ptrs-s: Руководство программиста

DocumentId:GradSoft-PR-07.03.2002

May 8, 2003

Contents

1 Введение

Пакет `ptrs` предназначен для организации техникой управления памятью в C++ программох. Он включает в себя набор шаблонов т. н. умных указателей (`smart_pointes`), с помощью которых можно обеспечить эффективную и удобную технологию управления памятью.

В этом документе подразумевается знакомство с основными концепциями C++, необходимое введение можно найти в [1], [3]

1.1 Об этом пакете

Данное ПО разработанно и поддерживается компанией GradSoft, Киев, Украина. Последняя версия этого пакета доступна на www-сайте: `http://www.gradsoft.com.ua` . Вы можете свободно использовать этот пакет и включать его в свои программы, в соответствии с лицензией, находящейся в файле `docs/LICENSE` (либо `docs/LICENSE.rus`

в дистрибутиве пакета GradSoft C++ ToolBox. При необходимости возможно коммерческое сопровождение пакета.

1.2 Об этом документе

Данное Руководство Программиста написанно для версии ptrs, входящей в состав GradC++ToolBox версии 1.5.0. Тут описано использование API с точки зрения программиста. Порядок инсталляции пакета описан в Руководстве Администратора пакета GradSoft C++ ToolBox [2]

2 Описание классов

2.1 Шаблоны указателей: общее место

Если Вы знакомы с концепцией "умных указателей", то Вы знаете, что `умный указатель` это класс C++ который содержит в себе собственно указатель, возможно вместе с некоторой дополнительноф инфрастуктурой, в котором перегуржены операторы разименования: `*, ->, ->*`.

В дополнение, для всех типов шаблонов указателей мы определяем метод `get()`, который возвращает нам указатель на внутренний объект.

2.2 Исключения

Пакет `ptrs` определяет еще одно исключение в стандартной иерархии исключений C++. Это `NullPointerException`, возникающее в том случае, когда мы пытаемся разъименовать пустой указатель во время работы с так-назывемыми безопасными (safe) классами указателей.

Схема:

```
std::runtime_exception
  |
 *->NullPointerException
```

2.3 safe_ptr

`safe_ptr` это просто оболочка вокруг указателя, который при попытке доступа к нулевому указателю генерирует исключение `NullPointerException` вместо переключения программы в режим неопределенного поведения.

Типичное его использование - использовать его как замену простого указателя для данных, который не принадлежат текущей подсистеме.

Пример:

```
void myFun(Something* x) throw(std::exception)
{
 safe_ptr<Somethins> sx(x);
 sx->do();
}
```

вместо

```
void myFun(Something* x) throw(std::exception)
{
 if (x==NULL) throw std::runtime_exception(string("x is null"));
 x->do();
}
```

2.4 safe_auto_ptr

Это полный аналог `std::auto_ptr`, только безопасный.

Заметим, что мы не включаем в состав `safe_auto_ptr` стандартный методы-шаблоны, так как в настоящее время их поддерживают не все компиляторы C++.

2.5 owned_ptr

Как вы можете определить из имени, `owned_ptr` - это указатель, 'принадлежащий' чему-либо. Что это означает - в `owned_ptr` мы храним указатель вместе с логическим флагом 'принадлежности'. Если этот флаг установлен в `true`, то при разрушении `owned_ptr` удаляется и внутренний объект.

'Безопасность' `owned_ptr` определяется вторым параметром шаблона, который должен быть одной из структур свойств: `ptr::safe` или `ptr::unsafe`.

- `owned_ptr<T,ptr::safe>` - безопасный `owned_ptr`
- `owned_ptr<T,ptr::unsafe>` - соответственно, опасный ;)

В отличие от других шаблонов указателя для `owned_ptr` не определен оператор присваивания и конструктор копирования, вместо этого используется метод `set` с двумя параметрами: первый параметр это объект для копирования, второй - флаг передачи ответственности за освобождение памяти.

Примеры:

```
MyClass* px = new MyClass();
owned_ptr<MyClass,ptr::safe> a(px,true);
MyClass* py = new MyClass();
owned_ptr<MyClass,ptr::safe> b(py,true);
a.set(b,true);
```

`py` сейчас принадлежит `a`, `b` указывает на `py`, но его разрушение никак не повлияет на `py`. а вот `px` удален при вызове `set`.

```
owned_ptr<MyClass,ptr::safe> a;
MyClass* px = new MyClass();
owned_ptr<MyClass,ptr::safe> b(px, true);
a.set(b,false);
```

`a` сейчас указывает на `px`, хозяин `px` по-прежнему `b`.

```
owned_ptr<MyClass,ptr::safe> a;
MyClass* px = new MyClass();
owned_ptr<MyClass,ptr::safe> b(px, false);
a.set(b,false);
```

`a` и `b` сейчас указывают на `px`, но `px` никому не принадлежит.

```
MyClass* px = new MyClass();
owned_ptr<MyClass,ptr::safe> a(px,false);
MyClass* py = new MyClass();
a.set(py,false);
```

утечка памяти (`px`).

2.6 counted_ptr

2.6.1 Теория

`counted_ptr` инкапсулирует известную идиому подсчета ссылок: объект существует до тех пор, пока на него ссылается, поэтому:

- давайте держать в каком-то месте пару из объекта и значения счетчика.
- когда мы создаем указатель на объект, увеличим значения счетчика.
- при уничтожении указателя будем уменьшать счетчик.
- при значении счетчика равным нулю на объект никто не ссылается, следовательно его можно удалять.

Эта идиома сильно упрощает управление памятью для ациклических структур.

2.6.2 Использование

Наш шаблон `counted_ptr` имеет два параметра: тип объекта и класс безопасности, как и `owned_ptr`.

В дополнение к обычным методам умных указателей, `counted_ptr` предоставляет еще метод `assign`, позволяющий изменить значение внутреннего указателя.

Пример:

```
counted_ptr<MyClass,ptr::safe> a(pA);
counted_ptr<MyClass,ptr::safe> b=a;
b.assign(pB);
```

`*pA` разрушен, как `a`, так и `b` указывают на `*pB`.

2.6.3 Ограничения

- шаблон `counted_ptr` не рассчитан на многопоточность. Для многопоточных приложений в `Threading` есть `counted_mt_ptr`.
- подсчет ссылок работает только на ациклических структурах, для работы с циклическими ссылками в памяти вы должны применять какую-то технику сбора мусора.

3 Перечень изменений

19.03.2002 - закончена первая версия.

07.03.2002 - создан.

References

[1] ANSI. *International Standard ISO/IEC 14882. Programming Languages - C++*, 1998. ISO/IEC 14882:1998(E).

[2] Ukraine GradSoft, Kiev. *GradSoft C++ ToolBox: Administration Guide*, 2000,2001. GradSoft-AD-e-04.09.2000-vC.

[3] David Harvey. *Smart pointer templates in C++*, 1996. http://web.ftech.net/ honeyg/articles/smartp.htm.