Grad-Soft ptrs: Programmers Guide

DocumentId:GradSoft-PR-e-07.02.2002

May 8, 2003

Contents

1 Introduction

Package **ptrs-s** is devoted for advanced techniques of memory management in C++. It includes set of smart pointers templates. Whith help of this templates developers can greatly reduce cost of attention to memory management issues during application development.

For reading of this document base knoweldge of C++ and memory management is required; for introductory matherial you can look at [1], [3]

1.1 About this package

This software is produsedby GradSoft company, Kiev, Ukraine. Last version of this package is aviable on GradSoft Web site `http://www.gradsoft.com.ua/eng/` .

You can free use this package and redistribute it with you programs, according to license, which situated in file `docs/LICENSE` inside distributive of GradSoft C++ ToolBox.

Commercial support of this package is aviable: call us for details.

1.2 About this document

This manual is writen for 1.5.0 version of GradSoft C++ ToolBox. Using ptr-s API from programmers point of view is described. Compilation and installation issues described in [2]

2 Class Hierarchy Description

2.1 Common Place

If you familiar with concepts of `smart pointers`, than you know, that `smart pointer` is a C++ class, which keep plain pointer, maybe some infrastructure data and overload pointer dereferencing operators: `*, ->, ->*`.

In addition method `get()` returning pointer itself is provided for all types of smart pointer templates.

2.2 Exceptions

Package ptr-s introduce one exception into standart C++ exceptions hierarchy. This is `NullPointerExceptions`, which is raised when we try to access to null pointer via `operator->` or `operator*` throught so-called **safe** pointer classes.

Hierarchy:

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

2.3 safe_ptr

`safe_ptr` is just wrapper arround pointer, which during applying of accessors throw exception `NullPointerException` instead switching program to undefined behaviour. Typical pattern of usage is keep in `safe_ptr` pointers to data, owned by other subsystems.

Example:

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

can be used instead:

```
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

This is full analog of `std::auto_ptr` with one change: throw `NullPointerException` during dereferencing of NULL pointer.

Note: we does not include ANSII C++ template methods for `std::auto_ptr`, since luck of compiler support for this language feature.

2.5 owned_ptr

As you can guess from name of template, `owned_ptr` is a pointer which 'owned' by some entity. What this mean: in `owned_ptr` we hold pointer itself and boolean flag which indicate 'ownity'. If ownity set to true, than destructor of holded class is called during `owned_ptr` destruction.

Safety of `owned_ptr` are denoted by second template parameter, which must be one of trait structures `ptr::safe` or `ptr::unsafe`.

- `owned_ptr<T,ptr::safe>` - for safe owned pointer.
- `owned_ptr<T,ptr::unsafe>` - for unsafe owned pointer.

Unlike other pointer templates, owned pointer have no overloaded `operator=` , instead we have method set with to parameters: one is pointer to set, other - boolean flag, which indicate passing of ownership.

Examples:

```
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` now owned by `a`, `b` point to `py` but can be destructed without affecting of `py`. `px` is destructed.

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

`a` now point to `px`, but `px` is still owned by `b`.

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

`a` and `b` now points to `px`, but `px` is not owned by anything.

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

`px` is leaked.

2.6 counted_ptr

2.6.1 Theory

`counted_ptr` incapsulate well-known idiom of reference counting: object is living while it's pointed, so:

- let's keep pair of object and counter value.
- when we create pointer to object let's increment counter.
- when we desctruct pointer to object let's decrement counter.
- when counter became zero, i. e. last pointer, which point to our object was destructed, let's destruct object.

This idiom greatly simplicified the task of memory management for acyclic structures.

2.6.2 Usage

Our `count_ptr` template have 2 template parameters: first is type of wrapped object, second - safety class, one of `ptr::safe` or `ptr::unsafe`.

In addition to usial pointer operations, counted-ptr provide additonal method: `counter_ptr<T,safety>::assign(T* new)` , which change value of internal shared pointer.

examlple:

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

now `*pA` is destroyed and both `a` and |b| point to `pB`.

2.6.3 Limitation

- our `count_ptr` template is not thread-safe. We provide thread-safe counted pointer as `count_mt_ptr` in Threading package.
- reference counting work only for acyclic structures, for cyclic memory objects you must use some garbage collection technique.

3 Changes

- 19.03.2002 - first english version complete.
- 07.03.2002 - created.

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.