Лекція 8. Структури

Оголошення структури, поля даних, конструктори.
Створення і використання екземплярів структури. Селектор імені.
Перевантаження оператора виведення структури.
Створення і використання масивів структур.
Функції для опрацювання структур.
Перевантаження арифметичних операторів.

Мова програмування С++ – це «мова С з типами користувача». Тобто, з можливістю для програміста оголошувати нові типи даних. Ми вже вміємо давати імена конструкціям оголошення типу за допомогою *typedef* чи *using*. Вони схожі на імена нових типів, проте є лише синонімами імені (alias name). Ми оголошували нові типи-переліки за допомогою *enum*. Проте у такого типу дуже обмежене коло можливостей. Нагадаємо, що *enum* доцільно використовувати для оголошення в одному місці декількох іменованих констант цілого типу.

«Справжній» новий тип у С++ оголошують за допомогою *struct* або *class*. Почнемо з простішого і поговоримо про структури. *Структура* – це тип даних, який об'єднує в одну сутність декілька іменованих, можливо, різнотипних значень. У програмах мовою С++ їх використовують для моделювання невеликих структурованих об'єктів, наприклад: тривалість у годинах і хвилинах, календарна дата, раціональне число, точка декартової площини тощо.

Схематично оголошення структури можна записати так:

```
struct <name_of_type>
{
     <type_1> <name_of_field_1>;
     <type_2> <name_of_field_2>;
     . . .
     [<constructor(s)>]
     [<destructor>]
};
```

Тут *type_X* – імена довільних типів: вбудованих, чи оголошених користувачем; *name_of_field_X* – імена частин структури (чи полів даних). Всередині структури можна оголошувати спеціальні функції: *конструктори* (їх може бути декілька) використовують для налаштування значень полів у момент створення, *деструктор* викликається автоматично в момент звільнення пам'яті від структури, тому його використовують для додаткового очищення пам'яті, виконання завершальних дій, якщо такі потрібні. Для зручного створення структур треба оголошувати конструктори. Потреба в деструкторі виникає не так уже й часто.

Наведемо приклад оголошення типу, що моделює раціональне число.

```
struct Fraction
{
     int nom;      // чисельник - ціле число
     unsigned den; // знаменник - натуральне
};
```

Тепер його можна використовувати для оголошення змінних:

```
Fraction half;              // виділення пам'яті для структури
half.nom = 1; half.den = 2; // задання значень полів структури
Fraction quarters;
quarters.nom = 3; quarters.den = 4;
cout << " half = " << half.nom << '/' << half.den << '\n';
cout << " quarters = " << quarters.nom << '/' << quarters.den << '\n';
```

До полів структури звертаються за допомогою *селектора імені*: *half.nom* – це поле *nom* структури *half*, *quarters.den* – це поле *den* структури *quarters*.

У наведеному прикладі створення екземпляра структури розділено на два етапи: виділення пам'яті, задання початкових значень. Так можна писати програми, але такий спосіб не дуже зручний, та ще й небезпечний: якщо забути задати значення котрогось з полів, чи зробити це неправильно, то який дріб ми отримаємо? Уникнути проблем допомагає оголошення конструкторів. Конструктор – функція, оголошена всередині типу з таким самим іменем, що й тип. Конструктор викликається автоматично в момент створення екземпляра. Удосконалимо оголошення типу *Fraction*.

```
// Раціональне число (множина Q)
struct Fraction
{
      int nom;      // чисельник - ціле число
      unsigned den; // знаменник - натуральне
      Fraction()    // конструктор за замовчуванням
      {             //   задає нуль в Q
            nom = 0;
            den = 1;
      }
      Fraction(int a, unsigned b)  // конструктор з параметрами
      {
            if (b != 0)
            {
                  nom = a;
                  den = b;
            }
            else
            {      // "виправлення" неправильно заданого дробу
                  nom = INT_MAX;
                  den = 1;
            }
      }
};
```

Тепер попередній фрагмент програми можна записати так:

```
Fraction half(1, 2);
Fraction quarters(3, 4);
cout << " half = " << half.nom << '/' << half.den << '\n';
cout << " quarters = " << quarters.nom << '/' << quarters.den << '\n';
Fraction zero;
cout << " zero = " << zero.nom << '/' << zero.den << '\n';
```

Ми задали два конструктори. Перший, без параметрів, називають конструктором *за замовчуванням*. Він спрацьовує тоді, коли користувач не задав значень чисельника і знаменника, і, за загальною домовленістю, присвоює екземплярові нульове значення типу. Для множини раціональних чисел це – значення 0/1. Конструктор за замовчуванням буде використано також тоді, коли користувач створить масив екземплярів структури без вказання ініціалізатора.

Конструктор з параметрами демонструє обдуману поведінку: він запобігає створенню екземплярів, що містять нуль у знаменнику. Ніхто з нас не збирається створювати такі числа свідомо, але ніхто не може гарантувати, що в результаті обчислення деякого знаменника в майбутньому не виникне раптом нуль. Тому в конструктор закладено додаткові перевірки.

Наведене вище оголошення структури з конструкторами є навчальним текстом. На практиці таке оголошення матиме дещо інший вигляд. По-перше, конструктор за замовчуванням, що тільки задає значення полів, можна задати значно коротше і ефективніше. По-друге, досить об'ємний конструктор з параметрами потрібно розділити на оголошення прототипу і власне визначення. Справа в тому, що оголошення типів зазвичай розташовують у файлах заголовків, а визначенню функції там не місце.

Щоб не переписувати програмний код багато разів, одразу доповнимо його ще однією можливістю: «навчимо» стандартний об'єкт *cout* виводити на консоль значення типу *Fraction*. Мова програмування надає програмістові чудовий засіб – перевантаження операторів для нових типів. Тобто, ми маємо змогу визначити функцію зі спеціальним іменем (для оператора виведення це ім'я – *operator*<<), що має серед параметрів хоча б один оголошеного нами типу, і використовувати її як звичайний оператор. На прикладі буде зрозуміліше.

файл "Fraction.h"

```
#pragma once
#include <iostream>
using std::ostream;

// Раціональне число (множина Q)
struct Fraction
{
	int nom;	// чисельник - ціле число
	unsigned den; // знаменник - натуральне
	Fraction():nom(0), den(1) {} // використано список ініціалізації
	Fraction(int a, unsigned b); // прототип конструктора з параметрами
};
// прототип оператора виведення
ostream& operator<<(ostream& os, const Fraction& q);
```

файл "Fraction.cpp"

```
#include "Fraction.h"

Fraction::Fraction(int a, unsigned b)
{
	if (b != 0)
	{
		nom = a; den = b;
	}
	else
	{
		nom = INT_MAX; den = 1;
	}
}

ostream& operator<<(ostream& os, const Fraction& q)
{
	os << q.nom << '/' << q.den;
	return os;
}
```

файл "Program.cpp"

```
#include "Fraction.h"
using std::cout;

int main()
{
	Fraction half(1, 2), quarters(3, 4), zero, copy;
	copy = half; // структури можна присвоювати одна одній!
	cout << " half = " << half << "\n quarters = " << quarters
		<< "\n zero = " << zero << "\n copy = " << copy << '\n';
	system("pause");
	return 0;
}
```

Тут конструктор за замовчуванням має порожній блок, оскільки всю роботу виконує список ініціалізації. Такий запис не тільки коротший, але й ефективніший, оскільки блок

виконується *після* створення екземпляра, а список ініціалізації – *в момент* створення. Значення зі списку компілятор використовує для ініціалізації пам'яті в момент виділення.

Визначення конструктора з параметрами розташовано в окремому файлі. Для того, щоб вказати на його зв'язок з типом *Fraction* використано *кваліфікатор імені*: *Fraction::Fraction*. Перше слово означає ім'я типу, друге – ім'я конструктора. Зверніть увагу на текст програми: середовище програмування навіть виділяє їх різними кольорами.

Прототип функції, що перевантажує оператор виведення має чітку структуру. Імена таких функцій завжди починаються зарезервованим словом *operator*, а після нього вказано символ самого оператора. Першим параметром оператора виведення завжди є посилання на потік виведення, другим параметром – значення або константне посилання на величину, яку потрібно вивести. Для передавання структур частіше використовують посилання. Результатом оператора виведення мав би бути той самий потік, який передали першим параметром. Це звичайна практика, оскільки результатом виконання оператора виведення в потік значення стандартного типу є потік виведення. Результатом *cout << "Yes!"* є об'єкт *cout*.

Запис у головній програмі *Fraction half(1, 2)* означає виклик конструктора для створення нової змінної, буквально *half = Fraction::Fraction(1, 2)*, виведення *cout << half* означає виклик функції: *operator<<(cout, half)*. Для всіх типів структур компілятор автоматично генерує оператор присвоєння. Такий оператор копіює значення відповідних полів.

Бачимо, що змінні оголошеного нами типу *Fraction* можна використовувати майже так само, як змінні стандартних типів. Їх легко створювати, виводити. Покажемо, що структури можна використовувати для створення масивів, екземпляри структур можна передавати на опрацювання у функції та отримувати їх з функцій як результат обчислень. Наприклад, масив раціональних чисел можна оголосити так:

```
Fraction P[3] = { { 1, 2 }, Fraction(3, 4) };
```

Тут для зображення початкового значення *P[0]* використано *ініціалізатор структури*, послідовність значень, розділених комами, обрамлена фігурними дужками. Компілятор використає ці значення з полями структури в порядку оголошення. Тому *P[0].nom==1*, *P[0].den==2*. Для створення *P[1]* явно вказано конструктор – тут все зрозуміло. Для *P[2]* не вказано початкове значення, тому для створення буде використано конструктор за замовчуванням.

Новий тип можна використати для створення динамічних змінних:

```
Fraction * seven = new Fraction(7, 1);
cout << " seven = " << *seven << '\n';
```

Для доступу до поля даних динамічно створеної структури можна використати два способи. Перший нам уже відомий через розіменування вказівника і селектор імені *(*seven).nom*, причому дужки тут обов'язкові, бо селектор має вищий пріоритет ніж розіменування. Другий спосіб набагато зручніший. Він використовує оператор доступу до імені за вказівником: *seven->nom*. Обидва записи еквівалентні, та ми будемо використовувати другий.

Продовжимо наші вправи з оголошенням нових типів. Оголосимо два нових, які зберігатимуть координати точки на площині в декартовій та полярній системах координат відповідно. Оголосимо також декілька функцій для опрацювання значень таких типів.

Декартові координати точки не накладають жодних обмежень на значення полів структури – це просто два дійсних числа. Полярні координати також записують парою дійсних чисел, але потрібно враховувати, що полярний кут набуває значень на проміжку [0°;360°).

```
// Декартові координати точки на площині
struct CPoint
{
      double x;
      double y;
};
```

```
// Полярні координати точки на площині
// кут задано в градусах
struct PPoint
{
	double ro;
	double phi;
};
```

Функції, що виконують перетворення координат з однієї системи в іншу.

```
PPoint CartToPolar(CPoint c)
{
	PPoint answer;
	answer.ro = sqrt(c.x*c.x + c.y*c.y);
	answer.phi = atan2(c.y, c.x) / M_PI * 180;
	return answer;
}

CPoint PolarToCart(PPoint p)
{
	double alpha = p.phi / 180 * M_PI;
	CPoint answer;
	answer.x = p.ro*cos(alpha);
	answer.y = p.ro*sin(alpha);
	return answer;
}
```

Поміркуйте, чи можна було б записати їх компактніше, якби у типів координат точки були оголошені конструктори.

Обчислення відстані між точками на декартовій площині.

```
double Distance(const CPoint& a, const CPoint& b)
{
	return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
}
```

Тепер можемо випробувати в дії оголошені типи і функції.

```
int main()
{
	// Вузли ламаної
	const int m = 5;
	CPoint chain[m] = { { 0, 0 }, { 2, 0 }, { 2, 2 }, { -1, 5 }, { -3, 3 } };
	// Довжина ламаної
	double Len = 0.0;
	for (int i = 1; i < m; ++i)
		Len += Distance(chain[i - 1], chain[i]);
	std::cout << " Length = " << Len << '\n';
	std::cin.get();
	// Перетворення координат
	PPoint R = CartToPolar(chain[2]);
	std::cout << "Cartezian point (" << chain[2].x << ','
		<< chain[2].y << ") in polar is: ro "
		<< R.ro << ", phi " << R.phi << " degrees\n";
	system("pause");
	return 0;
}
```

Поміркуйте, як зміниться програма, якщо визначити оператор виведення в потік для типів *CPoint* і *PPoint*.

На завершення лекції спробуємо допомогти студентові визначити, скільки часу триватиме його навчальний день, якщо він має чотири пари, між якими є дві звичайні і одна коротка перерва. Визначимо для цього тип *Time* та всі потрібні оператори.

файл "Time.h"

```
#pragma once
#include <iostream>

// Тип моделює тривалість у годинах і хвилинах
struct Time
{
	unsigned hours;
	unsigned minutes;

	Time() :hours(0), minutes(0){} // **конструктор за замовчуванням
	Time(unsigned h, unsigned m);  // **конструктор з параметрами
	Time(unsigned t);              // **конструктор з одним параметром перетворює ціле на Time
};

bool operator>(const Time & a, const Time & b); // тривалості можна порівнювати
Time operator+(const Time & a, const Time & b); // тривалості можна додавати
Time operator*(const Time & t, unsigned n);     // час можна множити на число
std::ostream & operator<<(std::ostream & os, const Time & t);
```

файл "Time.cpp"

```
#include "Time.h"

Time::Time(unsigned h, unsigned m)
{
	hours = h + m / 60; minutes = m % 60;
}
Time::Time(unsigned n)
{
	hours = n / 60; minutes = n % 60;
}

bool operator>(const Time & a, const Time & b)
{
	return a.hours > b.hours || a.hours == b.hours && a.minutes > b.minutes;
}
Time operator+(const Time & a, const Time & b)
{
	return Time(a.hours + b.hours, a.minutes + b.minutes);
}
Time operator*(const Time & t, unsigned n)
{
	return Time(t.hours*n, t.minutes*n);
}
std::ostream & operator<<(std::ostream & os, const Time & t)
{
	os << t.hours << (t.minutes > 9 ? ":" : ":0") << t.minutes;
	return os;
}
```

файл "Program.cpp"

```
#include "Time.h"

int main()
{	// Експерименти з часом
	Time Lecture(1, 20);
	Time Break(0, 20);
	Time shortBreak(15);
	Time Day = Lecture * 4 + Break * 2 + shortBreak;
	std::cout << "Student's learning day is " << Day << " long.\n";
	if (Day > 360) // 6 год. == 360 хв.
		std::cout << "It is longer then 6 hours.\n";
	else
		std::cout << "It's no so long.\n";
	system("pause");
	return 0;
}
```

Тут є над чим поміркувати. Чому оператор додавання працює коректно, якщо він не перевіряє, чи сума хвилин *a.minutes + b.minutes* не більша за 60? Те саме питання до оператора множення. І взагалі, як так сталося, що програма змогла порівняти тривалість *Day* і число 360?

Література

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