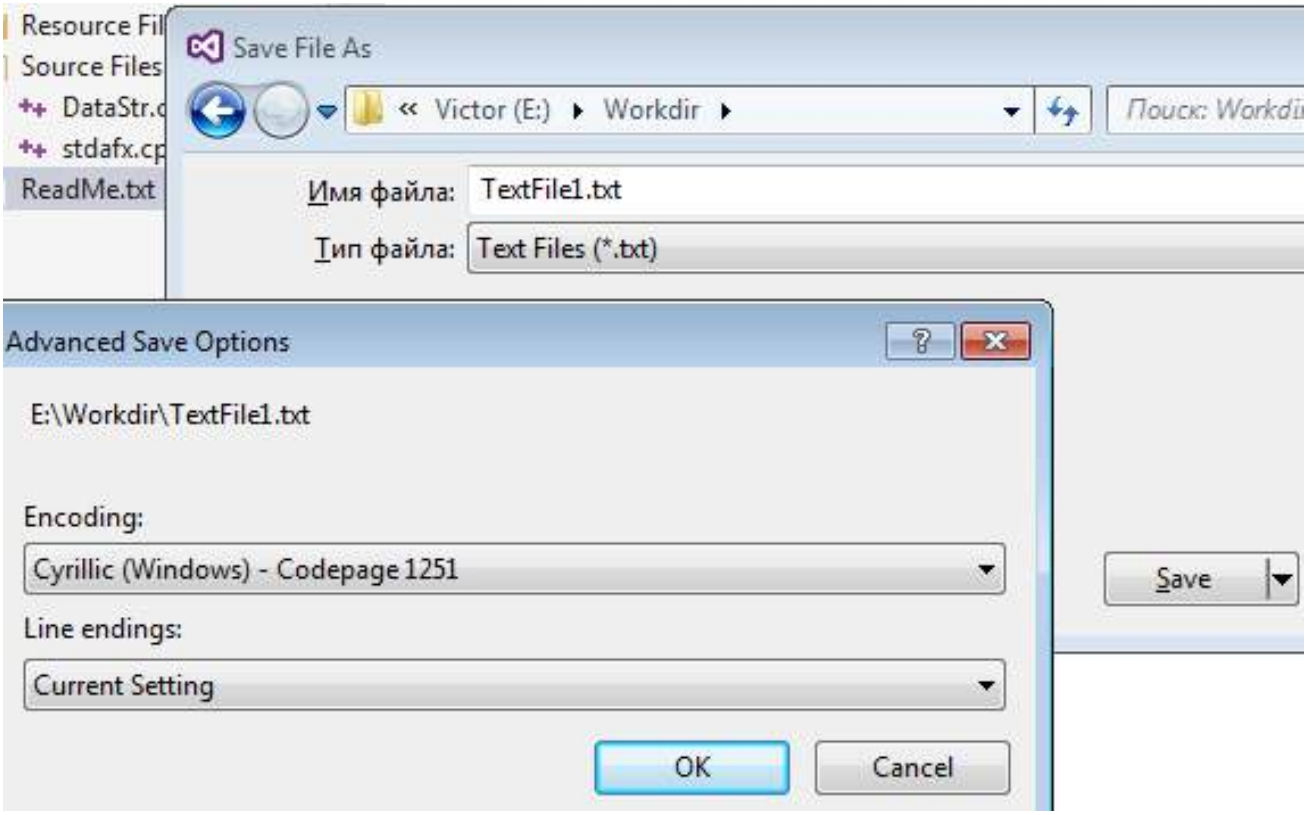

Рис. Д.1. Вибір кодування для файла, що зберігається

В інтегрованому середовищі MVS можна вибрати (або довідатися) кодування символів для додатка, якщо викликати вікно властивостей проекту, вибрати вкладку **Configuration Properties➔General** і бажане кодування в списку **Character Set**. За замовчуванням в MVS вибрано кодування **Unicode Character Set**, у якому для символу виділяється 2 байти. Вибір значення **Not Set** означає вибір кодування **ANSI (Windows-1252)**, у якому розмір символу становить 1 байт. Втім вибір того або іншого кодування впливає тільки на тип символів, які використовується деякими функціями або класами.

Для консольних додатків актуальним є той факт, що при виведенні символів кирилиці у вікно консольного додатка потрібно використовувати кодування **866 (Cyrillic DOS)**, а не яке-небудь інше. Це означає, що виконання оператора **cout<<"Привет!"** призведе до виведення зовсім інших «незрозумілих» символів: **╧Ёштхє!**

Реалізувати коректне виведення символів кирилиці можна за допомогою функцій **setlocale(), CharToOemA(), Ukr()**.

Для виведення символів російськомовного алфавіту досить викликати один раз на початку виконання програми функцію **setlocale()**, наприклад:

```
void main()
{
```

```
    setlocale(LC_ALL,"rus"); // або setlocale(LC_ALL,"");
    cout<<"Привет!";
    //…
}
```

Для виклику функції **CharToOemA()** зручно написати таку просту функцію-оболонку:

```
/*Rus – виконує перетворення символів кирилиці  з кодування
      1251 (Cyrillic Windows ) у кодування 866 (Cyrillic DOS)*/
char *Rus(char Str[])
{
    static char Buf[1024];
    CharToOemA (Str,Buf); return Buf;
}
```

Як видно з тексту функції **Rus()**, кількість символів у рядку має бути не більшою 1023, але це не проблема. Використання цієї функції може мати такий вигляд:

```
void main()
{
    cout<<Rus("Привет!");
SomeFunction(Rus("Слава Украине!"));
    //…
}
```

Перші два способи не дозволяють реалізувати виведення символів українського алфавіту і тому розробимо власну функцію **Ukr()**:

```
/*Ukr - Виконує перетворення символів кирилиці з кодування
      1251 (Cyrillic (Windows) ) у кодування 866 (Cyrillic DOS).
    Підтримує символи як російської, так і української мов */
char * Ukr(char Str[])
{
    static char Buf[1024];
    strcpy_s(Buf,strlen(Str)+1,Str);
    /* у наступному циклі замінюємо коди російських букв на коди
      українських,  не змінюючи кодів інших символів */
    #pragma warning (disable: 4309)
    for(unsigned int i=0;i<strlen(Buf);i++)
    switch (Buf[i])
    {
        case -78 : Buf[i]=73;  break; // I
        case -77 : Buf[i]=105; break; // i
        case -86 : Buf[i]=242; break; // Є
        case -70 : Buf[i]=243; break; // є
        case -81 : Buf[i]=244; break; // Ї
        case -65 : Buf[i]=245; break; // ї
        case -88 : Buf[i]=240; break; // Е
```

```
            case -72 : Buf[i]=241; break; // е
            default :
            {
              if((Buf[i]<=-17)&&(Buf[i]>=-64)) Buf[i]-=64; //А..Я,а..п
              else
                if((Buf[i]<=-1)&&(Buf[i]>=-16)) Buf[i]-=16; //р..я
            }
        }
        #pragma warning (default: 4309)
        return Buf;
    }
```

Використання цієї функції просте і має вигляд:

```
    void main()
    {
        cout<<Ukr("Привіт, їжаче!");
        //…
    }
```

У деяких випадках корисною може виявитися функція **WideCharToMultiByte()** або зворотна їй за призначенням функція **MultiByteToWideChar()**. Для функції **WideCharToMultiByte()** можна створити таку зручну функцію-оболонку:

```
    /*GetANSI - перетворює символи кирилиці, що задані у форматі Unicode,
                  у кодування MS DOS:  wchar_t->char *  */
    char * GetANSI(Cstring str)
    {
        static char Buf[1024];
        WideCharToMultiByte (866,WC_NO_BEST_FIT_CHARS,str,-1, Buf,
              sizeof(Buf),0,0);
        return Buf;
    }
```

Рядки типу **CString** визначені в заголовному файлі **atlstr.h**. Звісно, що замість кодової сторінки 866 можна використати будь-яку іншу.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Страуструп, Б. Язык программирования C++ Б. Страуструп; пер. с англ. С. Анисимова и М. Кононова под редакцией Ф. Андреева и А. Ушакова. – Спб. : М. : "Невский Диалект" – "Изд-во БИНОМ", 1999. – 991 с.
2. Прата, С. Язык программирования C++. Лекции и упражнения. 6-е изд. / С. Прата : пер. с англ. Ю.И. Корниенко, А.А. Моргунова под редакцией Ю.Н. Артеменко – М.: ООО "И.Д. Вильямс", 2012. – 1248 с.
3. Шилдт, Г. Искусство программирования C++ / Г. Шилдт : пер. с англ. – СПб.: БХВ-Петербург, 2005. – 496 с.
4. Кравець, П.О. Об'єктно-орієнтоване програмування: навч. посібник/ П.О.Кравець. - Львів: Видавництво Львівської політехніки, 2012. – 624 с.
5. Хортон, А. Visual C++ 2010: полный курс.: / А. Хортон пер. с англ. В.А. Коваленко – М.: ТОВ «И.Д. Вильямс», 2011.– 1216 с.
6. Дейтел Х.М. Как программировать на C++: 5-е изд. / Х.М. Дейтел, П.Дж. Дейтел : пер. с англ. под редакцией В.В. Тимофеева.5-е изд. — М.: ООО «Бином -Пресс», 2008 р. – 1456 с.
7. News, Status & Discussion about Standard C++ – [Електронний ресурс] – Режим доступу: https://isocpp.org/ (20.09.2019)
8. Рихтер, Дж. Windows via C/C++. Программирование на языке Visual C++.: / Дж. Рихтер, К. Назар.: пер. с англ. – Изд-во: Питер, Русская Редакция, 2009. – 896 с.

   Ковалюк, Т.В. Основи програмування : монографія /Т.В. Ковалюк. – Київ.: Видавнича група BHV, 2005. – 384 с.

Навчальне видання

Овсяннік Віктор Миколайович
Погудіна Ольга Костянтинівна

МОВА С++ НЕ ДЛЯ ЧАЙНИКІВ

Редактор Н. М. Сікульська

Зв. план, 2020
Підписано до видання 16.12.2020
Ум. друк. арк. 7,2. Обл.-вид. арк.8,06. Електронний ресурс

Видавець і виготовлювач
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17
http:// www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@khai.edu

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30.03.2001