Preprint submitted to ACM SIGPLAN Notices, 17.04.2007.

Raw Pointers in Application Classes of C++ Considered Harmful

Igor B. Smirnov*
Petersburg Nuclear Physics Institute

Abstract

In order to achieve good reliability, clarity, scalability and re-usability of the application high-level programs written in C++ the raw pointers should not be used as class members (fields), because the raw pointers are too dangerous in this role and they do not represent meaningful relations between objects. Instead of raw pointers two smart pointers should be used. The first one describes exclusive unique ownership with synchronization of objects copying and deletion. The second one describes inclusive references to independent alien objects with invalidating the reference at the deletion of the addressed object. All power of object-oriented programming is preserved, but difficulties and errors are eliminated.

Keywords: C++, methodology, raw pointers, smart pointers, dynamic storage.

1 Introduction

The C++ programming language is now one of the most popular and powerful languages, but its application can lead to serious problems with reliability of the programs. These problems are caused by frequent latent bugs in operations with raw pointers and in dynamic memory management. The raw pointers and "`new/delete`" seem to be too low-level constructs and commands, than it is acceptable for high-level application programs.

There is some similarity of harmfulness of the raw pointers and that of the "`goto`" statement [1, 2]. After the Dijkstra's classical manifesto "Go To Statement Considered Harmful" [1] the `goto`'s were abolished and replaced by "if-then-else" clauses, loops, and procedures in ordinary algorithm-oriented languages. But today we have potentially more powerful object-oriented languages, such as C++, which allow us to describe complicated data structures and which frequently rely on the raw pointers as means to link objects. However the raw pointers are not well suited for the purpose of establishing meaningful relations between objects. As well as `goto`'s, they are too low-level constructs, both mean jumps in the computer memory and both introduce dynamic variation in the sense of the surrounding context, whereas it was noticed that "our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed." [1]. Similarly to the `goto` instructions, the raw pointers and dynamic memory management cannot be just removed, but should be replaced by something.

The application of pointers in C++ is very diverse. One of their classical applications is their use as a sort of a "handle" for regular arrays. Today the regular arrays can be replaced by template containers, which hide the raw pointers from the user and permit only safe operations. The template containers can support more diverse data structures than plain arrays. The examples are many-dimensional arrays, lists, segmented one-dimensional arrays, which can grow without reallocation, but still provide reasonably efficient indexing. The template containers can perform all necessary checks such as index-out-of-bounds checks. They are supplied as standard components of compilers, they are also available from free-distributed libraries or can be easily programmed independently.

*Mail: I. B. Smirnov, Petersburg Nuclear Physics Institute, Gatchina 188300, Russia.
E-mail: Igor.Smirnov@cern.ch.

The notion of orthogonality [3], as a way of "reduction in the number of special rules and exception of rules", which also implies similarity of behaviour inherent to conceptually similar language constructs, motivates the assignment to general-purpose containers value and copy semantics, so that the elements of a container could be automatically copied and destroyed together with the container's "head" or "handle". This is similar to the class fields, which are copied and deleted simultaneously with the whole object of this class.

The considerations of orthogonality motivate also the requirement for containers to keep *a number* of *similar* elements, that is elements of the same type (cf. [4] §16.2.3, p.442). The elements of variable types (base or derived) can then be allocated with the help of the pointers in the dangerous regular style (also cf. [4] §16.2.3, p.442) or with the help of their safe replacement that will be described in this paper. This is similar to the work with classes and class fields: the class field of the variable type can also be expressed via the pointer in the regular style or via the pointer replacement in the methodology described.

So we need safe and convenient means for addressing *single* elements of possibly *variable* types (base and derived). For the sake of orthogonality we will divide the replacements of pointers by semantics into two opposite categories: one type of the replacements will follow value and copy semantics, whereas the other type will provide an almost pure pointer semantics: it will not clone or delete the referred object at copying or deletion of itself, but it will be *annulled* itself at the deletion of the object. The term "annulled" means here that an attempt to derefer such a pointer will trigger a run-time error with a clear diagnostic message (normally, but in "on-line" programs which are not desirable to terminate even in the case of an internal error this may be replaced by throwing an exception). This second type of the pointer replacement will be used to refer to an object "placed somewhere else", "owned by something else", "independent" of the current context, "located at some known address". The first type of the pointer replacement will emulate exclusive ownership of the referred object by the entity, which owns this replacement of the pointer. Normally this will be an object of some class or an instance of some container. For simplicity we can also say that this type of the pointer itself owns the pointee object.

While in some programs the pointers to a single objects may not be used at all, in others a large fraction of their data structures are described through such relations and hence by such pointers. This is especially actual for various modeling and simulation programs, in particular for that in high energy physics. Initialization of a large number of dynamically allocated objects having variable types with a limited lifetime and establishing various links between them is a common practice. The application of the regular C++ with its raw pointers and `new/delete` can lead to the loss of a large fraction of work time for explicit pointer and storage management and for tracking down pointer and storage bugs, whereas the products can be routinely used or shipped with many such bugs [5–7]. Such bugs can be latent, that is they may not exhibit themselves clearly enough. They may cause abnormal behaviour much later than the actual error occurs. They may not exhibit themselves at all until the program is modified. New bugs can be easily added at modification. Therefore it can be very difficult to make such a program work, to modify it correctly, to apply this program as a subroutine for a larger-scale program developed for a more general purpose, to re-use this program for another purpose, to prove its correctness in the end. "Dangling pointers, storage leaks, memory smashes, and out-of-bounds array indices are the bane of the C++ programmer and his customers" [5].

Dangling pointers can be excluded if we forbid to delete the object until all references to it disappear. One of the most consistent ways to provide this consists in abolishing value semantics in total and using only reference semantics with allocation of all objects in dynamic (heap) memory with application of a retention strategy, that is forbidding deletion of objects by the user and providing the automatic recognition and deletion of the unused objects. This however is difficult to achieve in general case although many algorithms have been proposed. In particular, this is because not all referred objects are necessary (for example, in the case of circular references). The most serious drawback of this concept is that the always-reference-semantic notations of class fields or container elements do not distinguish the *member* objects that are *unique* for this "header" object (or for this container) from the *references* to *alien* objects, which are *common* for many other objects or containers. So this approach solves the technical problem of dangling pointers, but conceptually makes the situation much worser. The numerous attempts to identify these different relations between objects somehow [8–15] result in rather awkward systems, and all seem to drift conceptually backward to value-semantics languages.

In C++ the retention strategy with a kind of garbage collection can also be implemented through a variant of well known "shared smart pointers" [16–18]. Circular references are also a problem here. In addition, the possibility of direct and interior references by raw pointers means that not all non-referred (through a shared smart pointer) objects are unnecessary. On the other hand, the shared smart pointer is inadequate to refer to an interior object (a field of a class or an element of a plain array) or to an object allocated in automatic or global memory, because such objects cannot be deleted by it. The expressiveness of the shared smart pointers is also a problem.

A kind of exclusive ownership is proposed in "resource management in C++" methodology by Milewski [19]. However, this methodology is implicitly based on the assumption that the "resources" are uncopyable. "A resource is something that can be acquired and released. Examples of resources are file handles and chucks of memory... ." The smart pointer of Milewski "owns" the resource or the object according to his "rule one", but *transfers* ownership at copying leaving the initial smart pointer empty. The `auto_ptr` pointer from contemporary standard template library has similar semantics, see [4] (§11.4.1, p.367). According to Stroustrup [4] this semantics is also called "destructive copy semantics"[1]. In practice such semantics is not convenient since it provokes errors due to unexpected disappearance of the objects, in particular due to their "sinking" in functions, when the smart pointers of this type are passed as function parameters [4, 16, 19].

2 Object management methodology

Thus there should be two smart pointers: the strong "owning" one and weak "non-owning" pointer to an "independent" object.

The "owning" smart pointer should copy (clone) the objects at the copying of self leaving both the source and target pointers with valid independent objects.

The frequently heard remark about inefficiency of deep copying is not consistent: the pointer of this type simply should not be copied when the complete copy of the object is not necessary; instead the second type of smart pointers can be used in such situations. We will not consider here various tricks, known as copy prevention or copy-on-write [20, 21]. As noted by Lea in [20], "in many applications, tricks like copy-on-write add more overhead than they save" (perhaps except problems with returns, which can be solved by other means).

The owning smart pointer should also delete the object at the deletion of self.

The owning smart pointer can be "cleared", which means that the object kept by it is deleted.

The pointee object for this type of pointers can be allocated only in dynamic memory.

The other kind of smart pointers, the pointer that refers to an independent object, should not clone or delete the referred object at copying or deletion of itself, but it should be annulled itself at the deletion of the object. The object (the pointee for this type of pointer) can be allocated in any kind of memory and can (and should) be deleted by any means, and this should automatically trigger the clearing of the pointer to it.

The both types of smart pointers can be `NULL` before initialization and after clearing. An attempt to derefer such a `NULL` pointer by ordinary ways triggers a run-time error. There should be a possibility to

[1] It is also called there "ownership semantics", but the use of such term would be rather confusing in the context of this paper and methodology. Similar notation "ownership transfer semantics" is better for our purposes.

determine whether the pointer is `NULL` without triggering the error.

In practice the choice of the correct category from these two, the ownership and the independence, almost never causes difficulty. Almost all references between objects necessary in programming can be described by one or the other kind of relations or their combination. It is very unlikely that one would need any other behaviour in addition to these two, and this behaviour could not be emulated, avoided, or implemented as an optional behaviour of these two pointers. Therefore, in particular, the necessity of practical applications of the more flexible "policy-based" smart pointers, first developed by Alexandrescu [22], does not seem to be well justified, although this idea is interesting. Another interesting in this context work is [6], which describes an error-detection system based on automatic recognition of ownership in programs based on raw pointers. One of the conclusions made in this paper is that "the object ownership model [the concept from this paper that assumes that each object is exclusively owned by some other object or block; note that the methodology that we discuss also assumes this] is used in all the three [studied] C++ applications." However they also admit that "it is sometimes appropriate to manage some objects in a program using other techniques, such as reference counting...". The numerical estimates of the number of errors which they found by this system in the tested programs are also very instructive.

Thus there are two primary relations: ownership and reference to an independent object. The corresponding smart pointers are convenient to call *active* and *passive* pointers. Earlier the author also called them controlling and protected pointers [23], but the corresponding abbreviated notations in real programs were found to be not so understandable. In programs it is convenient to denote their classes by names `ActivePtr< >` and `PassivePtr< >`, respectively.

The "strong" cloning pointers, similar to the active pointers described above, were discussed earlier by Barton and Nackman in [17]. A precise description of the smart pointers coinciding with our passive pointers has not yet been found in the literature. The nearest analogues found are the proposal of special "weak" smart pointers designed to work together with garbage collectors [5] and also weak references in Java and other reference-semantics languages. In contrast with such garbage-collector-related weak pointers (or references) to heap memory the passive pointers are assumed to refer to any object or a part of any object in any kind of memory (not only in the heap) and to be not interfaced with a garbage collector. All the shared, weak, and passive pointers can be implemented through some forms of technique, called in ref. [2] "reference counted tombstones". Note that the alternative approach with individual backward pointers from the pointee objects to each passive pointer cannot be used by considerations of efficiency. This observation, which was first mentioned in [24] in the context of improvement of core languages and hardware assistance, is equally valid for smart pointers and is firmly confirmed by practice. Therefore "generic defensive smart pointers" proposed in [25] and an example with something similar to an emulation of passive pointer lists through "path-based rules" and backward pointers, see [26], cannot be used as general-purpose tools. On the other hand the reference counting technique allows us to implement also the shared pointers as an alternative behaviour of the passive pointers to a particular object. Then the last passive pointer to this object will delete it at the destruction of itself. But it is not recommended to use this option in practice, because switching it on destroys the whole concept of ownership vs. independence and returns many technical problems such as restrictions on the allocation methods of the addressed object. The author has implemented this possibility in his library just "to make sure", but has never used it in practical applications yet. This variant of behaviour is simply kept switched off by a special macro-definition.

There is a lot of important details regarding the possible implementations of this methodology in standard C++. In this brief paper most of them should be omitted. We just mention, that the both smart pointers in standard C++ have to be intrusive, that is they need some ingredients in the addressed classes. But these ingredients are not burdensome. There are no problems with abstract classes and multiple inheritance. Both pointers can be elements of arrays, either regular or template. The passive pointer can also address elements of arrays or their interiors.

In total, the strongly shortened class definitions of smart pointer classes in the author's implementation look like:

```
enum Pilfer { steal };
enum Clone { do_clone };
enum Pass { dont_clone };

template<class X, class C = StandardCopyDefinition<X> >class ActivePtr
{public:                                            // Active pointer.
  inline X* operator->(void) const;   // With NULL check.
  inline X& operator*(void) const;    // With NULL check.
  inline X* getver(void) const;       // With NULL check.
  inline X* get(void) const;          // Without NULL check.
  inline void put(const X* fptr)      // Clone and take copy.
    {  // Here you can see an example of how copy() is applied:
      X* ptr_temp = (fptr != NULL ? C::copy(fptr) : (X*)(NULL));
      if(ptr!=NULL) delete ptr;
      ptr = ptr_temp;
    }
  inline void pass(X* fptr);          // Do not clone, take this.
  inline X* extract(void);            // Clear and return raw address.
  inline void clear(void);            // Clear, delete object.
  inline void pilfer(const ActivePtr<X,C>& f); // Extract from f and put here.
  void print(ostream& file, int l=1) const;    // Debug print.
  inline ActivePtr(void);
  inline ActivePtr(const X* fptr, Clone);      // Clone and take copy.
  inline ActivePtr(const ActivePtr<X,C>& f);   // Clone and take copy.
  inline ActivePtr(const ActivePtr<X,C>& f, Pilfer); // Extract from f and put here.
  inline ActivePtr(X* fptr, Pass);    // Take this.
  macro_copy_total(ActivePtr);        // Macro defining copy function in order to make
   // the smart pointer itself addressable by another smart pointer (which is scarcely needed).
   // Replaced by virtual ActivePtr* copy(void) const {return new (ActivePtr(*this);}.
  virtual ~ActivePtr();               // For the same purpose.
private:
  mutable X* ptr;  // The raw pointer to the target object,
                   // ``mutable'' is needed only for pilfer function and pilfer constructor.
};

template<class X>class PassivePtr   // Passive pointer.
{public:
  friend class RegPassivePtr;       // The base class of addressed object.
  inline X* get(void) const;        // Without NULL check.
  inline void put(const X* fptr);   // Register the raw pointer, do not clone, take this.
  inline PassivePtr(void);          // Create empty (NULL) pointer
  inline PassivePtr(X* fptr);       // Register the raw pointer.
  inline PassivePtr(X& fptr);       // Register the raw pointer.
  inline PassivePtr(const PassivePtr<X>& f);              // Copy the pointer.
  inline PassivePtr<X>& operator=(const PassivePtr<X>& f); // Copy the pointer.
  inline PassivePtr<X>& operator=(X* f);  // Register the raw pointer.
  template< class Y > PassivePtr< X >(const PassivePtr < Y >& f);
                                 // Works if Y is converted to X by default.
  inline void move_pointer(PassivePtr<X>& f); // Pass pointer from f to this clearing f.
  inline X* operator->(void) const;   // With NULL check.
```

```
  inline X& operator*(void) const;    // With NULL check.
  inline X* getver(void) const;       // With NULL check.
  long get_total_number_of_references(void) const;  // ... to the addressed object.
  void print(ostream& file, int l=1) const;         // Debug print.
  macro_copy_total(PassivePtr);
  virtual ~PassivePtr();
private:
  CountPP_ns::CountPassivePtr* cpp;    // Reference counter, through which
                                       // the object is addressed.
#ifdef USE_DOUBLE_PTR_IN_PASSIVEPTR
  X* ptr;         // Allows to address directly the object and to
#endif            // avoid dynamic casts from RegPassivePtr to X.
};
```

The whole C++ class definitions are currently kept in [27].

In the program written in C++ there should be *no* raw pointers defined as class members of user-defined classes[2] or as elements of template containers or built-in arrays. It needs to proceed from the axiom that *each* such simple pointer means a problem, either already existing, or inevitable in the future, after the next "improvement" of the program. So in the user program there should not be statements like this:

```
class AnyCompositeClass: ...
{ ...
  AnyClass* ptr;
  ...
};
AnyClass* built_in_array_of_raw_pointers[100];
AnyContainer< AnyClass* > container_of_raw_pointers;
              // Implying that container_of_raw_pointers[n] has type AnyClass*
              // (if the container were 1-dimensional array).
```

but may be statements like

```
class AnyCompositeClass: ...
{ ...
  ActivePtr< AnyClass > a_ptr;
  PassivePtr< AnyClass > p_ptr;
  ...
};
ActivePtr< AnyClass > built_in_array_of_active_pointers[100];
PassivePtr< AnyClass > built_in_array_of_passive_pointers[100];
AnyContainer< ActivePtr< AnyClass > > container_of_active_pointers;
AnyContainer< PassivePtr< AnyClass > > container_of_passive_pointers;
```

Pointers are used not only as class fields (class members) that describe the relations between objects, but also as variables in algorithms (functions) that provide the access to objects. While the simple pointer is not allowed in the first role, it is sometimes allowed in the second one. Indeed, if a regular simple pointer is obtained by safe dereferencing of a smart pointer (by `getver()` in the listing above) and the pointee object cannot disappear in the scope of the regular pointer, the careful use of the regular pointer for access to this object (except for deletion of it) can sometimes be reasonably safe. Typically such safe dereferencing and then the use of the obtained raw address is possible inside small functions which use information kept outside them in long-living data structures. It is sometimes justified both

[2]That is of an "application" classes, which do not belong to a category of thoroughly elaborated library classes of common use.

by the considerations of efficiency and by shorter notation. In addition, the simple raw pointer can also be required sometimes as a function argument [16]. Thus the careful use of simple pointers inside the bodies of functions or as their arguments is not strictly forbidden. The raw pointers should not be used for *description of the information structure*, but can be carefully used for *access* to it, as well as for other occasional operative purposes. The raw pointers are not forbidden at all in the application programs, but they are forbidden *in the role of class members* in the application programs. Needless to say, the raw pointers have to be used also in the implementation of the smart pointers as well as in containers and in a lot of other auxiliary library classes. Therefore all words in the specification "defined as class members of user-defined classes" in the beginning of the previous paragraph are significant.

Since the main notion of the described methodology is a copyable object, this methodology can be conveniently called the "object management methodology". We mean the methodology of programming rather than the methodology of "object-oriented analysis and design" as in [28], although the application of the programming methodology that we describe, has noticeable influence on the methodology of design and in fact changes it. The key ideas of the object management methodology were first described in the appendix of [23]. This methodology was adapted, tested, and practically used only in C++ so far. Therefore we attribute it here explicitly to programming in C++, although it seems to have much wider meaning. It might be considered as a structure of the next generation of object-oriented programming languages, which should allow us to classify not only *objects*, but also *relations* between objects.

3 Experience of application

The author has been using this methodology for many years in high-energy physics mostly for computer modeling of particle detectors and experiments (interactions of particles with matter, as described in [29,30]), for computer treatment and analysis of experimental data (see [31,32]), for some other mathematical and sometimes for system-related problems. The metrics of these scientific programs by the order of magnitude was 10^5 lines of the source code. It was observed that while the debug of programs based on raw pointers (with the use of raw pointers as class fields) and on manual memory management (with the calls of `new`/ `delete` from application programs) is painful and time demanding task, the debug of similar programs based on the described methodology is rather a formal and easy procedure. Rephrasing the statement form [25] — "the vast majority of pointer-specific bugs are detected at the point they are actually created" for our methodology it can be said that *the vast majority of pointer-specific or memory-management-specific bugs simply disappear and the minor rest is instantly detected at the point they are actually created and then easily corrected*. Moreover, it was noticed that the described methodology indirectly (but strongly) helped to deal with *other* kinds of bugs and problems, because it allowed the programmer to *exclude the possibility* of the latent bugs of these two kinds. Otherwise the programmer would need to check them again and again. The methodology also made the programs much more simple and understandable. In particular, it allowed to write complicated programs usually without any special design stage and without drawing class diagrams. It also allowed to reconfigure the programs quickly and painlessly. For example, to switch between ownership and reference one usually needs only to change the notation of the corresponding class field from `ActivePtr` to `PassivePtr` or vise versa, whereas the traditional methods require editing class constructors, destructors, assigmnent operatos and possibly other member functions, in which the programmer has a lot of chances to make a lot of latent errors. In particular, the manual coding of the assignment operators is not necessary at all, cf. the "orthodox canonical form" (OCF) from [21] (where it is necessary) and see [4], §11.3.4, p. 271 — "writing out the memberwise copy by hand is tedious and error-prone for classes with many data members".

Regarding the speed of calculations is it difficult to make precise predictions. As the first guess, the programmer can expect the speed reduction by something like 20% at the application of this methodology in "a normal" way, i.e. in a safe enough way, that is probably with switching on NULL-checks and with other precautions. The real number will depend on the program, computer, compiler, compiler options, numerous options related to the implementation and use of smart pointers (such as switching on and off NULL-checks) and on other numerous circumstances.

4 Conclusions

The simple raw pointers used as class members of application classes in C++ are very harmful (like "`goto`" statements in algorithms of high-level programs). But they cannot be just removed since they are used to establish meaningful relations between objects that cannot be established by the other means. The classification of these relations into the most basic categories: the exclusive ownership and the inclusive references to independent objects — make possible the full replacement in class fields of raw pointers by smart pointers of two types which emulate polymorphic inclusion and reference to independent objects, respectively. They should be applied together with template value- and copy-semantic containers. This methodology allows us to eliminate dangling pointers, memory leaks, to eliminate almost all segmentation faults, to reduce probability of other errors, to reduce drastically time required for debug, to increase reliability of the final programs and to make the programs much more clear and understandable.

References

[1] E.W. Dijkstra, Go to statement considered harmful. *Communications of the ACM* 1968; **11**(3): 147–148.

[2] D.B. Lomet, Making Pointers Safe in System Programming Languages. *IEEE Transactions on Software Engineering* 1985; **SE-11**(1): 87–96.

[3] B. T. Denvir, On Orthogonality in Programming Languages. *ACM SIGPLAN Notices*, 1979, **14**(7): 18–30.

[4] B. Stroustrup, *The C++ Programming Language.* (Special ed.), Addison-Wesley, 2000.

[5] J.R. Ellis and D.L. Detlefs, Safe, efficient garbage collection for C++. In *USENIX Sixth C++ Technical Conference*, 1994. http://www.usenix.org/publications/library/proceedings/c++94/full_papers/ellis_a'.

[6] D.L. Heine and M.S. Lam, A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, June 9–11, 2003, San Diego, California, USA.

[7] F.V. Tkachov, Programming education: a Russian perspective. In *Modular Programming Languages. Lecture Notes in Computer Science*, Springer-Verlag, 2003, **2789**: 69–77.

[8] J. Hogg, D. Lea, R. Holt, A. Wills, D. deChampeaux, The Geneva Convention On The Treatment of Object Aliasing. *OOPS Messenger*, April 1992.

[9] N.H. Minsky, Towards Alias-Free Pointers. *Proc. 10-th European Conf. Object-Oriented Programming*, Springer-Verlag, 1996, 189–209.

[10] P.S. Almeida, Balloon Types: Controlling Sharing of State in Data Types. In *ECOOP'97 — Object Oriented Programming*, II European Conf. *Lecture Notes in Computer Science* 1997; **1241**: 32–59. Springer.

[11] D.G. Clarke, J.M. Potter and J.Nobble, Ownership Types for Flexible Alias Protection. In *ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, Vancouver, Canada, October 1998.

[12] J. Noble, J. Vitek and J. Potter, Flexible Alias Protection. *ECOOP'98*, *Lecture Notes in Computer Science*, Springer-Verlag, 1998; **1445**: 158–185.

[13] J. Boyland, Alias burying: Unique variables without desctructive reads. *Software — Practice and Experience* 2001; **31**: 533–553.

[14] M.Barnett and D.A.Naumann, Friends Need a Bit More: Maintaining Inveriants Over Shared State. In *Mathematics of Program Construction*, *Lecture Notes in Computer Science*, Springer, July 2004.

[15] A. Banerjee and D. Naumann, Ownership: transfer, sharing, and encapsulation, In *ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP)*, July 2003.

[16] S. Meyers, *More Effective C++.* Addison-Wesley, 1996.

[17] J.J. Barton and L.R. Nackman, *Scientific and Engineering C++.* Addison-Wesley, 1994.

[18] G. Colvin, B. Dawes, P. Dimov, D. Adler, et al., The boost web site, smart pointers. 1999–2004. www.boost.org/libs/smart_ptr/smart_ptr.htm'.

[19] B. Milewski, Resource management in C++. *Journal of Object Oriented Programming* 1997; **10**(1): 14-22.

[20] D. Lea, The GNU C++ Library, *C++ Report*, June 1993, revised as http://gee.cs.oswego.edu/dl/libg++paper/libg++/libg++.html, 1995.

[21] A. J. H. Simons, Borrow, Copy or Steal? Loans and Larceny in the Orthodox Canonical Form. *ACM SIGPLAN Notices*, *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '98)* 1998; **33**(10): 65–83.

[22] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, 2001, See chapter at http://www.informit.com.

[23] I.B. Smirnov, Likelihoodlib — fitting, function maximization, and numerical analysis. *PNPI Preprint* 2001; **2421**. http://cdsweb.cern.ch/.

[24] D. Lomet, Scheme for invalidating references to freed storage. *IBM J. Res. Develop.* 1975; **19**(Jan): 26–35.

[25] A. Savidis, The implementation of generic smart pointers for advanced defensive programming. *Software-Practice and Experience* 2004; **34**: 977–1009.

[26] D. Litman, P.F. Patel-Schneider, A. Mishra, J. Crawford, and D. Dvorak, R++: Adding Path-Based Rules to C++. *IEEE Transactions on Knowledge and Data Engineering* 2002, **14**: 638–658.

[27] I.B. Smirnov, http://cern.ch/ismirnov/safetl, 2005–2007.

[28] G. Booch *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, 1994.

[29] I.B. Smirnov, Modeling of ionization produced by fast charged particles in gases. *Nuclear Instruments and Methods in Physics Research A* 2005; **554**: 474–493.

[30] I.B. Smirnov, http://cern.ch/ismirnov/heed, 2005.

[31] I.B. Smirnov, Track reconstruction for forward spectrometer of SPES4-π experiment. *PNPI Preprint* 2000; **2345**, http://cdsweb.cern.ch/, http://cern.ch/ismirnov/track.

[32] G.D. Alkhazov, V.V. Astashin, A.G. Atamanchuk, et al., SPES4-π: installation for exclusive study of nuclear reactions. *Nuclear Instruments and Methods in Physics Research A* 2005; **551**: 290–311.