[КЛАСИ ТА ОБ'ЄКТИ 1](#_Toc459789163)

[Визначення класу 1](#_Toc459789164)

[**Оголошення класу** 1](#_Toc459789165)

[**Реалізація класу** 5](#_Toc459789166)

[Створення об'єктів 8](#_Toc459789167)

[Реалізація доступу до членів класу 10](#_Toc459789168)

[Структури і об'єднання як різновиди класів 12](#_Toc459789169)

[**Класи і структури** 12](#_Toc459789170)

[**Об'єднання та класи** 13](#_Toc459789171)

[Конструктори і деструктори 14](#_Toc459789172)

[**Поняття конструктора** 14](#_Toc459789173)

[**Конструктор за замовчуванням** 16](#_Toc459789174)

[**Конструктор з параметрами** 17](#_Toc459789175)

[**Конструктор копіювання** 18](#_Toc459789176)

[**Конструктор перетворення** 19](#_Toc459789177)

[**Список ініціалізації** 20](#_Toc459789178)

[**Перевантаження конструкторів** 21](#_Toc459789179)

[**Поняття деструктора** 23](#_Toc459789180)

[Динамічні об'єкти 24](#_Toc459789181)

[**Класи з динамічними атрибутами** 24](#_Toc459789182)

[**Створення, ініціалізація і знищення динамічних об'єктів** 26](#_Toc459789183)

[Особливості роботи з об'єктами 27](#_Toc459789184)

[**Присвоєння об'єктів** 27](#_Toc459789185)

[**Передача об'єктів функціям** 30](#_Toc459789186)

[**Повернення об'єктів функціями** 34](#_Toc459789187)

[**Неявний параметр this** 37](#_Toc459789188)

[**Особливості використання конструкторів і деструкторів** 38](#_Toc459789189)

[Mасиви об'єктів 43](#_Toc459789190)

[**Створення та ініціалізація масивів об'єктів** 43](#_Toc459789191)

[**Динамічні масиви об'єктів і масиви покажчиків на об'єкти** 46](#_Toc459789192)

[**Приклади побудови статичних та динамічних об'єктів і структур даних** 48](#_Toc459789193)

[Константні члени класу і константні об'єкти 51](#_Toc459789194)

[**Константні атрибути** 51](#_Toc459789195)

[**Константні методи** 51](#_Toc459789196)

[**Константні об'єкти** 53](#_Toc459789197)

[Статичні члени класу 54](#_Toc459789198)

[**Статичні атрибути** 54](#_Toc459789199)

[**Статичні методи** 59](#_Toc459789200)

[Резюме 61](#_Toc459789201)

# КЛАСИ ТА ОБ'ЄКТИ

## Визначення класу

### **Оголошення класу**

Клас є абстрактним типом даних, що визначається користувачем. Як і будь-який тип даних, клас характеризується множиною даних, які описують внутрішню структуру інформації, що зберігається в об'єкті класу, і множиною програмних кодів, які описують алгоритми обробки цієї інформації.

Дані у класі оголошують у вигляді змінних, а програмні коди оформляють у вигляді функцій. Змінні та функції класу називають його *членами.* Відповідно до стандартної термінології ООП, для змінної, оголошеної в класі, часто використовують термін *атрибут класу*, а для функцій-членів класу - термін *метод класу.*

Оголошення класу являє собою узагальнення комбінованого типу даних і складається з двох частин: заголовка, що включає ключове слово **class**, і тіла, вкладеного у фігурні дужки:

**class** ім'я\_класу

{ тіло\_класу;

};

Тут *ім'я\_класу* означає ім'я "класового" типу. Після опису класу саме його використовують для побудови об'єктів цього класу. Усередині тіла класу оголошуються його члени: атрибути і методи.

Тіло класу визначає окрему область видимості. Оголошення членів всередині тіла класу поміщає їх імена в область видимості класу. Тобто, ідентифікатори членів класу є локальними по відношенню до даного класу.

Атрибути класу оголошуються майже так само, як і поля структури (запису):

тип атрибут.

Наприклад,

**class** TPoint *// інформація про точку*

{ **int** x, y; *// кординати*

**int** color; *// колір*

*…*

};

Атрибути класу можуть бути будь-якого типу, у тому числі і класового. Оскільки клас не вважається визначеним, допоки не закінчилося його тіло, то в ньому не може бути атрибутів його власного типу. Однак клас вважається оголошеним, як тільки розпізнаний його заголовок, тому в ньому допустимі члени, що є посиланнями або покажчиками на його тип. Іншими словами, атрибути класу не можуть бути представниками класу, що визначається, але можуть бути покажчиками або посиланнями на цей клас. Наприклад,

**class** TPoint

{ **int** x, y;

**int** color;

TPoint \*next; *// покажчик на наступну точку*

…

};

Слід відзначити, що структури і масиви “у чистому вигляді” в описи об'єктів, як правило, не включають. Справа в тому, що кожне дане, якщо воно включене до опису об'єкта, повинне оброблятися тільки сукупністю специфічних методів. Це означає, що атрибут-структуру можна і потрібно замінити атрибутом-об'єктом.

Класи можуть мати динамічні поля. Однак, динамічно виділяти пам'ять під прості змінні, що включаються в об'єкт, невигідно, оскільки пам'ять витрачається не тільки для розміщення самої змінної, а й під додаткову інформацію про виділення пам'яті. Тому динамічний розподіл пам'яті зазвичай використовують при роботі зі складними структурами даних, особливо, якщо розмір майбутнього атрибута заздалегідь не відомий (наприклад, залежить від вихідних даних і визначається в процесі обчислень).

Ініціалізація атрибутів при оголошенні класу не допускається, оскільки в момент оголошення класу пам'ять для розміщення його полів ще не виділена. Виділення пам'яті здійснюється не для класу, а для об'єктів цього класу, тому можливість ініціалізації полів з'являється тільки під час або після оголошення об'єкта конкретного класу.

Оголошення методів класу зазвичай представляється прототипами функцій-членів класу, наприклад,

**class** TPoint *// інформація про точку*

{ **int** x, y; *// кординати*

**int** color; *// колір*

…

**void** SetXY (**int**, **int**); *// введення кординат*

**void** Set\_Сolor(**int**); *// введення кольору*

};

Методи класу мають доступ до будь-яких інших методів і до будь-яких атрибутів даного класу. Однак, відповідно до принципа інкапсуляції, доступ до членів класу (атрибутів і методів) з об'єктів інших класів може бути обмеженим. Це забезпечуються введенням в опис класу областей доступу:

* **public***(відкритий/загальний)* – доступ з поза класу не обмежений;
* **protected***(захищений)* – доступ можливий тільки власним методам і методам похідних класів;
* **private***(закритий/приватний)* – доступ можливий тільки власним методам.

Члени класу, що знаходяться в закритій області (**private**), доступні тільки іншим членам свого ж класу і недоступні для використання з боку зовнішнього коду. Цей різновид доступу прийнято у класі за замовчуванням. Навпаки, члени класу, що знаходяться у відкритій секції (**public**), є доступними поза класом, тобто до них можна звертатися з інших класів або зовнішніх функцій (які не належать жодному класу). Вміст розділу **public** становить загальнодоступний інтерфейс класу. До захищених членів (**protected**) мають доступ лише деякі з об'єктів. Тому захищені елементи займають проміжне становище між відкритими і приватними.

При описі класу кожен член класу поміщається в одну з перерахованих вище областей доступу вказанням відповідного *специфікатора доступу*:

**class** ім'я\_класу

{ [ **специфікатор\_доступу\_1** ] : визначення\_атрибутів;

визначення \_методів;

**специфікатор\_доступу\_2**  : визначення \_атрибутів;

визначення \_методів;

…

**специфікатор\_доступу\_n** : визначення \_атрибутів;

визначення \_методів;

};

***Специфікатори доступу*** – це спеціальні синтаксичні конструкції, що явно задають область видимості кожного члена класу. Порядок вказання областей доступу і їх кількість в класі - довільні. Специфікатор, який визначає першу область доступу, може бути відсутнім. За замовчуванням ця область вважається закритою (**private**).

Дія будь-якого специфікатора поширюється до наступного специфікатора або до кінця класу. Можна задавати кілька секцій **private**, **protected** і **public**, порядок їх слідування значення не має. Однак, хороший стиль програмування рекомендує використовувати при визначенні класу кожен специфікатор доступу тільки один раз, що робить програму більш читабельною.

Виходячи із принципу інкапсуляції, атрибути класу завжди повинні бути захищені від несанкціонованого доступу. Для цього їх завжди доцільно робити закритими (**private**), у рідких випадках – захищеними (**protected**), щоб нащадки даного класу мали до них доступ.

Якщо елемент даних оголошений як **private**, то доступ до нього здійснюється тільки через методи цього ж класу. Тому методи, що реалізують інтерфейс класу із зовнішнім середовищем, слід робити відкритими (**public**). Якщо який-небудь метод має допоміжне значення для інших методів класу, його слід описати як закритий (**private**) або захищений (**protected**). Це забезпечує логічну цілісність інформації.

Інколи прості класи можуть містити тільки методи або тільки дані, проте більшість реальних класів містять обидві зазначені компоненти. Як приклад, оголосимо клас, що моделює графічний примітив - точку. Для цього потрібно задати її властивості (наприклад, кординати, колір) і поведінку (наприклад, перемістити точку у нову позицію, змінити її колір тощо). Оголошення такого класу може бути наступним:

**class** TPoint *// інформація про точку*

{ **int** x, y; *// кординати*

**int** color; *// колір*

**public** :

**void** GetX(); *// виведення кординати X*

**void** GetY(); *// виведення кординати Y*

**void** Get\_Сolor(); *// виведення кольору*

**void** Set\_Сolor(int); *// введення кольору*

**void** SetXY (**int**, **int**); *// введення кординат точки*

};

У цьому класі три прихованих атрибути - **x**, **y** і **color**, отримати значення яких ззовні можна за допомогою методів **GetX**(), **GetY**() і **Get**\_**Сolor**(). Хоча атрибути класу **TРoint** не визначені явно специфікатором **private**, однак вони є закритими, оскільки за замовчуванням усі члени, визначені в класі, є **private**-членами.

Оскільки доступ до закритих членів класу організовується через його відкриті функції, то усі методи класу **TРoint** визначені з специфікатором **public** і є відкритими, тобто доступними для інших об'єктів програми. Вони власне і складають інтерфейс даного класу.

Уміле використання рівнів доступу підвищує надійність програм і їхню здатність до змін, послабляючи взаємозалежність між об'єктами. Правильно описаними функціями-членами типу **public** можна змінювати приватні атрибути, не торкаючись програмного коду інших об'єктів.

Класи можуть бути вкладеними. Тобто, поля класу можуть бути атрибутами-об'єктами. Включення в структуру об'єкта іншого об'єкта називають *агрегацією*.

Об'єкти-атрибути призначені для збереження і спеціалізованої обробки даних. Як правило, кожен такий об'єкт поєднує в собі дані, що описують одну з комплексних характеристик конкретного реального об'єкта. З іншого боку, ця характеристика може використовуватися для визначення багатьох об'єктних типів. Тому вона є самостійною об'єктною одиницею і повинна бути описана і реалізована окремо.

Наприклад,

**enum** TMonth { Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };

**char** TName[20];

**class** TDate

{ **int**  Day, Year;

TMonth Month;

**public** :

TDate (**int** D, TMonth M, **int** Y);

**void** Print();

};

**class** TPerson

{ ТName Name;

TDate BirthDay;

**public** :

TPerson (TName UserName, TDate UserBirthDay);

**void** Print();

};

Тут клас **TDate** використовується в описі класу **TPerson**. Але, очевидно, що цей клас може бути включений у визначення всіх класів, що використовують дати.

### **Реалізація класу**

Реалізація класу відображає його внутрішній устрій і складається з визначення методів, оголошених в інтерфейсі класу. Кожен метод класу повинен бути визначений (описаний) в програмі. Визначити метод можна або безпосередньо (в тілі класу), або поза його межами (після тіла класу).

Хороший стиль програмування передбачає визначення методів поза межами класу. Такі методи називають *зовнішніми*.

У випадку реалізації зовнішнього методу, всередині класу слід вказати лише прототип методу, тіло ж методу повинно бути визначено в іншому місці програми (поза класом).

Оскільки різні класи можуть мати методи з однаковими іменами, при визначенні зовнішнього методу необхідно вказати до якого класу він належить. Для цього треба кваліфікувати ім'я метода іменем відповідного класу за допомогою операції розширення області видимості "::":

тип\_результату ім'я\_класу :: ім'я\_методу ( [параметри] )

{ тіло

};

Оператор "::" «прив'язує» ім'я члена до імені класу, тим самим однозначно ідентифі-куючи методи відповідного класу. Це дозволяє компілятору самостійно визначити до якого класу належить та чи інша функція. Наприклад, ось як можна записати код зовнішнього методу **Set\_Сolor**() класу **TPoint**:

**void** TPoint :: Set\_Сolor(**int** color\_new)

{ color = color\_new;

}

За своєю сутністю такий запис повідомляє компілятор про те, що дана версія методу **Set\_Сolor**() належить класу **TРoint** (функція **Set\_Сolor**() знаходиться у області видимості класу **TРoint**). Імена атрибутів у тілі метода класу використовують без явного вказання об'єкта.

При визначенні методу поза класом прототипи оголошення і визначення функції повинні збігатися.

Використання зовнішніх методів підвищує наочність тексту програми, особливо, у випадку значного обсягу коду в методах.

Метод класу можна також визначити як **inline**-функцію(*вбудовану функцію*). *Вбудованою* називають функцію, програмний код якої вставляється в те місце програми, з якого вона викликається (виклик функції замінюється її кодом). Це дозволяє збільшити швидкість (зменшити час) виконання програми за рахунок відсутності команд, пов'язаних з обробкою виклику функції та повернення з функції, які окрім передачі керування виконують дії по збереженню і відновленню контексту (вмісту основних регістрів процесора).

Однак, використання вбудованих функцій збільшує розмір програми в порівнянні з програмою без таких функцій, особливо, якщо така функція виявляється великою. Тому функції зі складною логікою не реалізуються як вбудовані.

Існує два способи реалізації вбудованого методу:

* використання у заголовку метода специфікатора **inline**;
* визначення коду метода безпосередньо в оголошенні класу.

Формат використання специфікатора **inline** у заголовку метода:

**іnline** тип\_результату ім'я\_функції ( [параметри] );

Наприклад,

**class** TPoint

{ **int** x, y;

**int** color;

**public** :

**void** GetX();

**void** GetY();

**void** Get\_Сolor();

**void** Set\_Сolor(**int**);

**void** Set\_XY(**int**, **int**);

};

**Inline void** TPoint :: GetX()

{ **cout** << "X= " << x;

}

**inline void** TPoint:: Set\_Сolor (**int** c)

{ color = c;

}

Слід зазначити, що специфікатор **inline** є рекомендацією, а не командою, за якою компілятор згенерує вбудований код. Існують різні ситуації, коли компілятор не може задовольнити відповідний запит. Так, вбудовуваними не можуть бути функції, які

* містять оператори циклу, конструкцію **switch** або оператор безумовного переходу **goto**;
* містять асемблерні вставки;
* реалізують рекурсивні алгоритми (рекурсивні функції).

У разі неможливості використання специфікатора **inline** для конкретної функції компілятор видає попередження і обробляє функцію стандартним способом або не видає ніяких повідомлень, ігноруючи цей специфікатор.

Будь-яка функція, яка визначається в оголошенні класу, автоматично стає вбудованою. У цьому випадку необов'язково специфікувати її оголошення ключовим словом **inline**. Наприклад, наведену вище програму можна переписати у такому вигляді:

**class** TPoint

{ **int** x, y;

**int** color;

**public** :

**void** GetX() { **cout**<<" X= "<< x; } *// автоматично вбудовані функції*

**void** GetY() { **cout**<<" Y= "<< y; }

**void** Get\_Сolor() { **cout**<<" Сolor - "<< color; }

**void** Set\_XY(**int** хх, **int** yy) { х = хх; y = yy; }

**void** Set\_Сolor(**int** c) { color = c; }

};

У цьому прикладі тіла функцій **GetX**(), **GetY**(), **Get\_Сolor**(), **Set\_XY**() і **Set\_Сolor**() визначені безпосередньо в оголошенні класу і тому автоматично є вбудованими методами. У загальному випадку, якщо тіло функції не надто складне і громіздке, то функція визначається безпосередньо в оголошенні класу.

Таким чином, специфікація класу здебільшого складається з двох частин:

* *оголошення класу* – в ньому описуються всі атрибути і методи класу;
* *визначення методів класу*, тобто реалізації конкретних функцій даного класу.

Хорошим стилем в С++ вважається винесення оголошення класу (його інтерфейсу) у окремий модуль - заголовний файл (.h-файл). Такий файл називають *файлом* *інтерфейсу* (або *інтерфейсом)*. При цьому в оголошенні класу, окрім атрибутів, повинні міститися тільки прототипи функцій. Визначення методів розміщується в окремому однойменному файлі з розширенням .cpp, який називають *файлом* *реалізації*. Така організація програми забезпечує незалежність всіх модулів, що використовують заголовний файл із оголошенням класу, від якихось змін у реалізації методів цього класу.

Основне завдання інтерфейсу (відкритих методів) полягає в тому, щоб дати клієнтам класу уявлення про його можливості (функціональність). При цьому клієнтів класу не повинно цікавити, яким чином клас виконує їх завдання. Закриті члени класу і визначення відкритих функцій-членів недоступні для клієнтів. Ці компоненти складають реалізацію класу.

## Створення об'єктів

Визначення класу не призводить до виділення пам'яті. Пам'ять виділяється тільки тоді, коли визначається конкретна змінна "класового" типу, яку називають *екземпляром класу* або *об'єктом*.

Способи створення об'єктів залежать від наявності або відсутності в класі спеціального методу, що визначає необхідні для створення об'єкта дії, – так званого *конструктора.* Зокрема, якщо в класі відсутній конструктор, але описані захищені (**protected**) або приховані (**private**) атрибути, то можливе створення тільки неініціалізованих об'єктів.

Створення об'єкта в програмі здійснюється оголошенням відповідної змінної (для *статичних* об'єктів) або динамічним розміщенням об'єкта в пам'яті (для *динамічних* об'єктів). Формати створення об'єктів:

ім'я\_класу ім'я\_об'єкта; *//створення статичного* *об’єкта*

ім'я\_класу \* покажчик\_на\_об'єкт = **new** ім'я\_класу *//створення динамічного* *об’єкта*

Наприклад, для оголошеного раніше класу **ТРoint** визначення

TPoint рoint; *// статичний* *об’єкт класу TPoint*

виділяє у статичній пам'яті область, достатню для зберігання атрибутів класу **ТРoint**, а визначення

TPoint \*р\_рoint = **new** TPoint; *// динамічний* *об’єкт класу TPoint*

- у динамічній пам'яті.

При оголошенні тільки покажчика на об'єкт, як і для звичайних змінних, пам'ять під об'єкт не виділяється. Це необхідно зробити окремо, використовуючи оператор **new**. Наприклад,

TPoint \*р\_рoint; // *покажчик* на *об’єкт класу TPoint*

р\_рoint = **new** TPoint;

Після роботи з динамічним об'єктом пам'ять необхідно звільнити, використавши оператор **delete**.

Об'єкти класу можна створювати і спільно з описом класу. Наприклад,

**class** TPoint

{ **int** x, y;

**int** color;

**public** :

**void** GetX();

**void** GetY();

**void** Get\_Сolor();

**void** Set\_Сolor(**int**);

**void** Set\_XY(**int**, **int**);

} рoint\_A, рoint\_B;

Після створення об'єктів класу кожен з них набуває власні копії атрибутів, які утворюють клас. Це означає, що кожен з об'єктів **рoint\_А** і **рoint\_В** матиме власні копії змінних **x**, **y** і **color**. Отже, дані, пов'язані з об'єктом **рoint\_А**, відокремлені від даних, які пов'язані з об'єктом **рoint\_В**.

На об'єкти поширюються загальні правила тривалості існування і області дії змінних, а саме:

* *зовнішні об'єкти* створюються до виклику функції **main**() і знищуються по завершенні програми;
* *локальні об'єкти* створюються щоразу при виклику функції, в якій вони оголошені, і знищуються при виході з неї;
* *динамічні об'єкти* створюються оператором **new** і знищуються оператором **delete**.

Під час або після оголошення об'єкта можлива ініціалізація його атрибутів - задання їм початкових значень. Способи ініціалізації атрибутів залежать від наявності або відсутності в класі конструктора, а також від того, в яких секціях класу описані атрибути.

Зокрема, за відсутності в класі конструктора і захищених (**protected**) або прихованих (**private**) атрибутів оголошення ініціалізованих об'єктів здійснюється так же, як і оголошення ініціалізованих структур, наприклад,

TPoint рoint = { 3, 4, 2 }.

Ініціалізуючі значення при цьому повинні перераховуватися в порядку задання атрибутів в оголошенні класу.

За відсутності в класі конструктора і наявності захищених (**protected**) або прихованих (**private**) атрибутів створити ініціалізований об'єкт неможливо.

Якщо в класі є конструктор, то він автоматично ініціалізує атрибути при створенні об'єкта. Способи створення об'єктів за наявності в класі конструктора будуть розглянуті пізніше.

Значення атрибутів неініціалізованих статичних та динамічних об'єктів можна задати і в процесі подальшої роботи з об'єктами: захищених (**protected**) і прихованих (**private**) - тільки за допомогою методів класу, а загальнодоступних (**public**) - за допомогою методів класу або безпосереднім присвоєнням в програмі.

Наприклад, створення ініціалізованих і неініціалізованих об'єктів за відсутності конструктора:

**class** Number

{ **public**:

**int** n;

};

**int** **main**()

{ Number N = { 56 }; *// ініціалізований об’єкт*

Number NN; *// неініціалізований об’єкт*

NN.n = 24; *// ініціалізація об’єкта* *безпосереднім присвоєнням*

…

}

## Реалізація доступу до членів класу

Реалізація доступу до членів класу визначається областю їх видимості відносно коду (методу), з якого здійснюється звернення.

Звертання до будь-якого члена класу із метода цього самого класу здійснюється безпосередньо за іменем атрибута чи метода, без явного вказання імені об'єкта.

Звертання до відкритих (**public**) членів класу - як атрибутів, так і методів – ззовні, тобто з методів інших класів або з функцій, не пов'язаних ні з яким класом, здійснюється за допомогою *повних імен*, кожне з яких має вигляд

ім'я\_об'єкта . ім'я\_класу :: ім'я\_члена\_класу;

або скорочених імен, в яких кваліфікатор доступу опущений і приналежність до класу визначається за типом об'єкта:

ім'я\_об'єкта. ім'я\_члена\_класу;

ім'я­\_покажчика\_на\_об'єкт -> ім'я\_члена\_класу.

Як видно, доступ до членів об’єкта є аналогічним доступу до полів структури. Для цього після імені об’єкта ставиться крапка “.”, а після неї зазначається ім’я атрибуту або методу. У такий спосіб член класу пов’язується з ім’ям об’єкта. При цьому дужки після імені методу є обов’язковими, навіть якщо немає параметрів усередині. Дужки “говорять” про те, що здійснюється виклик функції, а не використовується значення атрибуту. Операція крапка “.” є *операцією доступу* до члена класу.

Наприклад,

N. n = 15; *// перевизначення статичного об’єкта - числа N*

р\_рoint -> Set\_XY(3, 4); *// задання координат динамічного об’єкта* - *точки р\_рoint*

**cout** << рoint\_A .GetX(); *// виведення координати X статичного об’єкта - точки*  *рoint\_A*

У процесі виконання останнього рядка буде викликано метод **GetX**() для об'єкта-точки **рoint\_A**. Під час виклику **рoint\_A.GetX**() дії, визначені у функції **GetX**(), будуть спрямовані на копії даних, які належать тільки точці **рoint\_A**. Необхідно мати на увазі, що **рoint\_А** і **рoint\_В** – це дві окремі точки. Це означає, що ініціалізація атрибутів точки **рoint\_А** зовсім не приводить до ініціалізації атрибутів точки **рoint\_В**. Єдине, що зв'язує об'єкти-точки **рoint\_А** і **рoint\_В**, це те, що вони мають один і той самий класовий тип і методи класу, які обробляють їх дані.

Якщо атрибут оголошений як **private** або **protected**, то доступ до нього ззовні - із методів інших класів або із ззовнішніх функцій (не пов'язаних ні з яким класом) - здійснюється тільки через методи цього ж класу.

Такі методи, як правило, включають спеціальні функції для читання і запису атрибутів:

* *модифікатори* (*setters*) – для запису;
* *селектори* (*getters*) - для читання.

Методи-модифікатори і методи-селектори здійснюють підконтрольне зчитування і зміну атрибутів, забезпечуючи узгодженість внутрішніх даних об'єкта.

Так, якщо змінюється атрибут **size** об'єкта класу **stack**, який описує максимальний розмір стека, то необхідно здійснити ряд дій по узгодженій зміні інших атрибутів: виділення додаткової пам'яті і т.ін. Методи-селектори, зокрема, дозволяють не переписувати всю програму, якщо відбувається зміна в типі, способі зберігання і розміщенні елементів даних у класі – для цього достатньо внести зміни тільки у відповідну функцію.

Для підвищення читабельності програми, зазвичай селекторам і модифікаторам дають легко впізнавані імена: назви методів-селекторів починаються з Get, методів-модифікаторів – з Set. Так для раніше описаного класу **TРoint** були визначені селектори **GetX** (), **GetY** (), **Get\_Сolor** () і модифікатори **SetXY** () і **Set\_Сolor** ().

Розглянемо приклад визначення класу **TРoint** і реалізації доступу до його членів:

**class** TPoint

{ **int** x, y;

**int** color;

**public** :

**void** GetX() { **cout**<<" X= "<< x; }

**void** GetY() { **cout**<<" Y= "<< y; }

**void** Get\_Сolor() { **cout**<<" Сolor - "<< color; }

**void** Set\_XY(**int** хх, **int** yy) { х = хх; y = yy; }

**void** Set\_Сolor(**int** c) { color = c; }

};

**int** **main** ()

{ TPoint рoint\_A, рoint\_B;

рoint\_A. Set\_XY (19, 26);

рoint\_В. Set\_XY (28, 55);

**cout** << "Координати точки А: Х - " << рoint\_A.GetX() << ", Y - " << рoint\_A.GetY() << **endl**;

**cout** << "Координати точки В: Х - " << рoint\_В.GetX() << ", Y - " << рoint\_В.GetY() << **endl**;

**system**("pause");

}

Перший оператор функції **main**() визначає два об’єкти – **рoint\_A** і **рoint\_В** – класу **TРoint**. Наступна пара операторів здійснює виклики методу **SetXY**() для задання координат точок **рoint\_A** і **рoint\_В**. Наступні два оператори викликають методи **GetX**() і **GetY**() для об’єктів **рoint\_A** і **рoint\_В** і виводять на екран координати відповідних об’єктів, повернутих цими методами. Отже, результат виконання програми має вигляд:

Координати точки А: Х – 19, Y - 26

Координати точки В: Х – 28, Y - 55

## Конструктори і деструктори

### **Поняття конструктора**

Як уже зазначалося, ініціалізацію атрибутів об'єкта можна здійснити декількома способами, зокрема, за допомогою методів-модифікаторів (*setters*). Однак, використання таких функцій може спричиняти помилки: оскільки ніде не сказано, що об'єкт повинен бути проініціалізований, програміст може зробити це двічі або забути про це. Окрім того, вимога ініціалізації атрибутів класу є достатньо поширеною.

Тому у мовах програмування передбачено реалізацію цієї потреби при створенні об'єктів класу. Така автоматична ініціалізація атрибутів класу здійснюється за допомогою спеціального методу, що називають *конструктором*.

***Конструктор*** ***класу***  – це спеціальний відкритий (**public**) метод, що автоматично викликається при створенні об'єкта класу і виконує дії по виділенню пам'яті під об'єкт та, за необхідності, ініціалізацію його атрибутів.

Тобто, конструктор визначає операції, які необхідно виконати при створенні об'єкта. Традиційно такими операціями є ініціалізація полів класу і виділення пам'яті під динамічні поля, якщо такі в класі оголошені.

Слід відзначити, що явний виклик конструктора неможливий. Це в деяких випадках ускладнює створення ініціалізованих об'єктів.

Конструктор розпізнається по імені, яке обов'язково збігається з іменем класу. Ось, наприклад, як може виглядати модифікований клас **TРoint** після застосування у ньому конструктора:

**class** TPoint

{ **int** x, y;

**int** color;

**public** :

TPoint( ); *// оголошення конструктора*

**void** GetX();

**void** GetY();

**void** Get\_Сolor();

**void** SetXY(**int**, **int**);

**void** Set\_Сolor(**int**);

};

Код конструктора **TРoint**() може бути наступним:

TPoint() { x=0; y=0; color=2; }

Конструктор може мати аргументи, але не повертає жодного значення, навіть типу **void**. Це пояснюється тим, що конструктор автоматично викликається системою, а отже, не існує програми чи функції, яка його викликає, і якій конструктор міг би повернути значення. Отже, задавати значення, яке повертається, для конструктора немає сенсу.

Подібно функції-члену, конструктор може визначатися як всередині класу, так і за його межами. При реалізації зовнішнього конструктора його ім'я треба кваліфікувати іменем відповідного класу за допомогою операції розширення області видимості "::", наприклад,

TPoint::TPoint()

{ x=0; y=0; color=2;

}

У класі явно може бути не описано жодного конструктора. Тоді компілятор автоматично генерує «порожній» (без параметрів та операторів) конструктор. Особливістю такого конструктора є те, що він не присвоює початкові значення атрибутам класу, хоча пам'ять під об'єкт виділяє. Тому такий конструктор ще називають *неініціалізуючим*.

Як і будь-яка інша функція з параметрами, конструктор може бути перевизначений (параметрично перевантажений). Тому клас може мати декілька конструкторів, які дозволяють використовувати різні способи ініціалізації полів об'єктів.

Загалом конструктори можна класифікувати наступними чином:

1. за призначенням

* ініціалізуючий конструктор,
* неініціалізуючий конструктор,

1. за наявністю параметрів

* з параметрами,
* без параметрів,

1. за кількістю та типом параметрів

* конструктор за замовчуванням,
* конструктор з параметрами,
* конструктор копіювання,
* конструктор переміщення,
* конструктор перетворення.

### **Конструктор за замовчуванням**

Конструктор, який не має параметрів, називають *конструктором за замовчуванням*. Такий конструктор зазвичай ініціалізує атрибути класу константними значеннями.

Формат визначення конструктора за замовчуванням:

ім'я\_класу()

{ тіло\_конструктора

}

Наприклад, варіант зовнішнього конструктора за замовчуванням для класу **TPoint**:

TPoint::TPoint()

{ x=0; y=0; color=2; *// тут можна задати будь-які значення*

}

Роль конструктора за замовчуванням може грати конструктор, у якого всі параметри мають апріорні (константні) значення, наприклад,

TPoint::TPoint (**int** x=24, **int** y=12, **int** color=6).

Такий конструктор іноді називають *конструктором з параметрами за замовчуванням* *для всіх полів*.

Якщо в класі явно не описано жодного конструктора, то компілятор автоматично генерує «порожній» конструктор, який теж грає роль конструктора за замовчуванням.

Для класу, який має конструктор за замовчуванням (явно визначений або згенерований компілятором), об’єкт класу створюється оголошенням відповідної змінної, наприклад,

**TPoint** рoint\_A, рoint\_B.

Атрибути об'єктів одного класу, які створюються за допомогою конструктора за умовчанням, завжди мають однакові значення.

Клас може містити тільки один конструктор за замовчуванням. Якщо б таких конструкторів було декілька, то компілятор не зміг би вирішити, який з них повинен бути викликаний.

### **Конструктор з параметрами**

Конструктор, який разом із створенням об'єкту присвоює його атрибутам деякі початкові значення, отримуючи їх як параметри, називають *конструктором з параметрами*.

Формат визначення конструктора з параметрами:

ім'я\_класу (тип параметр\_1, …, тип параметр\_n)

{ тіло\_конструктора

}

Наприклад, варіант зовнішнього конструктора з параметрами для класу **TPoint**:

TPoint::TPoint (**int** x1, **int** y1, **int** color1)

{ x=x1; y=y2; color=color1;

**cout** << "Об'єкт ініціалізовано" << **endl**;

}

При використанні конструктора з параметрами атрибути класу ініціалізуються кожен раз, як тільки створюється екземпляр класу.

Щоб передати аргументи конструктору, необхідно пов'язати їх з об'єктом під час його створення. Мова програмування C++ підтримує декілька способів реалізації такого зв'язування. Найпоширенішим із них є спосіб, який передбачає, що аргументи повинні знаходитися за іменем об'єкта і поміщатися в круглі дужки:

тип\_класу ім'я\_об'єкта(аргумент\_1, …, аргумент\_n).

Ці значення передаються в конструктор класу, і ними ініціалізуються дані об'єкта цього класу. Наприклад, варіант створення об’єкта **рoint** класу **TPoint**:

TPoint рoint(5, 20, 3).

В результаті буде створення об'єкт-точку **рoint** зеленого кольору (color=3) з координатами х =5 і у =20.

Іноді в конструкторах доцільно використовувати параметри за замовчуванням - константні значення, які задаються в заголовку. Наприклад,

TPoint::TPoint (**int** x1, **int** y1, **int** color1=6).

Такі конструктори називають *конструкторами з аргументами за замовчуванням*.

Приклад.

**class** sstr

{ **char** str[50];

**public**:

**int** x, y;

sstr(**char** \*vs, **int** vx, **int** vy);

**void** print(**void**) { **cout** <<" x - " << x << " y - "<< y << " str -" << str << **endl**; }

};

sstr::sstr(**char** \*vs="Строка по умолчанию", **int** vx=80, **int** vy=90)

{ **strcpy**(str, vs);

x=vx; y=vy;

}

**int main**()

{ setlocale(0,"russian");

sstr a0; *// об'єкт з полями за умовчанням*

sstr a1("Строка"); *// об'єкт із заданим рядком*

sstr a2("Строка", 100); *// об'єкт із заданими рядком і x*

sstr a3("Строка", 200, 150); *// об'єкт із заданими рядком, x і y*

a0.print();

a1.print();

a2.print();

a3.print();

**system**("pause");

}

### **Список ініціалізації**

Особливістю конструктора з параметрами є те, що він не може бути використаний для ініціалізації атрибутів-констант чи атрибутів-посилань, оскільки їм не можуть бути присвоєні значення. Ініціалізація таких атрибутів усередині інтерфейсу класу неможлива, оскільки визначення класу задає тільки тип кожного із таких атрибутів, але не резервує для них реальної області пам’яті.

Для таких випадків застосовується спеціальна конструкція, що включається в опис конструктора, - *список ініціалізації*. Дана конструкція дозволяє ініціалізувати атрибути без використання операцій присвоєння значень.

Формат конструктора зі списком ініціалізації:

ім'я\_класу (параметри) : список\_ініціалізаціі

{ тіло\_конструктора

}

Конструктор з параметрами, що містить список ініціалізації часто називаюють *конструктором* *ініціалізації.*

Таким чином, основне призначення даного конструктора – ініціалізація атрибутів особливих типів (константних, посилальних). Окрім цього список ініціалізації застосовують для явного виклику конструкторів об'єктних полів (атрибутів-об'єктів) і атрибутів базового класу (конструкторів для базових класів, від яких успадкований описуваний клас). У останньому випадку список ініціалізації вказується в описі конструктора основного класу. Слід зазначити, що конструктор ініціалізації може застосовуватися і для ініціалізації звичайних полів класу.

Список ініціалізації в означенні конструктора має наступний формат:

ім'я\_атрибута (ініціалізатори) [, ім'я\_атрибута (ініціалізатори)],

де ініціалізатори визначають значення, які використовуються для ініціалізації зазначених частин створюваного об'єкта.

У разі використання списку ініціалізації тіло конструктора може виявитися порожнім (якщо всі атрибути ініціалізуються через список ініціалізації). Але і в цьому випадку воно обов'язково має бути присутнім. В загальному випадку конструктор може мати як список ініціалізації, так і оператори в тілі конструктора.

Наприклад,

TPoint(): x(0), y(10), color(5) { }.

В якості ініціалізаторів можуть виступати формальні параметри конструктора. Наприклад,

TPoint (**int** хх, **int** уу, **int** с): x(хх), y(уу), color(с) { } .

У першому випадку конструктор ініціалізації викликається при створенні об’єкта, зокрема, при його визначенні:

TPoint рoint.

У другому випадку при створенні об’єкта **рoint** буде викликано конструктор з відповідними, зазначеними у дужках параметрами:

TPoint рoint(5, 20, 3).

### **Конструктор копіювання**

При створенні об'єкта його атрибути можуть бути проініціалізовані значеннями атрибутів іншого об'єкта цього ж класу. Для такого способу створення об'єкта у С++ використовується так званий *конструктор копіювання*.

Конструктор копіювання має єдиний аргумент, що є поcиланням на вже існуючий об’єкт поточного класу. Формат визначення конструктора копіювання:

ім'я\_класу ( [**const**] ім'я\_класу & ім'я\_об'єкта).

Викликається конструктор копіювання, як і всі інші конструктори, під час оголошення об’єкта. У даному випадку об'єкт-ініціалізатор зазначається у круглих дужках після ідентифікатора створюваного об'єкта, наприклад,

TPoint рoint\_A (5, 20, 3);

TPoint рoint\_В (рoint\_A);

Об'єкт-точка **рoint\_В** після створення буде мати значення об'єкта-точки **рoint\_A**, бо її ініціалізація відбулася значенням об'єкта **рoint\_A** за допомогою конструктора копіювання.

Об'єкт, який створюється з використанням конструктора копіювання, може ініціалізуватися не тільки іменованими об'єктами, але і тимчасово створеними об'єктами, наприклад,

TPoint рoint = TPoint (5, 20, 3); *// ініціалізація об'єкта рoint тимчасовим об'єктом типу TPoint*

Якщо у класі немає явно описаного конструктора копіювання, то компілятор генерує його автоматично. Особливістю такого конструктора є те, що він виконує так зване *поверхневе копіювання*, тобто поелементне копіювання атрибутів класу. Наприклад, використання автоматично створюваного конструктора копіювання:

**class** TPoint

{ **int** x, y;

**int** color;

**public** :

**void** print(**void**)

{ **cout** << "Координаты точки: Х - " << x << ", Y - " << y << **endl**;

**cout** << "Цвет : "<< color << **endl**;

}

**TPoint** (**int** хх, **int** yy, **int** c) { х = хх; y = yy; color = c; }

};

**int** **main**()

{ **setlocale**(0,"russian");

TPoint рoint\_A(5, 20, 3);

рoint\_А.print();

TPoint рoint\_В = рoint\_A; *// виклик конструктора копіювання*

рoint\_В.print();

**system**("pause");

}

У даному прикладі компілятор автоматично створює конструктор копіювання, згідно якого атрибути об'єкта **рoint\_A** без зміни копіюються в атрибути об'єкта **рoint\_В**:

TPoint (**const** TPoint & obj) : х(obj.х), у(obj.у), color(obj. color) { }.

### **Конструктор переміщення**

Починаючи з С++11 при створенні об'єкта його атрибути можуть бути проініціалізовані значеннями *тимчасових об'єктів* - об'єктів , які не мають імені. Це стало можливим завдяки тому, що в С++11 додано новий тип посилання — посилання на тимчасовий об'єкт, яке ще називають ***rvalue*-посиланням** (*r-value reference*) або посиланням на *праводопустимий вираз* (той, що може стояти праворуч в операції присвоювання).

Його оголошення наступне:

ім'я\_типу &&.

*Rvalue*-посилання введені в стандарт С++11 для того, щоб відрізняти посилання на об'єкти, у яких немає імені, від традиційних посилань на об'єкти, у яких є імена (позначаються через &). Останні в стандарті С++11 почали називати ***lvalue*-посиланням** (посиланням на вираз, що може стояти ліворуч в операції присвоєння).

Дане нововведення дозволяє враховувати тимчасові об'єкти і реалізовувати семантику переносу (*move semantics*).

Для створення об'єктів на основі тимчасових об'єктів використовується так званий *конструктор переміщення* (*move constructor*), який замінює повне копіювання об'єкта переміщенням покажчика.

Він оголошується подібно конструктору копіювання, але його параметром є *rvalue-* посилання:

ім'я\_класу (ім'я\_класу &&тимчасовий\_об'єкт).

Наприклад,

**class** ТArray

{ **int** length; *// розмір масиву*

**int**\* data; *// динамічний масив*

**public**:

ТArray(**int** length\_p);

ТArray(ТArray&& k);

**void** Array\_out();

};

ТArray CreateArray(**int** n)

{ ТArray tmp(n);

**return** tmp;

}

**int main**()

{ ТArray a(CreateArray(20));

a.Array\_out();

system("pause");

}

ТArray::ТArray(**int** length\_p) : length(length\_p), data(**new** int[length\_p])

{ **for** (**int** i=0; i<length; i++)

data[i] = i;

}

ТArray::ТArray(ТArray&& k) : length(k.length), data(k.data)

{ k.data = **nullptr**;

k.length = 0;

}

**void** ТArray::Array\_out()

{ **for** (**int** i=0; i<length; i++)

**cout** << data[i] << " ";

**cout** << endl;

}

Константа nullptr, використана у прикладі, введена в С++11 з метою усунення можливих непорозумінь між константою 0 як цілочисельним значенням і покажчиком. nullptr має тип std :: nullptr\_t і його не можна переплутати із значенням іншого типу.

Конструктор переміщення введений для випадків, коли вихідний об'єкт є rvalue, наприклад, це значення, що повертається функцією. Даний конструктор служить для вирішення проблеми втрати ефективності, пов'язаної із створенням тимчасових об'єктів. Наприклад, копіювання об'єкта, який містить динамічний масив - це виклик функції, виділення пам'яті і цикл. Таке копіювання має сенс, коли дійсно потрібні дві копії масиву. Але в багатьох випадках це не так: часто масив копіюється з одного місця в інше, а потім видаляється стара його копія. Найчастіше так відбувається, коли об'єкт, що копіюється, є тимчасовим. Якраз для таких випадків і використовується конструктор переміщення.

### **Конструктор перетворення**

При створенні об'єкта його можна проініціалізувати об'єктом іншого типу, тобто перетворити об’єкт, тип якого відрізняється від типу конструктора, в об’єкт даного класу. Для такого створення об'єкту використовується *конструктор перетворення*.

Даний конструктор має рівно один параметр, тип якого відрізняється від типу поточного класу. Тіло конструктора вказує компілятору, як здійснити потрібне перетворення.

Можливі формати заголовка конструктора перетворення:

ім'я\_класу (ім'я\_класу ім'я\_об'єкта);

ім'я\_класу (ім'я\_класу &ім'я\_об'єкта);

ім'я\_класу (**const** ім'я\_класу &ім'я\_об'єкта).

Останній прототип служить для захисту від зміни переданого фактичного параметра в тілі конструктора, оскільки при отриманні посилання на фактичний параметр використовується власне переданий об'єкт, а не його локальна копія.

Наприклад,

**class** ТСube

{ **int** A;

**public** :

ТСube(**int** A1=10); *// конструктор перетворення*

**int** GetA();

**void** SetA(**int**);

**void** Area(**int**);

};

Для такого класу будуть вірні такі оголошення об'єктів:

ТСube сube1, сube2(20);

### **Перевантаження конструкторів**

У класі може бути визначено декілька конструкторів з одним і тим самим іменем. Тоді конструктор вважається *перевантаженим*. Єдина вимога: не може бути двох конструкторів з однаковим (за типами) набором формальних параметрів, зокрема, не може бути двох конструкторів за умовчанням.

Виконуватись буде той конструктор, параметри якого збігаються з аргументами, що були передані при оголошенні об'єкта, за кількістю і типом.

Щоб перевизначити конструктор класу, треба оголосити його в усіх потрібних форматах і визначити кожну дію, пов'язану з кожним форматом.

Розглянемо приклад використання перевизначених конструкторів.

#**include** <ctime> *// для використання системного часу і дати*

…

**class** timerClass

{ **int** s;

**public**:

timerClass (**char** \*t) { s = **atoi**(t); } *// задання секунд у вигляді рядка*

timerClass (**int** t) { s = t; } *// задання секунд у вигляді цілого числа*

timerClass (**int** xv, **int** sec) { s = xv\*60 + sec; } *// час в хвилинах і секундах*

timerClass (**int** hod, **int** xv, **int** sec) { s = 60\*(hod\*60 + xv) + sec; } *// час в год., хв. і секундах*

**void** Run(); *// таймер відліку часу*

};

**void** timerClass::Run()

{ **clock\_t** t1 = clock();

**while** ((**clock**()/**CLOCKS\_PER\_SEC** – t1/**CLOCKS\_PER\_SEC**)< s);

**cout** << "\a"; *// подання звукового сигналу*

}

**int** **main**()

{ timerClass ObjA(10), ObjB("20"), ObjC(1, 10), ObjD(0, 2, 8);

ObjA.Run(); *// відлік 10 секунд*

ObjB.Run(); *// відлік 20 секунд*

ObjC.Run(); *// відлік 1 хвилини і 10 секунд*

ObjD.Run(); *// відлік 0 годин 2 хвилини і 8 секунд*

**system**("pause");

}

У наведеному коді програми оголошено клас **timerClass**, який діє як таймер зворотного відліку. При створенні об'єкта типу **timerClass** таймеру присвоюється деяке початкове значення часу. При виклику функції **Run**() таймер здійснює відлік часу у зворотному порядку до нуля, а потім подає звуковий сигнал. Для даного класу конструктор перевизначається тричі, надаючи тим самим можливість задавати час як у секундах (причому або числом, або рядком), так і у хвилинах і секундах (за допомогою двох цілочисельних значень).

При створенні об'єктів **ObjA**, **ObjB**, **ObjC** i **ObjD** класу **timerClass** конструктор надає атрибуту **s** початкові значення чотирма різними способами, що підтримуються механізмом перевизначення конструкторів. У кожному випадку викликається той конструктор, який відповідає заданому переліку параметрів, і тому правильно ініціалі-зує "свій" об'єкт.

У цьому коді програми використовується стандартна бібліотечна функція **clock**(), яка повертає кількість сигналів, прийнятих від системного годинника з моменту початку виконання програми. Прототип цієї функції має такий вигляд:

**clock\_t** clock().

Тип **clock\_t** є різновидом довгого цілочисельного типу. Операція ділення значення функції **clock()** на значення **CLOCKS\_PER\_SEC** дає змогу перетворити отриманий результат у секунди. Як прототип функції **clock**(), так і визначення константи **CLOCKS\_PER\_SEC** містяться у заголовному файлі <**ctime**>.

Таким чином, кількість конструкторів класу визначається кількістю способів створення об’єкта цього класу. Який з конструкторів буде виконуватись, залежить від ситуації, в якій відбувається створення об'єкта, задовольняє їй по кількості і типах параметрів.

При заданні декількох конструкторів слід дотримуватися тих же правила, що і при написанні перевантажених функцій - у компілятора повинна бути можливість розпізнати потрібний варіант.

Механізм перевантаження конструкторів є зручним для динамічного створення об'єктів на основі інформації, яка стає відомою тільки у процесі виконання програми. Наприклад,

**int** **main**()

{ timerClass ObjA(10); *// створення об'єкта*

ObjA.Run();

**char** str[80];

**cout** << "Введіть кількість секунд: ";

**cin** >> str;

timerClass ObjB(str); *// створення об'єкта*

ObjB.Run();

**int** xv, sec;

**cout** << "Введіть хвилини і секунди: "; **cin** >> xv >> sec;

timerClass ObjC(xv, sec); *// створення об'єкта*

ObjC.Run();

**int** hod;

**cout** << "Введіть години, хвилини і секунди: "; **cin** >> hod >> xv >> sec;

timerClass ObjD(hod, xv, sec); *// створення об'єкта*

ObjD.Run();

**system**("pause");

}

Як бачимо, об'єкт **ObjA** створюється з використанням механізму ініціалізації цілочисельною константою. Основою для побудови об'єктів **ObjB**, **ObjC** і **ObjD** слугує інформація, яка вводиться користувачем у процесі виконання програми. Оскільки для об'єкта **ObjB** користувач вводить рядок, то є сенс перевизначити конструктор **timerClass()** для прийняття рядкової змінної. Аналогічно, при створенні об'єкта **ObjC** час вводиться у вигляді хвилин і секунд, тому для побудови об'єкта **ObjC** логічно використовувати формат конструктора, що приймає два цілочисельні аргумента. Для побудови об'єкта **ObjD**, час для якого вводиться у вигляді годин, хвилин і секунд, має сенс використовувати формат конструктора, що приймає три цілочисельні аргумента.

Наявність декількох форматів конструктора позбавляє програміста від виконання додаткових перетворень під час ініціалізації атрибутів об'єктів. Це дає змогу програмісту вибрати з наявних конструкторів той, який найкраще відповідає поточному формату даних.

З іншого боку, захаращення коду програми конструкторами для обробки ситуацій, які рідко виникають, як правило, негативно впливає на читабельність коду програми. Так перевизначення конструктора **timerClass**() для прийняття значення, вираженого, наприклад, у днях або наносекундах, навряд чи себе виправдає.

### **Поняття деструктора**

Доповненням до конструктора слугує *деструктор* – спеціальний метод, який автоматично викликається для коректного знищення об'єктів. У деструкторі визначаються операції, які необхідно виконати для звільнення пам’яті від даних після завершення роботи програми. Такі операції часто називають *збіркою сміття*.

Зазвичай деструктор потрібен, якщо конструктор або якісь методи класу динамічно розподіляють пам'ять, створюючи в ній якісь об'єкти. В цьому випадку під час знищення об'єкта для коректного звільнення пам'яті, виділеної під його динамічні поля конструктором, необхідно виконати певну дію або навіть певні послідовності дій, які і виконує деструктор.

Слід зазначити, що в деяких мовах програмування, наприклад, Java, є автоматичні збирачі сміття, що повертають втрачену пам’ять системі. В інших, як наприклад, С++, Object Pascal, пам’ять не звільниться до завершення програми, що є причиною нестабільності роботи, а часто і зависання програми.

Ім'я деструктора, як і ім'я конструктора, співпадає з іменем классу, але йому передує ще додатковий символ "~" («тильда»):

~ім'я\_класу() { … }.

У деструкторів немає параметрів. Подібно до конструкторів, деструктори не повертають значень, а отже, в їх оголошеннях відсутній тип значення, що повертається.

Клас може мати тільки один деструктор або не мати жодного. Якщо деструктор явно не описаний в класі, то компілятор автоматично генерує «порожній» деструктор, який не виконує ніяких дій.

На відміну від конструктора деструктор може викликатися явно.

Момент знищення об'єкта, а, отже, і автоматичного виклику деструктора визначається типом пам'яті, обраним для розміщення об'єкта: локальна, глобальна, зовнішня і т. д. Якщо програма завершується з використанням функції **exit**, то викликаються деструктори тільки глобальних об'єктів. При аварійному завершенні програми, що використовує об'єкти деякого класу, функцією **abort** деструктори об'єктів не викликаються.

Наприклад,

**class** String

{ **char** \*data;

**int** size;

**public**:

String (**int** sz) { data=**new char**[size=sz]; }

~ String () { **delete** data; }

};

## Динамічні об'єкти

### **Класи з динамічними атрибутами**

У класах, що використовують як атрибути дані структурованих типів (масиви, рядки, їх комбінації тощо), часто застосовують динамічний розподіл пам'яті. У цьому випадку відповідні поля містять покажчики на змінну відповідного типу. Покажчик отримує своє значення при виділенні пам'яті під структуру даних, для якої він визначений. Звільнення цієї пам'яті відбувається за адресою в покажчику відповідно до типу змінної.

При створенні класів з динамічними атрибутами необхідно враховувати велику кількість особливостей, зокрема, передбачити наявність конструктора, який здійснює динамічний розподіл пам'яті, і деструктора для звільнення розподіленої пам'яті при знищенні об'єкта. Наприклад,

**class** TNum

{ **int** \*pn;

**public**:

TNum(**int** n) { pn=**new int**(n); } *// конструктор*

~TNum() { **delete** pn; } *// деструктор*

};

Якщо в класі передбачений неініціалізуючий конструктор і в програмі з якоїсь причини динамічні поля можуть бути не створені, то в деструкторі потрібно перевіряти факт виділення пам'яті під поля перш, ніж її звільняти, інакше при виконанні програми буде згенерована помилка звільнення пам'яті. Крім того, для ініціалізації полів об'єкта, створеного за допомогою неініціалізуючого конструктора, слід передбачити метод класу, який виділяв би пам'ять під поля такого об'єкта і ініціалізував би їх.

Додаткові проблеми можуть виникнути і при необхідності створення копії такого об'єкта. Якщо у класі передбачено створення внутрішніх динамічних структур, використання тільки поверхневого копіювання призведе до помилки, так як атрибути-покажчики об'єкта і його копії матимуть однакові значення і вказуватимуть на одну і ту ж, розміщену в динамічній пам'яті, структуру (рис. 1, а). Відповідно при знищенні копії об'єкта деструктором єдина для двох об'єктів динамічна структура знищується, залишаючи основний об'єкт з атрибутом-покажчиком на неіснуючу структуру (рис. 1, б).

|  |  |
| --- | --- |
|  |  |
| а) | б) |

Рис. 1. Об'єкт з динамічним атрибутом:

а - після некоректного створення копії;

б - після видалення невірно створеної копії

У подібних випадках клас, окрім звичайного конструктора, в якому буде виділятися пам'ять під розміщення динамічної структури (рис. 1, а), повинен включати конструктор копіювання, який здійснює не тільки копіювання атрибутів об'єкта, а й самих динамічних структур. При цьому атрибути-покажчики в створюваному об'єкті повинні не механічно копіюватися з об'єкта-ініціалізатора, а вказувати на новостворені динамічні структури нового об'єкта. Такий спосіб копіювання об'єктів називають *глибоким копіюванням.*

Наприклад,

**class** TNum

{ **int** \*pn;

**public**:

TNum(**int** n) { pn=**new int**(n); }

TNum(TNum &Obj) { pn=**new int**(\*Obj.pn); } *// конструктор* *копіювання*

~TNum() { **delete** pn; }

**void print() { printf("%d \n",\*pn); }**

};

**int** **main**()

{ TNum A(1);

A .print();

TNum B(A);

B .print();

**system**("pause");

}

### **Створення, ініціалізація і знищення динамічних об'єктів**

Об'єкту деякого класу пам'ять може бути виділена статично - на етапі компіляції або динамічно – під час виконання програми. В останньому випадку виділення і звільнення ділянок пам'яті здійснюється при виконанні операцій **new** і **delete**.

При виділенні динамічної пам'яті для окремих об'єктів використовують такі форми звернення до операції **new**:

покажчик\_на\_об'єкт = **new** ім'я\_класу;

покажчик\_на\_об'єкт = **new** ім'я\_класу (список\_параметрів).

Перша форма застосовується для створення неініціалізованих динамічних об'єктів, друга – для ініціалізованих.

Формат оператора **delete**:

**delete** покажчик\_на\_об'єкт.

Наприклад,

ТNum \*a, \*b ; *// оголошення покажчиків на об'єкт*

a = **new** ТNum; *// виділення пам'яті під неініціалізований об'єкт*

b = **new** ТNum(5); *// виділення пам'яті під ініціалізований об'єкт*

ТNum \*с = **new** ТNum(b); *// копіювання об'єкта*

Динамічний об'єкт, який створюється з використанням конструктора копіювання, може ініціалізуватися не тільки іменованими об'єктами, але і тимчасово створеними об'єктами. Наприклад,

ТNum \*p = **new** ТNum(7); *// покажчик р отримує адресу динамічного об'єкта типу**ТNum*

*// (явний запуск конструктора з параметрами)*

ТNum р1 = \*р; *// розіменування покажчика на об'єкт для ініціалізації нового об'єкта*

При звільненні пам'яті автоматично буде викликаний деструктор класу, а при його відсутності автоматично буде згенеровано «порожній» деструктор виду

ТNum::~Т Num() { }

Приклад використання динамічних об'єктів з статичними полями.

**class** TVector

{ **int** x, y, z;

**public**:

TVector(){ } *// неініціалізуючий конструктор*

TVector(**int** ax, **int** ay, **int** az) { x=ax; y=ay; z=az; } *// ініціалізуючий конструктор*

~TVector(){ } *// деструктор*

**void** PrintVec();

**void** SetVec(**int** ax, **int** ay, **int** az) { x=ax; y=ay; z=az; } *// ініціалізуючий метод*

};

**void** TVector::PrintVec()

{ **cout**<<"Значение вектора: "<<**setw**(5)<<x<<" , "<<**setw**(5)<<y<<" , "<<**setw**(5)<<z<<"\**n**";

}

**void** **main**()

{ **setlocale**(0,"russian");

TVector \*a,\*b; *// покажчики на об'єкти класу*

a=**new** TVector(12, 34, 23); *// створення ініціалізованого об'єкта*

b=**new** TVector; *// створення неініціалізованого об'єкта*

b->SetVec(10, 45, 56); *// ініціалізація об'єкта*

a->PrintVec(); *// виведення: 12, 34, 23*

b->PrintVec(); *// виведення: 10, 45, 56*

**delete** a; *// виклик деструктора*

**delete** b; *// виклик деструктора*

**system**("pause");

}

## Особливості роботи з об'єктами

### **Присвоєння об'єктів**

Об'єкти одного і того ж класу можна присвоювати один одному. Наприклад,

**class** myClass *// оголошення класу*

{ **int** a, b;

**public**:

myClass() { a=b=0; }

**void** Set(**int** c, **int** d)

{ a=c; b=d; }

**void** Show()

{ **cout** << "a=" << a << "; b=" << b << **endl**; }

};

**int** **main**()

{ myClass ObjA, ObjB; *// створення об'єктів*

ObjA.Set(10, 20);

ObjB.Set(15, 12);

**cout** << "Before -" << **endl**;

**cout** << "ObjA: “; ObjA.Show();

**cout** << "ObjB: "; ObjB.Show();

ObjB = ObjA; *// присвоєння об'єкта ObjA об'єкту ObjB*

**cout** << “Аfter -" << **endl**;

**cout** << "ObjA: "; ObjA.Show();

**cout** << "ObjB: "; ObjB.Show();

**system**("pause");

}

Внаслідок виконання ця програма відображає на екрані такі результати:

Before –

ObjA: a=10; b=20

ObjB: a=15; b=12

Аfter –

ObjA: a=10; b=20

ObjB: a=10; b=20

При виконанні операції присвоювання за замовчуванням здійснюється поверхневе копіювання, тобто дані першого об'єкта порозрядно копіюються у другий.

При роботі з динамічними об'єктами слід пам'ятати, що присвоєння одного об'єкта іншому за допомогою покажчиків зводиться до копіювання адреси: після виконання операції перший покажчик містить ту ж адресу, що і другий. Старе значення в першому покажчику стирається, і, якщо він містив адресу деякого об'єкту, то пам'ять, виділена раніше під цей об'єкт, залишається зайнятою, але вже недоступною. Така помилка отримала назву «витік пам'яті», а недоступну внаслідок цього пам'ять прийнято називати «сміттям». До помилки також призведе спроба звільнити пам'ять за обома покажчиками, оскільки одна і та ж ділянка пам'яті, виділена під об'єкт, звільняється двічі.

Щоб уникнути цих помилок необхідно враховувати і контролювати подібні ситуації. Зокрема, можна передбачити наявність у класі конструктора копіювання, наприклад,

**class** myClass

{ **int** \*p;

**public**:

myClass(**int** c) *// визначення звичайного конструктора*

{ p = **new int**; \*p = c; }

myClass(**const** myClass &obj) *// визначення конструктора копіювання*

{ p = **new int**; \*p = \*obj.p; }

~myClass() { **delete** p; }

**void** Show() { **cout** << \*p << **endl**; }

};

**int** **main**()

{ myClass ObjA(10), ObjB (20); *// викликається звичайний конструктор*

**cout** << "Before -" << **endl**;

**cout** << " ObjA:" ;

ObjA.Show();

**cout** << " ObjB:" ;

ObjB.Show();

ObjB = ObjA; *// викликається конструктор копіювання*

**cout** << “Аfter -" << **endl**;

**cout** << " ObjA:";

ObjA.Show();

**cout** << " ObjB:";

ObjB.Show();

**system**("pause");

}

Внаслідок виконання ця програма відображає на екрані такі результати:

Before –

ObjA: 10

ObjB: 20

Аfter –

ObjA: 10

ObjB: 10

Слід зазначити, що присвоєння одного об'єкта іншому просто робить значення їх атрибутів однаковими, але ці два об'єкти залишаються абсолютно незалежними. Отже, подальша модифікація даних одного об'єкта не впливає на дані іншого.

З використанням операції присвоювання може бути виконана і ініціалізація об'єктом такого ж типу, наприклад,

TPoint рoint\_A (2, 4, 6); *// створення ініціалізованого об'єкта типу TPoint*

TPoint рoint\_В = рoint\_A; *//створення об'єкта рoint\_В − копії об'єкта рoint\_A*

У випадку ініціалізації за допомогою операції присвоювання динамічних об'єктів, слід передбачити наявність у класі конструктора копіювання, наприклад,

**class** myClass

{ **int** \*p;

**public**:

myClass(**int** c) *// визначення звичайного конструктора*

{ p = **new int**; \*p = c;

**cout** << "Allocate memory usual constructor" << **endl**;

}

myClass(**const** myClass &obj) *// визначення конструктора копіювання*

{ p = **new int**; \*p = \*obj.p;

**cout** << "Allocate memory copy constructor" << **endl**;

}

~myClass() { **delete** p; **cout** << "Free memory" << **endl**; }

};

**int** **main**()

{ myClass ObjA(10); *// викликається звичайний конструктор*

myClass ObjB = ObjA; *// викликається конструктор копіювання*

ObjA.Show();

ObjB.Show();

**…**

}

Внаслідок виконання ця програма відображає на екрані такі результати:

Allocate memory usual constructor

10

Allocate memory copy constructor

10

Результати виконання цієї програми вказують на те, що при створенні об'єкта **ObjA** викликається звичайний конструктор. Але коли об'єкт **ObjA** використовують для ініціалізації об'єкта **ObjB**, то викликається конструктор копіювання. Його використання гарантує, що об'єкт **ObjB** виділить для своїх атрибутів власну область динамічної пам'яті. Без конструктора копіювання об'єкт **ObjB** просто був би точною копією об'єкта **ObjA**, а атрибут **ObjA.р** вказував би на ту ж саму область динамічної пам'яті, що і атрибут **ObjB.р**.

Якщо об'єкт уже створений, то операція присвоювання здійснює не ініціалізацію створюваного об'єкта, а копіювання даних, наприклад,

TPoint рoint\_A (2, 4, 6); *// створення об'єкта рoint\_A*

TPoint рoint\_В; *//створення об'єкта рoint\_В*

рoint\_В = рoint\_A; *// копіювання об'єкта рoint\_A в існуючий об'єкт рoint\_В*

### **Передача об'єктів функціям**

Об'єкти можна передавати у функції так само, як і змінні будь-яких інших типів даних. За замовчуванням об'єкти класу передаються функціям за значенням. Тобто, у функцію передається не сам об'єкт, а його копія, яка стає параметром функції. Створення копії означає появу нового тимчасового об'єкта. Усі зміни, внесені в об'єкт-копію в процесі виконання функції, не впливають на вхідний об'єкт, який використовується як аргумент для функції. Наприклад,

**class** myClass

{ **int** с;

**public**:

myClass() { c = 0; }

**void** Set(**int** \_c) { c = \_c; }

**void** Get(**char** \*s) { **cout** << s << c << **endl**; }

};

**void** Fun(myClass obj) *// визначення функції- не члена класу*

{ obj.Get ("t2= "); *// виведення числа 10*

obj.Set(100); *// встановлює тільки локальну копію*

obj.Get ("t3= "); *// виведення числа 100*

}

**int** **main**()

{ myClass Obj; *// створення об'єкта класу*

Obj.Set(10);

Obj.Get ("t1= "); *// виведення числа 10*

Fun(Obj); *// передача об'єкта функції-не члену класа*

Obj.Get("t4= "); *// як і раніше, виводиться число 10*

**system**("pause");

}

Ось як виглядають результати виконання цієї програми:

t1=10

t2=10

t3=100

t4=10

Як підтверджують ці результати, модифікування об'єкта **obj** у функції **Fun**() не впливає на об'єкт **Obj** у функції **main**().

При передачі об'єкта функції потрібно використовувати поточний стан об'єкта, а не його початковий стан. Тому для створення локального (тимчасового) об'єкта, що є копією аргумента-об'єкта, викликається не звичайний конструктор, а *конструктор копіювання*, який визначає, як має створюватися копія об'єкта.

Якщо в класі безпосередньо не визначено конструктор копіювання, то він генерується за замовчуванням. Такий конструктор виконує *поверхневе копіювання*, тобто створює побітову копію об'єкта.

Коли виконання функції, яка містить аргумент-об'єкт завершується, копія аргумента руйнується, для чого викликається деструктор цього локального об'єкта. Потреба виклику даного деструктора пов'язана з виходом локального тимчасового об'єкта з області видимості функції, у якій він використовувався.

Якщо аргумент-об'єкт функції містить покажчик на динамічно виділену область пам'яті, то і покажчик, який належить локальній копії цього об'єкта, також посилатиметься на ту ж саму область пам'яті. Отже, якщо у копії об'єкта будуть внесені зміни у вміст цієї області пам'яті, то ці зміни стосуватимуться також оригінального об'єкта. Більше того, внаслідок завершення роботи функції локальна копія об'єкта буде зруйнована за викликом деструктора, що вплине на оригінальний об'єкт, оскільки деструктор звільнить ту ж саму область пам'яті, яку ще використовує (вже звільнену) даний об'єкт. Дана ситуація робить оригінальний об'єкт в подальшому непридатним для використання.

Розглянемо таку програму:

**class** myClass

{ **int** \*p;

**public**:

myClass(**int** c)

{ p = **new int**; \*p = c;

**cout** << "Виділення пам'яті звичайним конструктором" << **endl**; }

~myClass()

{ **delete** p; **cout** << "Звільнення пам'яті" << **endl**; }

**int** Put() { **return** \*p; }

};

*// У процесі виконання цієї функції виникає проблема*

**void** Get(myClass obj) *// передача об'єкта за значення*

{ **cout** << "\*p= " << obj.Put() << **endl**;

}

**int** **main**()

{ myClass ObjA(10);

Get(ObjA);

**system**("pause");

}

Ця програма містить принципову помилку, а саме: при створенні у функції **main**() об'єкта **ObjA** виділяється область пам'яті, адреса якої присвоюється покажчику **ObjA.р**. При передачі функції **Get**() об'єкт **ObjA** побітово копіюється в параметр **obj**.

Це означає, що обидва об'єкти (**ObjA** і **obj**) матимуть однакове значення для покажчика **р**. Тобто, в обох об'єктах (в оригіналі та його копії) член даних **р** вказуватиме на одну і ту саму динамічно виділену область пам'яті. Після завершення роботи функції **Get**() об'єкт **obj** руйнується за допомогою деструктора. Деструктор звільняє область пам'яті, яка адресується покажчиком **obj.р**. Але ця (вже звільнена) область пам'яті – та ж сама область, на яку все ще вказує член даних оригінального об'єкта **ObjA.р**.

Більше того, після завершення роботи програми руйнується об'єкт **ObjA** і динамічно виділена (ще під час його створення) пам'ять звільняється повторно. Однак, звільнення однієї і тієї ж самої області динамічно виділеної пам'яті удруге вважається невизначеною операцією, яка, як правило (залежно від того, яка система динамічного розподілу пам'яті реалізована), спричиняє непоправну помилку.

Одним із шляхів вирішення даної проблеми є передача у функцію не самого об'єкта, а покажчика або посилання на нього. У цьому випадку копія об'єкта не створюється. Отже, після завершення роботи функції деструктор не викликається. Наприклад,

**class** myClass

{ **int** \*p;

**public**:

myClass(**int** c)

{ p = **new int**; \*p = c;

**cout** << "Виділення пам'яті звичайним конструктором" << **endl**; }

~myClass()

{ **delete** p; **cout** << "Звільнення пам'яті" << **endl**; }

**int** Put() { **return** \*p; }

};

*// Ця функція НЕ створює проблем*

**void** Get(myClass &obj) *// передача об'єкта за посиланням*

{ **cout** << "\*p= " << obj.Put() << **endl**;

}

**int** **main**()

{ myClass ObjA(10);

Get(ObjA);

**system**("pause");

}

Оскільки об'єкт **obj** тепер передається за посиланням, то побітова копія аргумента не створюється, а отже, об'єкт не виходить з області видимості після завершення роботи функції **Get**(). Результати виконання цієї версії коду програми:

Виділення пам'яті звичайним конструктором

\*p= 10

Більш загальним рішенням даної проблеми є створення власної версії конструктора копіювання. Це дає змогу точно визначити, як саме потрібно створювати копію об'єкта і тим самим уникнути описаних вище проблем. Наприклад,

**class** myClass

{ **int** \*p;

**public**:

myClass(**int** c) *// визначення звичайного конструктора*

{ p = **new int**; \*p = c;

**cout** << "Виділення пам'яті звичайним конструктором" << **endl**; }

myClass(**const** myClass &obj); *// визначення конструктора копіювання*

{ p = **new int**; \*p = \*obj.p;

**cout** << "Виділення пам'яті конструктором копії" << **endl**; }

~myClass()

{ **delete** p; **cout** << "Звільнення пам'яті" << **endl**; }

**int** Put() { **return** \*p; }

};

// Ця функція приймає один об'єкт-параметр

**void** Get(myClass obj)

{ **cout** << "\*p= " << obj.Put() << **endl**;

}

**int** **main**()

{ myClass ObjA(10);

Get(ObjA);

**system**("pause");

}

Внаслідок виконання ця програма відображає на екрані такі результати:

Виділення пам'яті звичайним конструктором

Виділення пам'яті конструктором копії

\*p= 10

Звільнення пам'яті

У процесі виконання даної програми відбувається наступне: коли у функції **main**() створюється об'єкт **ObjA**, звичайним конструктором виділяється область пам'яті, адреса якої присвоюється покажчику **ObjA.р**. Потім об'єкт **ObjA** передається функції **Get**(), а саме – її параметру **obj**. У цьому випадку викликається конструктор копіювання, який для об'єкта **ObjA** створює його копію з іменем **obj**.

Конструктор копіювання виділяє пам'ять для цієї копії, а значення покажчика на виділену область пам'яті присвоює атрибуту **р** об'єкта-копії. Потім значення, яке адресується покажчиком **р** початкового об'єкта, записується в область пам'яті, адреса якої зберігається в покажчику **р** об'єкта-копії. Таким чином, області пам'яті, що адресуються покажчиками **ObjA.р** і **obj.р**, є різними та незалежними одна від одної, але значення (на які вказують **ObjA.р** і **obj.р**), що зберігаються в них, однакові. Якби конструктор копіювання у класі **myClass** не був завчасно визначений, то внаслідок створення за замовчуванням побітової копії члени **ObjA.р** і **obj.р** вказували б на одну і ту саму область пам'яті.

Після завершення роботи функції **Get**() об'єкт **obj** виходить з області видимості. Цей вихід супроводжується викликом його деструктора, який звільняє область пам'яті, яка адресується покажчиком **obj.p**. Нарешті, після завершення роботи функція **main**() виходить з області видимості об'єкт **ObjA**, що також супроводжується викликом його деструктора і відповідним звільненням області пам'яті, яка адресується покажчиком **ObjA.р**. Як бачимо, використання конструктора копіювання усуває негативні побічні ефекти, пов'язані з передачею об'єкта функції.

### **Повернення об'єктів функціями**

Якщо об'єкти можна передавати у функції, то з таким самим успіхом функції можуть повертати об'єкти, як результат. За замовчуванням об'єкт класу повертається із функції за значенням. Наприклад,

**class** strClass

{ **char** str[80];

**public**:

**void** Init(**char** \*s) { **strcpy**(str, s); }

**void** Show() { **cout** << "Рядок: " << str << **endl**; }

};

strClass Init() *// функція повертає об'єкт типу strClass*

{ strClass obj;

**char** str[80];

**cout** << "Введіть рядок: "; **cin** >> str;

obj.Set(str);

**return** obj;

}

**int** **main**()

{ strClass Obj;

Obj = Init(); *// присвоєння об'єкта, повернутого функцією Init(), об'єкту Obj*

Obj.Show();

**system**("pause");

}

У наведеному прикладі функція **Init**() створює локальний об'єкт **obj** класу **strClass**, а потім зчитує рядок з клавіатури. Цей рядок копіюється в рядок **obj.str**, після чого об'єкт **obj** повертається функцією **Init**() і присвоюється об'єкту **Obj** у функції **main**().

Щодо механізму повернення об'єктів функціями, то тут виникає ситуація, аналогічна ситуації з передачею об'єктів у функції. Якщо тип значення, що повертається функцією, є об’єктом класу, то під час виклику функції компілятор автоматично генерує тимчасовий об'єкт цього класу і використовує значення, яке визначено в операторі **return**, для ініціалізації цього об’єкта. Після повернення значення із функції негайно викликається деструктор тимчасового об'єкта і цей об'єкт руйнується.

Руйнування тимчасового об'єкта в деяких ситуаціях може викликати непередбачувані побічні ефекти. Так, якщо повернутий функцією об'єкт має деструктор, який звільняє динамічно виділену область пам'яті, то ця пам'ять буде звільнена навіть у тому випадку, якщо об'єкт, який отримує повернуте функцією значення, все ще використовує цю пам'ять. Наприклад,

**class** strClass

{ **char** \*s;

**public**:

strClass() { s = 0; }

~strClass() { **if**(s) **delete**[]s; **cout** << "Звільнення пам'яті" << **endl**; }

**void** Init (**char** \*str) { s = **new char**[**strlen**(str)+1]; **strcpy**(s, str); } *// завантаження рядка*

**void** Show() { **cout** << "s= " << s << **endl**; }

};

*// Ця функція повертає об'єкт типу strClass*

strClass Init()

{ strClass obj;

**char** str[80];

**cout** << "Введіть рядок: "; **cin** >> str;

obj.Set(str);

**return** obj;

}

**int** **main**()

{ strClass Obj;

Obj = Init(); *// ця настанова генерує помилку!!!!*

Obj.Show(); *// відображення "сміття"*

**system**("pause");

}

У даному прикладі виклик деструктора ~**strClass**() відбувається тричі. Вперше він викликається при виході локального об'єкта **obj** з області видимості у момент його повернення з функції **Init**(). Другий виклик деструкторa ~**strClass**() відбувається тоді, коли руйнується тимчасовий об'єкт, який повертається функцією **Init**(). У даному випадку цей об'єкт є побітовою копією об'єкта **obj**. Оскільки область пам'яті, що виділяється для зберігання рядка, який вводить користувач, вже була звільнена (причому двічі!), то під час виклику функції **Show**() на екран буде виведено "сміття". Нарешті, після завершення роботи програми викликається деструктор об'єкта **Obj**, який належить функції **main**().

Під час першого виклику деструктора звільняється область пам'яті, виділена для зберігання рядка, отримуваного функцією **Init**(). Два наступні звернення до деструктора ~**strClass**() пробують звільнити вже звільнену динамічно виділену область пам'яті. При цьому вони можуть зруйнувати систему динамічного розподілу пам'яті. Тому потрібно уникати повернення об'єктів у ситуаціях, коли це може мати згубні наслідки.

Одним із варіантів вирішення цієї проблеми є використання для повернення покажчика або посилання на об'єкт. Але здійснити це не завжди вдається.

Ще один спосіб вирішення цієї проблеми полягає у використанні конструктора копіювання. Наприклад, демонстрація механізму використання конструктора копіювання для створення тимчасового об'єкта, що повертається функцією:

**class** myClass

{ **public**:

myClass() { **cout** << "Звичайний конструктор" << **endl**; }

myClass(**const** myClass &obj) { **cout** << "Конструктор копії" << **endl**; }

};

myClass Fun()

{ myClass obj; *// викликається звичайний конструктор*

**return** obj; *// опосередковано викликається конструктор копії*

}

**int** **main**()

{ myClass ObjA; *// викликається звичайний конструктор*

ObjA = Fun(); *// викликається конструктор копії*

**system**("pause");

}

Внаслідок виконання ця програма відображає на екрані такі результати:

Звичайний конструктор

Звичайний конструктор

Конструктор копії

Тут звичайний конструктор викликається двічі: перший раз при створенні об'єкта **ObjA** у функції **main**(), другий – при створенні об'єкта **obj** у функції **Fun**(). Конструктор копіювання викликається в ту мить, коли генерується тимчасовий об'єкт як значення, яке повертається з функції **Fun**().

Для багатьох програмістів "приховані" виклики конструкторів копії зазвичай є не зрозумілими. Проте практично кожен клас у професійно написаних програмах містить певний конструктор копії, без якого не можливо уникнути побічних ефектів, які виникають внаслідок створення за замовчуванням побітових копій об'єкта.

### **Неявний параметр this**

Коли метод, що належить класу, викликається для обробки даних конкретного об'єкта, йому, крім явно оголошених параметрів, автоматично і неявно передається ще один прихований параметр: покажчик на той об'єкт, для якого викликається цей метод. У С++ цей покажчик має спеціальне ім'я **this** і неявно визначений в кожному методі класу наступним чином:

ім'я\_класу \***const** **this** = адреса оброблюваного об'єкта.

Явно описати чи визначити вказівник **this** не можна і не потрібно. Відповідно до неявного визначення, **this** є константним покажчиком, тобто змінювати його не можна. Проте в кожному методі, що належить класу, він вказує саме на той об'єкт, для якого викликають функцію. Кажуть, що покажчик **this** є додатковим (прихованим) параметром кожного нестатичного метода. Іншими словами, при вході в [тіло](http://ua-referat.com/%D0%A2%D1%96%D0%BB%D0%BE) метода, що належить класу, вказівник **this** ініціалізується значенням адреси того об'єкта, для якого викликана функція. Об'єкт, який адресується вказівником **this**, стає доступним усередині приналежного класу метода [саме](http://ua-referat.com/%D0%A1%D0%B0%D0%BC%D0%B5) за допомогою покажчика **this**. При роботі з членами класу усередині приналежного класу методу можна скрізь використовувати цей покажчик.

Наприклад, нехай визначено такий клас:

**class** myClass

{ **int** c;

**void** Fun() {...};

// ...

};

У тілі функції **Fun**() можна використовувати такий оператор для присвоєння члену c значення 5:

c = 5;

Насправді ж попередній оператор є скороченою формою такого оператора:

**this**->c = 5;

Відповідно, при оголошенні деякого об'єкта **А** виконується операція

**this** =&A,

а при виділенні пам'яті для розміщення динамічного об'єкта, що адресується покажчиком b, - операція

**this** =b.

При роботі з членами класу цей покажчик можна задавати явним чином. Але слід зазначити, що в цьому випадку використання **this** зазвичай не надає ніякої переваги, так як дані більшості конкретних об'єктів вже доступні функціях, що належать класу, по іменах.

Однак іноді явне застосування покажчика **this** корисне і навіть необхідне. Наприклад, без нього не можна обійтися, якщо в тілі приналежної класу функції потрібно явно задати адресу об'єкта, для якого вона була викликана.

Щоб зрозуміти, як працює покажчик **this**, розглянемо таку програму:

**class** TPoint

{ **int** x,y;

**public**:

**void** print(**void**)

{ printf("x=%5d y=%5d\n", x ,y);

}

TPoint \*fun1() *// повертає покажчик на об'єкт, для якого викликається*

{ x = y =100;

**return** this;

}

TPoint fun2() *// повертає об'єкт, для якого викликається*

{ x = y =200;

**return** \*this;

}

};

**int** **main**()

{ TPoint aa;

aa.fun1()->print(); *// виводить: 100 100*

aa.fun2().print(); *// виводить: 200 200*

**system**("pause");

}

Крім того, покажчик **this** явно використовують для формування результату при перевизначенні операцій, так як операція перевизначається для конкретного об'єкта, що викликає її.

Наприклад, функція, що повертає точку - середину відрізка, кінці якого задані:

TPoint& TPoint::Middle (TPoint A, TPoint B)

{ x = ( A.x + B.x ) / 2.0 ; y = ( A.y + B.y ) / 2.0 ; *// обчислює середину відрізка*

Print() ; *// виводить координати поточної точки*

**this** -> Print() ; *// усі три оператори еквівалентні*

(\***this**). Print() ;

**return** \***this**; *//повертає посилання на поточну точку*

}

### **Особливості використання конструкторів і деструкторів**

Узагальнимо особливості використання конструкторів:

1. Конструктор може визначатися явно користувачем або неявно компілятором. Неявне оголошення конструктора здійснюється в таких випадках:

* при визначенні нового об'єкта класу;
* при копіюванні об'єкта, у тому числі, при створенні тимчасового об'єкта (наприклад, при передачі об'єкта з функції);
* при динамічному створенні об'єкта (за допомогою оператора **new**).

1. Неявний виклик конструктора для створення тимчасових об'єктів зазвичай відбувається в наступних випадках:

* при ініціалізації;
* при виконанні операції присвоювання;
* для значень параметрів за умовчанням;
* при створенні та ініціалізації масиву;
* при створенні динамічних об'єктів.

1. Якщо програміст не оголосив жодного конструктора при визначенні класу, компілятор створює його автоматично як конструктор за замовчуванням без аргументів. Наприклад,

**class** CExample

{ **public**:

**int** a, b, c;

**void** multiply (**int** n, **int** m) { a=n; b=m; c=a\*b; };

};

Компілятор вважає, що клас **CExample** має конструктор за замовчуванням, тому якщо треба визначити об'єкти цього класу, необхідно просто оголосити їх без аргументів:

CExample ex;

1. Якщо в класі явно описаний хоча б один конструктор, то автоматичний порожній конструктор не створюється. Це означає, що, якщо за наявності в класі тільки конструкторів з параметрами, спробувати створити об'єкт, не задаючи аргументів, то буде отримано повідомлення про відсутність відповідного конструктора. Наприклад,

**class** Num

{ **int** n;

**public**:

Num(**int** an) { **cout** <<"Конструктор"<<**endl**; n=an; }

};

**void** main()

{ **setlocale**(0,"russian");

Num N(56);

Num cc; *// неініціалізований об'єкт*

**…**

**system**("pause");

}

При компіляції даної програми спроба створити об'єкт сс призведе до отримання повідомлення

error C2512:'Num': no appropriate default constructor available

1. Створення об'єктів без вказання аргументів вимагає, щоб у класі був заданий один з конструкторів:

* неініціалізуючий конструктор без параметрів («порожній»), наприклад,

Num () { }

* ініціалізуючий конструктор без параметрів, наприклад,

Num () { n = 0; }

або зі списком ініціалізації

Num (): n (0) { }

* ініціалізуючий конструкторз параметрамиза замовчуванням для всіх полів, наприклад,

Num (**int** an = 0) { n = an; }

1. Неприпустимо використання конструкторів з параметрами (навіть константами) при описі атрибутів, що є об'єктами інших класів. Якщо атрибути-об'екти класу необхідно ініціалізувати з використанням констант, то це можна зробити тільки в рядку ініціалізації.

Так, клас **z1** не може бути описаний наступним чином:

**class** z1

{ TPoint p2(2,3); // ПОМИЛКА!

TPoint p3;

**int** z;

**public**:

z1(**int** z2) : p3(p2), z(z2) { }

};

Правильний опис класу:

**class** z1

{ TPoint p2, p3;

**int** z;

**public**:

z1(**int** z2) : p2(2,3), p3(p2), z(z2) { }

};

Якби в класі **z1** були описані кілька конструкторів, то в рядок ініціалізації кожного з них необхідно було б помістити виклик конструктора з двома і більше параметрами для поля **p2**.

1. Конструктор копіювання викликається у наступних випадках:

* при ініціалізації створюваного об'єкта, наприклад,

box a(1,2,3); *// виклик конструктора з трьома параметрами*

box b(a); *// ініціалізація*

* при ініціалізації тимчасовим об'єктом, наприклад,

box c = box (3,4,5);

* при передачі параметрів-об'єктів у функцію за значенням (під час створення тимчасових значень в результаті передачі об’єктів класу як параметрів-значень), наприклад,

**int** f(box b);

* при поверненні результату роботи функції у вигляді об’єкта (компілятор викликає конструктор копіювання автоматично), наприклад,

box f();

1. Автоматично генеруємий конструктор копіювання не враховує особливостей об'єкта, а тому не завжди може застосуватися. Відомі два випадки, коли конструктор копіювання необхідно (!) описувати в класі явно:

* якщо при копіюванні атрибутів необхідно змінювати вміст хоча б деяких з них;
* якщо клас містить хоча б одне динамічне поле.

Наприклад, явне визначення конструктора копіювання, що змінює вміст поля при копіюванні:

**class** Num

{ **int** n;

**public**:

Num(**int** an) : n(an) { } *// ініціалізуючий конструктор*

Num(**const** Num & ob) *// конструктор копіювання*

{ n = ob.n\*3; }

**void** print(**void**)

{ **cout** <<" Поле n ="<< n <<" "<< **endl**; }

};

**void** **main**()

{ **setlocale**(0,"russian");

Num aa(10); *// викликається ініціалізуючий конструктор*

aa.print(); *// виводить Поле n = 10*

Num bb=aa; *// викликається конструктор копіювання*

bb.print(); *// виводить Поле n = 30*

**system**("pause");

}

Використане в останньому прикладі застосування конструкторів копіювання зустрічається досить рідко, оскільки робота з програмою при цьому істотно ускладнюється внаслідок необхідності пам'ятати особливості копіювання об'єктів.

1. Такі мови програмування, як Java і С#, не мають конструкторів копіювання, оскільки в жодній з них не створюються побітові копії об'єктів. Справа в тому, що як Java, так і С# динамічно виділяють пам'ять для всіх об'єктів, і програміст оперує цими об'єктами виключно через посилання. Тому при передачі об'єктів функції як параметрів або при поверненні їх із функцій в копіях об'єктів немає ніякої потреби.

Той факт, що Java і С# не потребують конструкторів копіювання, робить ці мови дещо простішими, але за простоту теж потрібно платити. Робота з об'єктами виключно за допомогою посилань (а не безпосередньо, як у C++) накладає обмеження на типи операцій, які може виконувати програміст. Окрім того, таке використання об'єктних посилань у Java і С# не дає змоги точно визначити, коли об'єкт буде зруйновано. У C++ об'єкт завжди руйнується при виході з області видимості.

C++ надає програмісту повний контроль над ситуаціями, які виникають у процесі роботи коду програми, тому вона є дещо складнішою, ніж Java і С#. Це – ціна за потужні засоби програмування.

Узагальнимо правила виклику конструкторів та деструкторів:

1. Конструктор викликається при створенні об’єкта, а деструктор — при його знищенні.
2. Явний виклик конструктора не можливий. На відміну від конструктора деструктор може викликатися явно.
3. Якщо об’єкт визначено глобально (поза будь-якою функцією), то конструктор викликається на самому початку виконання програми (до виклику функції **main**()), деструктор – по закінченні програми.
4. Якщо об’єкт визначено локально (всередині функції), то конструктор викликається кожного разу, коли виникає потреба в створенні такого об'єкта; деструктор — при виході за межі блоку, в якому визначено об’єкт (тобто коли об’єкт виходить з області видимості).
5. Якщо об’єкт визначено локально з використанням специфікатора **static**, конструктор викликається, коли потік управління вперше досягає визначення об’єкта, деструктор — в кінці програми.
6. Якщо об’єкт створено динамічно (з використанням операції **new**), то конструктор викликається при створенні об’єкта, а деструктор — коли об’єкт явно знищується з використанням оператора **delete**. Якщо цього не сталося, то деструктор не буде викликаний ніколи.

Чіткий порядок викликів конструкторів не залежить від того, використовуються конструктори по замовчуванням чи перевантажені конструктори, з аргументами чи без.

При створенні об'єкта конструктори викликаються в наступному порядку:

1) Спочатку викликаються конструктори базових класів, якщо клас для створюваного об'єкта є спадкоємцем інших класів, в порядку їх перерахування в списку наслідування (список, в якому після символу «:» перераховуються базові класи, розділені комами). Якщо в списку ініціалізації описуваного класу присутній виклик конструктора перетворення (або конструктора з двома і більше параметрами) базового класу, то викликається конструктор перетворення (або конструктор з двома і більше параметрами), інакше викликається конструктор за умовчанням базового класу.

2) Потім викликаються конструктори за умовчанням всіх вкладених атрибутів, які не перераховані в списку ініціалізації, і конструктори перетворення, копіювання і конструктори з двома і більше параметрами всіх вкладених атрибутів, які перераховані в списку ініціалізації. Всі перераховані в даному пункті конструктори (за умовчанням, перетворення, копіювання, з двома і більше параметрами) викликаються в порядку опису відповідних атрибутів в класі.

3) Після того, як будуть сконструйовані всі базові класи і атрибути, виконається власний конструктор класу.

Така послідовність виклику конструкторів логічно обґрунтовується тим, що в момент виконання власного конструктора всі інформаційні поля повинні бути вже проініціалізовані.

Описаний порядок застосовується рекурсивно, тобто першим конструюється базовий клас першого базового класу і так далі. Він не залежить від порядку, вказаного в списку ініціалізації членів.

Деструктори викликаються в зворотньому порядку:

1) Власний деструктор. У момент початку його роботи поля класу ще не очищені, і їх значення можуть бути використані в тілі деструктора.

2) Деструктори вкладених об'єктів у порядку, зворотньому порядку їх опису.

3) Деструктори базових класів у зворотньому порядку їх задання.

## Mасиви об'єктів

### **Створення та ініціалізація масивів об'єктів**

Масиви об'єктів можна організувати так само, як і масиви значень стандартних типів. Наприклад, у наведеному нижче коді програми створюється клас **displayClass**, який містить значення розширення для різних режимів роботи монітора. У функції **main**() створюється масив для зберігання трьох об'єктів типу **displayClass**, а доступ до них здійснюється за допомогою звичайної процедури індексування елементів масиву.

**enum** resolution { low, medium, high};

**class** displayClass

{ **int** width;

**int** height;

resolution res;

**public**:

**void** SetWH(**int** w, **int** h) { width = w; height = h; }

**void** GetWH(**int** &w, **int** &h) { w = width; h = height; }

**void** SetRes(resolution r) { res = r; }

resolution GetRes() { **return** res; }

};

**char** names[3][9] = { "Низький", "Середній", "Високий"};

**int** **main**()

{ displayClass Monitor[3];

Monitor[0]. SetRes(low);

Monitor[0].SetWH (640, 480);

Monitor[1]. SetRes(medium);

Monitor[1].SetWH (800, 600);

Monitor[2]. SetRes(high);

Monitor[2].SetWH (1600, 1200);

**cout** << "Можливі режими відображення даних: " << **endl**;

**int** w, h;

**for** (**int** i=0; i<3; i++)

{ **cout** << names[Monitor[i].GetRes()] << ": ";

Monitor[i].GetWH (w, h);

**cout** << w << " x " << h << **endl**;

}

**system**("pause");

}

Внаслідок виконання ця програма відображає на екрані такі результати:

Можливі режими відображення даних:

Низький: 640 x 480

Середній: 800 x 600

Високий: 1600 x 1200

У даному прикладі двовимірний символьний масив **names** використовується для перетворення значення перелічувального типу в еквівалентний символьний рядок. Оскільки в усіх перерахунках, які не містять безпосередньо заданої ініціалізації, перша константа має значення 0, друга – значення 1 і т.д., то значення, яке повертається функцією **GetRes**(), можна використовувати для індексації елементів масиву **names**. Це дає змогу вивести на екран відповідну назву режиму відображення на національній мові.

Багатовимірні масиви об'єктів індексуються так само, як і багатовимірні масиви значень інших типів.

Якщо при оголошенні масиву об'єктів класу, що включає конструктори, використовувати стандартну форму, то для кожного об'єкта масиву буде викликаний неініціалізуючий або ініціалізуючий конструктор без аргументів. Наприклад,

TPoint mas\_рoint [10]; *// десять разів викликається конструктор без аргументів*

Якщо клас **ТPoint** описаний з конструкторами, але при цьому в ньому відсутній конструктор, який можна викликати без аргументів, то при виконанні програми на етапі створення масиву буде отримано повідомлення про помилку *error C2512*, яке зафіксує відсутність у класі зазначеного конструктора. Якщо ж у класі взагалі відсутні конструктори, то помилка зафіксована не буде, оскільки автоматично буде створений неініціалізуючий конструктор без параметрів.

Якщо при створенні масиву використовувався неініціалізуючий конструктор, то всі об'єкти - елементи масиву будуть неініціалізованими. Застосування ініціалізуючого конструктора без параметрів або конструктора з параметрами, заданими за умовчанням, може:

* ініціалізувати всі створювані об'єкти однаково, як задано в ініціалізуючому конструкторі (рідко потрібно за умовою задачі);
* ініціалізувати всі створювані об'єкти випадковим чином (ще рідше потрібно за умовою задачі);
* ініціалізувати всі створювані об'єкти значеннями, що вводяться в процесі ініціалізації (введення в методі ініціалізації вважається нетехнологічним).

Для правильної (що відповідає умовам задачі) ініціалізації об'єктів у всіх трьох описаних вище випадках швидше за все знадобиться спеціальний ініціалізуючий метод, що задає значення атрибутів, наприклад,

**void** TPoint::setPoint(**float** af){ f=af; } *// ініціалізуючий метод*

Якщо клас містить параметризований конструктор, то масив об'єктів такого класу можна ініціалізувати різними значеннями. Наприклад, у наведеному нижче коді програми використовується клас **myClass** і параметризований масив об'єктів типу **array** цього класу, що ініціалізується конкретними значеннями.

**class** myClass

{ **int** a;

**public**:

myClass(**int** b) { a = b; }

**double** Put() { return a; }

};

**int** **main**()

{ myClass array[4] = { -1, -2, -3, -4 };

**for** (**int** i=0; i<4; i++)

**cout** << "array[" << i << "]= " << array[i].Put() << **endl**;

**system**("pause");

}

Результати виконання цієї програми:

array[0]= -1

array[1]= -2

array[2]= -3

array[3]= -4

Синтаксис ініціалізації масиву, виражений рядком

myClass array[4] = { -1, -2, -3, -4 };

є скороченим варіантом такого (довшого) формату:

myClass array[4] = { myClass(-1), myClass(-2), myClass(-3), myClass(-4)};

Формат ініціалізації, представлений у наведеній вище програмі, використовується програмістами частіше, ніж його довший еквівалент, проте необхідно пам'ятати, що він працює для масивів таких об'єктів, конструктор яких приймає тільки один аргумент. При ініціалізації масиву об'єктів, конструктор яких приймає декілька аргументів, необхідно використовувати довший формат ініціалізації.

Наприклад,

**class** myClass

{ **int** a, b;

**public**:

myClass(**int** c, **int** d) { a = c; b = d; }

**int** PutA() { **return** a; }

**int** PutB() { **return** b; }

};

**int** **main**()

{ myClass array[4][2] = { myClass(1, 2), myClass(3, 4),

myClass(5, 6), myClass(7, 8),

myClass(9, 10), myClass(11, 12),

myClass(13, 14), myClass(15, 16) };

**for** (**int** i=0; i<4; i++)

**for** (**int** j=0; j<2; j++)

{ **cout** << "array[" << i << "," << j << "] ==> a= "<< array[i][j].PutA() << "; b= ";

**cout** << array[i][j].PutB() << **endl**;

}

**system**("pause");

}

Внаслідок виконання ця програма відображає на екрані такі результати:

array[0,0] ==> a= 1; b=2

array[0,1] ==> a= 3; b=4

array[1,0] ==> a= 5; b=6

array[1,1] ==> a= 7; b=8

array[2,0] ==> a= 9; b=10

array[2,1] ==> a= 11; b=12

array[3,0] ==> a= 13; b=14

array[3,1] ==> a= 15; b=16

У наведеному вище прикладі параметризований конструктор класу **myClass** приймає два аргументи. У основній функції **main**() оголошується та ініціалізується масив **array** об'єктів шляхом безпосередніх викликів конструктора **myClass**(). Для ініціалізації масиву можна завжди використовувати довгий формат ініціалізації, навіть якщо об'єкт приймає тільки один аргумент. Коротка форма просто зручніша для застосування.

### **Динамічні масиви об'єктів і масиви покажчиків на об'єкти**

Масиви можуть формуватися і з динамічних об'єктів. Це можна зробити трьома способами:

* створити динамічний масив об'єктів - пам'ять під нього виділяють одним безперервним фрагментом рівним обсягу всіх об'єктів масиву (рис. 2, а), наприклад,

B mas[] = **new** B[n];

* створити статичний масив покажчиків на об'єкти і потім динамічно виділити пам'ять під елементи (рис. 2, б), наприклад,

B\*mas[n]; *// пам'ять під масив покажчиків виділена статично*

**for** (i=0; i<n; i++) mas[i] = **new** B; *// виділення пам'яті під об'єкти*

* створити динамічний масив покажчиків і потім також динамічно виділити пам'ять під елементи (рис. 2, в), наприклад,

B \*\*mas = **new** B \*[n]; *// пам'ять під масив покажчиків виділена динамічно*

**for** (i=0; i<n; i++) mas[i] = **new** B; *// виділення пам'яті під об'єкти*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  |  | |
| а) | б) | | | в) |

Рис. 2. Способи організації динамічних масивів об'єктів (сірим виділені

елементи структури, розміщені в динамічній пам'яті):

а - динамічний масив об'єктів;

б - статичний масив покажчиків на динамічно розміщувані об'єкти;

в - динамічний масив покажчиків на динамічно розміщувані об'єкти

Звільняти виділену пам'ять потрібно так, як вона була виділена:

* одним фрагментом - для динамічного масиву об'єктів:

**delete** [] mas;

* поелементно - для масиву покажчиків на об'єкти:

**for** (i = 0; i <n; i ++) **delete** mas [i];

* одним фрагментом, якщо пам'ять під масив покажчиків виділялася динамічно:

**delete** [] mas;

Наприклад, розглянемо програму, що створює масив з п'яти динамічних об'єктів, доступних через масив покажчиків на ці об'єкти, і масив з трьох динамічних об'єктів, доступних через покажчик на перший об'єкт масиву.

#**include** <locale.h>

#**include** <string.h>

#**include** <iostream>

using namespace std;

**class** sstr

{ **char** str[40];

**public**:

**int** x,y;

sstr() { } *// неініціалізуючий конструктор*

sstr(**int** vx, **int** vy, **char** \*vs="Cтрока по умолчанию") { setstr(vx, vy, vs); }

**void** setstr(**int** ax, **int** ay, **char** \*vs); *// ініціалізація полів об'єкта*

**void** print(**void**)

{ **cout**<<"Cодержимое полей : "<< **endl**;

**cout**<<"x= "<<x<<" y= "<<y<<" str= "<<str<<**endl**;

}

~sstr() { } *// деструктор*

};

**void** sstr::setstr(**int** ax, **int** ay, **char** \*vs)

{ if (**strlen**(vs)>=40)

{ **strncpy**(str, vs, 40); str[40]='\0'; }

else **strcpy**(str, vs);

x=ax; y=ay;

}

**void** **main**()

{ **setlocale**(0,"russian");

sstr \*a[5], *// масив покажчиків на п'ять динамічних об'єктів*

\*c; *// покажчик на масив динамічних об'єктів*

**int** i, j;

**char** \*vs="Строка"; *//виділення пам'яті і ініціалізація об'єкта*

**for** (**int** i=0; i<5; i++) *// створення масиву динамічних об'єктів*

a[i]=**new** sstr(10+i,10+2\*i);

**cout**<<" Массив объектов a"<<**endl**;

**for** (i=0; i<5; i++) *// виведення вмісту полів об'єктів*

{ **cout**<<"Элемент "<< i+1;

a[i]->print();

}

**for** (i=0; i<5; i++) **delete** a[i]; *// вивільнення пам'яті*

c=**new** sstr[3]; *// виділення пам'яті під три динамічних об'єкта*

**for** (i=0; i<3; i++) *// ініціалізація полів динамічних об'єктів*

( c+i)->setstr(15+i, 12+i\*2, vs+i );

**cout** <<" Массив объектов c"<<**endl**;

**for** (i=0; i<3; i++) *//* *виведення вмісту полів об'єктів*

{ **cout**<<"Элемент "<< i+1;

(c+i)->print();

}

**delete** []c; *// вивільнення пам'яті*

**system**("pause");

}

### **Приклади побудови статичних та динамічних об'єктів і структур даних**

Відмінності між статичними та динамічними, ініціалізованими та неініціалізованими об'єктами і масивами з них суттєво впливають на синтаксис їх опису та на роботу з ними, тому ці відмінності важливо добре розуміти.

Визначимо клас **ТPoint**, що включає два прихованих атрибути, ініціалізуючий і неініціалізуючий конструктори, ініціалізуючий метод і метод виведення вмісту атрибутів на екран:

**class** ТPoint

{ **int** x, y;

**public**:

ТPoint() { }

ТPoint(**int** ax, **int** ay): x(ax), y(ay) { }

**void** SetPoint(**int** ax, **int** ay) { x=ax; y=ay; }

**void** Print() { **cout** << x << " " << y << "\n"; }

};

А тепер створимо на його основі об'єкти і масиви об'єктів різних видів.

Статичні і динамічні об'єкти.

1. неініціалізований статичний об'єкт (рис. 3, а):

ТPoint А; *// оголошення об'єкта*

А.SetPoint(5, 10); *// ініціалізація полів*

А.Print(); *// виведення вмісту полів на екран*

Оскільки об'єкт **А** – статичний, то пам'ять під нього спеціально виділяти і звільняти НЕ треба, ця операція буде виконана автоматично.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| а) | б) | в) | г) |

Рис. 3. Результати операцій оголошення статичних (а-б) і динамічних (в-г) об'єктів:

а - статичний неініціалізованих об'єкт;

б - статичний ініціалізований об'єкт;

в - покажчик на нестворений динамічний об'єкт;

г - покажчик на динамічний ініціалізований об'єкт

1. ініціалізований статичний об'єкт (рис. 3, б):

ТPoint В(2, 3); *// створення об'єкта*

В.Print(); *// виведення вмісту полів на екран*

Об'єкт **В** відразу створюється ініціалізований, проблем з пам'яттю також немає.

1. неініціалізований динамічний об'єкт (рис. 3, в):

ТPoint \*C; *// оголошення неініціалізованих покажчика на об'єкт*

C = **new** ТPoint(3, 4); *// виділення пам'яті і ініціалізація полів*

C->Print(); *// виведення вмісту полів на екран*

**delete** C; *// звільнення пам'яті*

Під об'єкт необхідно окремо виділити пам'ять, відповідно її треба і звільнити.

1. ініціалізований динамічний об'єкт (рис. 3, г):

ТPoint \*D = **new** ТPoint (3, 4); */ \* оголошення покажчика на об'єкт, виділення пам'яті*

*і ініціалізація полів об'єкта \* /*

D-> Print (); *// виведення вмісту полів на екран*

**delete** D; *// звільнення пам'яті*

Обробка ініціалізованого об'єкта нічим не відрізняється від обробки неініціалізованого об'єкта, просто операції оголошення покажчика, виділення пам'яті і ініціалізації полів об'єкта поєднані в одному операторі.

Статичні і динамічні масиви об'єктів.

1. неініціалізований статичний масив об'єктів (рис. 4, а):

ТPoint c[4]; *// оголошення масива об'єктів*

**for** (**int** i=0; i<4; i++)

{ c[i].SetPoint(i\*i, i-5); *// ініціалізація полів*

c[i].Print(); *// виведення вмісту полів на екран*

}

Оскільки масив оголошується неініціалізованим, його об'єкти доводиться ініціалізувати, після чого вміст полів можна виводити. Проблем з пам'яттю немає.

|  |  |
| --- | --- |
|  |  |
| а) | б) |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| в) | г) | д) |

Рис. 1.4. Результати виконання операції оголошення масивів об'єктів:

а - неініціалізованих статичний масив об'єктів;

б – ініціалізований статичний масив об'єктів;

в - неініціалізованих динамічний масив об'єктів;

г - неініціалізованих статичний масив покажчиків на об'єкти;

д - ініціалізований статичний масив покажчиків на об'єкти

1. ініціалізований статичний масив об'єктів (рис. 4, б):

ТPoint d[2] = {ТPoint(2, 4), ТPoint(4, 5)}; */\* створення масива об'єктів*

*і ініціалізація їх полів \*/*

**for** (i=0; i<2; i++) d[i].Print(); *// виведення вмісту полів на екран*

Масив створюється ініціалізованим, вміст полів можна відразу виводити. Проблем з пам'яттю немає.

1. неініціалізований динамічний масив об'єктів (рис. 4, в):

ТPoint \*m=**new** ТPoint[3];

**for** (i=0; i<3; i++)

{ m[i].SetPoint(i, i+1);

m[i].Print();

}

**delete** [] m;

Пам'ять під динамічний масив виділяється одним шматком, отже і звільняти її треба також одним шматком.

1. неініціалізований статичний масив покажчиків на об'єкти (рис. 4, г):

ТPoint \*s[3];

**for** (i=0; i<3; i++)

{ s[i] =**new** ТPoint(i, i+1);

s[i]->Print();

}

**for**(i=0; i<3; i++) **delete** s[i];

В даному випадку сам масив покажчиків - статичний, тобто пам'ять (динамічну) під нього виділяти не треба. Пам'ять (динамічну) слід виділити під кожен з об'єктів, а по завершенні роботи звільнити її.

1. ініціалізований статичний масив покажчиків на об'єкти (рис. 4, д):

ТPoint \*q[] = { **new** ТPoint(2, 7), **new** ТPoint(1, 5), **new** ТPoint(4, 2)};

**for** (i=0; i<3; i++)

{ q[i]->Print();

}

**for** (i=0; i<3; i++) **delete** q[i];

У цьому випадку пам'ять під об'єкти також виділяється динамічно, а їх адреси заносяться в статичний масив покажчиків. Звільнення пам'яті виконується в циклі окремими шматками так, як вона виділялася.

## Константні члени класу і константні об'єкти

Як відомо, для захисту значень змінних стандартних типів від зміни їх оголошують константними. Для цього використовують ключове слово **const**. Аналогічним способом можна застосовувати модифікатор **const** і для членів класу, а також об’єктів класів.

### **Константні атрибути**

Для захисту від некоректної модифікації атрибути класу, як правило, поміщають у закритий (private) розділ і оголошують константними. Ключове слово const, яке вказують перед типом константного даного, інформує компілятор, що значення відповідного атрибута класу не повинне змінюватися після ініціалізації.

Константні атрибути класу не можна ініціалізувати у конструкторі звичайним способом. Єдиним методом ініціалізації даних-констант є список ініціалізації. Наприклад,

**class** ТPoint

{ **const** **char** name ; *// ім'я точки – константне дане-член*

**double** x, y ; *// координати точки*

**public**: *// відкриті члени класу*

ТPoint (**char** Name, **double** X, **double** Y) : name(Name), x(X), y(Y) { }

ТPoint (**char** Name) : name(Name), x(0), y(0) { }

} ;

### **Константні методи**

Одним із способів захисту атрибутів класу від змін є поділ методів класу на методи-селектори і методи-модифікатори. Методи-селектори не мають права змінювати значення атрибутів класу. Для цього їх оголошують константними. Методи-модифікатори можуть вільно присвоювати атрибутам нові значення.

Для того, щоб оголосити метод класу константним, необхідно вказати ключове слово **const** після прототипу функції, але до початку її тіла. Наприклад,

**class** ТPoint

{ **double** x, y ; *// координати точки*

**public** : *// відкриті члени класу*

ТPoint () { x = 0 ; y = 0 ; } *// ініціалізує координати нулями*

**double** Getx ( ) **const** { **return** x ; } *// константна функція повертає координату x*

**double** Gety ( ) **const** { **return** y ; } *// константна функція повертає координату y*

**void** Setx ( **double** X ) { x = X ; } *// встановлює задане значення координати x*

}

Якщо оголошення і визначення метода розділені, то модифікатор **const** необхідно вказати двічі – як при оголошенні метода, так і при його визначенні.

Коли константний метод описується зовні, а не всередині класу, то потрібно додати суфікс **const** наступним чином:

**inline** int Date::year() **const**

{ **return** y; }

Константний метод не тільки не може змінювати значення атрибутів класу, а й не може викликати неконстантні методи класу.

Таким чином, ті методи, які тільки зчитують дані з поля класу, є сенс робити константними, оскільки у них нема необхідності змінювати значення полів об’єктів класу.

Приклад використання константних методів.

**class** Distance

{ **int** feet;

**float** inches;

**public**:

Distance(): feet(0), inches(0.0) { } *//конструктор без аргументів*

Distance(**int** ft, **float** in): feet(ft), inches(in) { } *//конструктор з 2 aргументами*

**void** getdist()

{ **cout** <<"\**n**Введіть число футів "; **cin** >> feet;

**cout** << "дюймів "; **cin**>>inches;

}

**void** showdist() **const**

{ **cout** <<feet << "\' "<< inches <<"\''"; }

Distance add\_dist(**const** Distance) **const**; *// прототип*

};

Distance Distance::add\_dist(**const** Distance d2) **const** *//додавання d2 i d3*

{ Distance temp;

// feet=0; *//помилка не можна міняти поле*

// d2.feet=0; *//помилка не можна міняти поле*

temp.inches=inches+d2.inch*es;*

**if** (temp.inches>=12.0)

{ temp.inches-=12.0;

temp.feet++; }

temp.feet+=feet+d2.feet;

**return** temp;

}

**int** **main**()

{ **clrscr**();

Distance dist1, dist3;

Distance dist2(11, 6.25);

dist1.getdist();

dist3=dist1.add\_dist(dist2);

**cout** <<"\**n**dist1="; dist1.showdist();

**cout** << "\**n**dist2="; dist2.showdist();

**cout** << "\**n**dist3="; dist3.showdist();

**cout** <<**endl**;

**system**("pause");

}

В цьому прикладі обидві функції **showdist**() та **add\_dist**() є константними. В тілі функції **add\_dist**() перший із закоментованих операторів feet=0; демонструє, що компілятор видасть помилку при спробі зміни константною функцією поле об’єкта, з якого вона викликалася.

Константним можна оголосити і аргумент методу. В нашому прикладі це виглядає так:

Distance Distance::add\_dist(**const** Distance d2) **const**

Функція не може поміняти значення цього аргумента, що і видно на прикладі другого закоментованого оператора d2.feet=0.

За необхідності все ж таки можна змінити значення атрибута класу константним методом цього ж класу. Для цього відповідний атрибут слід специфікувати ключовим словом **mutable**. Наприклад,

**class** TComplex

{ **mutable** **double** Re;

**double** Im;

**public**:

TComplex(): Re(0), Im(0){ };

**void** change(**double** x) **const** { Re = x };

**void** print() { **printf**("Z = %**lf** + i\*%**lf** \**n**", Re, Im); }

} Z;

**int** **main**()

{ TComplex X;

X.print();

X.change(1.0);

X.print();

**system**("pause");

}

Результат роботи даної програми:

Z = 0.000000 + i\*0.000000

Z = 1.000000 + i\*0.000000

### **Константні об'єкти**

Аналогічним чином можна застосовувати модифікатор **const** і для створення константних об’єктів класів. Наприклад,

**const** Coord A(3, 5);

Ключове слово **const** інформує компілятор, що стан даного об'єкта не повинен змінюватися (об'єкт є недоступним для зміни). Як тільки для константного об'єкта викликається метод, який може змінити його атрибути, змінивши тим самим стан об'єкта, компілятор генерує повідомлення про помилку. Відповідно, для константного об’єкта можна викликати тільки константні методи, оскільки лише вони гарантують, що об’єкт не буде змінений.

Наприклад,

**class** Distance

{ **int** feet;

**float** inches;

**public**:

Distance(**int** ft, **float** in): feet(ft), inches(in) { } *//конструктор з 2 aргументами*

**void** getdist() *//неконстантний метод*

{ **cout** <<"\**n**Введіть число футів "; **cin** >> feet;

**cout** << "дюймів "; **cin**>>inches;

}

**void** showdist() **const** *//константний метод*

{ **cout** <<feet << "\' "<< inches <<"\''"; }

};

**int** **main**()

{ **clrscr**();

**const** Distance football(300, 0);

// football.getdist();

**cout** <<"Довжина поля=";

football.showdist();

**cout** <<**endl**;

**system**("pause");

}

У даній програмі об’єкт **football** оголошений як константний, тому для нього можна викликати тільки константні методи, наприклад, **showdist**(). Виклики неконстантних методів, таких як **getdist**(), є некоректними. Спроби зміни константного об'єкта відстежуються і присікаються компілятором.

Таким чином, константний метод можна викликати як для константного, так і для неконстантного об’єкта, тоді як неконстантний метод можна викликати тільки для об’єкта, який не є константим.

При створенні класу хороший стиль програмування вимагає оголошення, як константних, методів, що не змінюють атрибути об’єктів класу. Це дозволяє тому, хто використовує даний клас, створювати константні об’єкти класу, оскільки з цих об’єктів можуть викликатися тільки константні методи.

## Статичні члени класу

Члени класу (як методи, так і атрибути) можуть бути загальними (спільними) для усіх об'єктів даного класу. Тоді їх називають *статичними*.

### **Статичні атрибути**

*Статичні атрибути*використовуються для збереження даних, спільних для всіх екземплярів даного класу (значення яких є однаковими для всіх об'єктів даного класу). На відміну від інших членів, які присутні в кожному об'єкті як окремі елементи даних, статичні члени існують в єдиному екземплярі і пов'язані з самим класом, а не з конкретним його об'єктом. Це колективна сутність, доступна всім об'єктам одного класу.

Статичні атрибути також називають "змінними класу", оскільки вони є частиною класу, а не частиною об’єкта цього класу. Їхній вміст не залежить від об'єкта, в якому цей атрибут знаходиться.

Так лічильник числа створених об'єктів класу може існувати тільки у відриві від всіх об'єктів даного класу, хоча працювати з ним, зазвичай, доводиться одночасно з викликом методів, наприклад, конструкторів об'єктів. Тому описувати лічильник зручніше саме як елемент класу. Статичні атрибути зручно використовувати не тільки як лічильники об'єктів, а також як індикатори на зразок «зайнято» — «вільно». Об'єкти можуть по черзі змінювати значення статичного атрибута, повідомляючи своїм спадкоємцям, що вони виконали свою місію. Також, якщо в класі є деякі константи, однакові для всіх об'єктів класу, то нераціонально зберігати в кожному об'єкті власні копії цих констант. Спільним може бути покажчик на функцію обробки помилок для класу або, скажімо, покажчик на вільну пам'ять для його об'єктів тощо.

Статичний атрибут за своїми характеристиками нагадує статичну змінну: він видимий тільки всередині класу, але час його життя співпадає з часом життя програми. Таким чином, статичний атрибут існує навіть тоді, коли не існує жодного об’єкта класу. Тим не менше, на відміну від статичної змінної, призначеної для збереження значень між викликами функції, статична змінна класу використовується для зберігання даних, які спільно використовуються об’єктами класу.

Ство­рен­ня статичних атрибутів виконується в три етапи:

* оголошення (декларація);
* визначення (виділення пам’яті);
* ініціалізація.

При цьому оголошення статичних атрибутів здійснюється в тілі класу, а їх визначення – за межами класу. Визначення статичних атрибутів поза класом обумовлюється необхідністю виділити для них пам'ять до початку роботи програми, як глобальним змінним. Тільки тоді усі об'єкти класу посилатимуться на одне й те ж саме місце у пам'яті.

Це реалізується шляхом визначення статичних атрибутів в області визначення глобальних змінних (поза тілом класу), що призведе до резервування пам'яті для них на етапі компіляції. В іншому випадку на етапі збірки виконуваного модуля буде видана помилка про недозволене зовнішнє посилання, оскільки при обробці опису класу до створення конкретних об'єктів ніякі області пам'яті не відводяться.

Формат оголошення статичного атрибута передбачає використання службового слово **static**:

**static** тип змінна;

Тип статичного атрибута може бути будь-який: це можуть бути константні об'єкти, масиви, об'єкти класів і т.ін. Як і звичайні атрибути, вони можуть бути відкритими (**public**), закритими (**private**) або захищеними (**protected**).

Особливістю визначення статичного атрибута є те, що повторювати ключове слово **static** при цьому не можна. Воно повинно бути присутнім *тільки* в тілі класу. Наприклад,

**class** ТPoint

{ **int** x, y;

**public**:

**static int** n; *// оголошення статичного атрибуту n (кількість точок)*

ТPoint(**int** x1, **int** y1) { x=x1; y= y1; n++; }

**int** SetX(int x1) { x=x1; }

**int** SetY(int y1) { y= y1; }

**int** GetX() { **return** x; }

**int** GetY() { **return** y; }

~ТPoint() { n--; }

};

**int** ТPoint::n; *// визначення статичного атрибуту n*

Ініціалізація статичних атрибутів відбувається так само, як і глобальних змінних. Наприклад,

**int** ТPoint::n = 10;

За замовчуванням статичні атрибути ініціалізуються нулями.

Ініціалізацію статичних атрибутів можна поєднати з їх визначенням. Формат такого визначення:

тип\_змінної ім'я\_класу :: ідентифікатор [= ініціалізатор];

Слід відзначити, що константний статичний атрибут цілого типу, ініціалізований константою, - це константний вираз. Проектувальник може оголосити такий статичний атрибут, якщо всередині тіла класу виникає необхідність в іменованій константі.

Статичні атрибути можна визначати та ініціалізувати і у випадку, коли жоден об'єкт ще не створено.

У програмі може бути тільки визначення статичного атрибута. Це означає, що ініціалізацію таких атрибутів слід поміщати не в заголовні файли, а туди, де знаходяться визначення вбудованих методів класу.

Доступ до відкритого статичного атрибута може бути організований як звичайним способом (через об'єкт), так і за допомогою кваліфікованого імені «::» (через клас).

Наприклад, використання статичного атрибута в якості лічильника, який містить кількість створених екземплярів об'єктів цього класу:

**class** CDummy

{ **public**:

**static int** n; *// статичне поле класу*

CDummy () { n++; } *// конструктор*

~CDummy () { n--; } *// деструктор*

};

**int** CDummy::n=0; *// визначаємо статичне поле класу*

**int main**()

{ CDummy a;

CDummy b[5];

CDummy \*c = **new** CDummy;

**cout** << a.n << **endl**;

**delete** c;

**cout** << CDummy::n << **endl**;

**system**("pause");

}

Тут конструкції **a.n** і **CDummy::n** звертаються до однієї тієї самої змінної: до статичного атрибуту **n**, який належить до класу **CDummy**, і є спільним для всіх об'єктів цього класу.

Пряме звертання до приватних статичних атрибутів є забороненим. Для організації доступу до них вико­рис­товують три основні методи:

* через відкритий метод класу;
* через дружній клас чи дружню функцію;
* через визначення та ініціалізацію.

Наприклад,

**class** CDummy

{ **static int** n; *// статичне поле класу*

**public**:

CDummy () { n++; } *// конструктор*

~CDummy () { n--; } *// деструктор*

**int** count() { **return** n; }

};

**int** CDummy::n=0; *// визначаємо статичне поле класу*

**int main**()

{ CDummy a;

CDummy b[5];

**cout** << a. count() << **endl**;

**system**("pause");

}

Унікальна особливість статичного атрибута - те, що він існує незалежно від об'єктів класу, - дозволяє використовувати його такими способами, які для нестатичних атрибутів є неприпустимими. Зокрема, статичний атрибут може належати до типу того ж класу, членом якого він є; нестатичні ж - оголошуються лише як покажчики або посилання на об'єкт свого классу. Наприклад,

**class** Bar

{ **static** Bar mem1; *// правильно*

 Bar \*mem2; *// правильно*

 Bar mem3; *// помилка*

**public**:

// ...

};

Також статичний атрибут може виступати в ролі аргументу за замовчуванням для метода класу, а для нестатичних це заборонено. Наприклад,

**extern int** var;

**class** Foo

{ **int** var;

**static int** stcvar;

**public**:

**int** mem1(int =var); *//Помилка: трактується як Foo::var, але асоційованого об'єкта класу не існує*

**int** mem2 (int =stcvar);*// Правильно: трактується як static Foo :: stcvar,*

*// асоційований об'єкт і не потрібний*

**int** mem3 (int = ::var); *// Правильно: трактується як глобальна змінна var*

};

Розглянемо приклад, у якому змінні **Re** і **Im** оголошені статичними атрибутами класу **TComplex**.

**class** TComplex

{ **static double** Re;

**static** **double** Im;

**public**:

**void** print();

**void** init();

};

**double** TComplex::Re=0.0;

**double** TComplex::Im=0.0;

**int main**()

{ TComplex Z;

**printf**("sizeof = %d \n", **sizeof**(Z));

Z.init(); *// виклик функції-члена init()*

Z.print(); *// виклик функції-члена print()*

**system**("pause");

}

**void** TComplex::print() { **printf**("Z = %**f** + i\*%**f** \**n**", Re, Im); }

**void** TComplex::init() { Re = 1; Im = 2; }

Цікаво, що розмір класу **TComplex** дорівнює лише одному байту (як у зовсім порожнього класу).

Розглянемо програму, у якій створюються і виводяться на екран 25 комплексних чисел.

**class** TComplex

{ **double** Re;

**double** Im;

**public**:

**static** **int** counter;

**void** print();

**void** init(**double** x, **double** y);

};

**int** TComplex::counter=0;

**int main**()

{ **for** (**int** i=1; i<=5; i++)

**for** (**int** j=1; j<=5; j++)

{ TComplex Z;

Z.init((**double**)i, (**double**)j); *// виклик метода init()*

Z.print(); *// виклик метода print()*

TComplex::counter++; *// інкрементація статичного атрибуту*

}

**printf**("Counter = %**d**\**n**", TComplex::counter);

**system**("pause");

}

**void** TComplex::print() { **printf**("Z = %**lf** + i\*%**lf** \**n**", Re, Im); }

**void** TComplex::init(**double** x, **double** y) { Re = x; Im = y; }

Тут слід звернути увагу на синтаксичну конструкцію, прийняту для звертання до статичного атрибуту: **TComplex::counter**. Це підкреслює, що статичний атрибут не належить жодному об'єкту.

Для того щоб підкреслити, що кожен об'єкт має доступ до статичного атрибуту, можна оператор TComplex.counter++ замінити оператором Z.counter++.

Переваги статичних атрибутів у порівнянні з глобальними:

* статичний атрибут не знаходиться в глобальному просторі імен програми, отже, зменшується ймовірність випадкового конфлікту імен з іншими глобальними об'єктами;
* залишається можливість приховування інформації, оскільки статичний атрибут-об'єкт може бути закритим, а глобальний об'єкт - ніколи.

Використання статичних атрибутів скорочує витрати пам'яті і гарантує єдність даних у всіх об'єктах. Хоча статичні атрибути класу використовуються значно рідше, ніж автоматичні, однак існують ситуації, коли їх зручно використовувати.

### **Статичні методи**

Статичними можуть бути не тільки атрибути класу, але і його методи. *Статичні методи* – це методи, які є спільними для усіх об'єктів даного класу.

Статичні методикласу вирізняються тим, що мають доступ тільки до статичних членів (атрибутів і методів) даного класу. Це обумовлено тим, що, на відміну від інших функцій-членів, статичні методи мають глобальну область видимості і не прив'язані до жодного об'єкта класу. Як наслідок, їм не передається неявний покажчик this на поточний об'єкт. Тому явне або неявне звернення до покажчика this всередині тіла статичного методу викликає помилку компіляції. З цієї причини статичні методи не мають прямого доступу до нестатичних атрибутів класу (оскільки спроба звернення до нестатичного члена класу неявно вимагає наявності покажчика **this**) і не можуть викликати нестатичні методи (оскільки останні мають доступ до даних конкретних об'єктів). Однак, в останньому випадку зворотнє допустимо: нестатичні методи можуть викликати статичні методи.

Те, що статичні методи не прив'язані до об'єкта класу, дозволяє використовувати їх для отримання «загальної» інформації про об'єкти класу.

Оголошення статичного методу, як і статичного атрибуту, специфікується службовим словом **static**:

static тип\_результату ім'я\_класу :: ім'я\_функції ([ параметри]);

Специфікатор **static** вказується лише у декларації статичного методу. У визначенні метода (поза тілом класу) він не повторюється.

Статичні методи класу не можуть бути константними (не повинні мати кваліфікаторів **const**), але можуть бути **inline**-функціями.

Статичний метод класу може створювати об'єкти даного і будь-якого іншого класу. Це можна використовувати, зокрема, якщо необхідно в програмі заборонити створення об'єктів простим оголошенням або з використанням операції **new**. У цьому випадку конструктор і деструктор поміщаються в закриту область класу, а для створення і знищення об'єкта використовуються спеціальні статичні методи, які можна викликати, не маючи жодного об'єкту.

Оскільки статичний метод не використовує специфічного вмісту конкретного об'єкта, то звернення до нього може здійснюватися як через ідентифікатор об'єкта, так і через ідентифікатор класу:

ім'я\_класу :: імя\_функції(фактичні\_параметри);

Звертання до статичних методів можливе навіть тоді, коли ще не створено жодного об'єкту даного класу. У цьому випадку статичний метод можна викликати, кваліфікувавши його ім'ям класу (через «::»).

Слід зазначити, що застосування статичних методів обмежене ініціалізацією статичних атрибутів, яку необхідно виконати до створення реальних об'єктів.

Одне з очевидних застосувань статичних методів – робота з глобальними об'єктами і статичними атрибутами відповідного класу.

Розглянемо приклад, у якому для виведення значення статичного лічильника застосовується статична функція TComplex::HowMuch().

**class** TComplex

{ **double** Re;

**double** Im;

**public**:

**static** **int** counter;

**void** print();

**void** init(**double** x, **double** y);

**static void** HowMuch();

};

**int** TComplex::counter=0;

**int main**()

{ **for** (**int** i=1; i<=5; i++)

**for** (**int** j=1; j<=5; j++)

{ TComplex Z1;

Z1.init((**double**)i, (**double**)j); *// виклик метода init()*

Z1.print(); *// виклик метода print()*

TComplex::counter++; *// інкрементація статичного лічильника*

}

TComplex::HowMuch(); // Виклик статичної функції

**system**("pause");

}

**void** TComplex::print() { **printf**("Z = %**lf** + i\*%**lf** \**n**", Re, Im); }

**void** TComplex::init(**double** x, **double** y) { Re = x; Im = y; }

**void** TComplex::HowMuch() { **printf**("Counter = %**d**\**n**", TComplex::counter); }

## Дружні функції і дружні класи

### **Дружні функції**

Зазвичай, оголошення методів гарантує три логічно різні речі:

* метод має право доступу до закритої частини оголошення класу;
* метод знаходиться в області видимості класу;
* метод повинен викликатися для об'єкта класу.

Однак, у деяких випадках бажано забезпечити доступ до закритих (private) членів класу функціям, які не є членами даного класу. Це можна зробити, оголосивши відповідну функцію як «друга» цього класу.

*Дружньою*[*функцією*](http://ua-referat.com/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%97) класу називають функцію, яка, не будучи членом класу, має доступ до усіх його членів, у тому числі і захищених.

Щоб зробити функцію "другом" класу, потрібно в оголошення класу помістити її прототип зі специфікатором **friend**. Наприклад, у наведеному нижче фрагменті коду функція **Fun**() оголошується "другом" класу **myClass**:

**class** myClass

{ //...

**public**:

**friend** **void** Fun(myClass obj); *// "дружня" функція класу* *myClass*

//...

};

Розглянемо програму, що підсумовує два комплексних числа.

**class** TComplex

{ **double** Re;

**double** Im;

**public**:

**static int** counter;

TComplex(**double** x, **double** y) : Re(x), Im(y) { counter++; **printf**("Ctor %**d**\**n**", counter); }

TComplex(TComplex& z) { Re = z.Re; Im = z.Im; counter++; **printf**("Copy %**d**\**n**", counter); }

~TComplex() { **printf**("Dtor %**d**\**n**",counter); counter--; }

**double** getRe() { **return** Re; }

**double** getIm() { **return** Im; }

**void** putRe(**double** x) { Re =x; }

**void** putIm(**double** y) { Im = y; }

};

TComplex sum(TComplex,TComplex);

**int** TComplex::counter;

**int main**()

{ **printf**("Start main\n");

TComplex z(1,1), w(2,2), u(0,0);

u = sum(z,w); **printf**("End sum\n");

**printf**("w + u = %f + i\*%f\n",u.getRe(),u.getIm());

**printf**("End main\n");

**system**("pause");

}

TComplex sum(TComplex u, TComplex v)

{ **printf**("Start \n"); TComplex w(0,0);

w.putRe(u.getRe() + v.getRe());

w.putIm(u.getIm() + v.getIm());

**return** w;

}

У цій програмі для додавання двох комплексних чисел використовуються чотири інтерфейсні функції: **getRe**(), **getIm**(), **putRe**(), **putIm**(). Однак, програма стала б простішою, якби функція **sum**() мала прямий доступ до полів класу **TComplex**.

Для цього можна оголосити функцію **sum**() дружньою до класу **TComplex**. Наприклад,

**class** TComplex

{ **double** Re;

**double** Im;

**public**:

**static int** counter;

TComplex(**double** x, **double** y) : Re(x), Im(y) { counter++; **printf**("Ctor %**d**\**n**", counter); }

TComplex(TComplex& z) { Re = z.Re; Im = z.Im; counter++; **printf**("Copy %**d**\**n**", counter); }

~TComplex() { **printf**("Dtor %**d**\**n**",counter); counter--; }

**void print**() { **printf**("Z = %f + i\*%f\n", Re, Im); }

**friend** TComplex sum(TComplex, TComplex); *// "дружня" функція класу*

};

TComplex sum(TComplex,TComplex);

**int** TComplex::counter;

**int main**()

{ **printf**("Start main\n");

TComplex z(1,1), w(2,2), u(0,0);

u = sum(z,w); **printf**("End sum\n");

u.print(); **printf**("End main\n");

**system**("pause");

}

TComplex sum(TComplex u, TComplex v)

{ **printf**("Start \n");

TComplex w(0,0);

w.Re= u.Re + v.Re;

w.Im=u.Im + v.Im;

**return** w;

}

У даному прикладі функція **sum**() не є членом класу **TComplex**. Проте вона має повний доступ до **private**-членів цього класу. Зокрема, вона може безпосередньо використовувати значення u.Re, v.Im тощо. При цьому функція **sum**() викликається звичайним способом, тобто без прив'язування до імені об'єкта (і без використання оператора "."). Оскільки вона не є методом класу, то під час виклику її не потрібно кваліфікувати вказанням імені об'єкта.

Зазвичай "дружній" функції, як параметр, передається один або декілька об'єктів класу, для яких вона є "другом". Робиться це так само, як і у випадку передачі параметра функції **sum**().

"Дружня" функція одного класу може бути членом іншого класу. Наприклад, розділимо клас **TComplex** на два: **TNumber** і **TFunction**. У першому класі залишимо атрибути, а до другого віднесемо усі методи.

**class** TNumber; *// випереджувальне оголошення класу*

**class** TFunction

{ **public**:

**void print**(TNumber z);

};

**class** TNumber

{ **double** Re;

**double** Im;

**public**:

**static int** counter;

TNumber (**double** x, **double** y) : Re(x), Im(y) { counter++; **printf**("Ctor %**d**\**n**", counter); }

TNumber (TNumber & z) { Re = z.Re; Im = z.Im; counter++; **printf**("Copy %**d**\**n**", counter); }

~ TNumber() { **printf**("Dtor %**d**\**n**", counter); counter--; }

**friend void** TFunction::print(TNumber); *// "дружня" функція класу*

};

**int** TNumber::counter;

**int main**()

{ **printf**("Start main\n");

TNumber z(1,1);

TFunction a;

a.print(z); **printf**("End main\n");

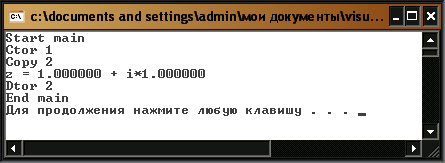
**system**("pause");

}

*// член класу TFunction і "друг" класу TNumber*

**void** TFunction::print(TNumber z) { **printf**("z = %f + i\*%f\n", z.Re, z.Im); }

У результаті одержуємо наступні повідомлення:



У даному прикладі функція **print**(), що належить класу **TFunction**, одержує параметр типу **TNumber**. Отже, цей тип повинен бути визначений раніше. Однак клас **TNumber** оголошує дружню функцію **print**(), що належить класу **TFunction**. Для розв’язання цієї проблеми в мові С++ передбачений *механізм неповного оголошення класу*.

Працюючи з дружніми функціями, потрібно враховувати наступні особливості:

* для дружньої функції несуттєві специфікації доступу;
* дружня функція при виклику не отримує покажчика **this**;
* об'єкти класів повинні передаватися дружній функції тільки явно через апарат параметрів;
* дружні функції не можуть мати специфікатори **static** і **extern**;
* у класі, функція-член якого є дружньою до не цілком оголошеного класу, слід розміщати тільки прототип; реалізація функції повинна знаходитися після повного оголошення класу;
* при виклику дружньої функції не можна використовувати операції вибору:

імя\_об'екта.імя\_функціі

указатель\_на\_об'ект-> імя\_функціі

* дружні функції не взаємні;
* одна і та ж функція може бути "другом" декількох класів;
* дружні функції не успадковуються.

Існує ряд обставин, при яких статус "дружньої" функції класу має велике значення. По-перше, дружні функції є корисними для перевизначення операторів певних типів. По-друге, вони спрощують створення деяких функцій введення-виведення. Третя причина частого використання функцій-"друзів" полягає у тому, що в деяких випадках два (або більше) класи можуть містити члени, які перебувають у взаємному зв'язку з іншими частинами програми.

Сфери використання дружніх функцій:

* одна і та сама функція є елементом двох різних класів;
* необхідне неявне перетворення типів.

### **Дружні класи**

Коли усі функції деякого класу є дружніми стосовно іншого класу, можна оголосити весь клас ***дружнім***. Наприклад,

**class** C

{ ...

**friend class** A;

**…**

}

Всі функції класу **A** мають доступ до закритих полів класу **C**. При цьому дружба не носить "наскрізного" [характеру](http://ua-referat.com/%D0%A5%D0%B0%D1%80%D0%B0%D0%BA%D1%82%D0%B5%D1%80) (не володіє властивістю транзитивності): якщо клас **A** [друг](http://ua-referat.com/%D0%94%D1%80%D1%83%D0%B3) класу **B**, а клас **B** друг класу **C**, то це не означає, що **A** - друг класу **C**.

Переробимо попередню програму в такий спосіб:

**class** TNumber; *// випереджувальне оголошення класу*

**class** TFunction

{ **public**:

**void print**(TNumber z); *// апеляція до не цілком оголошеного класу*

};

**class** TNumber

{ **double** Re;

**double** Im;

**public**:

**static int** counter;

TNumber (**double** x, **double** y) : Re(x), Im(y) { counter++; **printf**("Ctor %**d**\**n**", counter); }

TNumber (TNumber & z) { Re = z.Re; Im = z.Im; counter++; **printf**("Copy %**d**\**n**", counter); }

~ TNumber() { **printf**("Dtor %**d**\**n**", counter); counter--; }

**friend class** TFunction; *// оголошення дружнього класу*

};

**int** TNumber::counter;

**int main**()

{ **printf**("Start main\n");

TNumber z(1,1);

TFunction a;

a.print(z); **printf**("End main\n");

**system**("pause");

}

**void** TFunction::print(TNumber z) { **printf**("z = %f + i\*%f\n", z.Re, z.Im); }

Варто мати на увазі, що члени класу **TFunction** просто мають доступ до членів класу **TNumber**, але члени класу **TNumber** від цього не стають членами класу **TFunction**.

Іноді два класи настільки тісно зв'язані, що їм необхідний повний доступ до всіх членів. У цьому випадку їх оголошують *взаємно дружніми*. Наприклад,

**class** TNumber; *// випереджувальне оголошення класу*

**class** TFunction

{ **void print**(TNumber z);

**friend class** TNumber;

};

**class** TNumber

{ **double** Re;

**double** Im;

**public**:

**static int** counter;

TNumber (**double** x, **double** y) : Re(x), Im(y) { counter++; **printf**("Ctor %**d**\**n**", counter); }

TNumber (TNumber & z) { Re = z.Re; Im = z.Im; counter++; **printf**("Copy %**d**\**n**", counter); }

~ TNumber() { **printf**("Dtor %**d**\**n**", counter); counter--; }

**void** sum(TNumber y, TFunction z) { Re = Re + y.Re; Im = Im + y.Im; z.print(\***this**); }

**friend class** TFunction; *// оголошення дружнього класу*

};

**int** TNumber::counter;

**int main**()

{ **printf**("Start main\n");

TNumber n1(1,1), n2(1,1);

TFunction f;

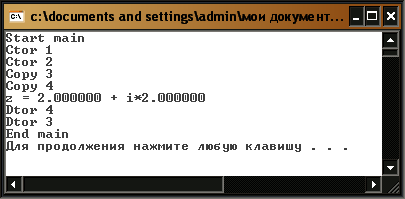
n1.sum(n2, f); **printf**("End main\n");

**system**("pause");

}

**void** TFunction::print(TNumber z) { **printf**("z = %f + i\*%f\n", z.Re, z.Im); }

У ході виконання цієї програми на екрані з'являться наступні рядки:



Як бачимо, функція **sum**(), що є членом класу **TNumber**, одержує вільний доступ до функції **print**(), оголошеної в закритому розділі класу **TFunction**. Якби класи **TNumber** і **TFunction** не були взаємно дружніми, це було б неможливо

## Структури і об'єднання як різновиди класів

### **Класи і структури**

Синтаксис класу в С++ збігається з синтаксисом структури С++:

**struct** ім'я\_структури

{ визначення\_членів\_структури;

}

По суті, класи і структури є спорідненими типами. За одним винятком, вони взаємозамінні, оскільки структура також може містити дані та програмні коди, які маніпулюють цими даними так само, як це може робити і клас. Єдина відмінність між С++−структурою і С++−класом полягає у тому, що за замовчуванням члени класу є закритими (**private**), а члени структури – відкритими (**public**).

Розглянемо приклад використання структури з властивостями, подібними до властивостей класу:

**struct** myStruct *// оголошення структурного типу*

{ **int** Get (); *// відкриті члени (за замовчуванням)*

**void** Set (**int** d);

**private**:

**int** c;

};

**int** myStruct:: Get ()

{ return c;

}

**void** myStruct:: Set (**int** d)

{ c = d;

}

**int** **main**()

{ myStruct Obj; *// створення об'єкта- структури*

Obj. Set (10);

**cout** << "c= " << Obj.Get () << **endl**;

**system**("pause");

}

У цьому коді програми визначається тип структури з іменем **myStruct**, у якій функції-члени **Set** () і **Get**() є відкритими (**public**), а член даних **c** – закритим (**private**).

У наведеному нижче прикладі показано еквівалентну програму, яка використовує замість типу **struct** тип **class**:

**class** myClass

{ **int** c; *// закритий член (за замовчуванням)*

**public**:

**int** Get ();

**void** Set (**int** d);

};

**int** myClass::Get () { return c; }

**void** myClass:: Set (**int** d) { c = d; }

**int** **main**()

{ myClass Obj; *// створення об'єкта класу*

Obj. Set (10);

**cout** << "c= " << Obj.Get() << **endl**;

**system**("pause");

}

С++−програмісти тип **class** використовують в основному для визначення форми об'єкта, який містить функції-члени, а тип **struct** – для побудови об'єктів, які містять тільки члени даних. Об'єкти, які не містить функцій-членів, іноді називають POD-struct (*Plain Old Data*).

Той факт, що і структури, і класи мають практично однакові можливості, створює враження зайвості. Причиною ж такого очевидного дублювання є намір зберегти мову програмування C++ сумісною з мовою С. У мові С, яка не містить ключових слів **public** або **private**, всі члени структури є відкритими. Ось чому і члени С++−структур за замовчуванням є відкритими (а не закритими). Оскільки ж конструкція типу **class** спеціально розроблена для підтримки інкапсуляції, то є певний сенс у тому, щоб за замовчуванням її члени були закритими.

Структури нарівні з класами можна використовувати для повноцінного опису типів даних, що включає описи операцій, які застосовуються до описуваного типу даних. Однак наявність інформаційних членів у відкритій секції порушує один з основних принципів ООП - принцип інкапсуляції.

### **Об'єднання та класи**

Той факт, що структури і класи – споріднені, зазвичай нікого не дивує; проте об'єднання також пов'язані з класами "близькими відносинами". Згідно з визначенням мови програмування C++, об'єднання – це, по суті, аналогічний клас, у якому всі члени даних зберігаються в одній і тій самій області. Об'єднання може містити конструктор і деструктор, а також функції-члени. Звичайно ж, члени об'єднання за замовчуванням відкриті (**public**), а не закриті (**private**).

Розглянемо код програми, у якій об'єднання використовують для відображення вмісту старшого і молодшого байтів короткого цілочисельного значення (передбачається, що короткі цілочисельні значення займають в пам'яті два байти).

**union** myUnion *// оголошення типу об'єднання*

{ myUnion(**short int** a); *// відкриті члени (за замовчуванням)*

**void** Show();

**short int** c;

**char** ch[2];

};

myUnion::myUnion(**short int** a) *// оголошення параметризованого конструктора*

{ c = a;

}

**void** myUnion::Show() *// відображення символів значення типу short int*

{ **cout** << ch[0] << " " << ch[1] << **endl**;

}

**int** **main**()

{ myUnion Obj(1000) *// створення об'єкта- об'єднання та його ініціалізація*

Obj.Show();

**system**("pause");

}

Подібно до структури, С++−об'єднання також виникло від свого С−попередника. Але у мові програмування C++ воно має ширші "класові" можливості. Проте тільки те, що мова програмування C++ наділяє "свої" об'єднання могутнішими засобами і більшою гнучкістю, зовсім не означає, що неодмінно треба їх використовувати. Якщо програміста цілком влаштовує об'єднання з традиційним стилем запису, то саме так його і треба використовувати. Але у випадках, коли можна інкапсулювати ці об'єднання разом з функціями, які їх обробляють, все ж таки варто скористатися С++−можливостями, що додасть програмі додаткові переваги.