Лекція 14. Класи (продовження)

1. *Перевантаження операторів.*
2. *Перетворення типу. Явні конструктори.*
3. *Коли необхідно оголошувати конструктор копіювання і оператор присвоєння.*

Продовжимо знайомство з класами C++. Головне гасло мови стосовно класів таке: *клас – це новий тип даних*. Це означає, що клас повинен мати такі ж властивості, як вбудований тип. Програміст може створювати змінні та константи нового типу, оголошувати масиви, передавати їх у функції як параметри та отримувати їх звідти як результат – усе це вже було в попередній лекції. Одна з відмінностей класу від вбудованого типу полягає в тому, що не існує способу оголошення літералу значення нового типу. Наприклад, для типу *int* літералами є 2019, –37; для типу *double* – 12.35, 1.e–6 тощо. Роль літералів для класу виконують його конструктори. Саме тому є сенс оголошувати декілька конструкторів з різними наборами параметрів, щоб можна було різними способами створювати екземпляри класу.

Які ще можливості класів ми не розглянули?

(Для вбудованих типів визначено оператори)&(клас використовують як вбудований тип)⇒ (треба вміти визначати оператори для класів). Точніше, перевантажувати визначені мовою оператори для новостворених класів. Таке завдання вирішують одним з трьох способів: 1) визначають у класі метод-оператор, або 2) дружню функцію-оператор, або 3) зовнішню функцію-оператор. Розглянемо всі три способи. Допоможе нам простенький клас, що моделює тривалість у годинах і хвилинах. (У попередніх лекціях ми використовували схожу структуру.)

```
class Time
{
private:
     int hours; int minutes;
public:
     Time();
     Time(int h, int m = 0);
     int hour() const { return hours; }
     int min() const { return minutes; }
     void show(std::ostream& os) const;
     Time operator+(const Time& t) const;
};
```

Тут оголошено метод-оператор додавання двох об'єктів класу *Time*. Наведемо його визначення та ще, мабуть, методу *show*. (Конструктори визначте самостійно.) Метод-оператор має тільки один параметр – це правий доданок. Лівим доданком є екземпляр, який виконує метод. Обидва екземпляри захищені від випадкових змін модифікатором *const*.

```
Time Time::operator+(const Time& t) const
{
     int sum = this->minutes + t.minutes;
     return Time(this->hours + t.hours + sum/60, sum%60);
}
void Time::show(std::ostream& os) const
{
     os << hours << " hours, " << minutes << " minutes";
}
```

Тепер обчислення суми тривалостей може мати такий вигляд:

```
Time A(2,35); Time B(3,42); Time C, D;
C = A + B; /*або*/ D = B + A; C.show(cout); ...
```

Компілятор перетворить другий рядок цього коду до вигляду

```
C = A.operator+(B); D = B.operator+(A); C.show(); ...
```

Очевидно, що об'єкти *C* і *D* зображатимуть однакові тривалості – результат додавання в обох випадках буде однаковим. Різниця лише в тому, хто з доданків є отримувачем повідомлення, а хто аргументом.

Правила перевантаження:

1. Перевантажувати можна лише оператори, визначені в мові C++, а саме + – * / % ^ & | ~ ! = < > + = –= *= /= %= ^= &= |= << >> >>= <<= == != <= >= && || ++ –– –>* , –> [] () new new[] delete delete[].
2. Хоча б один операнд має тип, визначений користувачем. Пріоритет і арність оператора зберігаються (як визначено мовою). Неможливо визначити новий оператор.
3. Не перевантажують . .* :: ?: typeid const_cast dynamic_cast reinterpret_cast static_cast.
4. Оператори = () [] –> перевантажують тільки функціями-членами.

Для нашого прикладу ми могли б вибрати будь-яке з наведених тут позначень операторів, але було б не дуже розумно позначати додавання тривалостей мінусом, зірочкою чи ще якось. Синтаксис не накладає таких обмежень, але семантика перевантажених операторів має бути добре зрозумілою.

Рухаємося далі. Що таке дружня функція, і як вона може допомогти нам з оператором додавання?

Дружніми класові називають зовнішні стосовно класу структури, наділені такими ж правами доступу до прихованої частини, як і всі члени класу. (Дружні структури «бачать», зокрема, *private*-поля.) Дружніми можуть бути функції, класи, методи.

Прототип дружньої функції оголошують в класі, визначають – поза ним, наприклад:

```
class Time
{
private:    .....
public:     .....
     friend Time operator+(const Time&, const Time&);
};

Time operator+(const Time& a, const Time& b)
{
     int sum = a.minutes + b.minutes;
     return Time(a.hours + b.hours + sum/60, sum%60);
}
// код C = A + B; компілятор замінить на C = operator+(A,B);
```

Компіляторові байдуже, який спосіб ви виберете. В обох випадках ви отримаєте однаково ефективний (чи ні) код. Зауважимо, що використання дружньої функції не порушує концепції приховування даних: дружня функція оголошена в інтерфейсі класу – в тому місці, де оголошують всіх, хто має доступ до полів.

Клас *Time* має методи для читання своїх полів, тому ми могли б використати і третій спосіб перевантаження оператора:

```
Time operator+(const Time& a, const Time& b)
{
     int m = a.min() + b.min();
     int h = a.hour() + b.hour() + m / 60;
     m %= 60; return Time(h,m);
}
```

Це зовнішня функція, клас ніяк не пов'язаний з її визначенням. Визначають її після оголошення класу, використовують не менш успішно, як два попередні варіанти. Не забудьте тільки, що на практиці треба зупинити свій вибір на одному з трьох запропонованих способів.

Давайте продовжимо наші вправи з перевантаженням операторів. Було б добре зайнятися виведенням об'єкта і замість нестандартного методу *Time::show* використовувати щось на зразок *cout<<time_instance*. Як визначити такий оператор? Щоб навчити об'єкт *cout* працювати з екземплярами класу *Time*, довелося б втрутитися у визначення класу *ostream*, а це нерозумно. Поширеною практикою є перевантаження оператора << за допомогою дружньої функції:

```
class Time
{
private:    .....
public:     .....
     friend ostream& operator<<(ostream&, const Time&);
};

ostream& operator<<(ostream& os, const Time& t)
{
     os<<t.hours<<" hours, "<<t.minutes<<" minutes";
     return os;
}
```

Для класу *Time* ми могли б використати і третій спосіб, оголосивши зовнішню функцію без зміни інтерфейсу класу:

```
ostream& operator<<(ostream& os, const Time& t)
{
     t.show(os);
     return os;
}
```

Ці функції не випадково повертають посилання на аргумент – потік виведення. Зроблено так для того, щоб ми могли, як звичайно, об'єднувати декілька повідомлень про виведення в один ланцюжок:

```
Time C = A + B; cout << "C = " << C << '\n';
```

То який спосіб перевантаження оператора виведення обрати? Скористаємося такою рекомендацією: якщо можна обійтися без дружніх функцій – обходьтеся! Потрібно віддати перевагу останньому наведеному варіантові: зовнішній функції без статусу дружньої, яка використовує доступний метод *show*. Це справді надійний спосіб, адже всю роботу виконує метод класу (а це і є його відповідальність!), а функція лише змінює спосіб виклику цього методу, робить виклик зручнішим, приводить його до традиційного способу виведення.

Давайте тепер відповімо на запитання, як визначити оператор *множення тривалості на число*? (1 год. 45 хв. × 2 = 3 год. 30 хв.) Спробуємо оголосити метод-оператор:

```
class Time
{private:   .....
 public:    .....
     Time operator*(double) const;
};

Time Time::operator*(double d) const
{    Time res;
     res.minutes = minutes * d;
     res.hours = hours * d + res.minutes / 60;
     res.minutes %= 60; return res;
}
```

Це працюватиме для виразів такого вигляду:

```
Time A(1,45); Time C = A * 2.0; // компілятор замінить на A.operator*(2.0)
```

Але з наступним рядком виникнуть проблеми:

```
Time A(1,45); Time C = 2.0 * A; /* компілятор спробує використати operator*,
     визначений для вбудованого типу double, адже тут об'єкт не є отримувачем
*/
```

Для того, щоб зробити можливим *множення числа на тривалість*, продовжимо перевантаження оператора *.

```
class Time
{
private:    .....
public:     .....
     Time operator*(double) const;
     ......
};

Time operator*(double d, const Time& t)
{
     return t * d;
}
```

Таке визначення локалізує відповідальність за правильне множення в методі класу.

Достатня кількість перевантажених операторів збільшує розмір коду, але забезпечує його ефективність.

Розглянемо ще одну можливість вирішення проблеми, що реалізує дії схожі до поведінки вбудованих типів. У виразі 2.5 + 1 другий доданок типу *int* буде автоматично перетворено до типу *double*. Мова C++ надає засоби для взаємного перетворення класів і вбудованих типів.

Перетворювачем значення вбудованого типу в об'єкт є конструктор з єдиним параметром цього (вбудованого) типу. Наприклад, для автоматичного перетворення *double* в *Time* треба визначити конструктор *Time(double)*. Якщо вам потрібен такий конструктор для звичайного створення об'єктів, і ви категорично проти автоматичних перетворень (іноді вони стаються несподівано і недоречно), забороніть автоматичне перетворення ключовим словом *explicit*: *explicit Time(double)*. Такий конструктор можна використовувати для явного перетворення типу.

Для перетворення об'єкта в значення вбудованого типу використовують спеціальні методи з прототипами вигляду *operator mun() const*. Наприклад, *operator double() const* у класі *Time*.

Зауважимо, що використання всіх згаданих інструментів не є обов'язковим і, що особливо важливо, не повинно призводити до неоднозначностей. Автоматичне перетворення типу у виразах є альтернативою перевантаженню операторів, дає коротший але повільніший код.

Для будь-якого класу компілятор генерує неявні функції-елементи:

1) конструктор за замовчуванням (потрібен, наприклад, для створення неініціалізованого масиву об'єктів);
2) конструктор копіювання (працює, коли ініціалізуємо новий об'єкт наявним, при передаванні аргумента за значенням, при поверненні результату функцією, можливо, під час обчислення виразів);
3) оператор присвоєння;
4) деструктор;
5) оператор адресування.

Наш перший приклад з класом *Time* не мав конструктора і деструктора, але все працювало. Нагадаємо, що явне визначення заміняє автоматично генероване, і що ми вже обговорювали питання оголошення конструкторів за замовчуванням і створення.

Жоден з наших прикладів не визначав конструктора копіювання і не перевантажував оператора присвоєння. Справді, в цих прикладах не було такої потреби. Неявні конструктор копіювання і оператор присвоєння просто копіюють всі відповідні поля об'єкта, і нас це влаштовувало.

Завдання для самостійної роботи. Розгляньте проект “VectorPrj” (vector.h, vector.cpp, mainVector.cpp, comment.txt). Клас *Vector* моделює геометричну сутність вектор на площині. Його особливістю є множинне подання вектора: екземпляр зберігає і Декартові координати, і полярні. У класі визначено звичайні методи, методи-оператори, дружні функції, оператори перетворення типу – проаналізуйте які, як саме вони працюють. Випробуйте клас у тестовій програмі, чи не виникають неоднозначності? Розгляньте роботу програми *mainVector*. Вона моделює розв’язок задачі випадкового блукання: «Деякий суб’єкт може виконувати кроки заданої довжини у випадковому напрямку. Скільки кроків доведеться йому виконати, щоб відійти від початкової точки на задану відстань?»

Розглянемо черговий приклад

```
class StringBad
{private:
     char * str;
     int len;
     static int num_strings;
 public:
     StringBad(const char* s);
     StringBad();
     ~StringBad();
     friend ostream& operator<<(ostream& os, const StringBad& st);
};

int StringBad::num_strings = 0;

StringBad::StringBad(const char* s)
{    len = strlen(s);
     str = new char[len+1];
     strcpy(str,s);
     num_strings++;
     cout<<num_strings<<": \""<<str<<"\" object created\n";
}
StringBad::StringBad()
{    len = 3;
     str = new char[4];
     strcpy(str,"C++");
     num_strings++;
     cout<<num_strings<<": \""<<str<<"\" default object created\n";
}
StringBad::~StringBad()
{    --num_strings;
     cout<<'\"'<<str<<" object deleted, "<<num_strings<<" left\n";
     delete [] str;
}
ostream& operator<<(ostream& os, const StringBad& st)
{    os<<st.str; return os;
}
```

Клас *StringBad* інкапсулює динамічні рядки. Кожен екземпляр містить вказівник на пам’ять рядка і його довжину. Крім того клас відслідковує кількість наявних екземплярів. Легко переконатися, що під час виконання наступного коду виникнуть проблеми з розподілом динамічної пам’яті.

```
int aFun(StringBad);
StringBad first("How do you do"); StringBad second = first;
int count = aFun(second);
```

Програма *mainStr.cpp* містить більше прикладів використання екземплярів класу, а файл *result.txt* – отримані результати з коментарями стосовно помилок етапу виконання.

Причиною помилок є те, що неявні конструктор копіювання і оператор присвоєння виконують «поверхове» копіювання об’єкта без врахування того, що об’єкт використовує не

лише власні поля, але й додатково розподілену динамічну пам'ять. Для коректної поведінки програми потрібно виконувати «глибоке» копіювання: копіювати не тільки поля, але і всі структури, пов'язані з об'єктом (виділяти пам'ять для копії і копіювати вміст).

Загалом можна сформулювати таке практичне правило. Якщо екземпляр потребує динамічної пам'яті для зберігання своїх даних, то: 1) конструктори резервують її або оператором *new*, або *new[]*, причому, всі конструктори використовують однаковий оператор; 2) деструктор звільняє динамічну пам'ять відповідно оператором *delete* або *delete[]*; 3) визначають явні конструктор копіювання й оператор присвоєння.

Для класу *StringBad* таке визначення матиме вигляд

```
class StringBad
{private:   .....
 public:
     StringBad(const char* s);
     StringBad();
     StringBad(const StringBad& st);
     StringBad& operator=(const StringBad& st);
     .....
};
/* конструктор копіювання */
StringBad::StringBad(const StringBad& st)
{    len = st.len;             // довжина у копії така ж
     str = new char[len+1];    // виділення пам'яті
     strcpy(str,st.str); // копіювання рядка
     num_strings++;            // оновлення статичного елемента-лічильника
     cout<<num_strings<<": \""<<str<<"\" object copy-created\n";
}
/* оператор присвоєння */
StringBad& StringBad::operator=(const StringBad& st)
{    if (this == &st) return *this;  // запобігти самоприсвоєнню !
     delete [] str;            // звільнити раніше зайняту пам'ять
     len = st.len;
     str = new char[len+1];    // виділити нову пам'ять
     strcpy(str,st.str); // копіювати рядок
     return *this;             // повернути отримувача
}
```

Структура визначених тут методів є доволі типовою. Її можна використовувати як зразок для оголошення інших класів. Зокрема, обов'язковими є перевірка на самоприсвоєння і звільнення пам'яті в операторі присвоєння.

Клас *StringBad* не випадково «бідний» на методи – він залишає доволі простору для власних експериментів з розширення функціональності класу. Скористайтеся нагодою, спробуйте визначити оператори для роботи з рядками (порівняння, конкатенації, підстановки), можливо, інші методи. Зокрема, випробуйте перевантаження оператора індексування для доступу до літер рядка:

```
class StringBad
{private:   .....
 public:    .....
     char& operator[](int);
};
char& StringBad::operator[](int i)
{    return str[i];
}
```

Перевірте, чи дає змогу цей оператор змінювати окремі літери рядка, друкувати їх? Чи запобігає він звертанню до неіснуючих літер? Чи можна з його допомогою працювати з константними об'єктами?

Ще одне завдання: зобразіть схематично розподіл пам'яті після виконання рядка

```
StringBad* favorite = new StringBad("Good luck!");
```

Вкажіть етапи, на яких працюватиме оператор *new* (*new[]*). Як відбуватиметься знищення об'єкта *favorite*?

На завершення пропонуємо вам ще один приклад і **завдання для самостійної роботи**.

Як повідомляє Стівен Прата, банк Bank of Heather виявив бажання встановити банкомат у супермаркеті Food Heap. Для того, щоб отримати згоду керівництва супермаркету, банкіри мусять повідомити завантаженість лінії зв'язку банкомата, середній розмір черги клієнтів, середній час очікування в черзі тощо. Для отримання таких даних було вирішено провести комп'ютерне моделювання ситуації. Працівники банку стверджують, що за їхніми спостереженнями обслуговування третини клієнтів триває одну хвилину, ще третини – дві хвилини, решти клієнтів – три хвилини. Відомо також, що клієнти надходять до банкомата через випадкові проміжки часу, але середня кількість клієнтів за годину є постійною величиною (принаймні, так можна припустити для моделювання). Припускається також, що клієнти не стають у заповнену чергу і не йдуть з черги, поки не отримають обслуговування.

Природньо для моделювання черги клієнтів використати структуру даних черга. Змоделюємо її класом *Queue*. Клієнтів моделюватиме клас *Customer*. Взаємодію клієнтів і черги та відстежування статистичних даних виконає головна програма.

Атрибутів класу *Customer* буде небагато: вистачить задавати час прибуття та тривалість обслуговування. Ці величини можна генерувати випадково.

```
class Customer
{
private:
      static int counter[3];
      long arrive;
      int processtime;
public:
      Customer() :arrive(0), processtime(0) {}
      static void showCounters();
      void set(long when);
      long when() const { return arrive; }
      int ptime() const { return processtime; }
};
ostream& operator<<(ostream& os, const Customer& customer);
```

Функціональність черги набагато різноманітніша:

- Черга зберігає впорядковану послідовність елементів.
- Потрібні операції додавання елемента в кінець черги та вилучення з початку.
- Новостворена черга не містить елементів. Довжина черги обмежена.
- Потрібні операції перевірки, чи черга порожня, чи черга заповнена.
- Потрібно визначати поточну довжину черги.

Для зберігання елементів черга використовує лінійний однозв'язний список. Зверніть увагу на те, що тип вузла списку оголошено всередині оголошення класу *Queue*. Справді, список – це *спосіб реалізації* черги, це її «внутрішня справа», про яку користувачі черги не мають знати. Завдяки такому приховуванню розробник черги в майбутньому може змінити реалізацію без жодного впливу на код клієнта. Можна використати, наприклад, масив, або двохзв'язний список, або стандартний контейнер.

```
typedef Customer Item;
class Queue
{
      struct Node // вкладене оголошення типу
      {
            Item item;
            Node * next;
            Node(Item val, Node* ptr = nullptr) :item(val), next(ptr) {}
      };
      enum {Q_SIZE = 10};
```

```
private:
     Node* front;              // вказівник на початок черги
     Node* rear;               // вказівник на кінець черги
     int items;                // поточна довжина черги
     const int qsize;   // максимальна довжина черги
     Queue(const Queue& q) : qsize(0) { } // заблокований конструктор
     Queue& operator=(const Queue& q) { return *this; }
public:
     Queue(int qs = Q_SIZE);
     ~Queue();
     bool isEmpty() const;     // перевірки
     bool isFull() const;      //
     int queueCount() const;   //
     bool enqueue(const Item& item); // занесення
     bool dequeue(Item& item);       // вилучення
     void show() const;
};
```

Розгляньте проект “Banking” (queue.h, queue.cpp – класи, test.cpp – тестова програма для попереднього випробування класів, bank.cpp – програма моделювання). Зрозумійте влаштування та роботу класів, запропонуйте власні доповнення чи модифікації. Поекспериментуйте з класами, отримайте результати моделювання.

Зверніть увагу на такі нові можливості-особливості:

1. Клас *Queue* для власних потреб оголошує структуру *Node*. Ім’я *Node* видиме лише в межах класу і не конфліктує з зовнішніми іменами.
2. Конструктор створення водночас є конструктором за замовчуванням. Копіювання черги не потрібне, навіть, шкідливе, тому, щоб запобігти використанню неявних конструктора копіювання і оператора присвоєння вони визначені явно але в закритій частині класу.
3. Екземпляри класу містять константне поле *qsize*, яке неможливо ініціалізувати звичайним оператором присвоєння після створення об’єкта, тому в конструкторі використано нову синтаксичну можливість – список ініціалізаторів. Його розташовують услід за двокрапкою після заголовку конструктора перед його блоком. У списку перелічують через кому імена полів і їхні початкові значення (в дужках). Ініціалізація відбувається перед виконанням тіла методу.

```
Queue::Queue(int qs) : qsize(qs), front(0), rear(0), items(0) { }
```

4. Деструктор класу *Queue* повинен виконати значний обсяг роботи. Як ви гадаєте, коли викликаються деструктор черги, деструктори елементів черги?

```
Queue::~Queue()
{
     Node* temp;
     while (front != nullptr)
     {
          temp = front;
          front = front->next;
          delete temp;
     }
}
```

Тип елемента черги *Item* визначено за допомогою *typedef*. У наступних лекціях цей спосіб ми замінимо визначенням шаблона класу.

Література

1. Бублик В.В. Об'єктно-орієнтоване програмування мовою С++.
2. Стивен Прата Язык программирования С++.
3. Бьерн Страуструп Язык программирования С++.