Лекція 11. Стек, черга, пріоритетна черга

1. *Особливості реалізації та інтерфейсу.*
2. *Приклади використання: обхід дерева, польський запис, моделювання обслуговування.*
3. *Доступ до реалізації. Експерименти з пріоритетною чергою.*
4. *Власна реалізація контейнерного адаптера.*
5. *Наслідування від стандартного контейнера для вдосконалення поведінки.*

У попередніх лекціях ми вже докладно обговорювали тему оголошення шаблона стека і наводили приклади різноманітних його реалізацій. Бібліотека STL містить власну, і настав час познайомитися з нею (заголовковий файл *stack*) та з реалізацією звичайної і пріоритетної черги (заголовковий файл *queue*).

Стек

Стек – динамічна структура даних, що змінює свій стан у результаті кожного читання-запису даних і працює за правилом «першим зайшов, останнім вийшов». Доступ можливий лише до вершини стека: сюди записують дані методом *push* і зчитують дані методом *pop*. У наших попередніх вправах ми реалізували саме такі методи. Розробники STL діяли трохи інакше, але про це згодом.

Всякий стек використовує послідовну пам'ять для зберігання даних і організовує специфічний спосіб доступу до них – лише з одного кінця послідовності. Ми теж так робили і створювали стек на основі масиву: статичного або динамічного. Якби у нашому розпорядженні тоді були послідовні контейнери STL, було б ще простіше. Адже у кожного з них є все, що потрібно стекові: методи *back*, *push_back*, *pop_back* можуть робити всю роботу. Але у послідовних контейнерів є більше, ніж потрібно стекові: доступ до довільного елемента контейнера, вставка і вилучення. Такі дії могли б зруйнувати стек, тому доступ до них потрібно закрити. Розробники STL так і зробили. Щоб не проектувати цілком новий контейнер, вони вирішили *адаптувати* наявні послідовні контейнери до нових умов використання – в якості стека. Саме звідси назва категорії контейнерів, які ми розглядаємо – контейнерні адаптери.

Як змінити інтерфейс класу? Потрібно включити його екземпляр до складу іншого класу. Новий клас визначить потрібний новий інтерфейс і переадресує виклики до методів до відповідних методів вкладеного об'єкта. Той функціонал вкладеного об'єкта, який потрібно приховати, просто не матиме відповідників у новому інтерфейсі.

Такий підхід нагадує ось яку історію. Був собі виробник елітних автомобілів. Справи йшли дуже добре. Виробництво налагоджене – щохвилини новий автомобіль. Одна біда у чоловіка: заміський будинок не мав дзвінка біля вхідних воріт. Автомобілів купа, а дзвінка нема. От і вирішив він розв'язати проблему наявними ресурсами: поставив біля входу новеньке авто, накрив його великим гарним ящиком, просвердлив у ньому дірки навпроти клаксона (щоб добре було чути) і вивів назовні зручний важіль, який натискав на сигнал на кермі. Тепер кожен відвідувач обійстя міг дати знати господареві про своє прибуття. ☺

Це історія про побудову адаптера. Ящик обмежує функціонал вкладеного об'єкта (не можна заводити двигун, перемикати передачі, кермувати тощо), а важіль і дірки – спосіб делегування функцій вкладеному об'єктові.

Клас *stack* містить вкладений об'єкт – послідовний контейнер – і використовує його можливості динамічного перерозподілу пам'яті. Назва «адаптер» не тільки говорить про спосіб реалізації стека, а й підказує про можливість адаптації *різних* контейнерів: основою для нього може бути *vector*, *list*, *deque*. Треба сказати, що розробники дуже добре виконали свою роботу. Стек є шаблонним класом з двома параметрами. Перший, як і треба було сподіватися, задає тип даних, які будуть зберігатися у стеку, другий – тип контейнера, який буде вкладено в стек. Користувач може задавати його на власний розсуд.

Синтаксис оголошення, інтерфейс

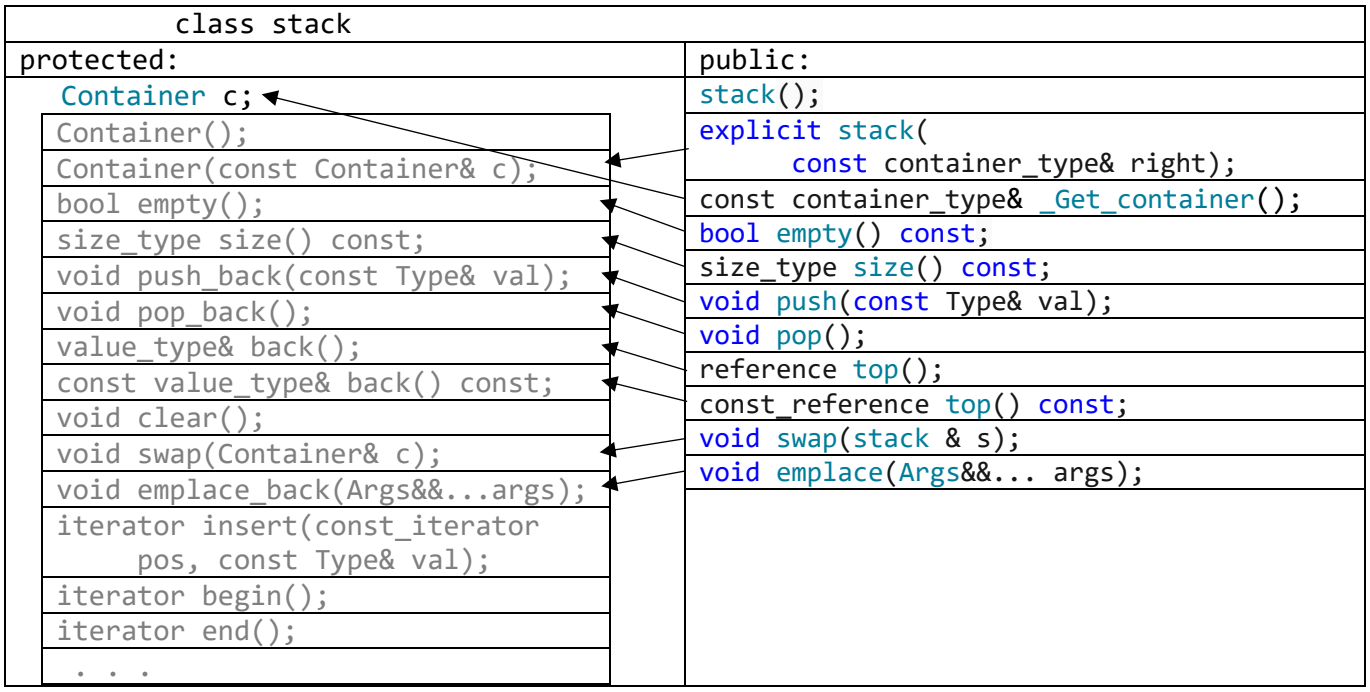
https://docs.microsoft.com/en-us/cpp/standard-library/stack-class?view=vs-2019

```
template <class Type, class Container= deque <Type>>
class stack
```

Другий параметр шаблона має значення за замовчуванням: зазвичай стек будують на основі дека. З'ясувалося, що механізми виділення-звільнення пам'яті дека найефективніший для потреб стека, тому обрали саме його. Проте користувач може задати список або вектор на власний розсуд.

```
stack<int> sid; // стек цілих на основі дека
stack<int, list<int>> sil; // стек цілих на основі списка
stack<double, vector<double>> sdv; // стек дійсних на основі вектора
```

Перелік методів стека подано в таблиці. Їх значно менше, ніж у послідовного контейнера



Метод *empty* повідомляє, чи стек порожній, *push* заносить дані до стека, *top* дозволяє подивитися, яке значення міститься у вершині стека (ми такий метод у власній реалізації стека називали *peek*). А от зчитування даних зі стека STL виконують у особливий спосіб, оскільки метод *pop* не повертає (!) даних. «Класичний pop» виконують за два кроки: методом *top* читають дані, методом *pop* вилучають їх зі стека. Це не надто зручно, але розробники стверджують, що так надійніше. Вони спиралися на підхід «один метод – одна відповідальність». Якщо станеться виняток під час виконання котрогось з них, то програмістові буде легше зрозуміти, у якому стані перебуває зараз його стек.

Варто також наголосити, що методи *top* і *pop* не можна викликати для порожнього стека, оскільки їхня поведінка в цьому випадку не визначена. Розробники дали користувачу метод *empty*, а заодно і відповідальність за його вчасний виклик.

Цікаво, що стек надає доступ до вкладеного контейнера. Метод *_Get_container* повертає константне посилання, яке можна використати для читання всіх елементів контейнера. Таким чином можна програмно дослідити весь вміст стека. Інший спосіб полягає в наслідуванні від стека. Підклас матиме доступ до захищеного поля *c* і зможе виконувати будь які дії з контейнером, що міститься у цьому полі.

Черга

Черга – динамічна структура даних, що змінює свій стан у результаті кожного читання-запису даних і працює за правилом «першим зайшов, першим вийшов». Дані записують з одного кінця черги, а зчитують – з іншого. За класичного підходу занесення до черги називають *enqueue*, вилучення – *dequeue*, але в STL знову не так.

Чергу STL влаштовано подібно до стека: вона збудована на основі послідовного контейнера, її методи називаються так само, тільки замість *top* є метод *front*.

https://docs.microsoft.com/en-us/cpp/standard-library/queue-class?view=vs-2019

```
template <class Type, class Container = deque <Type>>
class queue
```

Логіку розробників зрозуміти важко. Дивіться самі: занесення в чергу – метод *push*, так само, як у стека. Вилучення з черги – метод *pop*, так само, як у стека, і так само він не повертає даних. Перевірка, чи черга порожня – метод *empty*, так само, як у стека. А от читання даних з черги – метод *front*, не так, як у стека. Можливо, таким чином розробники хотіли підкреслити, що у стека є вершина, а в черги – початок. Зчитування даних з черги також виконують за два кроки: *front()* повертає значення, *pop()* – вилучає його.

Приклади використання стека, черги

Зміна порядку членів послідовності

Задача. Задано текстовий файл, окремі рядки якого починаються літерою '#'. Надрукувати спочатку всі звичайні рядки в оберненому порядку, а потім ті, що починаються з неї, в тому порядку, як вони були записані у файлі.

Для рядків з «чарівною» літерою використаємо чергу, а для всіх інших – стек.

```
int main()
{
	ifstream in("Stacks.cpp");
	stack<string> textLines;
	queue<string> directiveLines;
	string line;
	while (getline(in, line)) // зчитування та аналіз рядків
	{
		if (line[0] == '#')
			directiveLines.push(line + "\n");
		else
			textLines.push(line + "\n");
	}
	in.close();
	while (!textLines.empty()) // друк звичайних рядків зі стека
	{                          // в оберненому порядку
		cout << textLines.top();
		textLines.pop();
	}
	while (!directiveLines.empty()) // друк особливих рядків
	{                               // у вихідному порядку
		cout << directiveLines.front();
		directiveLines.pop();
	}
	cin.get();
	return 0;
}
```

Обхід дерева в глибину

Ми вже знайомі з рекурсивними функціями обходу дерева. Вони виконують обхід «в глибину» – від кореня до листка. Кожну з них можна переписати в ітеративному стилі з використанням стека. Стек запам'ятовуватиме ті вершини, до яких потрібно повернутися по дорозі назад – від листка до кореня.

```
// оголошення типу для моделювання вузлів дерева
struct treeNode;
typedef treeNode* Tree_t;

struct treeNode
{
	int val;
	Tree_t left;
	Tree_t right;
	treeNode() : val(0), left(nullptr), right(nullptr) {}
	treeNode(int x, Tree_t l, Tree_t r) : val(x), left(l), right(r) {}
};
// рекурсивна, друкує дерево як послідовність
void writeTreePre(const Tree_t T)
{
	// друк вмісту дерева в рядок, порядок обходу preorder
	if (T != nullptr)
	{
		std::cout << '\t' << T->val; // друк кореня
		writeTreePre(T->left);       // друк лівого піддерева
		writeTreePre(T->right);      // друк правого піддерева
	}
}
// друк - нерекурсивний обхід
// використано стек для запам'ятовування пропущених піддерев
void printPrefix(const Tree_t Tree)
{
	std::stack<Tree_t> subtrees;
	subtrees.push(Tree);
	while (!subtrees.empty())
	{
		Tree_t t = subtrees.top(); subtrees.pop();
		while (t != nullptr)
		{	 // друкуємо ліву вітку від кореня до листка
			std::cout << '\t' << t->val;
			// якщо по дорозі трапляються відгалуження, заносимо їх до стеку
			if (t->right != nullptr) subtrees.push(t->right);
			t = t->left;
		}
	}
}
// рекурсивна, друкує дерево як впорядковану послідовність
void writeTree(const Tree_t T)
{
	// друк вмісту дерева в рядок, порядок обходу inorder
	if (T != nullptr)
	{
		writeTree(T->left);          // ліве піддерево спочатку
		std::cout << '\t' << T->val; // корінь - посередині
		writeTree(T->right);         // праве піддерево завершує
	}
}
```

```
// Друк треба почати з крайнього лівого листка, тому все, що трапиться по дорозі
// має потрапити до стека. По дорозі назад опрацьовують корінь, а тоді – праве
// піддерево, у якому знову шукають крайній лівий листок
void printInfix(Tree_t Tree)
{
	std::stack<Tree_t> nodes;
	while (!nodes.empty() || Tree != nullptr)
	{
		if (Tree != nullptr)
		{
			nodes.push(Tree);
			Tree = Tree->left;
		}
		else
		{
			Tree = nodes.top(); nodes.pop();
			std::cout << '\t' << Tree->val;
			Tree = Tree->right;
		}
	}
}
```

Описані тут способи використання стека можна застосовувати в різних випадках обходу дерева. Нерекурсивні функції можуть виявитися ефективнішими за свої рекурсивні аналоги.

Обхід дерева в ширину

Обхід «в ширину» – це обхід за рівнями: спочатку корінь, тоді вершини першого рівня, за ними – їхні підвершини, або вершини другого рівня, і так до найглибшого рівня. Коли рухаєшся рівнем, потрібно якось запам’ятовувати вказівники на підлеглі вершини щойно опрацьованої. Для таких потреб добре підходить черга. Використаємо оголошений раніше тип вершини дерева. Тоді функції друку за рівнями можна записати так.

```
// друк - лівосторонній обхід за рівнями
// друкує дерево в один рядок
// використано чергу для запам'ятовування всіх вузлів нижчого рівня
void printByLevel(const Tree_t Tree)
{
	std::queue<Tree_t> nodes;
	nodes.push(Tree); // корінь – в черзі на опрацювання
	while (!nodes.empty())
	{
		Tree_t t = nodes.front(); nodes.pop();
		std::cout << '\t' << t->val;
		// наявні підлеглі вершини заносимо до черги
		if (t->left != nullptr) nodes.push(t->left);
		if (t->right != nullptr) nodes.push(t->right);
	}
}

// від друку за рівнями можна вимагати більшого:
// при переході на нижчий рівень дерева переходити на наступний рядок екрана
void printByLevelInLines(const Tree_t Tree)
{
	std::queue<Tree_t> nodes;
	nodes.push(Tree); int count = 1; // на першому рівні одна вершина
	while (count > 0)
	{
		int subcount = 0;
```

```
            for (int i = 0; i < count; ++i)
            {
                  Tree_t t = nodes.front(); nodes.pop();
                  std::cout << '\t' << t->val;
                  // підлеглі вершини потрібно запам'ятати і порахувати
                  if (t->left != nullptr)
                  {
                        nodes.push(t->left);
                        ++subcount;
                  }
                  if (t->right != nullptr)
                  {
                        nodes.push(t->right);
                        ++subcount;
                  }
            }
            std::cout << '\n'; // завершили рівень дерева
            count = subcount;
      }
}
```

Без черги такі функції написати було б дуже складно.

Використання стека для побудови польського запису

Побудову польського (постфіксного) запису арифметичного виразу з використанням стека та обчислення його значення докладно описано в навчальному матеріалі «Методи розробки алгоритмів / Метод часткових цілей» у вашій команді MS Teams (вкладка Файли).

Використання черги в задачах моделювання систем обслуговування

Є два хороші приклади, описані в літературі:

[ ] Брюс Эккель, Чак Эллисон Философия C++. ст. 378-382 «Задача касира»: клієнти банку шикуються в одну чергу. Їх по одному обслуговують касири. Кількість касирів пристосовується до кількості покупців. Змоделювати розвиток подій і дослідити, як змінюватиметься довжина черги та кількість касирів. Використано *queue* та *list*. Програма імітує багатопотокове виконання. Для доступу до пам'яті черги застосовано наслідування. Текст програми є в папці лекції (Adapters/BankQueue.cpp).

[ ] Стивен Прата Язык программирования C++. ст. 580-600 Емуляція черги до банкомата. Детально описано процес власної реалізації черги на основі списку. (Проект Banking також є в папці лекції.)

Пріоритетна черга

Порядок елементів у черзі може залежати не тільки від часу надходження, але й від їхньої «важливості», або пріоритету: значення з вищим пріоритетом повинні бути в черзі попереду значень з нижчим. Природньо сподіватися, що значення з однаковим пріоритетом стоятимуть у порядку надходження до черги. Але в класі *priority_queue* верх знову взяла реалізація, тобто, намагання зробити його якнайшвидшим, і порядок значень однакового пріоритету не визначено – він залежить від того, як спрацює алгоритм впорядкування.

https://docs.microsoft.com/en-us/cpp/standard-library/priority-queue-class?view=vs-2019

```
template <class Type, class Container = vector <Type>,
    class Compare = less <typename Container::value_type>>
class priority_queue
```

Бачимо, що *priority_queue* реалізовано на основі вектора. Третій параметр задає спосіб порівняння «пріоритетів». Слово взято в лапки, бо насправді порівнюють самі значення, які

додають до черги, за допомогою оператора <. Якщо до черги додають не числа, а складні об'єкти, одне з полів яких задає пріоритет, то потрібно належним чином визначити для таких об'єктів оператор порівняння. Можна також задавати третім параметром власний компаратор, наприклад, щоб отримати впорядкування від меншого до більшого, можна вказати *class Compare = greater <typename Container::value_type>*.

Набір методів у пріоритетної черги точно такий самий, як у стека. Знову маємо *top()* замість *front()* (але чому?!?). Поведінка – як у впорядкованої послідовності, яку можна читати тільки від початку. Реалізація – вектор, на який накладено швидке сортування без збереження взаємного порядку рівних значень. Порівняно з іншими контейнерними адаптерами пріоритетна черга має набагато більше конструкторів. Подивіться в документацію, яких.

Наступні приклади використання запозичено у Б. Еккеля і Ч. Еллісона.

```
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;

template <typename T>        // доступ до реалізації
class PQI : public priority_queue<T>
{
public:
      vector<T>& impl() { return c; }
};

template <typename T>        // друк контейнерів
void writeQueue(T& pq)
{
      while (!pq.empty())
      {
            cout << pq.top() << '\t';
            pq.pop();
      }
      cout << '\n';
}

int main()
{
      srand(time(0));
      priority_queue<int> pq1; // less as default
      priority_queue<int,vector<int>,greater<int>> pq2;
      PQI<int> pq3;
      for (int i=0; i<30; ++i)
      {
            int x=rand()%10;
            pq1.push(x); pq2.push(x), pq3.push(x);
      }
      cout << "Greater priority queue\n";
      writeQueue(pq2);
/*
Greater priority queue
0     0     0     1     1     1     1     2     2     2
2     2     3     3     4     4     4     4     5     6
6     6     7     7     7     8     8     8     9     9
*/
      cout << "\n\nUsual (less) priority queue\n";
      writeQueue(pq1);
```

```
/*
Usual (less) priority queue
9	9	8	8	8	7	7	7	6	6
6	5	4	4	4	4	3	3	2	2
2	2	2	1	1	1	1	0	0	0
*/
	cout << "\n\nContainer of child priority queue\n";
	copy(pq3.impl().begin(), pq3.impl().end(),
		ostream_iterator<int>(cout, "\t"));
/*
Container of child priority queue
9	9	8	8	6	7	6	8	7	4
4	7	4	5	2	2	6	0	1	1
2	1	2	3	1	0	2	4	3	0
*/
	cout << "\n\nChild (less) priority queue\n";
	writeQueue(pq3);
/*
Child (less) priority queue
9	9	8	8	8	7	7	7	6	6
6	5	4	4	4	4	3	3	2	2
2	2	2	1	1	1	1	0	0	0
*/
	cout<<endl;
	return 0;
}
```

Пріоритетну чергу можна будувати з довільних об'єктів, для яких визначено оператор <.

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;

class ToDoItem
{
	char primary;
	int secondary;
	string item;
public:
	ToDoItem(string td, char pri='A', int sec=1)
		: item(td), primary(pri), secondary(sec) {}
	friend bool operator<(const ToDoItem& x, const ToDoItem& y)
	{
		return x.primary>y.primary ||
			x.primary==y.primary && x.secondary>y.secondary;
	}
	friend ostream& operator<<(ostream& os, const ToDoItem& td)
	{
		return os<<td.primary<<td.secondary<<": "<<td.item;
	}
};
template <typename T>
void writeQueue(T& pq)
{
	while (!pq.empty())
	{
		cout << pq.top() << '\n';
		pq.pop();
	}
	cout << '\n';
}
```

```
int main()
{
	priority_queue<ToDoItem> toDoList;
	toDoList.push(ToDoItem("Empty trush",'C',4));
	toDoList.push(ToDoItem("Feed dog",'A',2));
	toDoList.push(ToDoItem("Feed bird",'B',7));
	toDoList.push(ToDoItem("Mow lawn",'C',3));
	toDoList.push(ToDoItem("Water lawn",'A',1));
	toDoList.push(ToDoItem("Feed cat", 'B',1));
	toDoList.push(ToDoItem("Feed student",'A',2));
	toDoList.push(ToDoItem("Sleep child",'A',2));
	toDoList.push(ToDoItem("Empty mind",'D',16));
	writeQueue(toDoList);
	return 0;
}
/*
A1: Water lawn
A2: Feed student
A2: Feed dog
A2: Sleep child
B1: Feed cat
B7: Feed bird
C3: Mow lawn
C4: Empty trush
D16: Empty mind
*/
```

Завдання з однаковим пріоритетом впорядковуються загадковим чином

Контейнери власного виробництва*

На цьому етапі навчання ми вже назбирали стільки цікавого досвіду, що важко втриматися від спокуси написати справжній контейнер. Ми вже і динамічний масив фігур робили, і списки реалізували, і складні випадки використовували. Чому б це все не матеріалізувати? Задумано – зроблено в проекті *ContainersByself*. Тут можна знайти два послідовні контейнери і один контейнерний адаптер:

- *Array<T>* є певним аналогом *vector<T>*, має метод вставляння значення на початок;
- *DblList<T>* є певним аналогом *list<T>*, має оператор індексування елементів;
- *Stack<T, Cont>* є певним аналогом *stack<T, C>*, реалізує дії зі стеком в класичному стилі.

Ці контейнери написані не для того, щоб перевершити STL, а щоб зрозуміти, як це робиться, і щоб переконатися, що написати щось схоже цілком можливо. І *Array*, і *DblList* мають повноцінні ітератори. Настільки, що з цими ітераторами можуть працювати всі стандартні алгоритми. Варто докладно розібратися з їхньою реалізацією.

У власних контейнерах ми можемо програмувати нові можливості, яких бракує в стандартних. Метод *Array<T>::putFirst(const T& x)* вставляє *x* на початок контейнера. Для його ефективної реалізації довелося «розмити» логіку перерозподілу пам'яті і включити відповідні алгоритми до тіла методу. Це не дуже добре архітектурне рішення, але дозволяє зменшити кількість переприсвоєнь удвічі: *n* замість 2*n*.

Метод *DblList<T>::operator[](size_t)* для підвищення ефективності використовує додатковий вказівник на останню відвідану оператором ланку. Зазвичай індексують сусідні елементи послідовності. В такому випадку доступ відбуватиметься за константний час, але в найгіршому випадку він буде таки лінійним.

Клас *Stack<T, Cont>* є прикладом оголошення шаблона з параметром шаблоном. Від вкладеного контейнера він вимагає наявності кількох методів: *size*, *back*, *putLast*, *getLast*.

* Додатковий матеріал для допитливих

Адаптація вектора під поліморфну колекцію

Настав час виконати давню обіцянку і навести приклад контейнера для зберігання динамічних об'єктів. У декількох прикладах вище ми бачили, що від контейнерів STL можна наслідувати, то ж використаємо саме такий підхід. Пригадаємо, які проблеми можуть виникнути при наповненні стандартного контейнерами вказівниками на об'єкти, і виправимо їх у підкласі. Отож, що не так робить *vector<T>*:

- деструктор не звільняє динамічну пам'ять від об'єктів, на які вказують елементи вектора;
- конструктор копіювання копіює вказівники, а не об'єкти;
- оператор присвоєння також копіює вказівники, а не об'єкти;
- метод *clear* не звільняє динамічну пам'ять від об'єктів, на які вказують елементи вектора.

Найлегше вирішити проблему звільнення пам'яті: потрібно додатково виконати оператор *delete* для кожного елемента вектора. Трохи складніше з «суперглибоким» копіюванням. Тут уже доведеться розраховувати на те, що об'єкти вміють виконувати *Clone*, що повертає їхню копію.

```
#include <vector>
using std::vector;

template <typename pT>
class pVector :public vector<pT>
{
private:
	void internalClear()
	{
		for (iterator it = begin(); it != end(); ++it) delete *it;
	}
public:
	pVector() : vector() {}
	~pVector()
	{
		internalClear();
	}
	pVector(const pVector& pV) : vector(pV)
	{
		for (iterator it = begin(); it != end(); ++it) *it = (*it)->Clone();
	}
	pVector& operator=(const pVector& pV)
	{
		if (this != &pV)
		{
			internalClear();
			vector::operator=(pV);
			for (iterator it = begin(); it != end(); ++it) *it = it->Clone();
		}
		return *this;
	}
	void clear()
	{
		internalClear();
		vector::clear();
	}
};
```

Програму, що порівнює *vector<T>* і *pVector<T>*, можна знайти в *PolimorphVector*.